Computer Graphics Tutorial 3

In this tutorial, we will handle some more in-depth WebGL code, and write some shaders for lighting effects and displacement mapping.

Running the Code

- Extract the given zip file from the course webpage (http://www.inf.ed.ac.uk/teaching/courses/cg/index2017.html#Tutorials).
- Open "tutorial_3.html" in a WebGL 2.0 compatible browser (recent versions of Firefox, Chrome, or Opera).
- If using Google Chrome, you must open it via the command line with `google-chrome -allow-file-access-from-files`, which allows the 3D model file to be loaded.
- If using Google Chrome, helpful debugging tools can be opened by pressing `F12`.
- If using Firefox, debugging is opened via `Ctrl + Shift + K`, or via Firebug.
- You can ignore `WebGL:INVALID_VALUE:` warnings for `enableVertexAttribArray` and `vertexAttribPointer` until you have completed all parts of the exercise.

Questions

1: Loading the 3D Model

In Tutorial 2, we loaded a simple 3D cube model, which was defined only by its vertex positions in 3D space. Now, we will load up a more complex model with vetex positions, normals, and texture UV coordinates.
The format of the `sphere.obj` file is similar to the previous `.obj` file we loaded, so you may refer back to that code, or look at the example solutions online.
The different types of lines in the `sphere.obj` file we want to load are as follows:

```
v 1.0 -1.0 -1.0
```

Lines starting with "v" define the 3D coordinates of each vertex in the model.

```
vt 0.5 0.5
```

Lines starting with "vt" define the 2D texture coordinates for each vertex in the model.

```
vn 1.0 -1.0 -1.0
```

Lines starting with "vn" define the 3D normal direction of each vertex in the model.

```
f 1/1/1 28/2/2 27/3/3
```

Lines starting with "f" define a triangular face of the model. Face lines have parts separated by spaces (one for each vertex in the triangular face). Each segment has 3 numbers separated by slashes:

```
pos/tex/norm
```

- `pos` - index of that vertex's 3D position from the list of "v" lines.
- `tex` - index of that vertex's 2D texture UV coordinates from the list of "vt" lines.
- `norm` - index of that vertex's 3D normal direction from the list of "vn" lines.

This is similar to the format of the `cube.obj` file from the previous tutorial, but instead of just a single index for for the vertex's position, 3 indices are given for its position, texture coordinates, and normal. NOTE: as before, the indexing scheme starts at 1.
Your task is to fill in the missing code in the `loadMeshData(s)` function of `load_obj.js` such that it fills in the `vertexBuffer` variable with the correct vertex positions, uvs, and normals. This buffer should be a single array of floats. Each face has 3 vertices. Each vertex has 8 floats (3 pos, 2 tex, 3 norm). The final `vertexBuffer` variable will have 122880 values in it. An example of the buffer should be filled in is shown below:

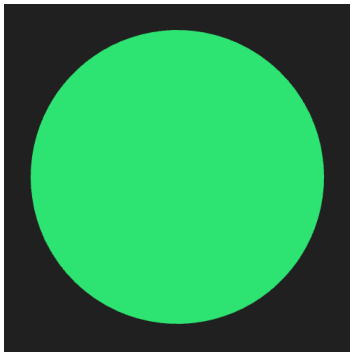
```
v 1.0 1.0 1.0
v 2.0 2.0 2.0
v 3.0 3.0 3.0
vt 0.7 0.7
```

```
vt  0.8  0.8
vt  0.9  0.9
vn  0.1  0.1  0.1
vn  0.2  0.2  0.2
vn  0.3  0.3  0.3
f  1/1/1  2/2/2  3/3/3
...
```

would have a vertex buffer filled in like:

```
[1.0,  1.0,  1.0,  0.7,  0.7,  0.1,  0.1,  0.1,
 2.0,  2.0,  2.0,  0.8,  0.8,  0.2,  0.2,  0.2,
 3.0,  3.0,  3.0,  0.9,  0.9,  0.3,  0.3,  0.3,   ...]
```

When finished, you should see a flat green sphere on the screen:



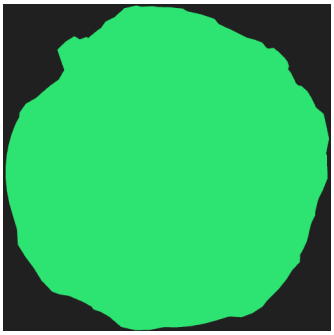
2: Displacement Mapping

Now, we will edit the vertex shader in `shaders.js` so that it performs displacement mapping. Edit the vertex shader so that it:

- Samples a colour from `displacementMapTexture` at the coordinates `vertUV`.
- Takes the "red" component of that colour (the image is greyscale, so r=g=b).
- Multiples this by the weighting factor of `displacementScale`

- Displaces the output vertex position by this amount along the normalized direction of the vertex normal from `vertNormal`

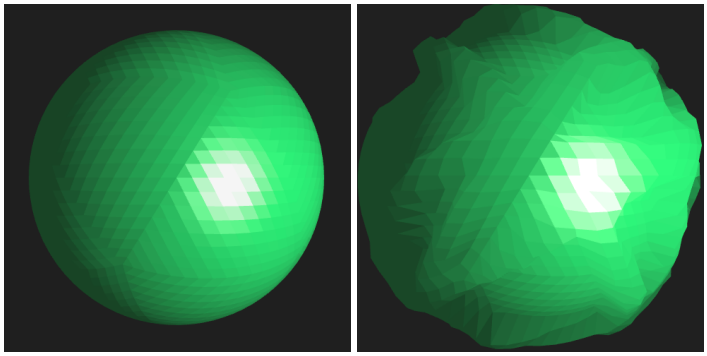
The displaced sphere should look like:



This displacement map is generated from the texture loaded from `displ.png`. Feel free to edit this, or examine the texture loading code in `common/texturing.js` and the binding code in the `renderModel()` function of `common/simple_drawing` if you are interested in how the texture is loaded and bound to the shader variables in WebGL.

3: Flat Phong Lighting

Edit the vertex shader in `shaders.js` to performs phong illumination (see the lecture slides and textbook). Use the transformed normal value `n`, the vertex's position in global space (apply the correct transformation matrix to it), and the included lighting and material properties. Assign the final colour to `outCol` for the fragment shader to use.

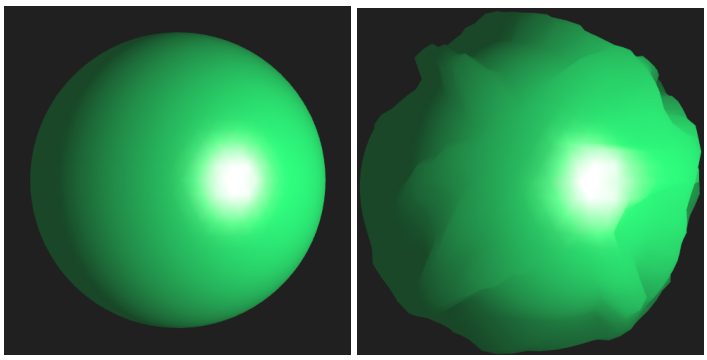


(a) Sphere (b) Displaced

4: Gouraud Shading

We can make this smoother by using per-vertex, rather than per-face lighting, and having the fragment shader interpolate the colours smoothly between the values at each vertex of the relevant triangle. This interpolated per-vertex lighting is known as Gouraud shading. Doing this is very simple in GLSL. By default, `out` variables from the vertex shader are interpolated between the triangle's vertices. This means the corresponding `in` variables already contain interpolated values by default.

However, the variable `outCol` has used the GLSL `flat` keyword when defining it. This means that a single value from a single vertex is used across the entire triangular face. To enable smooth Gouraud shading, simply delete the `flat` keyword in front of `outCol` in both the fragment and vertex shaders, and observe the results:



(c) Sphere (d) Displaced