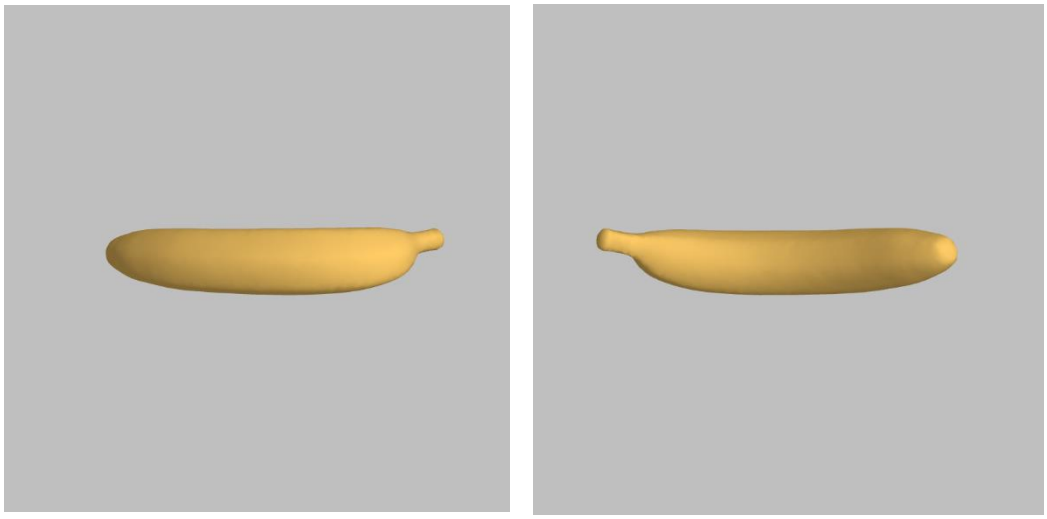
Computer Graphics Lab3 Report

Name(ID) : Erfan Rafieioskouei – 240842587

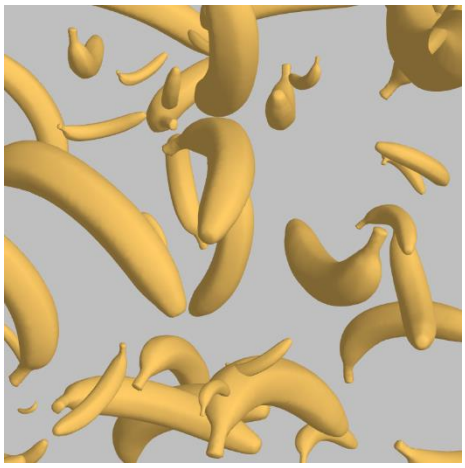
A1 — Modelview transformations

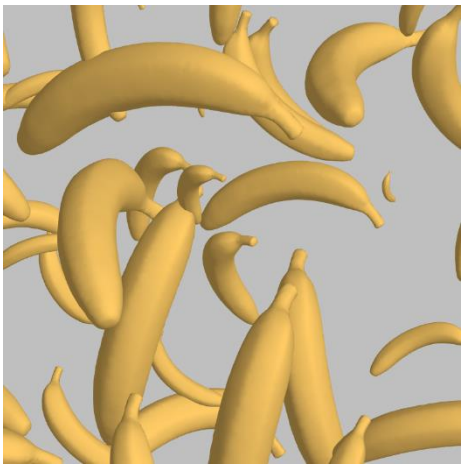


For this part I did exactly as the exercise wanted and made the changes in the render function. The updated code is as below :

```
        // average scene depth
        let z = (far + near) / 2;

        // scaling magnified by factor of 12
        let scaling = mat_scaling(vec_scale(12, [1, 1, 1]));

        // fixed rotation of 90 degrees around x-axis
        let fixed_rotation = mat_motion(Math.PI / 2, [1, 0, 0], [0, 0, 0]);

        // variable rotation around y-axis
        let motion = mat_motion(theta, [0, 1, 0], [0, 0, -z]);

        // rearranged modelview matrix
        modelview = mat_prod(motion, mat_prod(fixed_rotation, scaling));
```

A2 — Multiple models





So, to create random instances of the model I changed the num_models from 1 to 50. And in the init() function we have a loop that generates random transformation parameters for each model and stores them in transform array. As the code below :

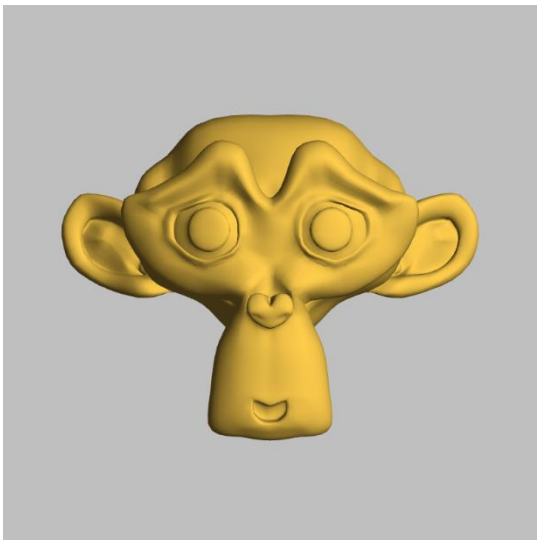
```
// random modelview parameters
   for(let k = 0; k < num_models; k++) {
       // empty struct
       transform[k] = {};
       // set random size, location, and rotation axis for each model
       transform[k].scale = vec_scale(random(1,10), [1,1,1]);
       transform[k].location = [random(-2,2), random(-2,2), -(far+near)/2 + random(-2,2)];
       transform[k].axis = [random(-1,1), random(-1,1), random(-1,1)];
   }
```

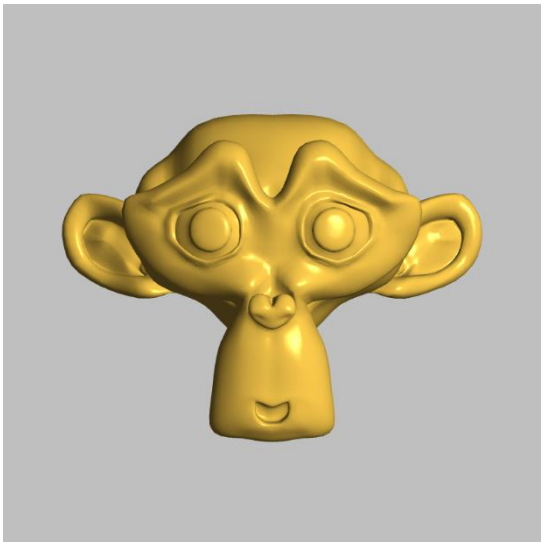
And then in render function in the loop I retrieve random parameters and assign them to the objects. The code is as below :

```
let scale = transform[k].scale;
       let location = transform[k].location;
       let axis = transform[k].axis;

       let scaling = mat_scaling(scale);
       let motion = mat_motion(theta, axis, location);

       modelview = mat_prod(motion, scaling);
```

The result creates the pictures given.

B1 — Phong shading





For the first picture I changed the material number based on the diagram given. The code is as below :

```
// B1 -- MODIFY
var material = {
    // banana
    ambient:  [0.24725, 0.1995, 0.0745, 1],
    diffuse:  [0.75164, 0.60648, 0.22648, 1],
    specular: [0.628281, 0.555802, 0.366065, 1],
    shininess: 51.2
};
```

For the second picture. In the fragment shader, I added the specular term which is a key component of the Phong reflection model. The original code only had ambient and diffuse lighting components, making surfaces appear matte. By implementing the specular calculation, I added the capability to render shiny highlights on surfaces.

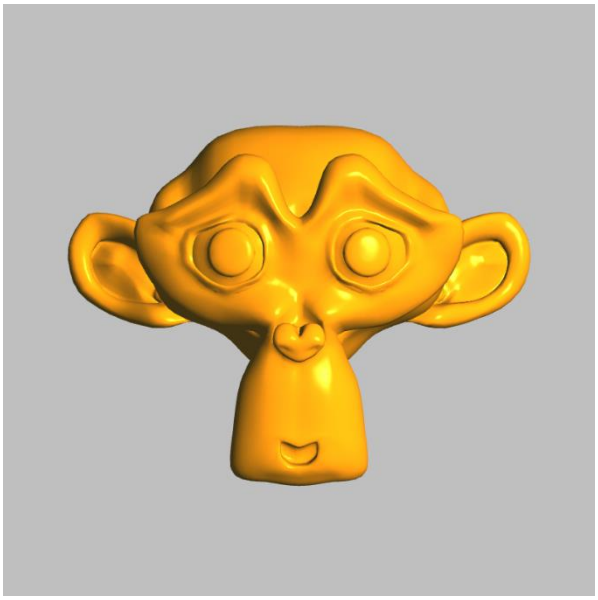
The implementation uses the reflection vector (r) and view vector (t) to determine where specular highlights should appear. The intensity of these highlights is controlled by the material's shininess value. When the reflection vector closely aligns with the view direction, a bright highlight appears, simulating how light reflects off shiny surfaces in the real world.

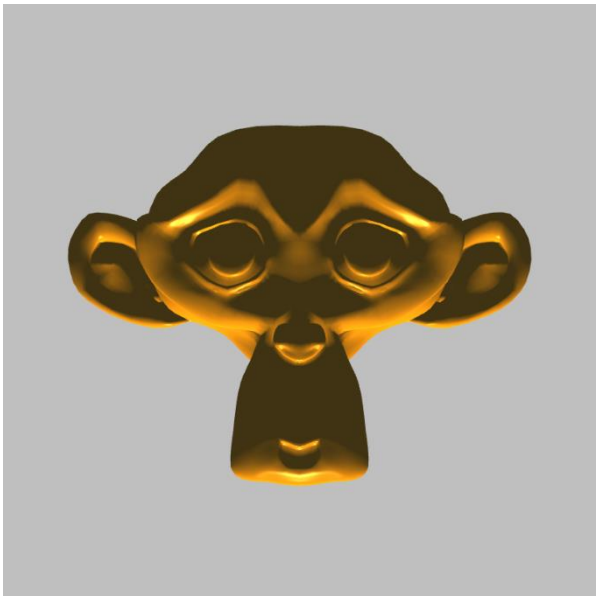
This specular term was then added to the existing ambient and diffuse components in the final color output (gl_FragColor), completing the full Phong illumination model. This creates more realistic lighting with proper highlights on shiny surfaces.

The updated code is as below :

```
void main()
{
    // renormalize interpolated normal
    vec3 n = normalize(m);

    // reflection vector
    vec3 r = -normalize(reflect(s,n));

    // phong shading components

    vec4 ambient = material.ambient *
                   light.ambient;

    vec4 diffuse = material.diffuse *
                   max(dot(s,n),0.0) *
                   light.diffuse;

    // B1 -- Implement specular term
    vec4 specular = material.specular *
                    pow(max(dot(r,t),0.0), material.shininess) *
                    light.specular;

    // B3 -- IMPLEMENT BLINN SPECULAR TERM

    // Output final color including ambient, diffuse and specular components
    gl_FragColor = vec4((ambient + diffuse + specular).rgb, 1.0);
}
```

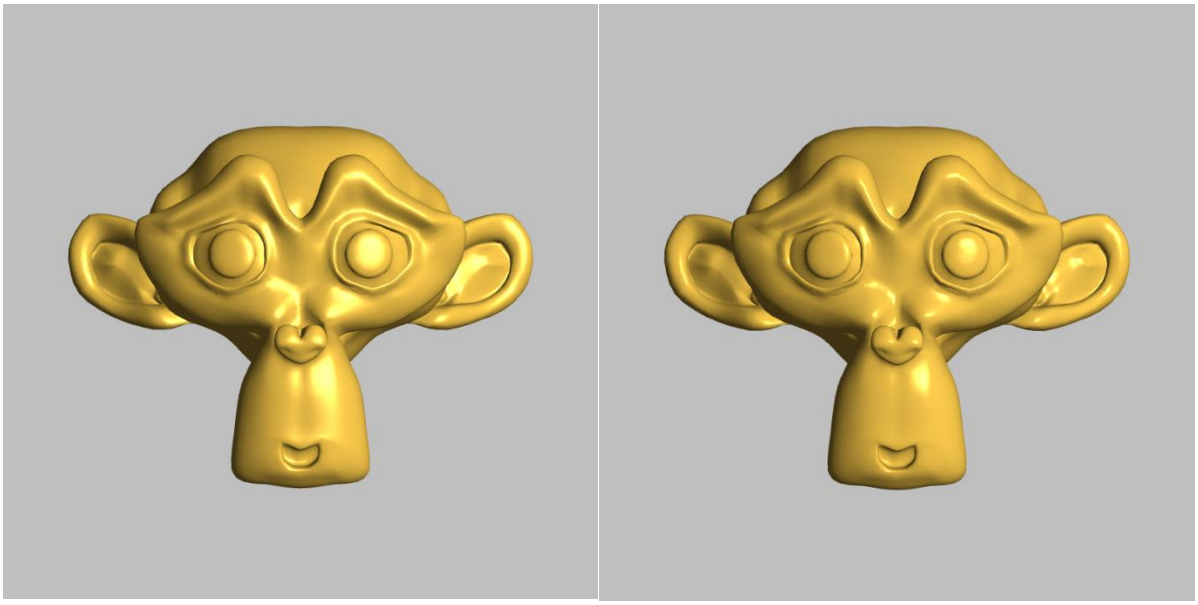
B2 — Lighting parameters





For the two pictures I did as the exercise wanted and changed the diffuse and position to new values.

B3 — Blinn-Phong shading



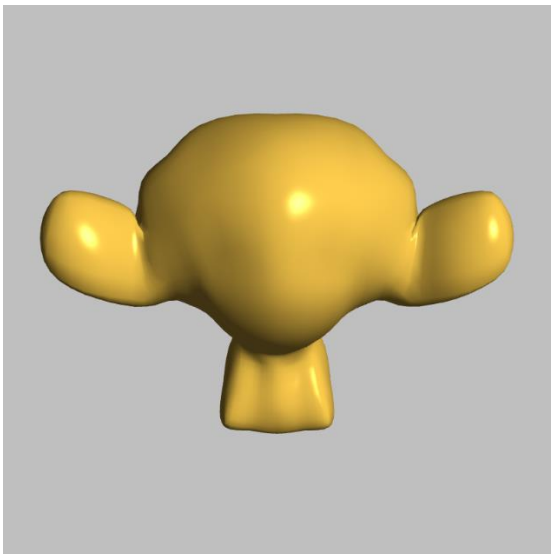
For this part, I removed the reflection vector calculation then added the halfway vector calculation and modified the specular term to use the halfway vector.

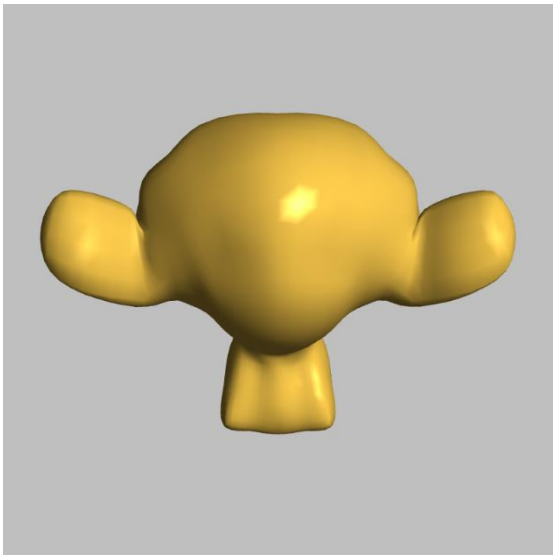
```
// compute halfway vector instead of reflection vector
    vec3 h = normalize(s + t);
 // B3 -- Blinn specular term implementation
    vec4 specular = material.specular *
                    pow(max(dot(h,n),0.0), material.shininess) *
                    light.specular;
```

and the ambient and diffuse components remain unchanged as they are the same in both models. And for the second picture I've just added a 4 multiplication.

The Blinn-Phong modification(B3-i) offers computational efficiency by replacing the reflection vector calculation with a simpler halfway vector, but produces noticeably broader specular highlights. This spread occurs because the angle between normal and halfway vector is typically smaller than the angle between reflection and view vectors. Multiplying the shininess exponent by 4.0(B3-ii) compensates for this geometric difference, achieving highlights that closely match the original Phong model(B1-ii) while retaining the performance benefits and physical plausibility of the Blinn-Phong approach.

B4 — Per-vertex lighting



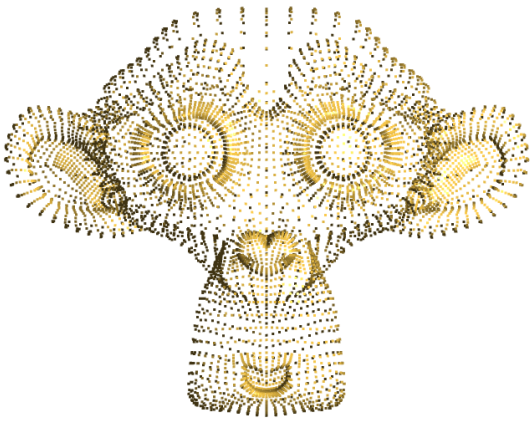


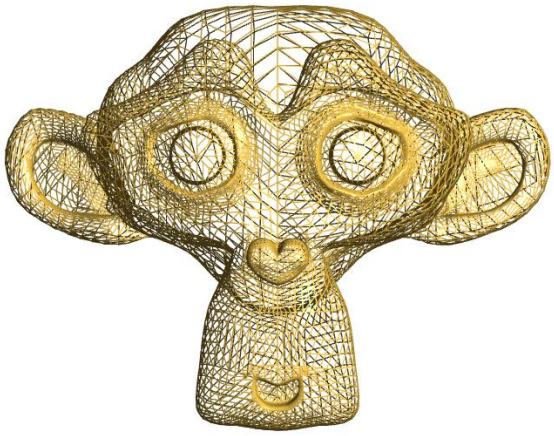
I did carefully as the exercise told and my description of the differences is that comparing images B4i and B4ii reveals the quality difference between per-fragment and per-vertex lighting implementations.

B4i (per-fragment) shows smoother, more natural specular highlights on the curved surfaces of the head and ears. In contrast, B4ii (per-vertex) exhibits more angular, faceted highlights due to lighting calculations being performed only at vertices with results interpolated across triangles.

This difference is most noticeable in the specular highlights because the Blinn-Phong specular component varies non-linearly across surfaces. While per-vertex lighting is computationally cheaper, the visual artifacts demonstrate why per-fragment lighting remains preferred for high-quality rendering, particularly for materials with prominent specular highlights.

C1 — Model structure





The resolution of the mesh varies significantly depending on the complexity of the surface:

High-Detail Areas: Regions with finer curvatures as with the eyes and the ears reveal a higher number of vertices and smaller triangle sizes. Thus, the detail of the model is secured to the best extent possible.

Flat or Simple Areas: Smaller and simple triangles cover more complex areas of the face, on the other hand large and simple vertices cover areas that need less detail for instance the cheeks and the forehead.

This is why studying the model in different modes helps to understand how the density of vertices is achieved optimally concerning the representation of the surface and its complexity.

C2 — Surface depths

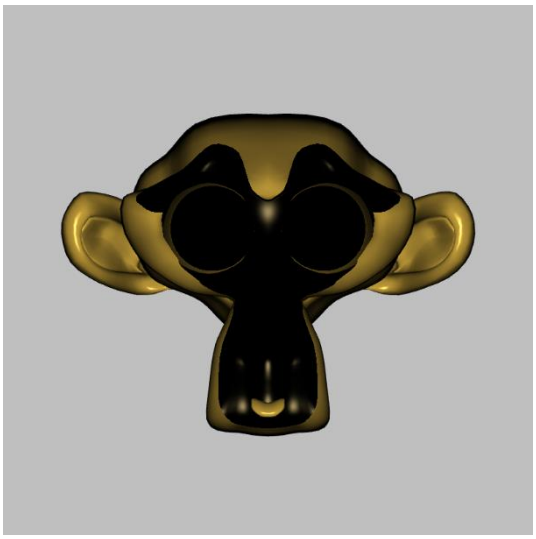
Smoothness of the Depth Map:

This was a greyscale depth map which looked fairly smooth since the change from one depth value to another was gradual across the diffuse models of the object.

Compared to the POINTS and LINES modes, which were more quantitative because of the vertex data structure, the depth shading provided a more integrated view of the 3D geometry.

But where there were complex geometric transitions, such as around the eyes, then the depth shading overall remained smooth because of finer mesh in these areas.

C3 — Surface directions



The unusual lighting effects can be attributed to the limitations of the basic rendering pipeline and the way light interaction is modeled in shaders.

No Light Scattering or Transmission:

In a realistically modeled environment, light would penetrate inside the model (subsurface scattering) or through translucent media. Standard shaders, for instance, presuppose that surfaces are opaque and always do not consider such processes.

Calculations of lighting are performed through surface normal and position with respect to light source. Light at near plane does not fall on interior surfaces thereby the interior is unseen and hence unlit.

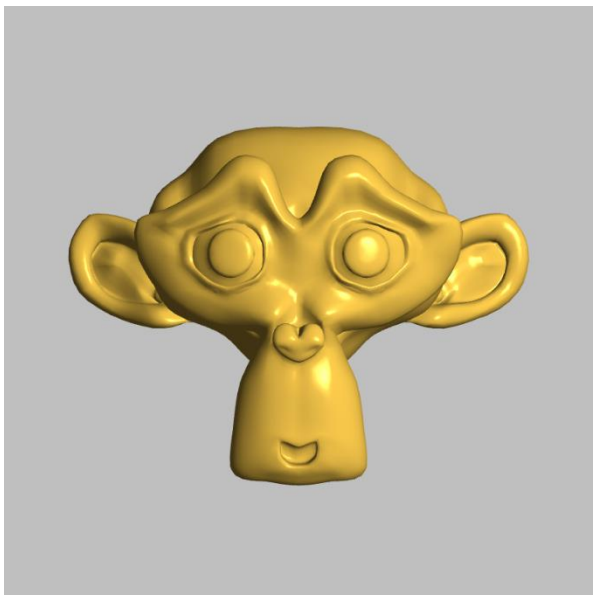
Back face Illumination:

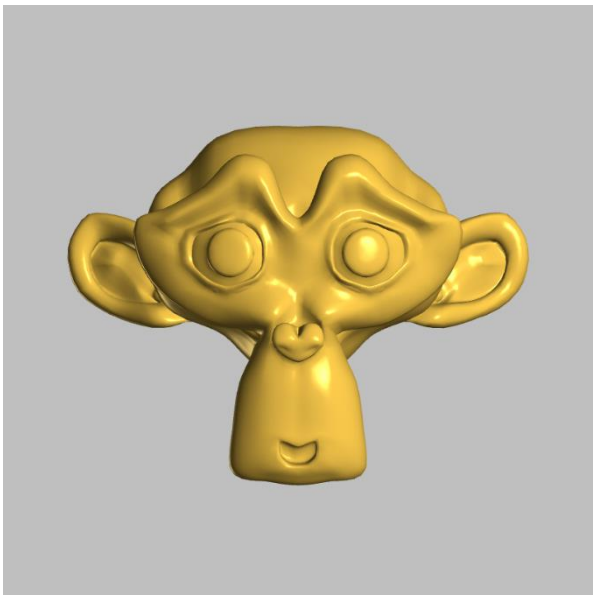
If the light is positioned behind the model placed, it then comes in contact with the interior faces to illuminate them. This is because its lighting algorithm does not consider the fact that some surfaces that are illumination are not physically reachable by the light.

Lack of Global Illumination:

Real-world lighting contains intervening light, a light which bounces off an object and gets to another; such would help light the interior. Without those global illumination techniques, such effects are missing and therefore physics violation occurs.

C3iii–iv.png :



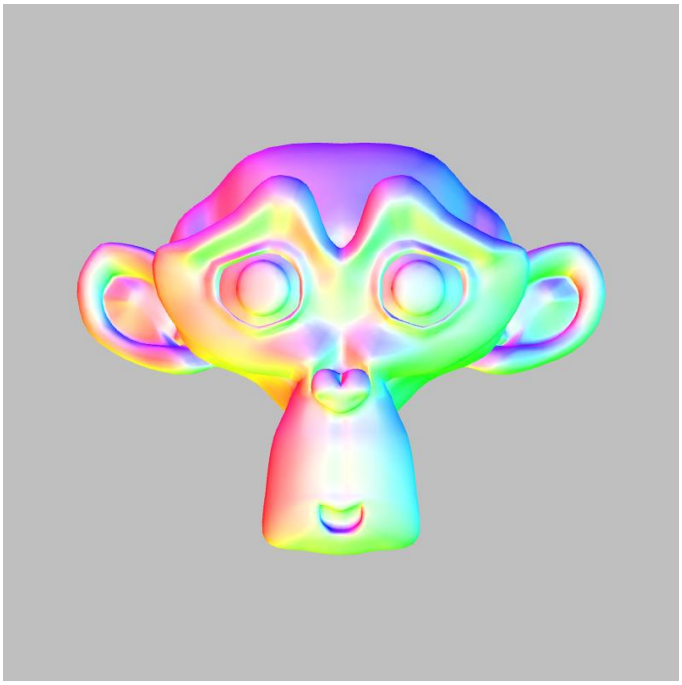


Switching of the specular term from max(dot(r, t), 0.0) to abs(dot(r, t)) brings non-physical behavior into the light equation. This leads to significant artifacts, especially sections of the reflection vector pointing in the wrong direction from the viewer resulting in wrong highlights.

Such mistakes are gradually exacerbated when the object is rotated since the artificial shine interferes with the look of the material.

As previously stated, the max() function was retained in order to keep the specular term realistic to light and visually appealing.

C4 — Surface orientations (advanced)



For the code part of this section, I did as the exercise wanted and the changes are as follows :

```
// HSV to RGB conversion function

vec3 hsv_to_rgb(vec3 c) {
    vec4 k = vec4(1.0, 2.0/3.0, 1.0/3.0, 3.0);
    vec3 p = abs(fract(c.xxx + k.xyz) * 6.0 - k.www);
    return c.z * mix(k.xxx, clamp(p - k.xxx, 0.0, 1.0), c.y);
}

void main()
{
```

```
    // renormalize interpolated normal
    vec3 n = normalize(m);

    // Calculate hue from x,y components of normal using atan
    // atan(y,x) returns angle in range [-π, π], so we need to map to [0,1]
    float hue = atan(n.y, n.x) / (2.0 * PI) + 0.5;

    // Calculate saturation based on z component
    // When normal points up (z=1), saturation should be 0 (white)
    // When normal is horizontal (z=0), saturation should be 1 (full color)
    float saturation = sqrt(1.0 - abs(n.z));  // Using pow(saturation,0.5) as suggested

    // Create HSV color with value=1.0
    vec3 hsv = vec3(hue, saturation, 1.0);

    // Convert HSV to RGB
    vec3 rgb = hsv_to_rgb(hsv);

    // Output final color
    gl_FragColor = vec4(rgb, 1.0);
}
```

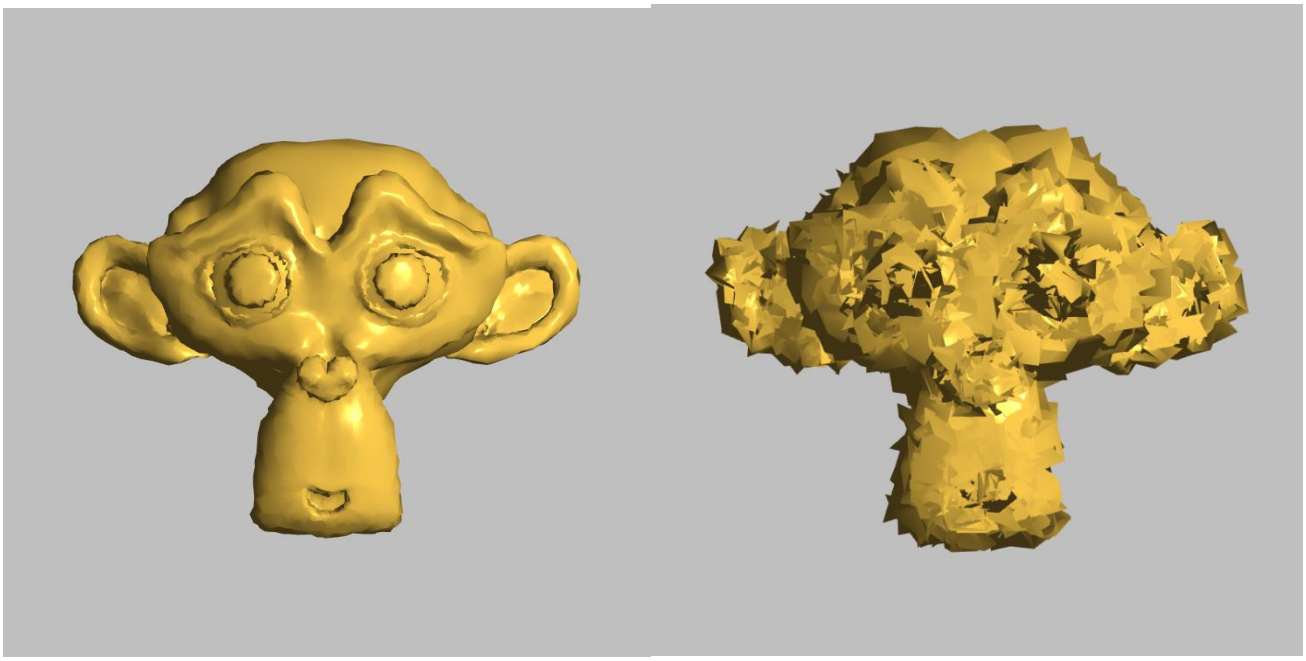
So, my comment is that the HSV normal visualization provides an intuitive way to understand surface geometry through color encoding, where hue represents the XY-plane orientation and saturation shows deviation from vertical.

Unlike Phong shading which depends on light position and view angle, this visualization reveals surface features consistently through direct color mapping of normal orientations.

While Phong highlights only appear at specific light-surface-view alignments, the HSV approach shows all surface orientations simultaneously, making it particularly useful for geometric analysis and quality assessment.

White areas indicate vertical normals (Z-aligned), while saturated colors reveal more horizontal surface orientations, providing immediate insight into the surface structure that complements traditional Phong shading's light-dependent representation.

D1 — Basic bump mapping



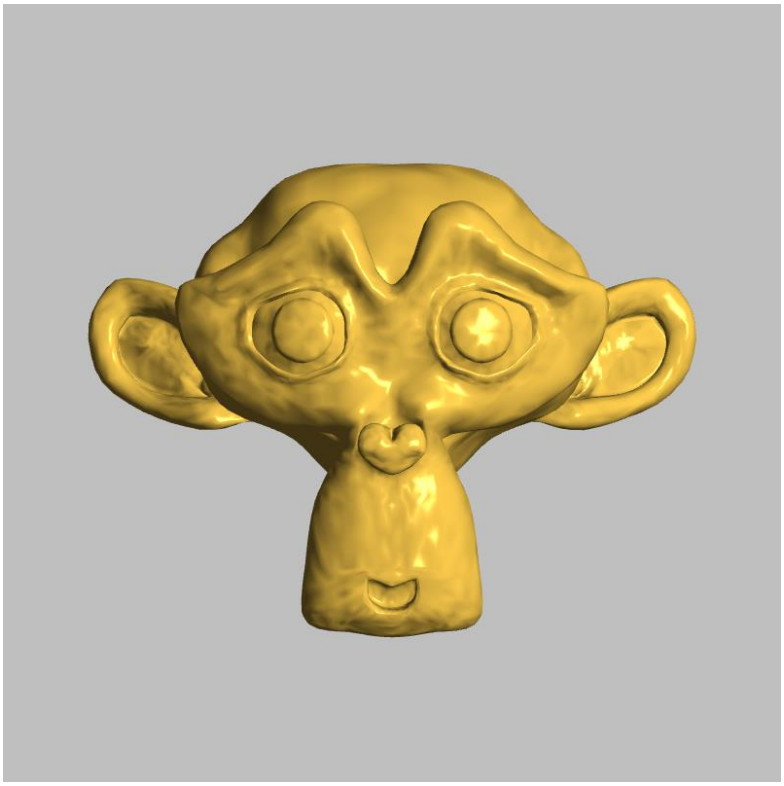
I’ve implemented the changes in code like wanted and the output changes based on the perturbation of the noise.

```
// Add random noise to the mesh vertices
    for(let i = 0; i < mesh.vertices.length; i++) {
        mesh.vertices[i] += 0.1 * random(-1, 1);
    }
```

For the next part, I’ve changed the for loop to the code below :

```
// Add random noise to the mesh vertex normals
    for(let i = 0; i < mesh.vertexNormals.length; i++) {
        mesh.vertexNormals[i] += 0.15 * random(-1, 1);
    }
```

And the output looks like this :

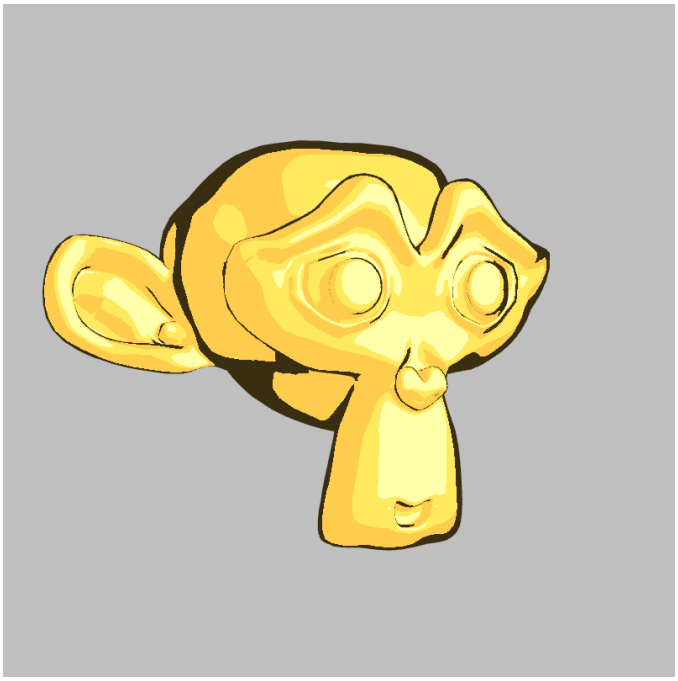


If we compare the two approaches shown in this exercise, the key differences in their results are :

Vertex perturbation technique generates actual geometric perturbation because of which the results are rather unrealistic, and it is clearly visible that triangle interlinks and mesh distortions are present. The most suggestive characteristic is the discontinuous contour which is especially evident during the model twirl, disrupting the continuity of the actual model.

Bump mapping using normal vector perturbation produces reasonable surface details and does not affect the structural aspect of the model. The critical observation here is to notice how the contour / silhouette of the surface is continuous and does not differ from the original model while surface is granular while seeking to express details.

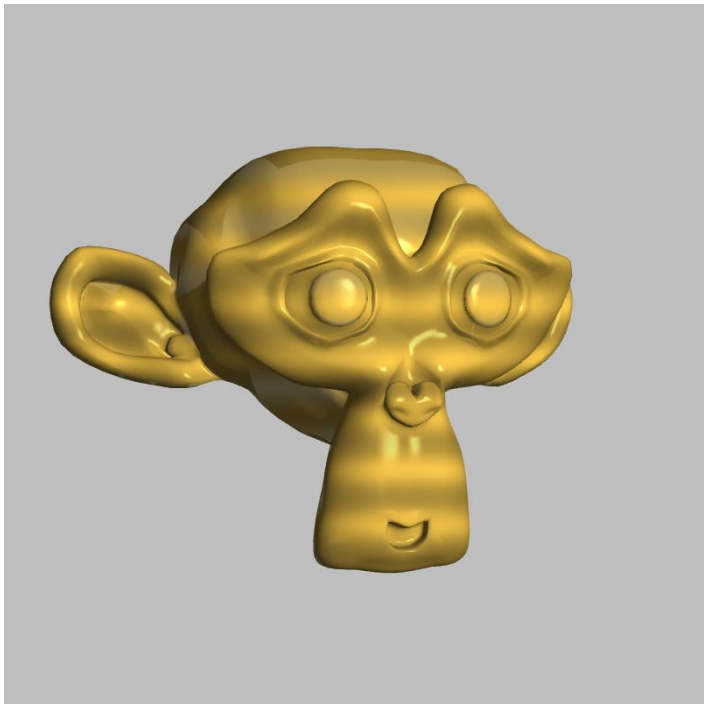
D2 — Basic cel shading



I’ve implemented the required things for the FS code.
We can clearly see the characteristic features of cel shading:

1. **Distinct color bands:** Notice how the yellow color doesn't smoothly transition but instead has clear "steps" or bands of shading. This is particularly visible on the cheek and forehead areas.
2. **Hard edges:** There's a stark black outline around the entire model, which was achieved by our view-dependent darkening (where dot(t,n) < 0.4). This creates the distinctive cartoon-like silhouette.
3. **Specular highlights:** You can see small, sharp highlight regions (the lighter yellow spots) particularly around the eyes and on the forehead. These aren't smooth like in Phong shading but appear as distinct patches, following our stepped implementation.

D3 — Artistic effects (advanced)



I transformed the original Phong lighting shader into one with a glitch aesthetic by implementing several visual distortion techniques while preserving the core lighting calculations.

First, I added a time uniform to drive the animations and created utility functions for generating pseudo-random values and glitch blocks. The main effect combines several layers: a displacement mapping that shifts pixels based on time-varying noise, RGB channel splitting that creates chromatic aberration effects, and scanlines that move vertically across the surface.

The glitch effect works by taking the original Phong-shaded color (ambient + diffuse + specular) and applying controlled distortions to it. The displacement is calculated using sine and cosine waves modulated by random noise, which creates a jittery, unstable look.

Color channels are independently shifted using different time scales, creating that characteristic digital corruption appearance. Finally, scanlines and random vertical shifts are added to simulate video signal interference. All of these effects are parameterized, allowing for easy adjustment of the glitch intensity and frequency.