CS 184: Assignment 2 — Scene Viewer

Ravi Ramamoorthi

1 Goals and Motivation

This is a more substantial assignment than homework 1, including more transformations, shading, and a viewer for a scene specified in a text file. There are also many potential extra credit options available. To make the workload manageable, we require that this *assignment be done in groups of two.* It is your responsibility to ensure that group members contribute equally, and that each individual understands all aspects of the assignment, even if they didn't write the code for that part. It is generally expected that you will keep the same partner for homework 3 (also to be done in groups of two), but you may switch partners; in any case, it would be good to look at the specification for homework 3 as well, and plan accordingly.

This assignment can require some work, so even though you have two weeks, please *start early.* As always, use the instructor, teaching assistants and newsgroup as resources. Do not post actual code however on the newsgroup.

The overall goal is to take an input scene description that specifies the camera, scene width and height, lights (up to 10 lights), object geometry (for simplicity, only spheres, teapots and cubes) and material properties. The program should render this scene with correct camera positioning and lighting. The user should be able to rotate the camera viewpoint as in the previous assignment, and should also be able to rotate the lights. In addition, they should be able to translate and scale the scene. The specific scene format is described below.

2 Logistics and Submission

There is very limited skeleton code for this assignment, since once goal is to learn to write a complete program in OpenGL. Your starting point will be the solution to homework 1. I provide some skeleton source code for additional transforms which may be helpful. We provide two scene files *hw1.txt* and *demo.txt*, the first of which corresponds to homework 1, and the second roughly to the demo OpenGL program. The output for *demo.txt* is provided in the image *demo.tif.* We are not releasing the solution program, since it includes the shaders in plain text, but the GSIs should be able to demonstrate it for you if you want. (You can also get extra credit if you find bugs in the solution). Please download the skeleton code above as a zip file from the course website. For this assignment you are allowed to make modifications to all the source files (and add new files if necessary).

As with previous assignments, run *submit hw2* from a directory containing your entire source and executable code. In addition, include a *README* file in that directory that lists any special instructions, comments, brief writeup on what you did etc. Furthermore, provide images for at least the two input files *hw1.txt* and *demo.txt* generated from your program. Please also try to include additional results (both scene files and images or videos) that showcase some nice examples of your program. Also, include a brief writeup with images at a URL on your CS 184 account, and include a link in the *README* file. Finally, if you have done anything worthy of extra credit, please document it and include images. To simplify grading, it is best if you are honest and document what does and does not work. Some points may be awarded for the clarity of documentation.

Limitations on OpenGL and GLM Functions: Since the goal is to write a modern OpenGL program with minimal deprecated features, and also to learn the concepts, you *should not use any OpenGL and GLM functions*, with the following exceptions (if in doubt ask):

- OpenGL functions for low-level drawing, as in homework 1. This includes glut functions for setting up buffers, window management and keyboard events as in the skeleton, basic functions like clearing the color or redisplaying.
- Basic GLM matrix and vector functions like multiplication, normalization, dot and cross products.
- Very limited use of OpenGL matrix stacks. In particular, you may set the OpenGL matrix mode to perspective or modelview and load a matrix (using a *mat4* from GLM). However, you *may not* call any standard OpenGL commands like pushing and popping transforms, translating, rotating, multiplying the matrix stack etc. You also *may not* use their GLM equivalents. One goal of this assignment is to maintain the matrix stack yourself, and indeed, this is what modern OpenGL requires.

In particular, OpenGL and GLM provide standard functions for camera positioning, perspective projection, translation, rotation, scaling etc. Your goal is to learn and re-create this pipeline, so you *may not* use these commands; moreover many of them are deprecated and so should not be used in modern OpenGL.

3 File Format

Your program should be run with a single argument, that is the scene file. Each line in the scene file should be treated separately and can be of the form:

- *blank line* Ignore any blank line (that just has whitespace)
- *# comment line* Any line starting with a # in the 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. If you do the extra credit options, you will likely add commands and/or parameters; please document.

In general, the program should respond gracefully to errors in the input, but parsing is not the point of this assignment, so we will mostly 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):

- *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. You may want to extend this command with *zmin* and *zmax*, which 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):

- *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. 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 glut commands for spheres, cubes and teapots, 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

For the sphere, you also need to specify a tessellation internally when calling *glutSolidSphere*. I use 20; make this large enough so the polygonization isn't obvious.

Finally, we can specify transforms that should be in effect when executing an object above. Note that this *also includes lights*. We will implement a fairly complete set of transformations.

- *translate x y z* A translation 3-vector
- *rotate x y z* θ A rotation of θ 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.

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.

4 User Interaction

When your program is run with a scene file, it must interpret it as above, and draw the scene. In addition, it should recognize the same keys as in homework 1 (namely 'h', '+','-',ESC), as well as the following extensions:

- *'g'* as before, this switches between the *glm::lookAt* and the user commands. Now, it should also switch between *glm::Perspective* and your command, and by default it should call your code.
- *'r'* should reset the transformation to the initial scene, including lights.

- *'v'* should set the motion to correspond to rotating the camera (this should be the default and what you did for homework 1). Note that the camera need no longer be looking at the origin as the center, but you will still move the camera about the origin, keeping the center or look at point fixed.
- *'t'* should set the motion to translating the scene. In this case, the arrow keys should now move the scene in the x and y directions respectively. The *amount* should be scaled by *0.01* for this purpose.
- *'s'* should do the same for scaling the scene, again in x and y directions. Again, *amount* should be scaled by *0.01*. By pressing 't', 'v' or 's' one controls whether you modify translation, scale, or view. If one modifies say view, and then hits 't' to modify translation, the system should remember and keep the view change, i.e., should not revert to the original view.
- *'0'. . . '9'* should select the lights 0,. . . ,9 (depending on the order in the file) and should now move them, as above. The same crystal ball interface can be used. If the user hits a light that hasn't been specified, print an error message and continue (do not abort the program).

Note that when you move the lights, unlike the camera, there is no "up" direction. Instead, one should create a coordinate frame using the current camera up direction (might be different because of motion from that in the scene file) and the location of the light. Since up and light may not be orthogonal, one needs to go through the full steps of making a coordinate frame. Thereafter, one can simply rotate the lights as we do for the camera. One needs only the first three coordinates, for point and directional lights, since this rotation does not affect the distance of the light from the center.

5 Implementation Hints

The assignment itself has been completely specified above, 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).

5.1 Additional Transforms

You should start with implementing additional transforms, beyond those required in homework 1. We call this part (a), and skeleton code is provided. 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. Note also how the skeleton code handles the translation and scale, accounting for the column-major order in OpenGL. You should now have homework 1, except you've written all the transformations yourself, and you can scale and translate the teapot.

5.2 Lighting

Your next challenge is to implement lighting. At this point, you may want to include a very simple parser that only handles the *light* command. The important aspect is to develop the fragment shader. The example shaders for the demo 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 shader also takes uniforms for the material properties. The shading equation you should implement is:

$$I = A + E + \sum_i L_i \left[D \max(N \cdot L, 0) + S \max(N \cdot H, 0)^s \right], \tag{1}$$

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.

For moving the lights, as noted you need to take the up vector and make it orthogonal to the light location; then simply implement the standard crystal ball interface. It is important to note that the current transformation and model view matrix acts on lights, just as it does on geometry. Therefore, you must transform the light array properly before passing it to the shader (using *transformvec()* or an equivalent function).

5.3 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. 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.

The reflectance parameters are easily implemented simply by keeping the current state of reflectance 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. This is rather confusing because of the row-column switch, so the code actually left-multiples. My code uses a function with body as follows: *mat4 & T = transfstack.top() ; T = M * T ;* , where M is the transformation matrix. Push and pop transforms just operate on the stack in the standard way.

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. Again, remember row and column major for matrix-vector multiplication (see the hint in homework 1).

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 one remaining element is parsing the scene file. 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. This could also be done easily with standard C I/O. You are welcome to use something fancier.

5.4 Extra Credit

There are a number of opportunities for extra credit, with a small number of points proportional to effort. In essence, you want to build a complete scene viewer and manipulator. Opportunities include:

Lighting and Shading: Include a fancier lighting and shading model, such as spot lights, environment lighting etc. Include more complicated shading such as shadows with shadow mapping, reflections, textures and so on.

Scene Definitions: Include more complete support for scene graphs and instancing, to for example define the sphere in the example and then instance it to new copies. Look up scene graph methods and editors and build a more complete hierarchical scene graph.

User Interaction: Let the user visually modify the input file (at least given an option to edit and reload). Include more possibilities, at least a zoom key (maybe with the mouse) and ways to pick and scale/translate/rotate individual objects in the scene.

Geometry: Include support for more objects, at least triangle meshes with normals. For each object, store it in a vertex buffer and send it properly to the graphics card.

A really good extra credit solution would include the usability of something like ivview (this is a software on the SGIs from a time long ago, and you may want to look it up and try to emulate it), with full support for scene graphs and picking. Given the capabilities of today's programmable graphics cards, you should also include a variety of more complex visual effects. You may want to look for inspiration at the ray tracing assignment, and at least parse everything in that assignment (thus, your rasterizer in this assignment and raytracer in a later assignment could both take the same input file). It would be cool if you could provide raytrace quality images (not so hard, since you just need to do shadow mapping for which there are many online tutorials).