

CS 354

Project 5 - shaders

Due Friday, April 19, 5:00 pm

Overview

In this project, you'll gain experience writing programmable shaders. You'll write a vertex shader that bends a flat 2D grid into a torus. You'll also write several fragment shaders. Each fragment shader will implement an increasingly sophisticated lighting model. All the shader code you'll write will be written with OpenGL Shading Language (GLSL). In order to give your torus a bump appearance, you'll also write a small amount of C++ code to convert texels in a height map texture into normal vectors to be stored in a normal map. Your fragment shaders implementing bumpy lighting models will then be able to sample the resulting normal map textures.

The purposes of this assignment are:

1. Understand vertex and fragment shaders
2. Deepen understanding of texture mapping

In the project directory you'll find four subdirectories:

1. `cg4cpp` a C++ template library implementing Cg/HLSL-style vector, matrix, and other data types with accompanying standard library routines.
2. `glew` the OpenGL Extension Wrangler library assists in resolving OpenGL extension entry points.
3. `stb` public domain code for loading various types of image file formats.
4. `shader_scene` the project's C++ executable code.

Consult the `ReadMe.txt` file in the `shader_scene` subdirectory. The project is intended to build on the CS Ubuntu Linux machines. These machines have NVIDIA Quadro GPUs so can easily execute the project code, and this is the platform upon which the project case is tested. You can build this project code with Visual Studio 2008 for Windows operating system and that should work ok. Make sure you have OpenGL 2.0 or better for your PC. You might be able to make the `GNUmakefile` build for Mac OS X but that's an unsupported configuration.

Building the Project

Use the `GNUmakefile` in the `shader_scene` subdirectory to build the project code. Simply type `make` and the debug version of `shader_scene` will be built in the `bin.debug64` subdirectory. You can rebuild and run the project executable with

```
make drun
```

Driving the Project

Run the program from the `shader_scene` directory by typing

```
bin.debug64/shader_scene
```

Running the program for the first time you'll see a red square with a "cloudy hills" backdrop.

Mouse Controls

When you click the right mouse button, you'll get a pop-up menu that includes several sub-menus. With these sub-menus, you can select between various decals, normal maps, environment maps, light colors, and fragment shaders. (There's just a single vertex shader all the fragment shaders use.)

Clicking and dragging the left mouse button moves the viewer (or camera) location. Moving left-right rotates the viewer around a circle. Up and down moves the viewer up and down but always looking at the object. Essentially the viewer is constrained to move on a cylinder surrounding the object-soon-to-be-torus.

Clicking and dragging the middle mouse button (that is, clicking the track wheel) moves the light position. The light source is visualized as a sphere. The light source too is constrained to a virtual cylinder.

Holding down Ctrl and then clicking and dragging the left mouse button can spin the object-soon-to-be-torus. The object spins as if connected to a trackball and keeps spinning when you release. Ctrl-Left-Clicking and releasing without motion will stop the spinning. (There is also a menu option explicitly to stop the spinning motion.)

Keyboard Controls

There are a few keys supported. Pressing 'w' toggles the wireframe mode. This might be helpful to debug your modifications to `torus.vert`.

Pressing '0' through '9' switches between shaders which are numbered 00 through 09 in the `shader_scene/glsl` directory.

Modifying the GLSL Shaders

You'll be modifying the GLSL shaders in the `shader_scene/glsl` subdirectory. You'll find ten fragment shaders with the `.frag` suffix and a single vertex shader with the `.vert` suffix.

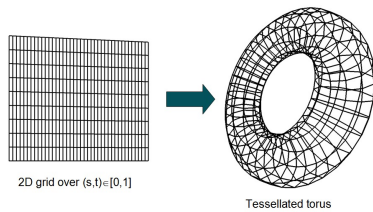
Tasks

Task 0: Rolling up the Square into a Torus

Modify the `torus.vert` vertex shader implementation so that the square patch with vertex positions ranging from `[0..1]` in both the `x` and `y` (or `u` and `v`) components is rolled up into a torus.

Research the torus to find a parametric function $F(u, v) = (x, y, z)$ for a torus. *Hint:* Wikipedia is a fine place to look.

Be sure to adjust `u` and `v` if their range is not the `[0..1]` range of the square patch vertex components. This figure visualizes what the vertex shader's job:



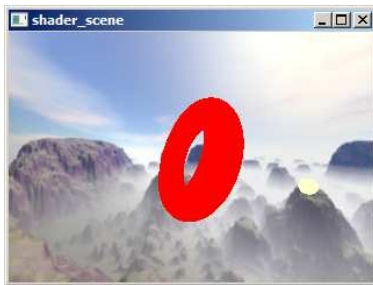
The incoming (u, v) attribute to `torus.vert` is named `parametric`.

Get the outer and inner radius of the torus from the respective x and y components of the `torusInfo` uniform variable. Notice that the existing code for `torus.vert` declares several attribute, uniform, and varying variables that are used to communicate with the C++ application and the downstream vertex shader.

When you compute your (x,y,z) position for the vertex on the torus in object-space, then transform this vertex position by the current modelview-projection matrix. *Hint:* GLSL provides built-in variables prefixed with `gl_` that track current OpenGL fixed-function state such as the modelview-projection matrix.

For subsequent tasks, you'll need to compute additional varying quantities in `torus.vert` for use by the downstream fragment shader. The first fragment shader is `00_red.frag` that unconditionally outputs red as the fragment color so initially you can simply output `gl_Position` for the torus (x,y,z) position in clip-space.

When you complete this task, your program should render a result like:



Task 1: Applying a Texture Decal

Once you can roll the red square into a torus, your next task is to shader the torus with a decal. This will require generating texture coordinates as a function of the parametric attributes. Output from your vertex shader to the `normalMapTexCoord` varying 2-component vector (s,t) .

Then in the `01_decad.frag` fragment shader, use this texture coordinate set to access the decal sampler.

Make sure the decal tiles 2 times in the inner (smaller) radius and 6 times in the outer (larger) radius. Assuming there are more fragments generated than vertexes transformed, would this scaling be more efficiently performed in the vertex or fragment shader?

When you complete this task, your program should render a result like:



Try picking other decals from the “Decal texture...” menu.

Task 2: Diffuse Illumination

In this task, you’ll shade the torus with a per-fragment ambient + diffuse illumination term by modifying the `02_diffuse.frag` GLSL shader.

To compute diffuse illumination, you’ll need a surface normal vector and a light vector.

Both of these vectors should be normalized. (GLSL has a `normalize` standard library function to normalizing vectors is easy in GLSL.) The dot product of these two normalized vector (clamp to zero if negative) models the diffuse lighting contribution.

You must make sure the light and surface normal vectors are in the same coordinate system (or sometimes stated “in the same coordinate frame”). This could be object space, eye space, or surface space. For efficiency reasons (and to facilitate normal mapping, particular environment mapping of normal mapping), it makes sense to choose surface space. In surface space, the (unperturbed) surface normal is always in the direction (0,0,1) pointing straight up in the direction of the positive Z axis.

The uniform vector `lightPosition` provides the position of the light in object space. The (un-normalized) light direction vector is the vector from the vertex position to the light position.

To transform an object-space direction into a surface-space, version you must construct a orthonormal basis (a rotation matrix) that can rotate directions from object space to surface space.

First compute the gradients of $F(u, v)$ in terms of u and v . You want:

$$\frac{\partial F(u, v)}{\partial u}, \frac{\partial F(u, v)}{\partial v}$$

Don’t trust yourself to differentiate a complicated function involved trigonometric functions?

<http://www.wolframalpha.com> can differentiate for you! As a simple example, try “diff(u², u)”.

We call the normalized gradient of F in terms of u the *tangent*. The cross product of these two normalized gradients is the (normalized) normal to the surface in object space as a function of (u, v) .

In general, the cross product of the normal and tangent vector is a normalized vector mutually orthogonal to both the normal and the tangent called the binormal.

These three normalized vectors T, B , and N for the tangent, binormal, and normal respectively can be used as column vectors of a 3x3 matrix M useful for converting directions and positions to and from object and surface space. So

$$M = [T \ B \ N]$$

When this matrix M is multiplied by a surface-space vector, the result is an object-space vector. Because M is orthogonal, the inverse of M is the transpose of M so

$$M^{-1} = \begin{bmatrix} T^T \\ B^T \\ N^T \end{bmatrix}$$

So multiplying M^{-1} by a vector in object space is the same as pre-multiplying that vector by M to convert that vector into surface space.

In GLSL, you can construct a 3x3 matrix with the `mat3` constructor with three `vec3` (3-component vector) treated as column vectors.

With this approach, the vertex shader can compute the object-space light vector (simply the light position minus the surface position, with both in object-space), and the transform this light vector into surface space. There is no need to normalize this vector in the vertex shader indeed, it is better to normalize it in the fragment shader after interpolation. The vertex and fragment shaders have a `lightDirection` varying vector intended to interpolate the surface-space light vector.

Computing the diffuse contribution in surface space is easy. The (unperturbed) surface normal is always $(0,0,1)$ so the Z component of the interpolated and normalized `lightDirection` is the diffuse lighting coefficient.

(Later for some of the bumpy shaders using normal mapping, the shader will substitute a perturbed normal obtained from a normal map texture to use instead of the unperturbed $(0,0,1)$ surface space normal.)

In the accompanying diffuse fragment shader for this task, we need to normalize the interpolated `lightDirection` and use the Z component as the diffuse contribution.

Because the diffuse coefficient is a magnitude and should not be negative, the fragment shader should clamp the coefficient to zero with the GLSL `max` standard library function.

The `LMa`, `LMd`, and `LMS` uniform variables provide an RGB color that is the light color (hence the `L`) and the material color (hence the `M`; with `a`, `d`, and `s` indicating the ambient, diffuse, and specular material color) modulated on a per-component basis. See the `Torus::draw` method to see where these uniforms are set.

In order for the diffuse shading to reflect the light and material colors, you should modulate `LMd` by the diffuse coefficient and add `LMa` to output a final fragment color for this task.

When you complete this task, your program should render a result like:



Try changing the “Material...” and “Light color...” settings to verify this shader is operating correctly. Move the light or spin the object. The region of the torus most facing the light surface should be brightest.

Task 3: Bumpy Diffuse

For this task, you should modify the `03_bump_diffuse.frag` shader so it operates in the same manner as the shader in Task 2 except rather than using an unperturbed surface normal, a perturbed normal sampled from a normal map is used instead.

Instead of sampling the decal sampler as in Task 1, sample the `normalMap` sampler to fetch an RGB value that represents a perturbed surface normal, based on a height map converted to a normal map.

Normals are stored as signed components but RGB textures store $[0..1]$ values. For this reason, the fragment shader in this task needs to expand the $[0..1]$ RGB values to be $[-1..+1]$ normal components.

You should also use the scale parameter to scale the height field Z component. By increasing scale, the bumpiness becomes more pronounced. When the scale is zero, all bumpiness should disappear. If the scale is negative the sense of the bumpiness will reverse, so outward bumps become inward bumps and vice versa. The ‘b’ key increases the scale while the ‘B’ key decreases the scale.

Once the normal map texels are generated correctly and the `normalMap` is sampled properly in the shader, the shader

needs to compute the dot product of the sampled perturbed normal and the interpolated and normalized lightDirection vector. This dot product result becomes your diffuse coefficient once clamped to zero avoid negative values.

When you complete this task, your program should render a result like:



Try varying the “Bump texture...” setting. Make sure when the normal map is “Outward bumps” that the bumps appear to bump outward consistently over the entire torus. Make sure the bump lighting on the torus responds to changes in “Material...” and “Light color...” menus.

Task 4: Specular

For this task, you should modify the `04_specular.frag` shader so the shading just shows a specular contribution.

Compute a Blinn specular contribution. For this you need to compute the dot product between the (unperturbed) surface normal and the normalized half-angle vector. The half-angle vector is the average of the light vector and view vector.

Whereas Tasks 2 and 3 computed the object-space light vector and transformed it into surface space, Task 4 requires doing the same for the half-angle vector.

You have two choices: A: Compute the half-space vector at each vertex and interpolated this value at each vertex, or (more expensively) B: Interpolate the view vector and light vector and compute the half-angle vector at every fragment.

Choice B is fairly expensive so the `shader_scene` examples have a `halfAngle` varying to interpolate the half-angle vector, but the view vector is also available so you can choose either approach.

Use the `shininess` uniform to control the specular exponent. Remember to force the specular coefficient to zero if the diffuse contribution is non-zero. Also remember to force the specular coefficient to zero when it is negative.

Modulate the specular color result by the `LMs` uniform value.

Note: The initial “Material...” has zero specular color so you should switch to a different material to see the specular highly properly. Otherwise you won’t see a specular highlight if you are modulating by `LMs`.

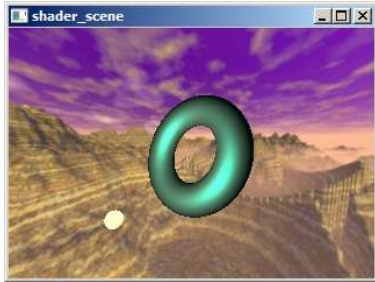
When you complete this task, your program should render a result like:



Task 5: Specular + Diffuse + Ambient

For this task, you should modify the `05_diffspec.frag` shader to include the ambient, diffuse, and specular lighting contributions assuming an unperturbed normal.

When you complete this task, your program should render a result like:



With this task, the lighting should change as the “Material...” and “Light color...” selections change but should not depend on the “Environment map”, “Bump texture”, or “Decal texture” choices.

Task 6: Bumped and Lit

For this task, you should modify the `06_bump_lit.frag` shader to include the ambient, diffuse, and specular lighting contributions with a perturbed normal from the `normalMap` sampler and with a decal color from the `decal` sampler.

Think of this task as combining Tasks 1, 3, and 4.

When you complete this task, your program should render a result like:



Task 7: Environment Reflections

For this task, you should modify the `07_reflection.frag` shader to reflect the object’s surroundings based on the `envmap` environment map and the surface’s unperturbed normal. First, find the reflection of `eyeDirection` off the surface. This will require the surface normal, but recall from task 2 that you can simply use $(0, 0, 1)$.

The reflected view vector is in surface coordinates. Convert it to world coordinates. This will use the `objectToWorld` uniform matrix and also M that you computed in the vertex shader. Use variables `c0`, `c1`, and `c2` to pass the tangent, binormal and normal vectors to the fragment shader and you can then reconstruct M . Remember, to convert a vector v from object to surface coordinates to premultiply v by M . To convert back, simply postmultiply.

Use the `reflect` GLSL standard library call to compute a reflection vector. Use the `textureCube` GLSL call to compute the color.

Task 8: Bumpy Environment Reflections

For this task, you should modify the `08_bump_reflection.frag` shader to reflect the object's surroundings based on the `envmap` environment map and the surface's perturbed, expanded normal from the `normalMap` sampler.

Task 9: Everything

For this task, you should modify the `09_combo.frag` shader to combine bumpy ambient, diffuse, and specular with bumpy reflections too. To avoid oversaturation, combine 50% of the ambient+diffuse, 50% of the specular, and 60% of the bumpy reflection.

Scoring

1. 10% - Task 0
2. 10% - Task 1
3. 10% - Task 2
4. 10% - Task 3
5. 10% - Task 4
6. 10% - Task 5
7. 10% - Task 6
8. 10% - Task 7
9. 10% - Task 8
10. 10% - Task 9

turnin

Submit and verify your submission as follows:

```
turnin --submit vkeshari project5 glsl  
mkdir test  
cd test  
turnin --verify vkeshari project5
```

Submit ONLY the `glsl` subdirectory. You are graded only on your shader code.