CMSC 23700 **Introduction to Computer Graphics** **Project 1**
Winter 2012 **January 10**

Geometry, Trackball, and Transforms

Due: Wednesday, January 25 at 10:00pm

(v1.1 Jan 12: pairs allowed, clarified requirements for designed scene, added hint)

1 Summary

You can complete this project individually or in pairs (but not groups of three or more). The purpose of this project to is to thoroughly test your understanding of transforms, to introduce you to the mechanics of rendering with shader-based OpenGL, and create a trackball interface that may be useful in future projects. The starting point for your work is demo C code that uses the GLFW library to create the OpenGL context and display window, and uses the `spot` library functions to create and draw some simple geometry.

2 Description

You will create and apply a range of transforms: model, view, and projection (both orthographic and perspective). You demonstrate your implementation by building one or more scenes from the primitives provided and supporting interactive adjustment of parameters from the keyboard and mouse. In total you should involve at least *four* of the different primitives supplied (cone, cylinder, sphere, and cubes with hard or rounded corners).

The rendering of the scene is performed by a vertex and a fragment shader (`simple.vert` and `simple.frag`) which you should not have to change, but reading their source may be informative. The vertex shader implements a single diffuse light, with a *world-space* position passed in through variable `lightDir`, which is the vector pointing towards the light (make sure this maintains unit-length).

2.1 Keyboard Interface

1. Arrow keys: 3 degree rotations around the viewspace horizontal **U** vector (up and down arrows) and the vertical **V** vector (left and right arrows). The front-most part of the scene should move in the direction of the arrow. In this and all other rotations performed by the view transform, the look-at ("`at`") point stays fixed, and the eyepoint "`from`" location is rotated.

2. "`u`": toggle whether the "`up`" vector is fixed in world space, or whether it is transformed along with the eyepoint ("`from`") vector during viewpoint changes. Default is that "`up`" is fixed.

3. "`p`": toggle between orthographic and perspective projection transforms. Default is orthographic.

4. "`1`", "`2`", "`3`": trigger display, and printing textual description, of scenes (or different viewpoints of scenes) that you create to debug your implementation of the transforms, as described in Suggested Process and Scene Description below.

5. "`v`": Subsequent mouse interactions update camera and view transform (the default)

6. "m": Subsequent mouse interactions update (by translation in the model transform) one object of your choice in the scene you create

7. "l": Subsequent mouse interactions update (by rotation) the light direction.

8. "q" or "Q" to quit.

With "v", "m", and "l", provide some output (via `printf` to `stdout`) as to which mode the mouse interactions are in.

2.2 Mouse Interface

The interface will perform rotations and translations according to the movement of the mouse while the button (any button) is pressed down. The action performed depends on where in the window the initial click happens. Holding down shift will modify the action of the movement.

1. Within the *bottom* 1/5th of the window, horizontal motion induces:
   (a) without shift: rotation around viewspace **N** vector, applied to either the eye point (mode "v") or to the light direction (mode "l")
   (b) with shift: translation along viewspace **N**, of both the eye point and look-at point (mode "v") or the transformed object (mode "m").

2. Within the *left* 1/5th of the remaining portion of the window, vertical motion induces changes to the view (in mode "v", no effect in modes "m" or "l"):
   (a) without shift: "zooming" in and out (controlling field-of-view angle with perspective projections, and scaling the size of the front face of the axis-aligned view volume with orthographic projection).
   (b) with shift: scaling the difference between the near and far clipping planes (their mean distance to the eye stays fixed). Provide textual output in `stdout` of far-near.

3. Within the remaining upper right 4/5ths of the window, horizontal and vertical motion induces:
   (a) without shift: rotation around viewspace **U** (vertical motion) and **V** (horizontal) vectors, applied to view (mode "v") or light (mode "l").
   (b) with shift: translation along viewspace **U** and **V** vectors, applied to both eye and look-at points (mode "v") or the transformed object (mode "m").

With all of these, you will have to make some aesthetic judgements about how much motion creates how much change. For the rotations, one rule of thumb is that motion across the full width or height of the window should induce about 360 degrees of rotation.

3 The Software

The provided code will compile and run on the Maclab. Even though it may not be hard to port it to Linux or Windows, we can't help with that (but we'd be interested in hearing about your experiences).

3.1 `proj1.c` demo code

A skeleton of the Project 1 code is provided to you, and contains many "`YOUR CODE HERE`" sections. A walkthrough of this code and its overall organization will be given in the Lab section. Feel free to organize functions to new source files; such changes will have to be reflected in the Makefile and `extern` function declarations in `proj1.c`.

Note this project uses a "core" OpenGL 3.2 context (none of the deprecated functionality from previous OpenGL versions is available); see the setup for this at the beginning of `main()`. Changing the OpenGL version is not needed and not allowed for this project!

3.2 GLFW

We use GLFW (instead of the unmaintained GLUT) to manage a window and an OpenGL context, and to receive information about mouse and keyboard events. Please see the GLFW documentation for more information: http://www.glfw.org/documentation.html. You will probably need to consult the Reference manual for details on how to, for example, detect when the arrow keys are typed.

3.3 `spot` library functions

To handle some of the less intuitive parts of rendering geometry with modern OpenGL, we have organized some utility functions into a library called "`spot`" (the name is not meaningful). For now, the documentation for the library is the `spot.h` header file. You are responsible for reading through this to learn what functions are available and what they do, and how the functions are distributed amongst the source files. You shouldn't have to modify anything about `spot` to complete the project. If you do make modifications to `spot`, justify these in comments around the modification.

4 Suggested Process and Scene Description

Carefully review the demo code to understand how it works, and experiment with making targetted changes to see what effect they have. This project will develop your skills of *visual debugging*: understanding and fixing your code through visual rather than textual output. We recommend proceeding via the following steps:

1. Modify the given scene geometry by directly transforming the `spotGeom->xyz` coordinates of one of the objects (perhaps translation and uniform scaling; things that won't induce any change in the normals). By varying positions and colors of a few simple objects, you can make a scene for which the rendered appearance will tell you something about the correctness of the view and projection transforms. Start with the simplest possible orthographic projection.

2. In the same way that `demoM` is manually set to a fixed matrix in the distributed code, you can set fixed values for the `modelMatrix`, `normalMatrix`, `viewMatrix`, and orthographic `projMatrix`, according to the formulae in the book and in lecture.

3. If something about the transforms doesn't seem to work, you can (in C) apply the transforms in sequence to a single vector (`GLfloat testvec[4]`), printing the vector value before and after matrix multiplication, to debug the behavior of the transforms.

4. Instead of directly transforming (in C) the vertex positions, produce the same transform by passing a per-object model transform to the vertex shader.

5. Document the scene geometry that illustrates that your transforms are working as expected (you can correctly predict what the rendered image will look like). Make pressing "`1`" trigger the creation of these scene geometry and rendering, and printing of your description to `stdout`.

6. Implement rotations of the orthographic view transform based on arrow key input. It will be more reliable to recompute the view transform from camera parameters with each viewpoint change, rather than trying to update the view transform matrix by matrix multiplication. In fact, no transform that is adjusted by user input should need to be computed by repeated and incremental matrix-matrix multiplication (i.e. something of the form $[M] = [\text{adjustment to } M][M]$).

7. Design a logical framework for organizing the response to mouse movement, which will have to depend on: mode, clicked window region, and shift-click vs regular click.

8. Implement rotations of the orthographic view transform based on mouse movement.

9. Implement perspective view transforms and toggling between orthographic and perspective.

10. Document the scene geometry and camera parameters you used to confirm the correctness of your perspective transform, and make pressing "`2`" show this, and print your description.

11. Implement rotation of the light direction from mouse input.

12. Implement translations in the view transform and in the model transform based on mouse input.

13. Implement scalings or shears in the per-object model transforms, and verify that the normals are being correctly updated. Read the note about the ellipsoid shape in `spot.h` for a hint about some geometry that might help debug this.

14. Make pressing "`3`" show your demonstration of correct normal transforms, with printed description. Include an object (besides a sphere) which has been scaled along a *different* axis than one of the model-space coordinate axes.

15. Clean up your code and reorganize it as needed to reflect your new understanding of how to build this kind of program. Document it with complete sentences in comments.

You will not be able to finish if you do not start soon. Get your feet wet immediately by trying the first few steps, to help develop a realistic plan about how long subsequent steps will take. As you will likely learn, it is easy to make changes that give you a blank screen or an otherwise uninformative image. Take thoughtful and tactical steps in editing your code, constantly re-compiling and re-running it, so that you know exactly which change created a problem. Use `svn` to make backups of different versions as you go (you can also recover specific versions). When you ask for help, saying "I changed X from A to B, and that produced unexpected change Y" should be the starting point of discussion. Saying "It doesn't work" gives us little to go on, and will frustrating for everyone.

5 Style Considerations

The correctness portion of the grade (70%) will be based on how you implement the user interface functions above. For the style portion (30%), consider the following:

1. The code you hand in should not achieve any transformations by directly modifying (in C) the per-vertex positions and normals. All transforms should be implemented by passing matrices to the vertex shader.

2. With all warnings enabled `-W -Wall`, your code should compile without warnings.

3. Any modifications to `spot` should be well-justified.

4. Significant constant values for parameters shouldn't be hidden in the code; information should be communicated by variables that are initialized in well-defined locations.

5. Avoid introducing new global variables.

6. Any resources that were created and initialized should be cleaned up prior to exit.

7. In the context of the main loop of interaction and rendering, think when values change, about where they are learned and used. Try to avoid re-setting things in an inner loop that can be set once in an outer loop.

8. In reading the documentation of the code in your comments, we should be able to understand the effect of the functions you implement, and the basic steps taken by the function. Provide the information that some unfamiliar with your code and this project would need in order to understand the organization and functioning of the code.

6 Submission

We will set up a Pheonixforge repository for each student; for details on checking out the demo code and submitting your project, see http://people.cs.uchicago.edu/~glk/23700. We will collect the projects at 1:00pm on Wednesday January 25th from the repositories to begin grading.

Remember the late policy: late programming projects will be docked 12% every 6.0 hours, in increments of 6.0 hours, based on the clock on the Pheonixforge server. This is linear not an exponential process; after 72 hours the project will receive a zero.