CS488
Assignment 2

"Can I use a dialog box to implement rotations?"
– A student working on Assignment 2 (who was promptly told 'No')

This assignment is due **Wednesday, February 1 [Week 5]**.

If you still need the provided code for this assignment run `/u/gr/cs488/bin/setup A2` from your CSCF account.

1 Topics

- Modeling and viewing transformations.
- Perspective and homogeneous coordinates.
- 3D line/plane clipping.
- Menus, menubars, valuators, and message areas.

2 Statement

You are to write a program that displays and interactively manipulates a wire-frame box that you should construct with vertices at the 8 unit points (+–1,+–1,+–1).

You are to provide a set of coordinate axes (a **modelling gnomon**) that you should construct with three lines, drawn from (0,0,0) to (0.5,0,0), (0,0.5,0), and (0,0,0.5) respectively. This gnomon will represent the local **modelling coordinates** of the box, and it must be subjected to every modelling, viewing and projection transformation applied to the box *except scaling*.

You are also to draw a separate set of axes for the **world coordinates**, which are the same size (coordinates) as the modelling gnomon, but are aligned with the world coordinate axes instead of the modelling axes. The world gnomon should be at (0,0,0). You should subject this **world gnomon** only to the viewing and projection transformations.

Note that the gnomon are merely graphical representations of the coordinate axes; for the coordinate axes themselves you should use orthonormal bases.

You are to apply **modelling** transformations to the box (*rotations*, *translations*, and *scales*) and **viewing** transformations to the eyepoint (*rotations* and *translations*). Transformations will be

menu-selected and will be applied according to mouse interactions. Specifically, x motion of the mouse will be used as a *valuator* to control the amount of each transformation, the mouse *buttons* will be used to select the axis of the transformation, and a menu will be used as a *choice* device to determine the major modes of the program's execution.

You will need to maintain four distinct coordinate systems in this assignment. Three of these coordinate systems are 3D and one is 2D: the box ("model") coordinates (3D), the eyepoint ("view") coordinates (3D), the universal ("world") coordinates (3D), and the display ("screen") 2D normalized device coordinates (which arise from the perspective projection of the eye's view onto the eye's x, y plane).

The modelling transformations apply with respect to the model coordinates (i.e. a *model mode rotation* about the the x axis will rotate the box about its current x axis, not the world's x axis). The viewing transformations apply with respect to the view coordinates (i.e. a *view mode rotation* about the x axis will appear to swing the objects of the view up or down on the screen, since the eye's x appears parallel to the screen's horizontal axis). None of the modelling transformations will change the world coordinates (i.e., the world gnomon never changes its location, though, of course, it may drift out of our view as a result of viewing transformations).

You are to form all matrices yourself and accumulate all matrix products in software. You will also do the perspective projection yourself, including the division to convert from 3D homogeneous coordinates to 2D Cartesian coordinates. This means that you will have to do a 3D near-plane line/plane clip in the viewing coordinate system to avoid dividing by zero or having line segments "wrap around".

3 The Interface

As with the previous assignments, this assignment is implemented in C++ with `gtkmm`. We have provided some code for you that sets up a window and draws an example set of lines for you. You will need to add the parts that do all the 3D transformations, projections, etc.

We suggest that you add a few matrices to `Viewer` (in `viewer.hpp`) – which ones you add and what each means is up to you. At the very least, you will need a modelling matrix and a viewing matrix.

You should implement the stub functions already found in `Viewer` and perhaps add some more of your own. You will also want to implement the missing matrix functions in `a2.hpp` and `a2.cpp`, and use these in your `Viewer`.

Your viewer should have an on-screen indication of where the near and far planes are located (from a call to `Viewer::set_perspective`). Simple textual Gtk labels are fine. Document how you display this in your manual.

4 Drawing lines

In `draw.c`, we provide the following C++ routines to draw lines and set colours in a Togl widget:

- `draw_init(int width, int height` — call this before drawing any line segments. It will clear the screen and set everything up for drawing.

- `draw_line(Point2D p, Point2D q)` — draw a line segment. `p` and `q` are in window coordinates.
- `set_colour(Colour col)` — set the colour of subsequent calls to `draw_line`.
- `draw_complete()` — call this after you are done drawing line segments.

These routines use OpenGL. Your assignment should not contain any further OpenGL calls. Further, you should not modify `draw.cpp`. You should call these routines from `viewer.cpp` with the GL widget active. There is already example code in `viewer.cpp` that does this for you.

5 Top Level Interaction

The menubar will support (at least) two menus: the `Application` and `Mode` menus. The `Application` menu will have two selections: `Reset` (keyboard shortcut `A`), which will restore the original state of all transforms and perspective parameters, and set the viewport to its initial state; and `Quit` (keyboard shortcut `Q`), which will terminate the program.

The `Mode` menu selections will be used to determine what effect mouse dragging on the viewing area will have on the transformations. The `Mode` menu will consist of a list of radiobuttons, which select among the viewing and modelling modes, and a viewport mode. There should be an on-screen indication of what mode is currently active (eg., a message bar).

In any View and Model interaction modes, transformations are initiated with the cursor in the 3D viewing area, upon a button down event. Relative motion of the cursor is tracked and the transformations are continuously updated until a button up event is received. The current interactive mode should be presented in a message bar somewhere on the display widget (hint: use a `Gtk::Label` widget). You should also show the locations of the near and far plane.

If multiple mouse buttons are held down simultaneously, all relevant parameters should be updated in parallel. For rotation, you may apply the rotations in a fixed order, as opposed to composing multiple infinitesimal rotations. However, this ONLY applies to the case where multiple mouse buttons are held down; in general, you will want to be able to compose an *arbitrary* sequence of transformations.

These interaction modes are a bare minimum, and form a poor 3D user interface. We'll look at better ways to create an interface to 3D rotations in Assignment 3.

6 View Interaction Modes

The following view interaction modes should be supported:

Rotate (keyboard shortcut `O`): Use x-motion of the mouse to:

- **B1:** Rotate sight vector about eye's x (horizontal) axis.
- **B2:** Rotate sight vector about eye's y (vertical) axis.
- **B3:** Rotate sight vector about eye's z (straight into eye).

Translate (keyboard shortcut `N`): Use x-motion of the mouse to:

B1: Translate eyepoint along eye's x axis.

B2: Translate eyepoint along eye's y axis.

B3: Translate eyepoint along eye's z axis.

Perspective (keyboard shortcut `P`): Use x-motion of the mouse to:

B1: Change the FOV over the range of 5 to 160 degrees.

B2: Translate the near plane along z.

B3: Translate the far plane along z.

A good default value for the FOV (Field Of View) is 30 degrees.

7 Model Interaction Modes

The following model interaction modes should be supported:

Rotate (keyboard shortcut `R`): Use x-motion of the mouse to:

B1: Rotate box about its local x axis.

B2: Rotate box about its local y axis.

B3: Rotate box about its local z axis.

Translate (keyboard shortcut `T`): Use x-motion of the mouse to:

B1: Translate box along its local x axis.

B2: Translate box along its local y axis.

B3: Translate box along its local z axis.

Scale (keyboard shortcut `S`): Use x-motion of the mouse to:

B1: Scale box along its local x axis.

B2: Scale box along its local y axis.

B3: Scale box along its local z axis.

The initial interaction mode should be model-rotate, and this mode should be restored on a reset.

The amount of translation, rotation, or scaling will be determined from the relative change in the cursor's x value referenced to the value read at the time the mouse button was last moved. Make sure your program doesn't get confused if more than one button is pressed at the same time; all the motion events should be processed simultaneously, as specified above, although individual "incremental" transformations can be composed in a fixed order.

You should use appropriate scaling factors to map the relative mouse motion to reasonable changes in the model and viewing transformations. For example, you might map the width of the window to a rotation of 180 or 360 degrees.

Do not limit the *accumulation* of rotations and translations; i.e., there should be no restriction on the cumulative amount of rotation or translation applied to an object, or to the number of sequential transformations.

8 Viewport mode

The viewport mode allows the user to change the viewport. Assume that you have a square window into the world, and that this window is mapped to the (possibly non-square) viewport. Don't bother trying to preserve the aspect ratio. You should draw the outline of the viewport so that we can tell where it is.

In the Viewport mode, the left mouse button is used to draw a new viewport. The left-mouse-button-down event sets one corner, while the left-mouse-button-up event sets the other corner. You should be able to draw a viewport by specifying the corners in any order. If a mouse up position is outside the window, clamp the edges of the viewport to the visible part of the window.

The initial viewport should fill 90% of the full window size, with a 5% margin around the top, bottom, left and right edges. This is important so that we can verify that your viewport clipping works correctly – if you do not do this, you may lose marks in two places. The user should be able to set the viewport to any portion of the window, including sizes large than the original size. Note also that the viewport is to be reset to the initial size when the reset option is selected from the file menu.

The keyboard shortcut for viewport mode should be `V`.

9 Projective Transformation

You will need to implement a projective transformation. This will make the cube look three-dimensional, with perspective foreshortening distinguishing front and back. You may use a projective transformation matrix, if you wish. However, note that for this assignment there is no need to transform the z-coordinate. You can use the mappings x/z and y/z, although note that some additional scaling will be necessary to account for the field-of-view.

10 Orthographic View

If you cannot get your projective transformation matrix working, you may implement an orthographic view (no perspective) instead. However, you will not get a mark for objective 7. You may also want to implement an orthographic view first and do your projective transformations last.

11 Line clipping

You will need to clip your lines to the viewing volume. There are several ways to clip, any of which will suffice for this assignment. Note, however, that you *must* clip to the near plane before completing the perspective projection, or you will get odd behaviour and coredumps. You may find it easiest to clip to the remaining sides of the viewing volume after you complete the projection (since you will be clipping to a cube), but you may clip at any point in your program. Note that we will be testing clipping against all sides of the view volume.

12 Donated Code

In `/u/gr/cs488/data/A2`, you will find

- `draw.cpp` – The drawing routines.
- `appwindow.cpp` – A window to which you may add widgets.
- `viewer.cpp` – A widget that will display your rendering. This is where the core part of your code will go.
- `algebra.cpp` – Routines for geometry, lumpy toads, etc.
- `a2.cpp` – Matrix routines you should implement and use.
- `Makefile` – Used to build your code with `make`

These files have been copied to your `handin/A2/src` directory.

13 Deliverables

Executables:

Your source should be in the directory `cs488/handin/A2/src`. Your executable should be in `cs488/handin/A2`.

Additional Documentation: Be sure to note the following in your documentation:

- How you set up the view volume clips, and what you called the function that implements these clips;
- What matrices you chose to store in `viewer.cpp` and their purpose.

14 Objectives: Assignment 2

Due: Wednesday, February 1 [Week 5].

Name: _____

UserID: _____

Student ID: _____

___ **1:** All model transformations are carried out with respect to the box's local origin. (This means, for example, that an x translation will not necessarily be parallel to the world's x axis, if the box has been rotated about its y or z axis.)

___ **2:** Viewing transformations work as expected according to the eye's coordinates. This is indicated by where the world gnomon is displayed.

___ **3:** Model transformations are applied to the box gnomon, except that the box gnomon is carried along **unscaled**.

___ **4:** The transformations in all modes act smoothly **while** the mouse is being moved. Pressing two buttons at the same time results in the two transformations being performed together.

___ **5:** Rotations, translations, and scales can be invoked in any order. Interaction modes may be entered and left as often as desired. There are no restrictions that prevent model transformations from being applied after the view has changed, or view transformations after the box has been transformed. No matter what sequence of transformations is entered, the box never distorts so that its edges fail to meet at right angles (in 3D).

___ **6:** A menubar with pulldown menus is used, with the functionality specified in the assignment description, including a reset for all transformations, the use of radiobuttons, and on screen feedback indicating the current interaction mode, and near and far plane locations.

___ **7:** The perspective transformation has been correctly implemented, and the field-of-view can be changed as specified in the assignment description.

___ **8:** The viewport user interface and the viewport mapping works as specified in the assignment description, and the initial viewport is about centered with 90% maximum size.

___ **9:** Lines are clipped to the near and far planes. The near and far planes can be changed as specified in the assignment description.

___ **10:** Lines are clipped to the sides of the viewing volume.

Declaration:

I have read the statements regarding cheating in the CS488/688course handouts. I affirm with my signature that I have worked out my own solution to this assignment, and the code I am handing in is my own.

Signature: