CS 184: Assignment 1 — Transformations

Ravi Ramamoorthi

Goals and Motivation

The purpose of the homework is to *fill in the code to allow rotation of the viewpoint around a scene,* using what is known as a crystal ball interface. *This homework is to be done individually.*

The assignment was designed with two main goals, as a gentle introduction to the course and initial material on transformations. The first goal is to understand how viewing and other transformations are used to render scenes. You will implement a simple crystall ball interface, standardly used for viewing, and also see how to build up the standard transformation matrices for this purpose.

The second goal is to get some initial familiarity with OpenGL so you can be set up for the next assignments. In particular, the assignment skeleton is written in OpenGL and GLSL. It also uses the modern GLM OpenGL Mathematics library, available at `http://glm.g-truc.net/`. The library and zip file are included in the skeleton to save you the trouble of downloading it. For this assignment itself, you do not strictly need to know much about OpenGL or shaders, but try to look through the main program and shaders to get a sense of how it all works. You are welcome to play around with it and explore; try to understand the solution framwork. However, do note for the final submission, the requirement is to *not submit any modifications to the main assignment framework.* This requirement is to provide standardization and for grading.

While the actual coding work is minimal, some thinking is needed, so you will want to *start early.* You may also run into unexpected problems with OpenGL, GLM, or GLSL for this first assignment. Post any questions you have to the newsgroup, since other students will want to see the answers too. However, do not post anything resembling code.

Logistics and Submission

Download the assignment (you should already have gotten the skeleton working in HW 0). Now, compile and run the assignment. You should see the same teapot as in the solution on a blue background, but the arrow keys won't work. Also, if you hit 'g' to toggle to using your own *lookAt* function, you will get something weird. Your job will be to make the arrow keys and viewing work.

You should have debugged issues with compilation already in homework 0. We will discuss OpenGL and GLSL programming in class, but what you actually need to do in this assignment doesn't strictly require that information. We do provide the GLM math library, and in fact, your inputs and outputs will be GLM vectors and matrices. The operation of the library is quite intuitive, except for caveats below. Documentation can be found online, or in the *doc/* subdirectory of the GLM installation. We have also provided a pre-compiled solution in *transforms_sol* (it is *transforms_sol.exe* on windows, *transforms_sol_osx* for Mac OS X, and *transforms_sol_linux* for linux. Note that binary incompatibilities may prevent it running on some machines). Your program should match that exactly (this is important mainly for nailing down the sign conventions). Please do not attempt to decompile or otherwise reverse engineer the solution.

For the final submission **only make changes to** *Transform.cpp*. Do not add any *#include* lines to the code. Do not change *Transform.h*. If you do make changes to any of the other files, make sure your solution works without those changes. (In particular, some operating systems like Windows may require additional includes. You may need to remove these includes before submitting to our feedback servers.)

Submission: First, you *must* use the feedback servers and provide a link to the output you obtain. (Details on how to access the feedback servers will be provided on Piazza). Second, you should submit your *Transform.cpp* (this is mainly so we can test your code on additional test cases if needed, and inspect your code in case you have errors). Run *submit hw1* from a directory containing only two files: *hw1.txt* that just includes a link to the feedback server output (for *both* code and image graders), and *Transform.cpp*.

Assignment Specification

You will be implementing a classic crystal ball interface. This simulates a world in which the viewer is glued to the outside of a transparent sphere, looking in. The sphere is centered at the origin, and that is the direction towards which your eye is always pointing. At the origin, there is something interesting to look at, in this case, a teapot.

You can change the viewpoint by rotating the crystal ball in any direction about the origin. Usually this is done with a mouse, but you will be using the keyboard for this assignment to make things easier. You must think about how the position of the eye and direction of the up vector change with left-right or up-down rotations.

Fill in the parts of *Transform.cpp* that say "//FILL IN YOUR CODE HERE". First, you should fill in *left()* and *up()*. Once these are working, fill in *lookAt()*. You must also fill in the helper function *rotate()* and you can use it in your code (this function simply sets up the rotation matrix for rotation about a given axis; we will discuss construction of this matrix in class).

The compiled correct solution has been provided for you with name *transforms_sol* (see above for exact filename on different platforms). Your solution must behave identically. In addition, the skeleton code allows the use of the 'g' key to toggle between using the system (GLM) lookAt command and yours (initially set to the system command). One verification is to press the arrow keys a few times and then hit 'g' to toggle and verify that your *lookAt()* matches the system version. This verification is usable even if you are on a different platform and cannot get the pre-compiled solution to work. Note though, that you still need to follow the pre-compiled solution in terms of its sign conventions and behavior for *left()* and *up()*.

You also need to use the feedback servers (details to be provided on Piazza), which will enable you to fully test that you match the true solution exactly. As usual, you can submit multiple times to the feedback servers; when satisfied, include links to the output for both code and image feedback in your submission. Also as usual, note that we may test on more inputs, and read your code to assign you a fair grade (grading need not be based solely on feedback server output; we may also adjust the feedback server thresholds for grading to fairly differentiate different systems versus conceptual errors).

GLM Libraries: Documentation, Restrictions and Caveats

In the course of modifying *Transform.cpp*, you will want to make use of the helper classes and functions from the GLM libraries. The documentation describes them, but for the most part is intuitive (see caveats below and in code comments though). In particular, you have *vec3*, *vec4*, *mat3* and *mat4* classes for matrix and vector storage and operations. The functions *rotate* and *lookAt* must return matrices. In fact, these are all in the *glm* namespace and we use typedefs in *Transform.h*.

By default, matrices are stored in row-major order (but see caveats below) and can be indexed simply at *matrix[0][3]* for example, to get the first row and fourth column of a *mat4* matrix. Vectors can simply be indexed as arrays. Matrix-vector and matrix-matrix multiplication, addition etc. work simply as overloaded operators, but see caveats below. A matrix or vector can be initialized with a constructor that specifies its elements, for instance *vec4 V(a,b,c,d)* for a *vec4* or a matrix constructor with 16 arguments for a *mat4*. Initializing with a single value also repeats the elements for example *vec3 v(1.0)*. For a matrix, it instead repeats the elements on the diagonal. An identity matrix can be created using for example, *mat4 I(1.0)*. In general, GLM is designed to have the same syntax and be compatible with OpenGL and GLSL.

Helper Functions and Restrictions: You are welcome to use *glm::dot*, *glm::cross* and *glm::normalize* to operate on vectors for standard dot-products, cross-products and normalization. You do not strictly need to, but you may also use *glm::radians* and matrix-vector or matrix-matrix multiplication from the glm library (via overloaded operators). **You may not use other glm or OpenGL functions**, except for overloaded operators for arithmetic operations and array subscripting. You may not use any OpenGL or glu functions.

Caveats on Row vs Column Major: OpenGL stores matrices in column-major order, while standard GLM uses row-major but is designed to be compatible with OpenGL. In addition, matrix multiplication in OpenGL right-multiplies, which is opposite from standard conventions. All of this can get pretty confusing. As far as this assignment is concerned, one key caveat is in writing a matrix-vector product like $y = M * x$ where M is a matrix, while x and y are vectors. In the default case, GLM will treat M as column-major effectively using its transpose. The "correct" way is to write $y = x * M$ which seems to be handled as expected. Because of this confusion, we recommend that you use matrix-vector multiplication sparingly, and avoid matrix-matrix multiplication altogether, unless you really know what you are doing for this assignment.

Please also note that you will return a row major matrix from *lookAt* (that the skeleton code automatically transposes for compatibility with OpenGL. In constrast *glm::lookAt* already does the transpose to be compatible with older OpenGL code). Since GLM is just a header library, it's actually easy to also just inspect the source code to see what it's actually doing if you are confused. If you didn't understand this last paragraph at all, you don't need to worry about it.

Hints

We include below some optional hints that may be of interest. You are not required to use or refer to any of the material here, however. This mainly pertains to the material needed in *Tranform.cpp*. *Main.cpp* is reasonably documented, and one goal is for you to try to learn some OpenGL if you want to look at it in more detail. You may want to augment it, for example adding options for scale and translation as well. However, your final submission should only include *Transform.cpp*, and should work with unmodified skeleton code.
Rotate: Rotate just implements the standard axis-angle formula to create a rotation matrix. We will discuss the formula in class; this is a good exercise in correctly coding vectors and matrices. You will of course need to use standard trigonometric functions and convert degrees to radians.
Left: The simplest function to fill in is left. The input is the degrees of rotation, the current eye 3-vector and current up 3-vector. Note that you may need to convert degrees to radians (in the standard way) to set up a rotation matrix. Your job is to update the eye and up vectors appropriately for the user moving left (and equivalently right). This function should not require more than about 3 lines of code to do the appropriate rotations.
Up: The up function is slightly more complicated, but satisfies the same basic requirements as left. You might want to make use of helper functions like *glm::cross* and auxiliary vectors. Again, you need to update the eye and up vectors correctly.
lookAt: Finally, you need to code in the transformation matrix, given the eye and up vectors. You will likely need to refer to the class notes to do this. It is likely to help to define a *uvw* coordinate frame (as 3 vectors), and to build up an auxiliary 4×4 matrix M which is returned as the result of this function. Consult class notes and lectures for this part.

Acknowledgements

This assignment was developed with Aner Ben-Artzi (a former Berkeley undergrad) a few years ago. The skeleton has been rewritten in modern OpenGL using GLM and GLSL and a few clarifications have been added. Nicholas Estorga developed the feedback system based on Brandon Wang's efforts in Spring 2012.