HOMEWORK 2: BASIC IMPLEMENTATION HINTS

Homework 2: Basic Implementation Hints

The assignment itself has been completely specified in the previous pages, along with skeleton code and some examples. What follows below are hints about how to approach the assignment in a step-by-step fashion (the second and third parts are largely independent and can be done in either order). However, you do not strictly need to proceed per the guidance below; it is only highly recommended (and is the basis for the solution program).

Note that these are basic implementation suggestions from the instructor. The next page has more detailed hints from previous students Kolega and hekimgil which you may also find (more) useful.

Additional Transforms

You should start with implementing additional transforms, beyond those required in homework 1. The skeleton implements the basic functionality; all you need to do is add the appropriate routines to the *Transform* class for translation and scale. In addition, you must implement the perspective transform yourself, rather than calling the GLM command. These modifications should not be too hard to do. You should now have homework 1, except you've written all the transformations yourself, and you can scale and translate the teapot. (If you can't yet get the teapot to display with the HW 2 framework, you may want to use the HW 1 framework instead for now so you can actually see it. That is, simply add the transformations to homework 1 and debug; then you can simply use the debugged `Transform.cpp` almost as is in homework 2 afterwards).

Lighting

Your next challenge is to implement lighting. At this point, you may want to fill in just enough of the parser to handle the *light* command. The important aspect is to develop the fragment shader. The example shaders for the demo (homework 0) and homework 1 already include the basic ability to deal with point and directional lights. You just need to declare a uniform to store all the 10 lights, and loop over them, adding the color to the final output. The skeleton includes the basic framework to set up the fragment shader. The shader also takes uniforms for the material properties. The shading equation you should implement is:

$$I = A + E + \sum_i L_i \left[D \max(N \cdot L, 0) + S \max(N \cdot H, 0)^s \right]$$

where I is the final intensity, A is the ambient term, E is the self-emission, and the diffuse D and specular S are summed over all lights i with intensity L_i. N is the surface normal, L is the direction to the light, H is the half-angle, and s is the shininess. You need to be able to compute the direction to the light and half-angle. Finally, please note that the attenuation for point lights, briefly mentioned during lecture, is *not* used in these shading equations (or equivalently it is set equal to 1; light intensity does not fall off with distance, so please use the equation above).

Geometry and Transforms

Finally, you need to implement the full file format. The scene and camera commands are pretty easily implemented, simply by inputting the values and then using them in the initialization routine. (Much of this is already done in the skeleton and the parts you need to code are clearly indicated). Note that the camera up vector need not be orthogonal to the direction connecting the eye and center, and you should create a full coordinate frame. (Support for this is already provided in the skeleton and the helper function in *Transform.cpp*). The material property parameters are easily implemented simply by keeping the current state of material properties and updating it as needed.

For the transform commands, one must maintain a stack of transformations. I recommend doing so with the C++ standard template library (you should look this up if you don't already know it). In particular, my code says

```
stack <mat4> transfstack; transfstack.push(mat4(1.0)) ;
```

This defines a stack of *mat4* and sets the initial value to the identity. Then, when you encounter a transform, you set up the corresponding glm vector and right-multiply the stack. My code uses a function with body:

```
mat4 & T = transfstack.top() ; T = T * M;
```

where M is the transformation matrix. Push and pop transforms just operate on the stack in the standard way. (Many of these functions are already available to you in the skeleton code).

All of this is relevant when an object definition (or light) is reached. For objects, you will store them in an array that includes the material properties (in effect when the object call happens) and the current transformation. Note that this transformation does not include the camera commands in my implementation; instead I multiply that in properly in the *display()* function. For lights, you will similarly multiply by the transformation matrix to store the transformed light.

Finally, the display routine loads the camera look at matrix, then sets up the lights, transforming them by this matrix. It then loops over the objects, setting the reflectance properties in the shader, and setting the correct transformation matrix. To set the correct transformation matrix, one needs to consider the overall translation, scaling, as well as camera matrix and the object transform, and concatenate them all properly. Do this yourself, and load it into OpenGL. Finally, you actually draw the object using *glutSolidCube, glutSolidSphere, glutSolidTeapot* with the size argument (the drawing is already in the skeleton code).

The one remaining element is parsing the scene file, for which the skeleton already provides an almost complete framework. You may use any method, this is not the core part of the assignment. I simply start with the code to read the shader in the demo, and turn each line into a string stream in C++. I check for blank and comment lines using:

```
if ((str.find_first_not_of("\t\r\n") != string::npos) && (str[0] != '#'))
```

and if so I just do `stringstream s(str) ; s >> cmd ;` where *cmd* is a string that is the command. A sequence of if statements then deals with each command. For parsing the remaining parameters, I have a simple function to read a specified number of parameters from a string and return a failure code if not.