

CSE 167 (WI 2021) Homework 2

Download the code frameworks.

In this homework, we will build a complete scene viewer using OpenGL. This is a more challenging assignment than homework 1, and will probably take at least twice as much time.

Our 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. In addition, they should be able to translate and scale the scene.

2.1 Assignment Overview

We provide a complete skeleton code. It is more complicated than in homework 1, and one should try to understand how it works early on. As noted in the skeleton code, you will be making changes where indicated in four files below:

- `Transform.cpp`: This is similar to the `Transform.cpp` file in HW1, but now we add code stubs for translations, scales, and perspective projection.
- `readfile.cpp`: This file is for reading the input file (the parser structure is in place, but you need to deal with various other aspects.)
- `display.cpp`: This file is for displaying the scene.
- `shaders/light.frag.glsl`: This file is for the actual fragment shader.

We provide two scene files `hw1.txt` and `demo.txt`, the first of which corresponds to the scene seen in homework 1, and the second roughly to the demo OpenGL program.

OpenGL and GLM Limitations

Since the goal is to write a modern OpenGL program without deprecated feature, and also to learn the concepts:

- You are allowed to use OpenGL functions for low-level drawing, as in HW1. This includes glut functions for setting up buffers, window management and keyboard events.
- You are allowed to use Basic GLM matrix and vector functions like multiplications, normalization, transpose, dot, cross, and outer products. (There are some helpful routines for multiplications in the skeleton code; their use is optional.)
- No use of OpenGL matrix stacks. You **may not** call any standard OpenGL commands like pushing and popping transforms, translating, rotating, multiplying the matrix stack. Do not use their GLM equivalents either. One goal of this assignment is to maintain the matrix stack yourself, and indeed, this is what modern OpenGL requires.
- 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.

Submission

Submission will be done similarly to HW1, where we ask you to produce screenshots and submit code.

Run your HW2 solution with “`demo.txt input.txt`” as the command line arguments. The first argument specifies the scene file, while the second argument is our test case file. The second argument is only required if you wish to output the test case images.

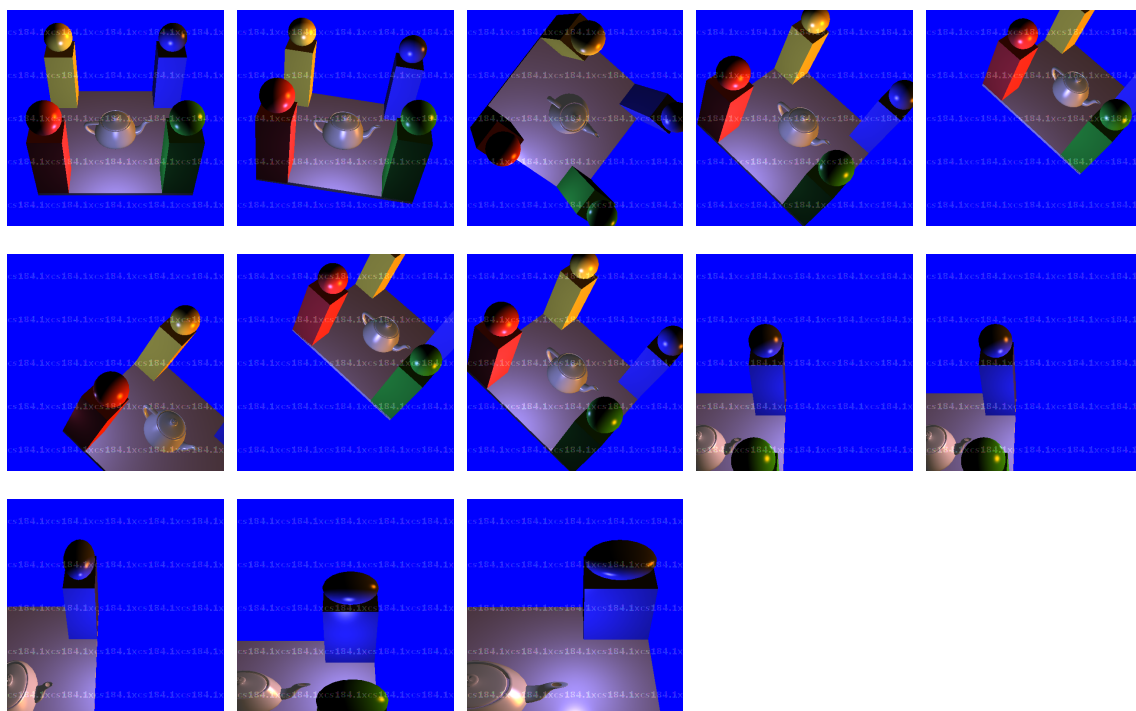


Figure 1 For your reference, the final exported images should look like these. (000–012 left-to-right, row majored)

In the Visual Studio, the command line argument is already specified (under Project → Properties → Configuration Properties → Debugging → Command Arguments).

Once the program starts, press and the program will automatically output all the required images (such as “demo.txt.input.txt.000.png”). Upload the specified images and the 4 cpp files you were asked to modified.

2.2 File Format

Your program can run with a single argument: the scene file. (A second argument is used only if you want to export images.) 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.

2.2.1 General Format and Starter

First of all,

- **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 parameter 2 ...:** 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 com-

mands 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.

2.2.2 Lighting

Next, you want to 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; not that this is a 4-vector, not just a 3-vector (for now, just set the *a* or *alaph* component to 1).

If a user specifies more than 10 lights, you may skip the *light* lines after the first 1- lights are input.

2.2.3 Material Properties

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.
- **r g b a** gives the emissive color of the surface.
- **shininess s** specifies the shininess of the surface.

2.2.4 Geometry

Next, we 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.

2.2.5 Transformation

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 θ** 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 (so that it represents going down closer to the leaf of the scene tree graph). Recall that the first transformation applied to a point position is the last one in the code, whereas the last transformation would be the view matrix (and the projection matrix).

The skeleton code has helper functions for the right-multiplication, which may also help explaining 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.

2.3 User Interaction

When your program is run with a scene file, it must interpret it as above, and draw the scene. Most of the basic for user interaction are already in the skeleton. In particular, the program should recognize the same keys as in homework1 (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. The amount of transformation given by keyboard input should also be reset.
- `[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 (in world space) respectively. The *amount* should be scaled by 0.01 for this purpose. (Note that *amount* is initially set to 5).

- `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.

2.4 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).

The Appendix has more detailed hints from previous students which you may also find useful.

2.4.1 Additional Transforms

We suggest that you 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).

2.4.2 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:

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

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 . The vector 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).

2.4.3 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. (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. (The 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. We recommend doing so with the C++ standard template library (you should look this up if you don't already know it). In particular, the skeleton 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. The skeleton code uses a function `rightmultiply` 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 this implementation; instead we multiply that 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 *solidCube*, *solidSphere*, *solidTeapot* with the size argument (the drawing is already in the skeleton code).

2.4.4 Parser

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. The skeleton simply starts with the code to read the shader in the demo, and turn each line into a string stream in C++. It checks for blank and comment lines using:

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

and if so it just does `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, there is a simple function to read a specified number of parameters from a string and return a failure code if not.

A Detailed Steps and Hints

Below are step-by-step instructions and hints (from Fall 2012 student KolegA), and further below also very detailed hints from hekimgil and from Spring 2013 student Aldur. Obviously, these hints are optional and not “official course policy” (and they used somewhat earlier skeleton versions... It has been updated to the current skeleton) but you will likely find them very useful. Also, this is a very long list of things, and you don’t necessarily need to read this section before starting. You may want to start the homework and refer to the basic hints first as you go along, and then look at the more detailed hints only if you get stuck.

A.1 Detailed Steps and Hints (Courtesy KolegA)

Step 0. Blue screen.

Compile framework, run, you should see window with blue background. It’s normal.

Step 1. Where is my teapot?

Add functionality little by little, make sure it works and then move to the next. So let us show something on screen. Select file `hw1.txt` as command line argument. Since we can’t see blue teapot on blue screen we need to change background color (first line in `display()` function) or object’s color (finalcolor variable from file `light.frag.glsl`). For example change object colors to red (this step has already been done in the current skeleton, with the shader setting the color to black if you do nothing; you can always change it to red if you prefer). Comment line `modelview = transf` from `display()` function, run it, press **G** to switch to `glm::LookAt` and `glm::Perspective` functions. Now you should see red (or black if you made no changes to the shader) teapot on blue background.

Step 2. Copy-paste HW1

Copy transform functions from your `hw1`. Run, press **G**, check if everything works like in `HW1`.

Step 3. I want bigger teapot

Implement scale and translate matrices, uncomment `modelview = transf` and think how to combine current `modelview` matrix, scale and translation. Review lecture about composing transformations if you have troubles with this step. Don’t be upset if you can’t make it work from the first try. Run, press **G**, make sure you can change size and position.

Step 4. Perspective teapot

Review lecture on `gluPerspective`. There should be enough information to implement the perspective matrix. Compare it with `glm::Perspective`.

Step 5. Turn on the light

Fill the arrays `lightposn` and `lightcolor` in `readfile.cpp`. Use for-loop, `memcpy` or `std::copy`. Pass them to the `light.frag.glsl` using `glUniform4fv`. Don’t forget to transform `lightposn` first, use array `lighttransf`, function `transformvec` and `hw0` and `hw1` as example. Implement shader (file `light.frag.glsl`) like in `hw0` but with for-loop. There are point lights and directional lights. Directional lights have zero in last element ‘w’ of the vector. Compare picture with picture from `hw0`.

Step 6. Almost done

Camera and transformations left. Every object has `matrix transform` change it in `readfile.cpp` (almost everything done by skeleton code), combine it with `transf matrix` and `set modelview` for every object in `display.cpp`. (In the current skeleton, the drawing routines like `solidCube` take care of setting the appropriate matrix transform for the modelview matrix in the shader, so your code in `display` just needs to update modelview but you do need to set the shader uniforms for the material properties explicitly).

A.2 Detailed Steps and Hints (Courtesy Aldur)

A.2.1 General

- it is best to get transformations right before moving to lighting.
- you can use `hw1.txt` to get the teapot outline and then play with all the transformations checking if they don't seem weird.
- once you have the transformations really working (you see 4 pillar outlines and such), dump the screenshots and compare autograder output; if the outline is not accurate, continue to work on transformations until you get it 100% right.
- for your shader editing you can use editor with GLSL syntax highlight (e.g. `notepad++`).
- if you really end up stuck, search thoroughly through the discussion, you may not get exactly the answer you seek, but a hint good enough to get back on track.

A.2.2 Transform.cpp

- you can start by copying the `rotate`, `left`, `up` and `lookat` from HW1
- `rotate` needs a small tweak: axis needs to be normalized before use
- implementing remaining functions can be done by strictly following math from the lectures, it helps a lot to write the function contents similarly to lecture slides, for example to initialize `mat4` with full 16-argument constructor, write them as if you were writing math matrix, then transposing them before returning (because `glm` is column major, not row major)
- remember that any angles need to be converted to radians before being used
- it is crucial to implement these functions correctly straight away, else they will haunt you later and possibly interfere with finding source of subtle bugs

A.2.3 readfile.cpp

- at every point where you are asked to write code, you can use `values[]` array to actually get the data, that has been read from file; slightly above `readvals()` has a comment indicating what values are available in what order, for example

```
readvals(s, 8, values); // Position/color for lts.
```

means that `values[0]` to `values[3]` hold a single light position corresponding to 4 vector coordinates, then `values[4]` to `values[7]` light color by analogy.

- light position and color vectors need to be sequentially added to `lightposn[]` and `lightcolor[]`, one coordinate per array index, therefore a single `vec4` takes 4 indices in such array, this means every 4 indices consist of a vector of one light (0–3 first light, 4–7 second light and so on)
- make good use of *unused* variable when saving light position and color to their respective array indexes
- I did not need to use `upvector()` for camera implementation

- translate, scale, rotate are also straightforward, use `rightmultiply()` and as second argument the transform stack itself, be sure to *understand* why this is working

A.2.4 `display.cpp`

- if you decide to use `transformvec()` for light transformations, you need to first load into opengl the matrix created via `Transform lookAt()`; this is because `transformvec()` obtains this matrix directly from opengl, check its definition! [This is no longer very relevant in the modern OpenGL skeleton, and `transformvec` takes the modelview matrix].
- transforming light positions by view matrix is indeed necessary and can be done for example via `transformvec()` function, you need to extract pos vector values from appropriate array one by one, apply this function to each set of 4 and then add results to `lightransf` array in exactly the same order and positions.
- you need to provide all light colors and transformed (!) positions in a SINGLE `glUniform` call per array, this call and the process of passing data to shader is quite unintuitive, therefore I suggest reading `glUniform` documentation and also I've found a very helpful hint to avoid certain error, available here (http://www.opengl.org/wiki/GLSL_%3a_common_mistakes)
- below lighting block, you need to construct transform matrix (`transf`), by multiplying translate, scale and modelview (`mv`) matrices (not necessarily in this order!), lectures provide all the information necessary to figure the correct order.
- you can access object fields by using "`obj->fieldname`" notation, this is C++ specific, remember to finally load matrix into modelview that furthermore correctly includes current object transform and your general transform! This makes perfect sense since you need to include both camera orientation and object orientation! If this still seems confusing, imagine that objects are drawn one after another, each with their own transformations and material properties applied beforehand (this is not entirely true, but serves as a good visualization of how this code flows).
- make sure to supply shader with shininess and other material properties using float version of `glUniform`, not the int!

A.2.5 `llight.frag.glsl`

- this may look like a very hard part, but in essence you just need to apply HW0 shader of the same name, then modify the light code to instead iterate over all light sources and apply them one after another to the result.
- you should use shader code from HW0, not HW1, as the latter has only 2 point light sources and does not show how to handle directional lights. Also lectures L7V3 around 6-7 minute and L7V4 describe this in even further detail.
- If in doubt on how to implement lighting calculation "pipeline", follow the Lighting formula Eq. (1).
- You should treat attenuation as if there was none (even for point lights). If you are using HW1 shader as base, then it's already coded for no attenuation and you do not need to change anything in that regard.