Computer Graphics Lab 3

Name: James Davies
Student ID: 200682354

October 2023

Contents

1 Exercise A

1.1 A1 - Modelview transformations

A1 revolves around the modelview matrix, as we change the perspective in which we view the banana model; first rotating around the y-axis at an extended distance, then translating this to a close up viewpoint rotating around the x-axis.

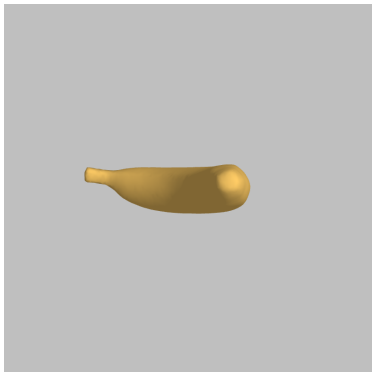

Figure 1: A1-i

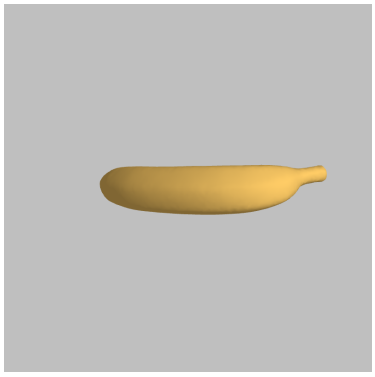

Figure 2: A1-ii

To achieve the first translation, in which we view the banana model from a closer perspective, we can use the following line of code:

```
let scaling = mat_scaling(vec_scale(12,[1,1,1])); // A1
```

From the code snippet we can see how utilising the vec_scale function allows us to pass a scaled vector (by a factor of 12) to mat_scaling instead of the original [1,1,1] vector. To implement the next aspect of A1, in which we change the axis on which the model rotates, we can use the following line:

```
let motion = mat_motion(theta, [0,1,0], [0,0,-z]); // A1
```

This is after we have already defined the rotation variable, using Math.PI/2 passed to the mat_motion function. The finalisation of this transformation is via a updated definition of the modelview:

```
modelview = mat_prod(mat_prod(motion, rotation), scaling); // A1
```

in which we call mat_prod firstly on the motion *and* rotation, before passing it again to get the matrix product of the prior result and scaling.

1.2 A2 - Multiple models

A2 builds off what we changed in A1, using the banana model from a close up perspective, but we will be utilising the fact the drawing functions are within a for loop in the JS code. This enables us to increase the number of iterations and therefore models rendered.

This is implemented alongside the random modelview parameters generated within the init() function of the JS file. First we have to set the number of models to render via the num_models variable (in my case 50), and then turn to the for loop within the render() function. As mentioned prior, the for loop iterates for as many models as there are.

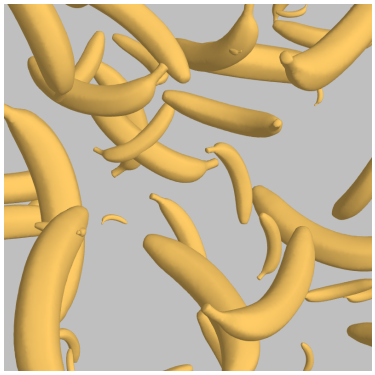

Figure 3: A2-i

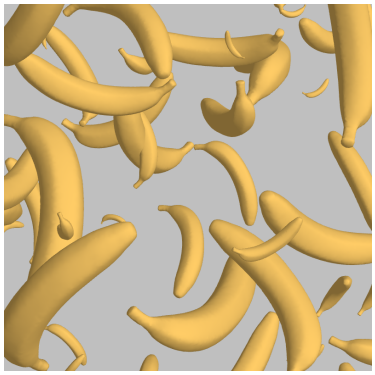

Figure 4: A2-ii

Within this for loop, we must change the motion, rotation and scaling variables to allow for each model to be rendered. The chosen iterative variable is 'k', and so the code passed is as so:

```
let motion = mat_motion(theta, transform[k].axis, transform[k].location); // A2
let rotation = mat_motion(Math.PI/2, transform[k].axis, [0,0,0]); // A1 + A2
```

with the scaling variable altered in the same manner with transform[k].scale.

2 Exercise B

2.1 B1 - Phong shading

From exercise B onwards, we change models from the banana model to Suzanne. We begin with a simple rendering of 'gold' material without the specular term implemented, after which we implement the specular of the Phong shading model.

Figure 5: B1-i

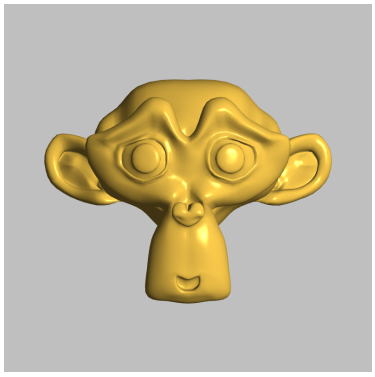

Figure 6: B1-ii

The specular term is implemented via the declaration and initialization of a vec4 variable; using: power function, max (returns max value between two) and dot product functions with the reflection/shininess coefficients, alongside the specular terms of the material and lighting as values.

```
// B1 -- IMPLEMENT SPECULAR TERM
vec4 specular = material.specular *
                pow(max(dot(r,t),0.0), material.shininess) *
                light.specular;
```

2.2 B2 - Lighting parameters

Staying with the Phong shading model, we begin to experiment with changing the lighting parameters at the top of the JS file. This is implemented by changing the lighting properties; specifically the diffuse component for the 'warmth' element, and position for positioning of the light.

Figure 7: B2-i

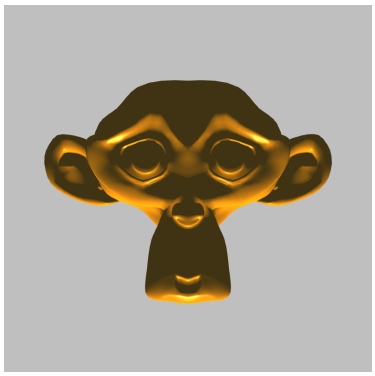

Figure 8: B2-ii

2.3 B3 - Blinn-Phong shading

In B3, we begin to entend the Phong shading model, implementing the Blinn modification to the existing model. This allows us to remove the reliance on the reflection vector, and instead use a unit vector which is halfway between the light source and target; creating a better sense of realism.

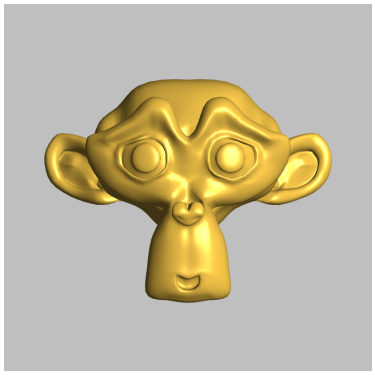

Figure 9: B3-i

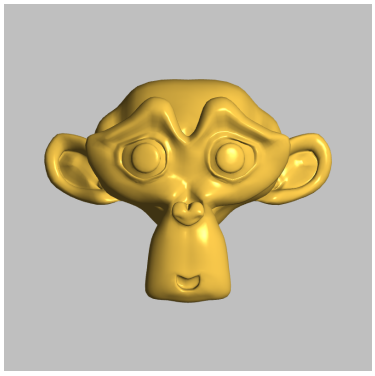

Figure 10: B3-ii

We first have to declare an initialize a 'halfway' variable, in which we will eventually pass to the specular term. We can calculate this via the use of the

normalization function, to which we pass the addition of source (light source) and target (viewer) variables.

```
// halfway vector
vec3 halfway = normalize(source+target); // B3
```

This halfway variable, instead of using the dot product of the reflection coefficient in the specular, is now passed to the dot() function. The renormalized interpolated normal is also passed to this function, so we now have the product of the halfway and normal as one of the values in the specular vec4 calculation.

2.4 B4 - Per-vertex lighting

The modern standard of shading involves lighting computations within the fragment shader (aptly referred to as per-fragment shading), with old methodologies performing calculations in the vertex shader (Gouraud shading). In B4 we will experiment with the two methodologies and illustrate the differences via the back of the Suzanne model.

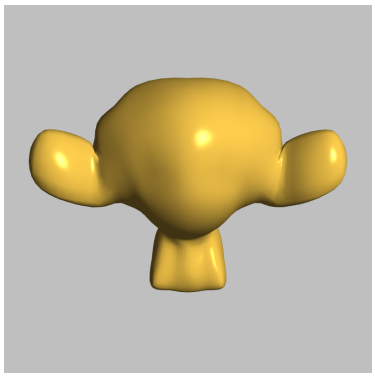

Figure 11: B4-i

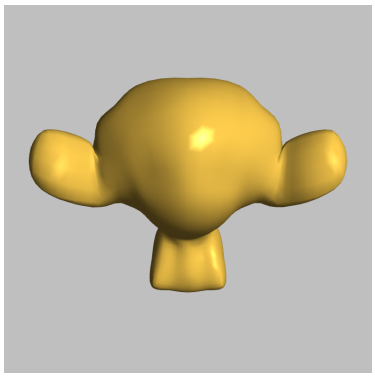

Figure 12: B4-ii

To implement Gouraud shading, we first need to copy all of the lighting calculations and data from the fragment shader, into a new vertex shader. We then declare colour as a varying vec4 type variable before main(). Colour is already declared and set as the value for gl_FragColor in a new pass-through fragment shader which does nothing else apart from the aforementioned functionality.

```
colour = vec4((ambient + diffuse + specular).rgb, 1.0); // B4
```

The core changes to the code for B4 is the fact that we no longer pass the vector with ambient, diffuse, specular and 1.0 as values to gl_FragColor. Instead we now set this vector to the colour varying vec4, as the code snippet above shows.

3 Exercise C

3.1 C1 - Model structure

In exercise C, we will use a series of changes to the underlying structure to understand the inner workings of the Phong model; firstly visualising the structure of Suzanne via points instead of triangle mode rendering, and the using lines to create a mesh.

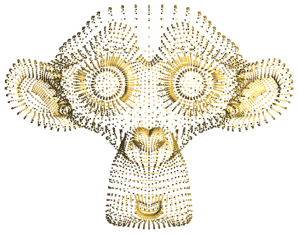

Figure 13: C1-i

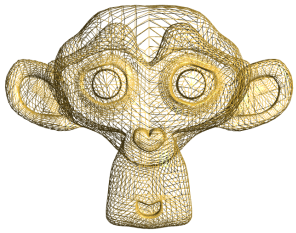

Figure 14: C1-ii

In both C1-i and C1-ii, the way in which we change the rendering is by altering the declaration of gl.drawElements(). The former, we can change the rendering mode to gl.POINTS and the latter gl.LINES.

```
gl.drawElements(gl.TRIANGLES, mesh.indexBuffer.numItems, gl.UNSIGNED_SHORT, 0);
```

In order to visualise these changes more accurately, like specified by the guidelines, it is helpful to change gl.clearColor() to white.

3.2 C2 - Surface depths

We will now use a GLSL function to visualise surface depth using a greyscale rendering, as we pass the depth coordinate of gl_FragCoord to this function (scene_depth).

Figure 15: C2

We first, as part of the core functionality of exercise C2, need to pass the depth coordinate (z) of gl_FragCoord to the scene depth function we used from lecture 6. However, like specified in the guidelines, it is also useful to use a linear transformation to aid visualisation. In this manner I have performed the transformation and the requisite function call within the same variable declaration and initialisation to scaled_depth.

```
// scaled depth + linear transformation
float scaled_depth = (far - scene_depth(gl_FragCoord.z)) / (far - near);
gl_FragColor = vec4(vec3(scaled_depth), 1.0); // C2
```

We are then able to use the scaled_depth variable as a vec3 type to pass to a vec4 type alongside 1.0 for alpha, which is in turn set to gl_FragColor.

3.3 C3 - surface directions

To visualise surface direction, we can essentially remove the front of the Suzanne model, by ceasing to render the closest fragments to the viewpoint. This is achieved via the given if statement that checks whether the depth coordinate of gl_FragCoord is less than 0.5, and if this condition is met then utilising the 'discard' functionality. We can then alter the position and ambient values of the lighting positions in the JS file to accomplish the model renderings from C3i-iv.

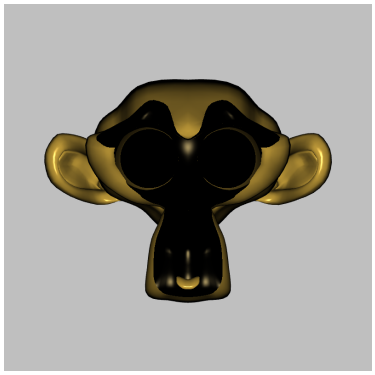

Figure 16: C3-i

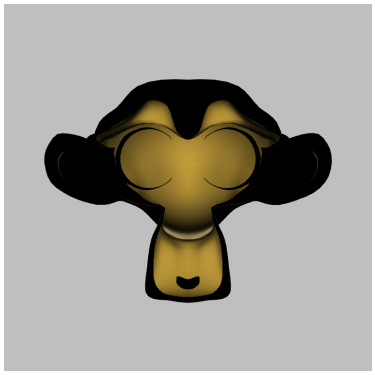

Figure 17: C3-ii

In regards to why the lighting effects don't align with the real world, I can only assume that the reasoning is because the 'lighting' isn't actually directional from a real-world physics perspective. As the lighting coefficients affect each fragment that it would be exposed to if it *were* a physically directional light source.

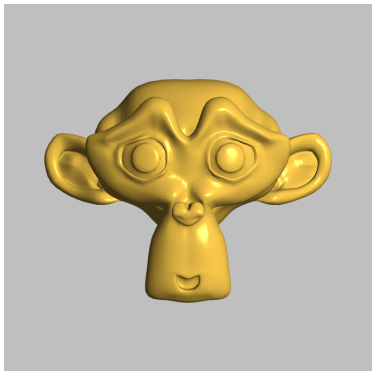

Figure 18: C3-iii

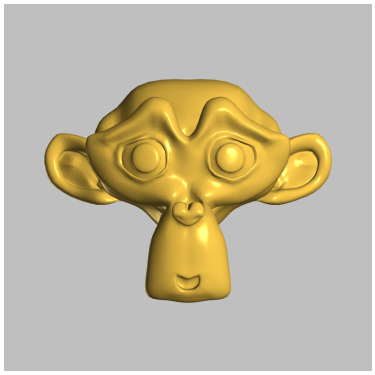

Figure 19: C3-iv

Hence in C3-i, the near fragments aren't there and the 'light' doesn't reach the rear fragments, so we don't see into the head. However, in C3-ii the rear frag-

ments *are* there and are affected by the light source at the backside of the head, but in this case the we're not seeing the head light up per se. The assumption is in this case we are just seeing the back side of the head 'illuminated', and we're seeing that through the head as the near fragments aren't rendered.

The difference between the two renderings C4-iii and C4-iv, the former using max() and the latter using abs() within the declaration of the specular term, is primarily to do with clamping and negative values. As in the prior method, max() will clamp negative values to 0.0; with abs() the positive value of the dot product will be taken even if the value is negative. This results in incorrect lighting/shading, especially in places where you don't expect specular illumination (i.e. under Suzanne).

3.4 C4 - Surface orientations (advanced)

In C4, we utilise a secondary GLSL function given called hsv_to_rgb() which takes a vector of length 3 (vec3), which like the function name suggests maps HSV values to RGB values. HSV stands for hue, saturation and value; the end result being a more detailed visualisation of how the surface is actually rendered as normals aligned to different axes will map to different levels of saturation.

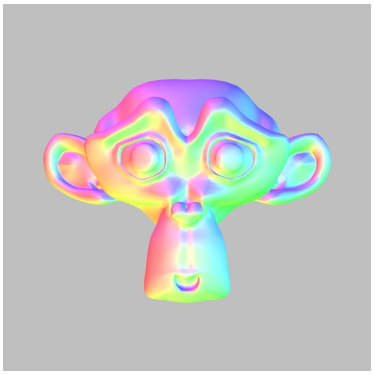

Figure 20: C4-i

We first implement this by calculating the hue and saturation. The hue we can calculate by using atan() to calculate the arctangent values of x and y of the normal, and then dividing this value by (2*PI)+0.5.

```
// calculations for hue and saturation
float hue = atan(n.y, n.x) / (2.0 * PI) + 0.5;
float saturation = acos(n.z) / PI;
```

HSV operates under the property of cylindrical geometry and how colours are affected by light, with: the colour white at the centre, increasing saturation towards the edges of the cylinder, and increased 'brightness' towards the top of the cylinder. The hue is based on where on the cylinder you take the value from, therefore we can calculate this via taking the arctangent divided by the degrees (converted to radians in our calculation with 2*Pi). Likewise, we can calculate the saturation by taking the arccosine of the depth value of the normal; giving us the placement between the centre and edge of the cylindrical structure representing HSV.

We then modify the saturation to improve visibility, taking the current saturation to the power 0.5.

```
// modified saturation to improve visibility
float modified_saturation = pow(saturation, 0.5);
```

Finally we declare and initialise the vec3 variable hsv as specified in the guidelines, with the brightness value set to 1.0.

```
// hue, saturation, value (hsv) initalisation
vec3 hsv = vec3(hue, modified_saturation, 1.0);
```

The end result, colouring wise, maps in an analogous fashion in regards to lighting, specular illumination and shadows i.e the more prominent the effect, the higher the saturation levels. In general, the darker elements of the Phong model lean towards pinks/reds, with the lighter elements leaning towards yellows/greens.

4 Exercise D

4.1 D1 - Basic bump mapping

In exercise D we return to our standard Phong shading model we implemented in B1, however in these exercises we will be using the GL buffers to visualise different forms of bump mapping. In D1-iii we implement a basic form of this, but firstly we begin D1-i and D1-ii by erroneously inserting random noise into mesh.vertices. We can then create a more realistic form of this methodology by inserting random noise into mesh.vertexNormals instead.
In D1-i and D1-ii, we can see how the attempt at bump mapping is naive, more so in the latter image where we have increased the number of perturbations.

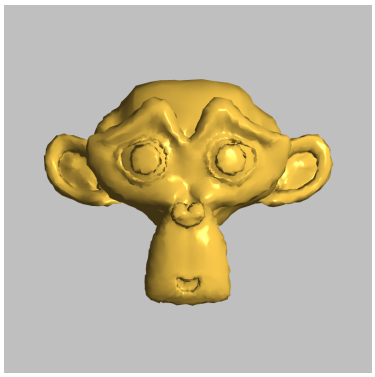

Figure 21: D1-i

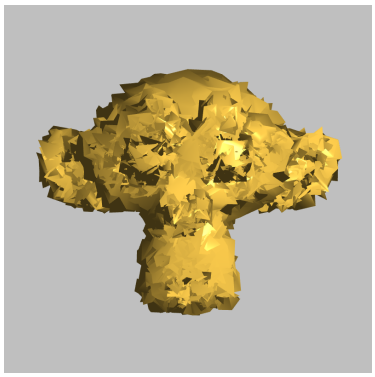

Figure 22: D1-ii

This is because we are attempting to create texture effects by randomly altering the size of vertices in the mesh, which is bound to cause issues as there are no limiting factors in which to keep the effect realistic. Even at small values the rendered triangles are occasionally jagged.

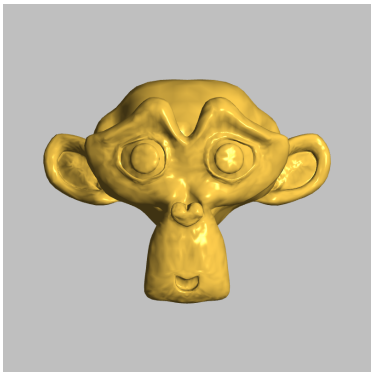

Figure 23: D1-iii

However, if we instead randomly alter the normals and leave the vertices in place (like in D1-iii) we get a more realistic effect. This effect is called bump mapping. The realism of this effect can be recognised via the outline of the Suzanne model; in D1-i the surface is sometimes angular, (completely so in D1-ii) and isn't what you would expect from a textured effect of a skull. However, in the model where bump mapping hs been successfully implemented, we can see that the 'bumps' are suitably rounded off and more natural.

4.2 D2 - Basic cel shading

Our final rendering is an attempt an 'non-photorealistic' or 'stylized' rendering. This is implemented via something called cel shading, used to shade the surface with solid colours.

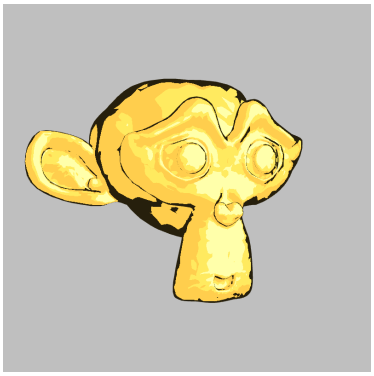

Figure 24: D2

From the pseudo-code given in the guidelines, we are able to define the following if statements to implement the cel shading. The conditions within the if/else if statement revolve around checking whether the dot product of source (light) and the normal is more than a given number. This either results in the material's specular value being assigned to gl_FragColor, or in the latter case, 0.2 * material's specular value being assigned to gl_FragColor. We then write a further if statement with a dot product as a condition. However, in this case the two values passed are the target (viewer) and the normal, and the condition that needs to be met is that it is less than 0.4. Within the actual if statement, it is the rgb value of gl_FragColor that is being assigned to, and material's diffuse.rgb value that is being assigned.

```
if (dot(source,n) > 0.9) {
    gl_FragColor += material.specular;
} else if (dot(source,n) > 0.75) {
    gl_FragColor += 0.2 * material.specular;
}

if (dot(target, n) < 0.4) {
    gl_FragColor.rgb = (0.2 * material.diffuse.rgb);
}
```