Project 3

1.1 Overview

In this assignment, you will learn how to program GPU shaders. A Windows-only executable is provided to allow for reloading shaders and displaying the result, so you only need to write shader code in an ASCII editor of your choice (notepad or similar).

Shaders are small programs that are executed by the GPU. They replace functional blocks of the traditional OpenGL pipeline. In particular, we are concerned with ***vertex shaders*** and ***fragment shaders*** (sometimes also called ***pixel shaders***). They fit into the rendering pipeline in the following places:

1) ***Scene traversal*** – your program issues a draw call that traverses the scene and sends each vertex of the model to the GPU for transformation.
2) ***Vertex shader*** – takes one vertex (with attributes) as input and outputs a transformed vertex with some different attributes. Attributes may contain position (mandatory), color, normal, etc., all optional. Vertex shader outputs are available to the fragment shader (see below).
3) ***Primitive assembly*** – depending on your draw call, group vertices into ***primitives*** such as triangles, quads, lines, etc. For this assignment, we deal exclusively with triangles.
4) ***Rasterization*** – this step computes the overlap between pixels and the primitives. For each pixel, a ***fragment*** is generated. Vertex attributes are interpolated to form fragment attributes.
5) ***Fragment shader*** – takes one input fragment (with attributes) and outputs one or more colors. In our examples, the fragment shaders will output at most one color. Fragments can also be discarded, in which case they are removed from what follows.
6) ***Fragment tests*** – fragment tests can conditionally discard fragments. A common example is the depth test that discards occluded fragments. Non-discarded fragments end up as pixel colors on the screen.

Shaders are written in GLSL, a variant of C. In addition to normal data types, GLSL also specifies the following:

`vec2, vec3, vec4` – floating point vectors with 2,3,4 components

`mat2, mat3, mat4` – floating point matrices with 2x2,3x3,4x4 components

vector components can be accessed using .x, .y, .z, .w. They can be selected quite flexibly, for instance:

```
vec3 a;
vec3 b = a.xxz;
```

This is equivalent to `b.x = a.x; b.y = a.x; b.z = a.z;`

Furthermore, larger vectors can be generated from smaller ones:

```
vec4 a = vec4(b.xyz,1.0);
```

Floating point numbers have to be specified as such, e.g., 1.0f or 1.0, but not 1.

Shaders also have certain variable modifiers:

- **`uniform`** – the variable is the same for all vertices or all fragments and does not change per vertex/fragment.
- **`in`** – input variable that changes from vertex to vertex or fragment to fragment
- **`out`** – variable that is handed on to the next stage

For instance,

```
uniform vec3 light_direction;
in vec3 normal;
float diffuse_intensity(void) {
      return dot(light_direction,normal);
}
```

would use a static light direction (does not change for current frame) and a normal that changes from vertex to vertex or pixel to pixel.

There are more "built-in" functions like **`dot()`**, these include (but are not limited to):

- **`dot()`**, **`normalize()`**, **`cross()`**
- **`pow()`**, **`sin()`**, **`cos()`**
- etc.

Vertex Shader

The bare minimum vertex shader looks like this and transforms an object-space position to screen-space

```
#version 150    // GLSL version to use

uniform mat4 ModelViewProjection;// Projection*Model*View Matrix
in vec3 position;                // per-vertex position

void main() {
      gl_Position = ModelViewProjection*vec4(position.xyz,1.0);
}
```

Fragment Shader

The bare minimum fragment shader assigns a color:

```
#version 150    // GLSL version to use

out vec4 color;                  // output color

void main() {
      color = vec4(1.0,0.0,0.0,1.0);   // red w/ alpha=1 (opaque)
}
```

Computations performed in the vertex shader can be forwarded to the fragment shader by using in/out modifiers on variables with identical names:

```
// Vertex Shader
...
in vec3 color;
out vec3 frag_color;

void main() {
...
      frag_color = color;   // this just pass through the color
}

// Fragment Shader
...
in vec3 frag_color;         // interpolated vertex color from
vertex shader
out vec3 result;            // resulting color
void main() {
      result = frag_color;  // output the interpolated vertex
color
}
```

The provided executable will give you some diagnostic messages when loading shaders. In particular, you will be alerted of syntax errors. For your convenience, hitting the SPACE bar will reload the most recently loaded shader. Shaders, Models, and Normal maps can be loaded using the menu on the middle mouse button. Shaders are collected in the shaders folder.

1.2 Spaces

In this assignment, we adhere to the following conventions:

- Input vertex positions and normal are in object space
- Input light position is in world space
- Lighting is performed in camera space

Therefore, the first step of each problem is to compute camera space versions of all input vectors required for lighting. **Unit vectors (such as normals) should be normalized in both the vertex shader and the fragment shader. Try to understand why this should make a difference. Consider that rasterization performs a linear interpolation and be prepared to explain to the TA what a linear interpolation between two 2D vectors looks like.**

Hint: In GLSL, there is a matrix-vector product and a vector-matrix product:

```
vec4 v;
mat4 M;
vec4 result = v*M;  // transpose(M)*v
```

1.3 Variables

The Framework provides access to all needed variables (e.g., Model, View, Projection and their inverses).

1.4 Required Functionality

1.4.1 Familarize yourself with the provided executable

a) Make sure the executable for this assignment runs on your machine. Contact the TA immediately if you encounter problems. Check the menu on the middle mouse button to load models, shaders, and normal maps. Only some combinations might make sense for now.

b) Familiarize yourself with the GLSL shading language. Searching for "GLSL", the internet offers plenty of tutorials. Make sure you understand:
   - GLSL's data types and built-in functions
   - The interactions between vertex shader and fragment shader

   Be prepared to explain these to the TA.

c) Look at the provided simple shader that comes with the project. The vertex shader is in a text file called **"vsSimple.txt"** and the fragment shader is in a text file called **"fsSimple.txt".** Be prepared to explain to the TA what it does and look at the variables that are available in the shader.

1.4.2 Halflife Shader

a) Write your first shader. **In the vertex shader, transform position, normal, and light direction to camera space.** Write the transformed vectors to the respective output registers; do not forget to normalize attributes that require normalization. In the fragment shader, compute a simple "Half-Life" light model. The diffuse intensity for this model is float I = 0.5*dot(N,L)+0.5; Be prepared to explain to the TA what this does. Multiply the light intensity with the interpolated vertex color and write the resulting color to the output register**. Edit files vsHalflife.txt and fsHalflife.txt for this example.**

1.4.3 Phong Lighting

a) Now, implement a full Phong light model in the **vertex shader.** Output a per-vertex color including light contributions to the fragment shader. The fragment shader should only output the interpolated per-vertex color. **Edit files vsPhong_a.txt and fsPhong_a.txt for this example.**

b) Move your implementation of the Phong light model to the **fragment shader.** Use the interpolated per-vertex normal and evaluate the light model per-pixel. What are the differences? Be prepared to explain these differences and why they happen to the TA. **Use vsPhong_b.txt and fsPhong_b.txt for this example.**
Hint: GLSL has a reflect() instruction.

c) What happens to the highlight if the specular exponent increases? Be prepared to explain this to the TA.

1.4.4 Blinn-Phong Lighting

a) **Use vsBlinnPhong.txt and fsBlinnPhong.txt to implement Blinn-Phong lighting in the fragment shader**. This model differs from Phong in that it uses a halfvector instead of the reflection vector to compute the specular component.

b) Compare the result of Blinn-Phong with Phong. Try to understand the differences.

1.4.5 Toon Shading

a) Starting with the per-pixel Phong (P3b), implement Toon Shading. Toon Shading does not use continuous light intensities I = dot(N,L) but "quantizes" them, e.g. if (I>0.75) I = 1 else if (I>0.5) I = 0.75 else if (I>0.25) I = 0.5 else if I>0 I=0.25. Use three quantization levels for the diffuse intensity and two quantization levels for the specular intensity. It may make sense to override the object color to a bright, constant color of your choice. **Hint: GLSL has instructions like floor(), fract() and modf(). Use vsToon.txt and fsToon.txt for this example.**

b) Now, highlight silhouettes by looking at the dot product between view vector (normalized vector position-camera) and surface normal. If this dot product is smaller than a threshold (say, 0.25), output black color, otherwise the result of P5a. What is the problem of this simple approach to highlight silhouettes ? Discuss ideas how to improve this simple algorithm with the TA.

1.4.6 Parametrics and Normal Mapping

a) Select the parametric model "uv-Torus" from the menu of the executable. This will provide you with data in the in_parameter variable in the vertex shader, but with no positions and normals. Use the uv parametrization (u,v=0..2π) to compute positions on a Torus (Equations here: https://en.wikipedia.org/wiki/Torus):

$$p(u,v) = \begin{pmatrix} (R + r\cos v)\cos u \\ (R + r\cos v)\sin u \\ r\sin v \end{pmatrix}$$

Modify vsUV.txt and fsUV.txt for all of problem 6. Render the Torus, use R=1.0 and r=0.25.

b) Compute tangent, bitangent and normal for the Torus. Using the previous equations, the equations are:

$$Tangent(u,v) = \frac{\partial}{\partial u} p(u,v)$$

$$Bitangent(u,v) = \frac{\partial}{\partial v} p(u,v)$$

$$Normal(u,v) = Tangent(u,v) \times Bitangent(u,v)$$

Normalize all three quantities, transform them to camera-space, and output them for use in the fragment shader. In the fragment shader, use per-pixel Phong lighting (copy from 1.4.3).

c) Add "fake geometry" using a normal map. To do this, use (4*u/2 π,v/2 π) as texture coordinates. Use the predefined texture texNormalMap in fsUV.txt to fetch normal data. The texture contains one normal for each pixel, nx = red, ny = green, nz = blue. Color data is stored in the range [0,1], so you need to expand after fetching it and before use: n = normalize(2.0*n-vec3(1.0,1.0,1.0)). The normal from the texture is in ***tangent space*** and has to be transformed before using it. Tangent space is given by Tangent(u,v) [x-axis], Bitangent(u,v) [y-axis] and Normal(u,v) [z-axis]. These three axes define a 3x3 rotation matrix. Transform the normal from the texture to camera space (multiplication with the inverse tangent space matrix) and then replace the normal from 6b) with the transformed normal from the texture. Be prepared to explain to the TA how this works.

1.4.7 Anisotropic Lighting

a) Download and read this short paper: http://www.cs.ubc.ca/~heidrich/Papers/IMDSP.98.pdf. Focus on the math in Paragraph 2 and not the OpenGL implementation. Be prepared to explain the key idea to the TA.

b) **Using vsAniso.txt and fsAniso.txt,** implement this light model in the fragment shader. Evaluate the equations from Paragraph 2 for each fragment. Compute micro-scratches from brushing using the following algorithm (T-vector in the paper). **Do this in the fragment shader:**
   - **Given an object position p, normalize p (projection onto the unit sphere)**
   - **Compute polar coordinates from p=(x,yz) (coordinates on the unit sphere)**

$$\begin{pmatrix} r \\ u \\ v \end{pmatrix} = \begin{pmatrix} 1 \\ \arccos z \\ \arctan \frac{y}{x} \end{pmatrix}$$

     If necessary, mind special cases for y/x.

   - **Now express p=(x,y,z) again in terms of these polar coordinates**

$$p(u,v) = \begin{pmatrix} \sin u \cos v \\ \sin u \sin v \\ \cos u \end{pmatrix}$$

   - **Finally, compute** $T = \frac{\partial}{\partial u} p(u,v)$

Explanation: This algorithm projects the object space point p onto a unit sphere. It then computes a tangent direction on the sphere along the u parameter direction. This will give radial scratches that are then mapped to the object again.

HINT: GLSL has a atan(y,x) function to compute the arctan(y/x) reliably and with special cases.

Evaluate this equation per fragment. This requires interpolating the object-space position from the vertex shader and forwarding it to the fragment shader. Remember that the T vector computed in this fashion is still in object space and has to be transformed to camera space in the same fashion as a surface normal would be transformed.
Be prepared to explain to the TA for which model this shader looks best. Try to understand what makes it look "right" for some models and "wrong" for others. How would you fix this?

1.4.8 Cook-Torrance Light Model

a) **Using vsCookTorrance.txt and fsCookTorrance.txt**, implement the Cook-Torrance light model in the fragment shader. Reference:
https://en.wikipedia.org/wiki/Specular_highlight#Cook.E2.80.93Torrance_model

Equations:
N: camera-space normal
L: camera-space light vector
V: camera-space view vector
H: normalized halfway vec tor between L and V
m: rms slope of microfacets
ρ_0: reflectivity of the object at incident angle

Beckmann distribution

$$\alpha := \arccos(N.H)$$
$$\gamma := \frac{1 - \cos^2(\alpha)}{\cos^2(\alpha) m^2}$$
$$D(N, L, V, m) := \frac{1}{\pi m^2 \cos^4(\alpha)} e^{-\gamma}$$

Fresnel Term (Schlick's approximation)

$$\varphi := \arccos(\mathrm{N.V})$$
$$F(\rho_0, \varphi) := \rho_0 + (1 - \rho_0)(1 - \cos\varphi)^5$$

Geometry term

$$G(N, V, L) = \min\left(1, \quad 2\frac{(H.N)(V.N)}{V.H}, \quad 2\frac{(H.N)(L.N)}{V.H}\right)$$

Full specular component

$$I_{spec} := \frac{DFG}{\pi (V.N)(N.L)}$$

First implement the D-term. Use only Ispec = D / (4(V.N)(N.L)). Load the Torus. Try at least the following values for m: 0.05,0.4,0.5,0.6. What changes ?
Now use only Ispec = F/(π(V.N)(N.L)). Use the following values for ρ_0: (1.0,1.0,1.0); (0.8,0.8,0.8); (0.5,0.5,0.5); (0.1,0.1,0.1). What changes ? Now, use a vector for ρ_0. Select different values for the RGB channels.
Now use all three terms for Ispec and optimize the code to use as many dot products as possible and minimize the use of arccos / cos. Use m = 0.25 and ρ_0=0.8 for presentation to the TA.