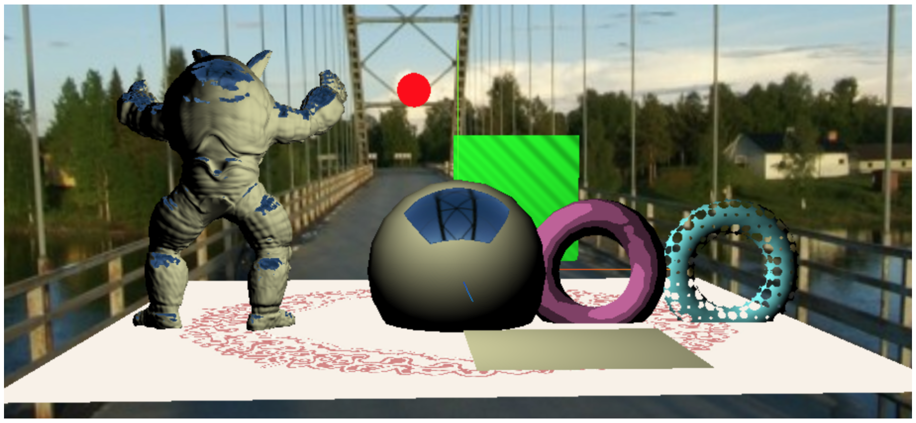
CPSC 314 Assignment 5: Shaders, Texture Mapping, and Basic Lighting

Out: Tue March 13, 2018
Due: Mon March 26, 2018



[24 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, and as described here: `https://threejs.org/docs/#Manual/Getting_Started/How_to_run_things_locally`.

   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. You will need to experiment with different rotations to get these faces into place. Note: the `posx` and `negx` images, i.e., positive and negative x-axis images, are (for reasons unknown) labelled backwards, so the `posx` image is located along the negative x-axis, and vice-versa. The labels for all the other skybox images are correct. For this 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 red 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 `toonShader` code as uniforms, i.e., `uniform vec3 lightPosition;` and `uniform vec3 myColor;`.

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`. 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, i.e., an image which uses only a small number of colors to render the image. This can be trivially done by producing a discretized version of the intensity, i.e., $i = 0, 0.25, 0.5, 0.75, 1$. This can be done with the help of 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.

(e) (4 points) Create a simple procedural bump-map shader. Use a copy of your `toonShader` again, this time to give an initial diffuse-shading starting point for your `myBumpShader` shader. Use the fixed normal that is provided, i.e., `vec 3 N = normalize(surfNormal);`. Create small procedural perturbations to the surface normal, using `sin()` and `cos()`, as a function of the object position. This should give the square the appearance of a bumpy or wavy surface.

(f) (4 points) Complete a basic environment map shader, which is applied to the Armadillo and the sphere. Your shader only needs to render the environment map for reflected 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. The remainder of the sphere (or Armadillo) should be simply rendered as a diffuse-shaded object.

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 vector 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, and the texture color can be retrieved using `texture2D(myTexture, vec2(u,v))`.

(g) (4 points) Develop an idea of your own for augmenting the scene. Ideas include: adding animated warps to the geometry in the vertex shader, creating procedural shaders, implementing a bump map using a texture, or creating an animated texture. The instructors and TAs will not respond to any questions that ask us to define this more precisely.

Submit your code using `handin cs314 a5`.
Include a README.txt file that contains your name, your student number, and any comments and explanations that you wish to include.