Vertex and Fragment shaders – Fun with the GPU!

This is a long project description, but *don't panic* – it's mostly instructional information to make the project easier, and explanations of multiple options you only have to implement a subset of.

For this *individual* project, you will be programming your GPU using **vertex** and **fragment** hardware shaders to accomplish various visual effects.

- Fragment shaders are used to render individual "fragments" on the screen (basically, they render individual pixels). They are typically used to modify the coloring and lighting of the polygons being rendered.
- Vertex shaders are used to modify the vertex positions (x, y, z) of the polygons being rendered.

  Vertex shaders cannot introduce additional vertices, or delete existing ones – they are limited by the meshes they are given. (Unlike geometry shaders, which are outside the scope of this project).

Vertex and fragment shaders are typically used together in pairs – that is, you use one vertex shader and one fragment shader to render something in a particular style.

For this project, you will be implementing *two* pairs of Vertex/Fragment shaders. That is, you must implement two fragment shaders, and two vertex shaders, from the choices below. Note that some pairs of vertex/fragment shaders won't work well together, because one will obscure the effects of the other (such as the "x-ray" fragment shader and the "fear-of-light" vertex shader). Choose pairs that do not interfere with each other.

Please allow switching between the pairs by pressing the '1' and '2' number keys (where '1' loads the first pair of vertex/fragment shaders, and '1' loads the second pair).

Fragment options

1. Change the lighting from diffuse to cel-shading ("toon" lighting). This is usually done by thresholding the intensity of the diffuse shading at various values (so that the possible light intensity values are reduced to a discrete set, e.g. [0.0, 0.3, 0.6, 1.0]).

   Note that "intensity" here refers to the scalar value between 0 and 1 that scales the albedo of the surface (making it lighter or darker depending on the light).

   (Use the typical formula for diffuse shading, with intensity = $\mathbf{max}(0.0, \vec{N} \cdot \vec{L})$, where $\vec{N}$ is the surface normal, and $\vec{L}$ is the vector from the fragment to the light.)

2. Add specular highlights to the sphere. Add in an "imaginary" light source in addition to the "real" one so that you get two specular highlights. It can be anywhere you choose, as long as it creates a visible specular highlight on the sphere.

   (Use the Phong illumination specular formula $\mathbf{max}(0.0, \vec{R} \cdot \vec{V})^n$, where $\vec{R}$ is the reflected light, $\vec{V}$ is the vector form the fragment to the viewer, and n is the shininess exponent).

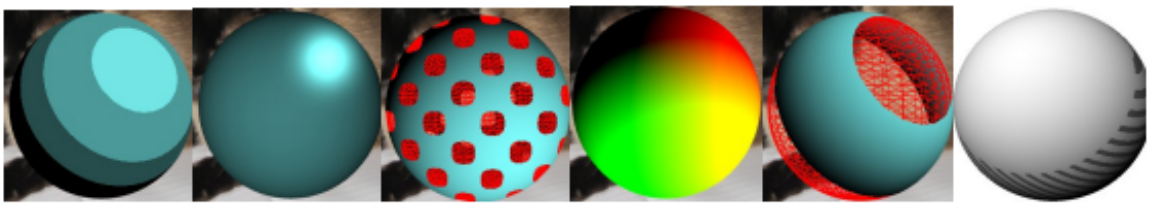
3. Turn the sphere into a "wiffle ball" by adding transparent holes all over it. (Hint: you can use the *discard* glsl function to "throw away" a particular pixel in a fragment shader, so that it won't be rendered at all). The holes should be circular, but you have some discretion with their radius and spacing (see the example below).

4. Color each pixel on the sphere as a function of its x, y position in space (any way that clearly uses both components). Make sure the light still affects the color intensity.

5. Simulate an "x-ray light" by making the sphere transparent where the light is brightest. You might also want to make the back of the sphere (i.e., where the light intensity = 0) transparent so you can see all the way through it to the background.

6. Make the ball "pulse" by changing its color as a smooth function of time.

7. Add a "hatching" pattern to the darker half of the sphere, to make the shading look kind of hand-drawn.



(Fragment shader possible results)

Vertex options

1. Modify the vertices of the sphere so that they "warble" by moving the 'y' component of each vertex with a **sin** function of the 'x' component and time (or mouse position).

2. Like 1, but make the vertices ripple "outward" radially like a pebble dropped into a pond.

   (You may have to change the normals as well to see this clearly).

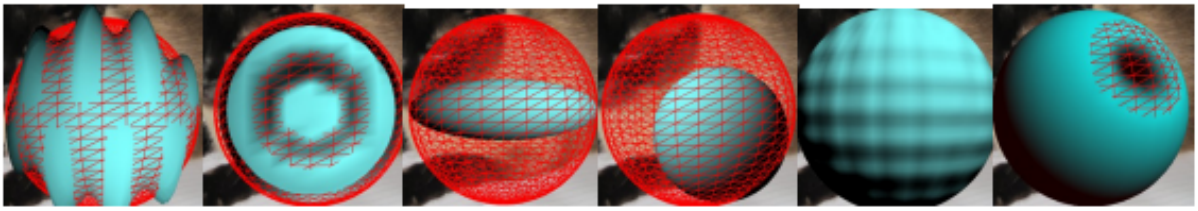
3. "Squash" your sphere as a function of time (or mouse position).

4. Make your sphere move in a circle on the x, y plane (translating, not rotating) as a function of time or mouse position.

5. Modify the normal vector to make the sphere look "bumpy" (you may want to increase the sphereDetail in the processing code to get this to look good).

6. Make the sphere "afraid" of the light by moving points away from the light based on how directly the light is shining on the sphere (this will make a dent where the light is bright).

   (Again, you may have to modify the normal vectors as well to see this clearly).



(Vertex shader possible results)

Base code

The base code for this project includes a sphere in the foreground, with a textured background behind it. The sphere has a red "wireframe" drawn over it for reference (so that you can see where the original sphere vertices are before your vertex shader changes it).

The base code includes BasicFrag.glsl and a BasicVert.glsl, which have some boiler-plate template code to get you started (basically, the *public static void main(..)* of shaders). Most of your work will be modifying (or copying) these .glsl files, which you should edit in your favorite plain-text editor.

The base code uses the 'TextFrag' and 'TextVert' shaders to render the textured plane in the back, as an example of how to use shaders in Processing. You'll have to modify the Processing code slightly to load and apply your own shaders to the sphere.

Passing information from Processing (such as the time, or the mouse position) to your shader program requires the use of *uniform variables*. Instructions for using them can be found at the top of BasicVert.glsl in the base code.

Extra credit

You may earn extra credit by implementing more than one pair of vertex/fragment shaders, allowing the user to switch between them by pressing different number keys. For this, feel free to come up with options not listed in this project description (e.g., texture mapping, edge-detection, bump-mapping), and to combine multiple fragment or vertex shader effects into one (e.g., combining vertex options [1] and [4] to make a warbling orbiting sphere). If you do this, *make sure to show/mention it clearly in your video and report.*

Reference material/help

- https://www.khronos.org/files/opengl44-quick-reference-card.pdf

  OpenGL quick reference card. GLSL stuff starts on page 8.

- https://processing.org/tutorials/pshader/

  Processing shader reference.

Deliverables

Code

- Working code showing off your shaders as described above. Make sure you can switch between shaders with the number keys.

Report

- Clearly state all fragment/vertex shader pairs you implemented, and the details of anything special you did for extra credit.
- Write a brief description of why you might want to use fragment/vertex shaders, rather than modifying the texture data and mesh vertices in a normal programming language (like in Processing, C#, C++, etc).
- Also include your name and picture, as usual.

Video

- Clearly show all pairs of your shaders working. The video is very important for this project, because shaders that work on your machine may not work on the TAs machines due to hardware differences (e.g., your GPU may permit more lenient syntax than ours).