

COMPUTER GRAPHICS ASSIGNMENT2

In this assignment, you will explore mesh subdivision, simple lighting techniques, and using simplified normal mapping to fake additional geometry.

1: Loading the Model (5%)

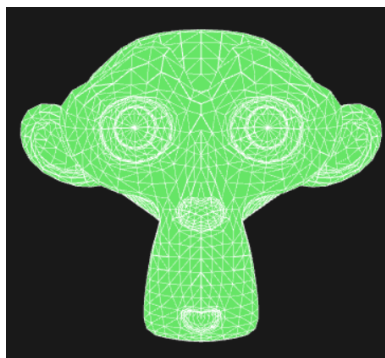


Figure 1: The 3D monkey model loaded from `monkey.obj`

First, we will load in our 3D model from the `monkey.obj` file. To do this, implement the missing code for the `loadMeshData(s)` function in `01_load_obj.js`.

You should fill in and return the `vertexBuffer` array such that it is a single flat list of floating point numbers. Each vertex in the buffer should be composed of 8 values - 3 for the vertex's 3D coordinates, 2 for the 2D texture UV coordinates, and 3 for the normal vector at that vertex. The model is composed of 3752 faces with 3 vertices each, so the final contents of `vertexBuffer` should contain 90048 floating point values.

For more information on the format of `.obj` files and how to load their contents, see Tutorial 3. You can view the final results by opening `cg_assignment_2.html` in a WebGL 2.0 compatible browser.

NOTE: To open with Google Chrome, add the command-line flag:
`-allow-file-access-from-files` to ensure the 3D cube model file can be loaded.

2: Phong Shading (30%)

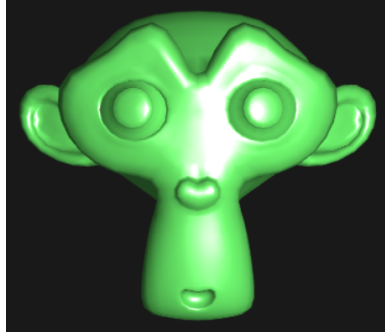


Figure 2: The model with per-fragment phong illumination.

We can make our model look more interesting by adding illumination to it. In Tutorial 3, we performed per-vertex lighting and interpolated the colour results across the fragments within each triangle. Now, we will perform this lighting calculation once per fragment instead, which will give it a smoother appearance. You will need to edit the GLSL vertex and fragment shaders in `shaders.js` here.

(a) Normals (10%)

Add a pair of `in/out` variables to the vertex and fragment shaders to pass the normal from the vertex to the fragment shader for later the lighting calculations.

(b) Phong Illumination (10%)

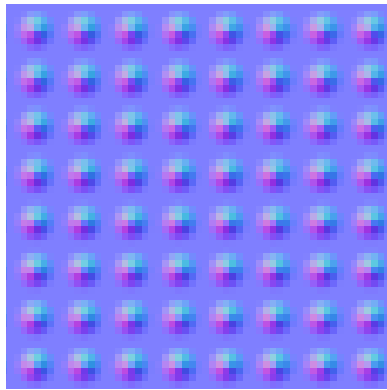
Perform phong lighting in the fragment shader using the interpolated normal from above, and the relevant material and lighting properties provided in the fragment shader for you. Both the Phong formulation, and the Blinn-Phong formulation using half-vectors are acceptable here.

(c) Altering Shininess (10%)

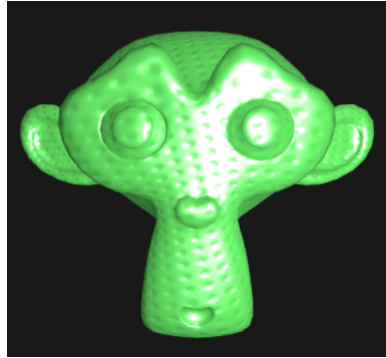
Change the variable `shininess` in the fragment shader from a `const` value to a `uniform` value so it can use values provided from the javascript host code. Edit

the `03_render.js` file to pass in the value from the `shininess` slider the fragment shader so you can dynamically adjust this lighting parameter.

3: Simple Normal Mapping (30%)



(a) Normal map texture



(b) Normal map applied

Normal mapping is commonly used to add extra surface details without any additional vertex geometry. You will implement a simplified version of this technique by reading from a texture and perturbing the vertex normal by that amount.

A normal map with a basic repeated dent pattern is supplied for you in `bump.png`. Feel free to experiment with different normal mapped textures to see how they effect the model. One method for generating such normal maps from ordinary images is to use a plugin for GIMP 2.

(a) Texture UVs (10%)

Add `in` and `out` variables to the vertex and fragment shaders to pass the texture uv coordinates from the vertex shader so that the fragment shader can use them to read in the normal map texture.

(b) Perturbing Normals (10%)

Sample the colour from `normalMapTexture` at the correct texture uv coordinates you passed from the vertex shader above. Multiply the xyz (or rgb) components

of this sampled colour by `displacementScale`, and add the result to the normal you passed in for question 3(a).

(c) Phong Lighting (10%)

Use the perturbed normal in the phong illumination calculation from part 3. You should be able to alter the scale of this normal mapping effect using the slider provided.

4: Subdivision (35%)

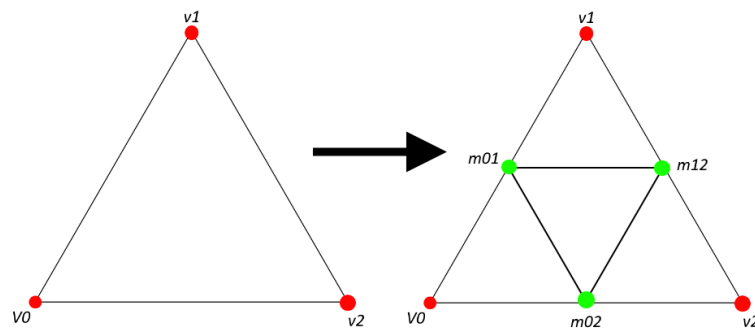


Figure 3: The triangle v_0, v_1, v_2 subdivided into 4 triangles at edge midpoints

We can add more detail to our mesh by subdividing its triangles into many smaller ones. This is done using the subdivision scheme shown in Figure 3, where we add extra vertices at the midpoints of the triangle's edges. This splits each single triangle into 4 smaller ones, and this process can be repeated to recursively subdivide meshes.

(a) Adding new vertex data (20%)

Fill in the missing section of `subdivideMesh` in `02_subdiv.js` so that uses the data from `originalVertexData` (which is a flat float array in the same format as you generated in part 1), and returns a new float array in the same format containing the new vertex data about the subdivided triangles. You should linearly

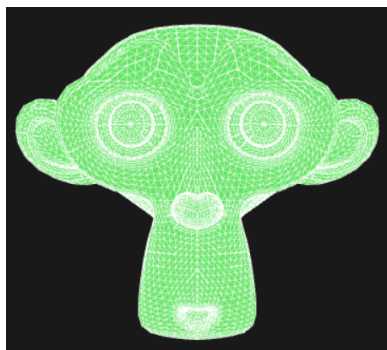
interpolate the positions, textureUVs and normals for each of the new points you add to the triangle's edges. You may wish to use some of the vector arithmetic helper functions in `common/maths_utils` here.

(b) Normal Displacement (5%)

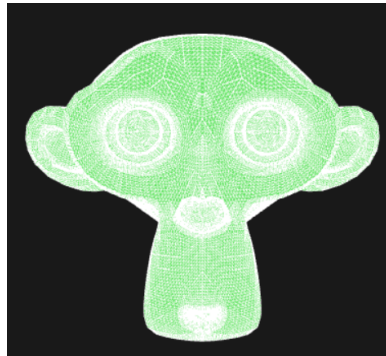
Edit the subdivision code such that every vertex position in the new buffer gets displaced a distance of `alpha` along the newly interpolated vertex normals.

(c) Multiple subdivisions (10%)

Edit the subdivision function to recursively subdivide the 3D mesh `numSubdivisions` times (up to twice). Zero subdivisions returns the original vertex buffer, 1 subdivision splits triangles into 4, and 2 subdivisions splits triangles into 16.



(a) 1 Subdivision



(b) 2 Subdivisions

Submitting your assignment:

You can submit your assignment using the `submit` command on a DICE machine:

```
submit cg cw2 ./folderWithYourAnswerFiles
```