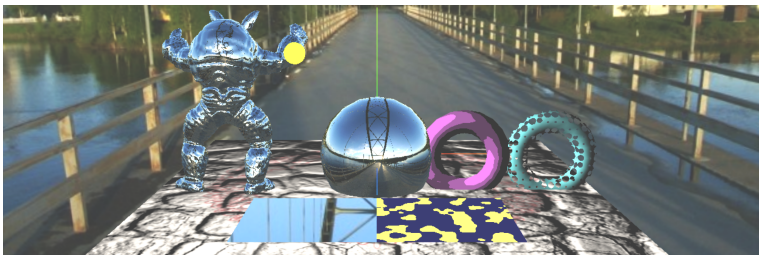
CPSC 314 Assignment 5: Shaders, Texture Mapping, and Basic Lighting

Out: Wed November 13, 2019
Due: Wed November 27, 2019



[27 marks total; worth 8%]

1. The objective of this coding question is to gain hands-on experience with vertex shaders, fragment shaders, texture mapping, and lighting. See the course web pages for the starting template.

   The code runs in the browser and can thus be run by opening `a5.html`. You will need to enable local file access, as you did for Assignment 1. Also, see `https://www.shaderific.com/glsl-functions/` for a good summary of GLSL functions.

   The template code comes with the following keybindings:

   `a,d,w,s`: moves the light source to the left, right, up, and down

   You will be making changes to the javascript, `a5.js`, and the various vertex and fragment shaders in the HTML file, `a5.html`. After making edits, a page reload on your browser will then run your code again. Error messages will be displayed on the javascript console. Debugging shaders and their compilation errors can be tricky, and so **we recommend testing your code after every change**. Something as simple as a missing semicolon in the shader code requires some effort to find. Always look at the development console to understand the problems. Scroll to the top to find the line number of the first error encountered.

   Lighting computations are best done in VCS, i.e., eye or camera coordinates. Note that the shading language allows for component-wise multiplication using a statement such as: `vec4 c = a*b;`, where `a` and `b` are also of type `vec4`.

(a) (1 point) Observe the carpet texture while rotating the scene. Note some of the artifacts of the rendered texture, particularly when seen at glancing angles. Change the `a5.js` code to use a better filter. To see a list of the available `three.js / WebGL` filters, look at `threejs.org -> Documentation -> Textures -> Texture` and look at the `.minFilter` texture constants that can be assigned.

(b) (3 points) One of the faces of a small skybox is already in place, the `posx` face. Complete the construction of the skybox for the other 5 faces using similar code. First simply focus on getting them into the right positions. Then you can experiment with different rotations to get these faces properly oriented. Note that the second page of the lab notes on environment mapping provides diagrams of the correct orientation of the image for each face, where `(u,v)` correspond to the `(x,y)` axes of the image. For the assembly, use the current `size=10`, and spin the camera around the outside to ensure that all the images join correctly (continuously) at the edges. Once you have all the pieces in place, change to `size=1000`.

(c) (4 points) We'll now be making changes to the fragment shaders, beginning with the yellow torus. First, let's add some parameters that can be set from javascript, via a `uniform` variable. As defined on the javascript side, the `toonMaterial` definition defines two uniforms: `lightPosition` and `myColor`. Add these to the `toonFragShader` shader code (in `a5.html`) as uniforms, i.e., `uniform vec3 lightPosition;` and `uniform vec3 myColor;`. `Uniform` variables are used to pass constant variables to shaders.

Now set the fragment color to `myColor`. Because `gl_FragColor` is `vec4`, you'll need to use something like `gl_FragColor = vec4(myColor,1.0);` which will append a fourth opacity value of 1.0 to the color. Verify the results.

Next, we'll implement a simple diffuse shading model, given by $i = N \cdot L$. First, compute a normalized `vec3 L` that points towards the light source. Note that the location of the current surface point, in VCS, is given by `vcsPosition`, and the eye is at `(0,0,0)` in VCS. Next, compute a normalized version of the VCS normal. Compute a dot product of these two quantities to give a scalar intensity i. Now use this intensity to multiply `myColor` to give a diffuse-shaded torus of the right color. Verify that the diffuse lighting follows the light source as you move it using the keypresses given above.

Finally, change your shader to a *toon* shader (another name for 'cartoon' shader). These use only a small number of colors to render the image. A simple way to achieve this is to discretize the intensity to one of five values, i.e., $i = 0, 0.25, 0.5, 0.75, 1$. This can be done with a combination of multiplications, divisions, and the floor function, which rounds down, i.e., `floor(3.22) = 3.0`.

(d) (4 points) Use a copy of your `toonShader` (perhaps without the final "toon" part) to create a diffuse-shading starting point for your `holeyShader` shader. This new shader will procedurally generate holes in the object being rendered. First, learn how to `discard` a fragment based on its position (both object and VCS coordinates are available via `varying` types). For example, if the position lies within a sphere of a certain radius, the use of `if (condition) discard;` will create a visible hole. Next, use the `floor` function to help create an implicit function that defines a regular 3D grid of spheres, i.e., via the use of the fractional coordinates. This should produce rendered surfaces full of holes. Hint:
`dx = x - floor(x+0.5); dy = y - floor(y+0.5); dz = z - floor(z+0.5);`
`r2 = dx*dx+dy*dy+dz*dz;`
computes the square of the radial-distance to the nearest integer `x,y,z` 3D grid location. You may want to scale x,y,z by some constant before doing this computation in order to get a grid spacing that you like.

(e) (4 points) Complete the `floorFragShader` so that it applies a given normal map to give the floor a bumpy appearance. First, view the the file `image/stone-map.png`. Each texel in the normal map stores the (N_x, N_y, N_z) components of a unit normal using the (r, g, b) values. Because $N_x, N_y, N_z \in [-1, 1]$ and $r, g, b \in [0, 1]$, the normals are stored according to $r = (N_x + 1)/2, g = (N_y + 1)/2, b = (N_z + 1)/2$. Thus first, extract the texture map normals by doing a normal-map texture lookup, and undoing the transformation just described. To verify that these are being computed correctly, visualize the normals by directly using these as the fragment color. Note that negative values will be clipped to zero. Next, we need to transform the normals because the normal-map assumes that the surface to be rendered lies in the xy-plane, i.e., that the default normals are in the $+z$ direction. However, in our scene, y is up. Thus, you will want to compute a 90-degree rotated version of the normals so that y is up. [advanced-topics note: usually this transformation is done by specifying a full coordinate frame using tangents and bitangents]. Then compute the lighting direction, L using the known VCS `lightPosition` and the current surface point `vcsPosition`. Compute a simple diffuse lighting using `N.L`. Lastly, use the diffuse lighting to scale the standard texture map color. This will give a surface that still retains the texture map colors while having visible bumps as specified by the normal map.

(f) (3 points) Complete the `envmapFragShader` that implements a basic reflective *environment map* shader. This is used to shade the Armadillo, the sphere, and the square mirror on the floor. Begin your shader by handling rays that exit the top of the skybox. I.e., the top of the skybox should be visibly reflected off the top of the sphere. For the time being, the remainder of the fragments can be rendered with a constant-color default.

First, compute the incident ray direction, in VCS. This is given by the `vcsPosition` and the eye position, which is at the origin of VCS. A reflected ray can be computed directly as follows: `R = reflect(I,N)`, where the `reflect()` function is provided for you in the GLSL shading language. Next, we need to know the reflected vec-

tor in world coordinates in order to do the texture lookup. A multiplication by `matrixWorld` achieves this. GLSL directly supports matrix multiplication, i.e., `newvec = matrix*vec`. Lastly, if the positive y component is the largest component of the reflected vector, then it will exit through the top plane of the cube. So, for this case, the u and v coordinates should then be computed; see the lab notes on this. The texture color can be retrieved using `texture2D(uPosyTexture, vec2(u,v))`. This texture color is then directly used as the fragment color. When debugging, it may be useful to replace the `posyTexture` specified in `a5.js` with the image `ABCD.jpg`, whose orientation is easier to determine.

(g) (3 points) Complete `envmapFragShader` for the remaining faces.

(h) (2 points) Procedural shaders can use computation to produce image complexity with a very compact model and without texture map images. The `pnoiseFragShader` in the template code implements *Perlin* noise. Here our goal will be to briefly experiment with it. (i) The given version uses viewing coordinates to index the noise, and thus the noise moves when the scene is rotated. Change this to use the `ocsPosition`. What happens when the scene is rotated? Add your answer as a comment in the code. (ii)Change the fragment color so that it uses a hard-threshold on the colour, using a final color given as follows:
`float j=floor(i+0.9); gl_FragColor = vec4(j,j,0.3,1.0);`. (iii) What is the result of using more levels, i.e., `levels=5` ? Add your answer as a comment in the code.

(i) (3 points) Develop an idea of your own for augmenting the scene. Possible ideas include: create a procedural, animated normal map that creates virtual ripples on an object; create "puddles" of water by discarding fragments in the Perlin noise shader, and using environment map computations for the non-discarded fragments; animated object motion; your own ideas! The instructors and TAs will not respond to any questions that ask us to define this more precisely.

Submit your code using `handin cs-314 a5`.
Include a README.txt file that contains your name, your student number, and any comments and explanations that you wish to include.