CS 4620 Programming Assignment 4
Shaders

out: Wednesday 14th October 2015

due: Thursday 29 October 2015 (11:59pm)

1 Introduction

In this assignment, you will implement some shaders in GLSL, a graphics-specific language that lets you write programs that run on the graphics processor (GPU). We provide a framework and different scenes, so you can try your shaders with different configurations. Your task consists of mostly shader programming, the framework takes care of sending the required data from the CPU to your shaders.

There are four main components to the assignment:

1. Implement a Cook-Torrance shader with texture-mapped diffuse component. This can produce more realistic illumination than the Blinn-Phong shader you implemented in the Ray 1 assignment.
2. Implement specular (mirror-like) reflection under environment lighting.
3. Implement a normal mapping shader (without environment mapping).
4. Implement a displacement mapping shader (also without environment mapping) and compare results with normal mapping.

2 Interface Overview

You will use the same interface you used for the Scene assignment. We have provided the implementation of Blinn-Phong shading (TestScenePhong.xml, Phong.vert, Phong.frag) as an example. We have created the shader files for you, so you just have to fill in the shader implementations. We have also prepared some scene files which you can use to test your shaders. You should have one scene file for each task in the assignment. The white balls indicate point light sources. Correct renderings of the scene files are shown below.

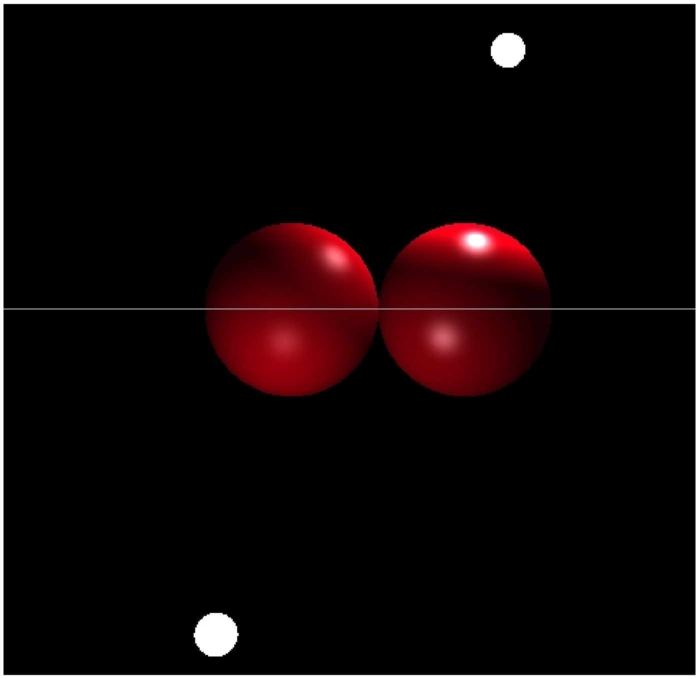

Figure 1: Correct rendering of the TestSceneCookTorrance.xml scene file, the sphere at the left has Cook-Torrance shading, the other one has Blinn-Phong shading

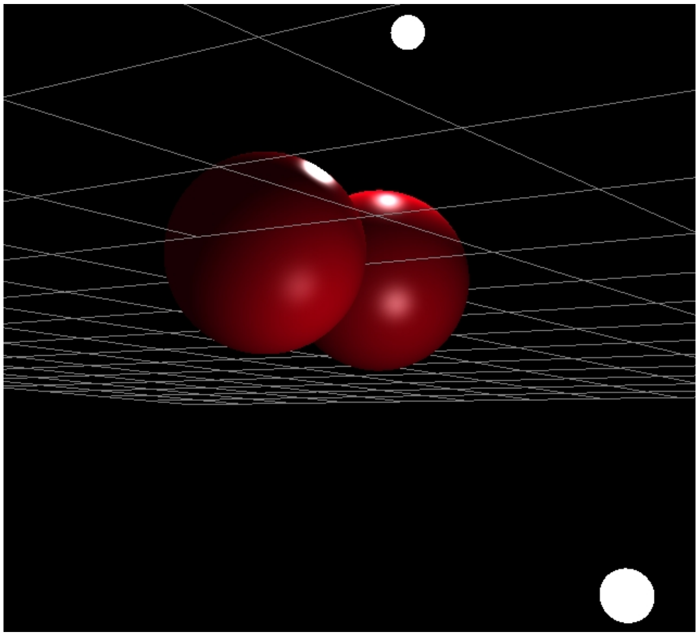

Figure 2: Correct rendering of the TestSceneCookTorrance.xml scene file, camera moved to show the lights from grazing angle

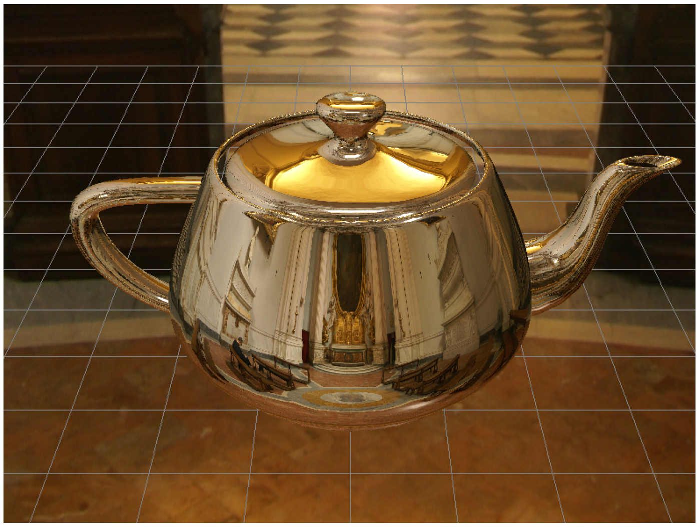

Figure 3: Correct rendering of the TestSceneReflection.xml scene file

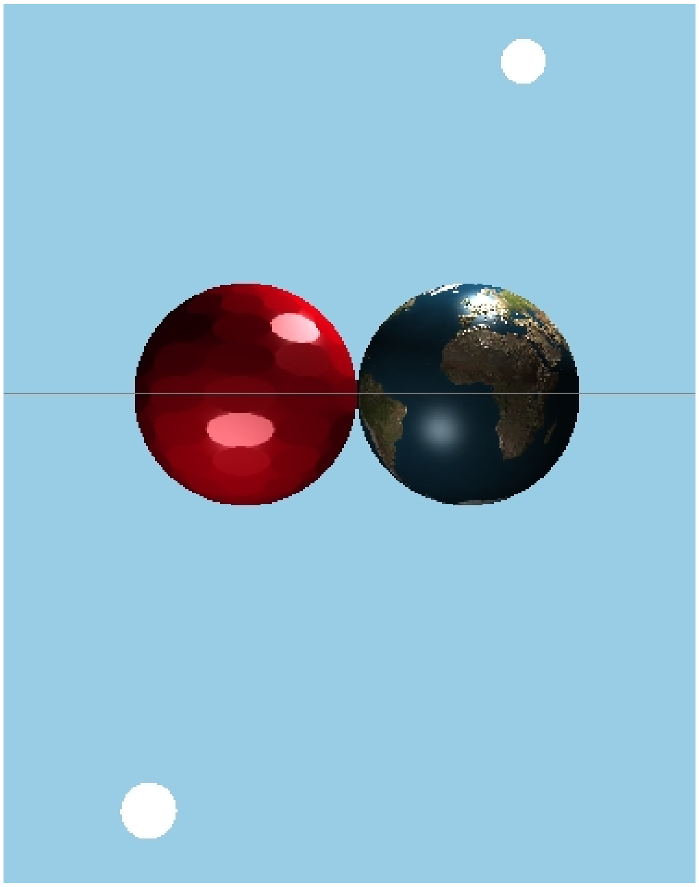

Figure 4: Correct rendering of the TestSceneNormalMapped.xml scene file

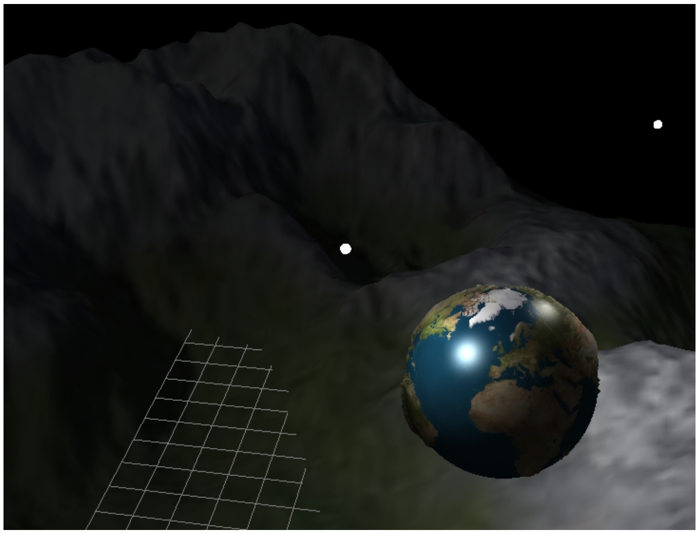

Figure 5: Correct rendering of the TestSceneDispMapped.xml scene file

3 Requirements

The assignment is broken into four parts: Cook-Torrance shader, specular reflection under environment mapping, normal mapping, and displacement mapping.

3.1 Problem 1: Cook-Torrance Shader

Blinn-Phong shading can give you nice results and it's fast and easy to implement – but it's far from the reality of light reflection. For more realistic appearance, we should use more sophisticated models like Cook-Torrance shading. Note the difference between Blinn-Phong and Cook-Torrance shading in Figure 1 and 2.

The Cook-Torrance shading model starts with the same Lambertian diffuse shading as Blinn-Phong, but it models the surface's microstructure to give a more physically accurate approximation for the specular term. The microfacet model used in Cook-Torrance shading assumes that the surface consists of randomly aligned small smooth planar facets. The model also takes the Fresnel refraction-reflection term and the shadowing effect of the microfacets into account. If you are interested in this topic, we suggest reading [1].

To compute Cook-Torrance shading, you will need the same vectors which you used in Blinn-Phong shading: the viewing direction vector, the normal vector and the light direction vector (for each light source). The shading equation is the following in case of one light source [2]:

$$color = \left(k_s \frac{F(\beta)}{\pi} \frac{D(\theta_h) G}{(N \cdot V)(N \cdot L)} + k_d \right) max(N \cdot L, 0) \frac{I}{r^2} + k_d * I_a$$

where F is the Fresnel term, D is the microfacet distribution, G is the geometric attenuation, N is the normal vector of the surface, V is the viewing direction, L is the light direction, I is the light intensity, r is the shaded point's distance from the light source, I_a is the ambient light intensity, k_s

and k_d are the diffuse and specular reflectance of the surface, and the angles β and θ_h are as defined below.

You can get the value of k_s and k_d using *getSpecularColor* and *getDiffuseColor* functions in the shader program, you can find more details about these in 4.2.

3.1.1 Fresnel Term

The Fresnel equations describe the reflective/refractive behavior of the light when it reaches a smooth surface. For this assignment we'll use an approximation of the reflectance part here and ignore refraction. This approximation uses β which is the angle between the viewing direction V and the half vector H:

$$H = \frac{L+V}{||L+V||}$$

$$F(\beta) = F_0 + (1-F_0)(1-\cos\beta)^5 = F_0 + (1-F_0)(1-(V\cdot H))^5$$

where F_0 is the specular reflectance when light arrives perpendicularly to the surface; we'll use 0.04.

3.1.2 Microfacet Distribution

The determine the distribution of the microfacets, we use the Beckmann distribution function. θ_h is the angle between the normal N and the half vector H:

$$D(\theta_h) = \frac{1}{m^2\cos^4\theta_h} e^{-(\frac{\tan\theta_h}{m})^2} = \frac{1}{m^2\cos^4\theta_h} e^{-\frac{1-\cos^2\theta_h}{m^2\cos^2\theta_h}} = \frac{1}{m^2(N\cdot H)^4} e^{\frac{(N\cdot H)^2-1}{m^2(N\cdot H)^2}}$$

where $m \in [0,1]$ is the roughness term, which controls the sharpness of highlights. Roughness has the same function as Phong exponent, but the effect goes in the opposite direction; 0 is a perfect mirror surface, and 0.1-0.2 is a slightly glossy surface.

3.1.3 Geometric Attenuation

This term captures the self-shadowing of the microfacets, for detailed explanation see [1].

$$G = min(1, \frac{2(N\cdot H)(N\cdot V)}{V\cdot H}, \frac{2(N\cdot H)(N\cdot L)}{V\cdot H})$$

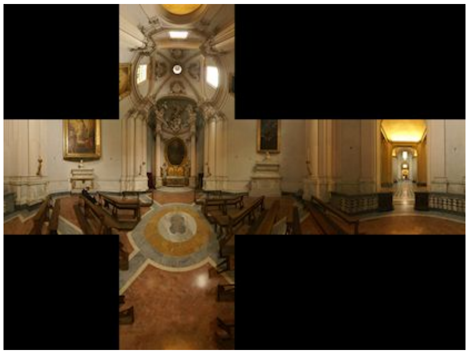

Figure 6: An example of unfolded cube map (work of Emil Persson, aka Humus)

3.2 Problem 2: Environment Mapping

You used point lights in the Raytracer and Scene assignments. Real world scenes usually have much more complex lighting conditions than that and environment mapping is a clever technique to capture complex lighting in a texture. The idea of an environment map is that the illumination from the environment depends on which direction you look, and an environment map is simply a texture you look up by direction to answer the question “how much light do we see if we look in that direction?”. Your task here is to implement specular reflection under a given environment map.

Environment lighting is the lighting conditions surrounding the rendered object. In OpenGL, there is a special kind of texture called cube map, which stores images of the distant environment. A cube map consists of six faces, and each of them has a 2D texture (see Figure 6). You can sample a point on the cube with a 3D vector in order to get environment light intensity in a certain direction. See section 11.6 of the book. In this assignment, we’ve done the part of building the cube map and you can use *getEnvironmentColor(vec3 l)* to sample it in the shader.

When using environment mapping, the image looks more realistic if you can see the environment being reflected in the background of the scene – that is, rays that don’t hit the object should just be looked up in the environment. We use a sky box to add the background. The sky box is just a huge box surrounding the scene that is shaded by the shader *Environment.frag* and *Environment.vert*. When you look at the background, you are actually looking at the inner faces of the box. In *Environment.frag* shader, you simply look up the color of the environment in the direction of the viewing ray. Implementing the sky box is a good starting point for environment mapping: once it is done, you will be able to look around the environment simply by rotating the camera.

The next step is to implement specular reflection under environment lighting. Given a viewing direction and a normal direction, we can use the equation in the written part (i.e., Mirror Reflection) to compute the reflection direction, and then use it to sample the cube map. The shader file for specular reflection is *ReflectionMap.frag* and *ReflectionMap.vert*.

The scene will look like Figure 3 after implementing the reflection.

3.3 Problem 3: Normal Mapping

Creating detailed models with thousands of polygons is a time consuming job and rendering them in real-time can be also problematic. However, with normal mapping, we can make simple, low resolution models look like highly detailed ones. In addition to the polygon model, we provide a high resolution normal map which we can use in the fragment shader instead of the interpolated normals, adding detail to the low resolution model. Normal maps can be generated procedurally in the shaders or read from an image and used as a texture.

In this task, you have to implement a normal map texture generator which generates normals for a sphere which has smaller flat discs on its surface. This texture will be used in the fragment shader to retrieve normals instead of using the interpolated normals. You will see two spheres in the test scene *TestSceneNormalMapped.xml*, one uses your generated normal map, the other uses a normal map which is loaded from an image.

The normal map texture generator should have two parameters:

1. $resolution$: The number of flat discs for each row and column on the generated normal map.
2. $bumpRadius$: The radius of the flat discs. If $bumpRadius$ is 0.5 the discs are tangent to each other, if it is larger, they intersect. If $bumpRadius$ is ≥ 1.0 then the sphere should look like it has not smooth but rectangular surface.

You have to implement the *getColor* function of the *TexGenSphereNormalMap* class. This function will be called with u and v sweeping from 0 to 1 and the returned colors will be saved to a texture, which will be passed later to the shader.

As a first step, we suggest that you generate the sphere normals without discs. You should see the same result as you get in TestScenePhong.xml, if you modify the shader to render the normal vectors as color.

Now, you can add the discs, whose centers are on a $resolution$ by $resolution$ grid in texture space. Thus, you should have $resolution * resolution$ number of circles with the same radius on a regular grid in texture space. In world space, the normal vector inside a disc should be constant along the disc's surface and equal to the normal vector of the sphere at the disc center. Outside all discs the normal is simply the normal of the sphere. However, the value of the normal that must be sent to the texture must be in tangent space, so a conversion from world space to the object's tangent space is necessary.

You can get the normal map value for a texture coordinate by calling *getNormalColor(texture coordinate)*. See 4.2 for detailed explanation about this function. Since textures store RGB color and not arbitrary vectors, we have to transform our normal vector components from [-1, 1] to [0, 1] when we set the texture color. If you use *Colord* to store the converted [0, 1] values, you can easily convert them to [0, 255] using the *Color* class. Don't forget that your shader will get the normal map as a texture, which means that the values are in [0, 1] (GLSL uses the [0, 1] continuous color range instead of [0, 255]). Thus, you have to convert these "color channels" back into the three coordinates of the normal vector in the normal mapping fragment shader to obtain normal vectors again. The recovered normal vector is in tangent space, which means you have to transform it to world space using a tangent space matrix and normal matrix.

3.4 Problem 4: Displacement Mapping

Displacement mapping takes a step further towards accurate rendering of a bumpy surface, and moves the vertices of the mesh along the normal vectors. We call the map which stores the magnitude values *height map*. We use the normal map to hold the displacement magnitudes, so you

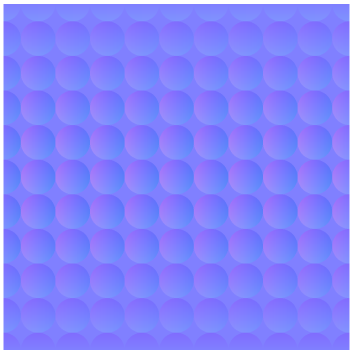

Figure 7: The normal map texture with $bumpRadius = 0.5, resolution = 10$

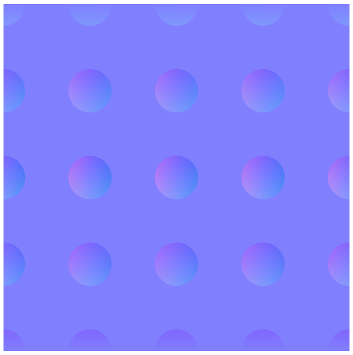

Figure 8: The normal map texture with $bumpRadius = 0.25, resolution = 4$

can query height map values using *getNormalColor(texture coordinate)*. The displacement magnitude should be the average of the channels of the retrieved color from the height map texture. You can have a look at the height map we used in the scene file to generate terrain from a plane with displacement mapping (Figure 9).

The vertex shader from Phong.vert is a good starting-point. The task here is to modify that shader to move each vertex along the normal direction by a distance proportional to the value in the height map. The *dispMagnitude* uniform variable determines the scale of the displacements: a value of 1.0 in the height map means the vertex should be moved by a distance *dispMagnitude*. The displacement should take place in model space, before transforming to world coordinates, so that the displacement magnitude is measured in object-space units.

3.5 Additional implementation notes

We use a scaling factor for all pixel values of the final image. This can be used to fine tune the final image brightness. It is called $exposure$ and it can be reached from all shaders. You should multiply the computed pixel color with $exposure$, before setting the final pixel color.

$$gl\_FragColor = computedColor \cdot exposure;$$

Figure 9: The height map texture used for terrain generation.

4 Framework

The framework has a standard set of uniform variables which are passed to each shader:

`mWorld` The model matrix, which transforms the points into world coordinates.

`mWorldIT` The inverse transpose of the previous matrix, it is used to transform normals to world coordinates.

`mView` The view matrix, it is used to transform points from world coordinates to camera coordinates.

`mProj` The projection matrix.

`mViewProjection` The product of the view and projection matrices.

`worldCam` The position of the camera in world coordinates.

`exposure` A scaling factor for all pixel values of the final image.

`shininess` The shininess of the surface being illuminated (Phong exponent).

`roughness` The roughness of the surface being illuminated (used in the Cook-Torrance shader).

`dispMagnitude` Scale for magnitude of the displacement (used in the displacement mapping shader).

`getDiffuseColor(vec2 uv)` The diffuse color of the surface being illuminated. Can be indexed with texture coordinates. Each component is between 0.0 and 1.0.

`getSpecularColor(vec2 uv)` The specular color of the surface being illuminated. Can be indexed with texture coordinates. Each component is between 0.0 and 1.0.

`getEnvironmentColor(vec3 l)` The environment lighting from a certain direction. Use a 3D vector to sample the cube map.

`getNormalColor(vec2 uv)` The normal map of the surface being illuminated. Can be indexed with texture coordinates. Each component is between 0.0 and 1.0. This is also used to store the height map in the displacement mapping task.

`numLights` The actual number of point lights.

`lightPosition` The position array of the lights.

`lightIntensity` The intensity array of the lights.

`ambientLightIntensity` The intensity of the ambient light source.

We also standardize five vertex attributes:

`vPosition` The position of the vertex.

`vNormal` The normal of the vertex.

`vUV` The texture coordinates of the vertex.

`vTangent` The tangent-space X-axis found in world space.

`vBitangent` The tangent-space Y-axis found in world space.

In addition to these variables, you might want to use *varying* variables. These variables are the output of the vertex shader and before arriving to the fragment shader input, they are interpolated across triangles. They are how your vertex and fragment shaders communicate, and deciding what variables you need is up to you.

4.1 Bindings

We defined bindings to each of the preceding variables. As a result, we have a handle (similar to a handle in Windows API) for each of them, which we can use to assign values to the variables and pass them to the shader program. You can look at most of these bindings in *RenderMaterial.java*. Before sending the vertices of an object through the rendering pipeline, we set all of the attribute and variable values which are used in the current material's shader program. You can look at the code in *Renderer.java:draw()*

4.2 Samplers

Samplers are used to access values of a texture in GLSL. They represent a texture which is bound to the OpenGL context. The OpenGL context has a certain number of texture units, which you can use to bind a texture to. For each texture type there is a corresponding texture sampler. E.g. for a 1D texture (GL_TEXTURE_1D) the sample type is *sampler1D*. We use *sampler2D* and *samplerCube* in this assignment. The CPU code of the framework takes care of the texture bindings and setting up samplers. For further reading, we suggest `https://www.opengl.org/wiki/Sampler_(GLSL)`.

We might have textured surfaces or sometimes they just have a constant color. We want the same shader code to work for all combinations of textured/untextured inputs. Thus we define a function for each of these possibly textured values: *getDiffuseColor(vec2 uv)*, *getSpecularColor(vec2 uv)*, *getNormalColor(vec2 uv)* and *getEnvironmentColor(vec3 l)*. Before compiling the shader code, we append the appropriate code for each function, depending on whether that particular input is textured. If you would like to see how this works, you can find the code in *Rendermaterial.java:addSpecProviders*.

4.3 Shader compilation

All the shaders you will write are loaded and compiled by classes we have provided. In case of a shader compile error, you will see a message in the console with the error.

You can add an eclipse plugin for shader syntax highlighting. It works with Eclipse Luna Release (4.4.0), may work with other Eclipse versions. Download from here, installation instructions here.

4.4 Additional Notes

Note that color values in the range $[0, 1]$ in GLSL correspond to the full range of brightnesses that can be displayed. If you need more information about OpenGL or GLSL, there is an excellent webpage: `https://www.opengl.org/wiki/GLSL`. You can also look at `http://www.lighthouse3d.com/opengl/glsl/`, where you can find a tutorial and plenty of examples. Beware that there are lots of different GLSL versions out there with different syntax. We use GLSL 1.2 in this assignment.

Also, we have marked all the functions or parts of the functions you need to complete in with TODO A4 in the source code. To see all these TODOs in Eclipse, select Search menu, then File Search and type TODO.

5 What to Submit

Submit a zip file containing your solution organized the same way as the code on CMS. Include a readme in your zip that contains:

- You and your partner's names and NetIDs.
- Any problems with your solution.
- Anything else you want us to know.

Also, there will be a written part (A4 Shaders Written). Submit the solutions in pdf format on CMS.

References

[1] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, January 1982.

[2] Philip Dutre, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination*. A. K. Peters, Ltd., Natick, MA, USA, 2002.