HOMEWORK 2: ASSIGNMENT SPECIFICATION

Homework 2: Assignment Specification

File Format

Your program should be run with a single argument: the scene file (a second argument is used only if you want to output images for the autograder/feedback system; that part is handled by the skeleton). Each line in the scene file should be treated separately and can be of the form below. The skeleton code includes the core of a simple parser, and you only need to fill in a few items.

- **Blank line**: Ignore any blank line (that just has white space).
- **# comment line**: Any line with # as its first character is a comment, and should be ignored

- **command parameter1 parameter2 ...**: The first part of the line is always the command (to allow formatting, it may be preceded with white space, as in the example scene files). Depending on the command, it takes in some number of parameters. If the number of parameters is not appropriate for that command (or the command is not recognized) you may print an error message and skip the line.

In general, the program should respond gracefully to errors in the input, but parsing is not the point of this assignment, so we will test your program with well-formed input. The commands that your program must support are the following. We first start with general commands (you may require these be the first commands in the file, as in our examples, but you really shouldn't need to do that. In any case, you can use the core parser in the skeleton):

- **size width height** specifies the width and height of the scene.
- **camera lookfromx lookfromy lookfromz lookatx lookaty lookatz upx upy upz fovy** specifies the camera in the standard way. *zmin* and *zmax* are currently hardcoded. Note that this defines both the perspective (given the aspect ratio of *width/height* and *fovy*) and modelview transforms for the camera.

Next, you must support the lighting commands. There can be up to 10 lights, that are enabled simply by adding light commands to the file (your program should also gracefully disable lighting if no light commands are specified; the skeleton already includes support for this):

- **light x y z w r g b a** has 8 parameters, the first 4 of which specify the homogeneous coordinates of the light. It should be treated as directional (distant) if *w = 0* and as a point light in homogeneous coordinates otherwise. The colors are specified next; note that this is a 4-vector, not just a 3-vector (for now, just set the a or *alpha* component to 1).

If a user specifies more than 10 lights, you may skip the *light* lines after the first 10 lights are input.

Shading also requires material properties. Note that these properties will in general be different for each object. Your program must maintain a state (default values can be black), and update the relevant property each time it is specified in the scene file. *(Note that this is a simple state which is over-written. It is not affected by geometric transformation commands like* *push/pop etc.)* When an object is drawn, it takes the current material properties. It is possible for example to change only the diffuse color between objects, keeping the same specular color by using only a *diffuse* command between objects.

- **ambient r g b a** specifies the ambient color
- **diffuse r g b a** specifies the diffuse color of the surface
- **specular r g b a** specifies the specular color of the surface
- **emission r g b a** gives the emissive color of the surface
- **shininess s** specifies the shininess of the surface

We next need commands to specify object geometry. For simplicity, you can assume a maximum number of objects (this should be at least 10; there is no reason not to have a larger number of objects). For now, we are going to use a modern OpenGL implementation of the old glut commands for spheres, cubes and teapots, (all the relevant drawing code is already within the skeleton) so the commands are simply

- **teapot size** makes a teapot of given size
- **sphere size** makes a sphere of given size
- **cube size** makes a cube of given size

Please see the skeleton code for the specifics of how to call the relevant functions, and how these primitives are drawn; you don't need to worry about it for the most part.

Finally, we can specify transforms that should be in effect when executing an object above. Note that this *also includes lights*, as in standard OpenGL (that is, the lights are also acted on by the modelview matrix in standard OpenGL). *However, note that the overall scene translation, camera rotation and scale, specified with the keyboard do not act on the lights; only the transforms in the scene file act on both geometry and lights as in standard OpenGL.* We will implement a fairly complete set of transformations.

- **translate x y z** A translation 3-vector
- **rotate x y z $\boldsymbol{\theta}$** A rotation of θ degrees about the axis x y z
- **scale x y z** A non-uniform scaling

Note that the transformations can be combined with a sequence of transform commands, e.g., translation, rotation, translation, scale. Any transformation command should *right-multiply* the current transformation matrix, following OpenGL convention. This convention is confusing, since the *first transformation applied is the last one in the code* (or the transformation closest to the object command). For example, if one gives commands for translation, rotation, scale (which is the conventional order), then one scales first, then rotates, and then translates. See the skeleton code for helper functions to do the right multiplication, which may also help explain the concepts.

To allow for a hierarchical scene definition, we also define

- **pushTransform** "pushes" the current transformation onto the stack (after we are done with our transforms, we can retrieve it by popping).
- **popTransform** "pops" the transform from the stack (i.e., discards the current transform and goes to the next one on the stack).

Your program must initially load the transform stack with the identity. Note that there are no commands to explicitly set the transformation to the identity or make it a specific value. This is largely to get you to practice good design. In essence, as in the examples, the commands for each object should lie within a *pushTransform ... popTransform* block. These blocks may also be nested.