University of Calgary, CPSC453
Assignment 2
Graphics Pipeline

Release Date: Monday, October 5, 2015
Due Date: Monday, October 26, 2015 at 10:59 am
Weight of this assignment: 10%
Total Marks: 100 Marks (+ 20 Bonus)

Overall Description

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 alinged 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 controller to the amount of each transformation, and the mouse *buttons* will be used to select the axis of the transformation, and a menu will be used as the *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 eye point ("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 (ie. a *model mode rotation* about the x axis will rotate the box around its current x axis, not the world's x axis). The viewing transformations apply with respect to the view coordinates (ie. 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 (ie. the world gnomon never changes its location, though of course it may drift out of our view as a result fo the viewing transformations).

You are to form all the 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 coordiantes. 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."

The Interface

As with the previous assignments, this is implemented in C++ with Qt. 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. Note that **algebra.h, algebra.cpp** contains custom in-house class implementations of Point2D, Point3D, Vector3D, Matrix4x4 and Colour. You are to use (and if needed expand) these classes rather than their Qt implementations, to demonstrate your understanding of the transformation matrices. Additionally, for drawing, you will not be using VAOs, VBOs, or any other OpenGL commands. Instead you will use the **set_colour** and **draw_line** commands for any drawing. Examples of use are found in **Renderer::paintGL**.

We suggest you add a few matrices to the **Renderer** - 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 **Renderer** and perhaps add some more of your own. You will also want to implement the missing matrix functions in **a2.h** and **a2.cpp** and use these in your **Renderer**.

Drawing Lines

In **draw.cpp**, we provide the following C++ routines to draw lines and set colours in an OpenGL window:

- `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(const Point2D& p, const Point2D& q)` - draw a line segment. p and q are in window coordinates.
- `set_colour(const 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 **renderer.cpp** with the GL widget active. There is already example code in **renderer.cpp** that does this for you.

Top Level Interaction

The menubar will support (at least) two menus: the **File** and **Mode** menus. The **File** menu will have to 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 the **Quit** (keyboard shortcut **Q**) which will terminate the program. The **Quit** menu item and shortcut are already implemented in the provided code; be sure not to break it.

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 status bar).

In any View and Model interaction modes, transformations are initiated with the cursor in the 3D viewing area, up on 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 status bar somewhere on the display widget.

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 infintesimal 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 for 3D rotation in Assignment 3.

View Interaction Modes

The following view interaction modes should be supported:

- **Rotate** (keyboard shortcut **O**): Use x-motion of the mouse to:
  - **LMB**: Rotate sight vector about eye's x (horizontal) axis.
  - **MMB**: Rotate sight vector about eye's y (vertical) axis.
  - **RMB**: Rotate sight vector about eye's z (straight) axis.
- **Translate** (keyboard shortcut **N**): Use x-motion of the mouse to:
  - **LMB**: Translate eyepoint along eye's x axis.
  - **MMB**: Translate eyepoint along eye's y axis.
  - **RMB**: Translate eyepoint along eye's z axis.
- **Perspective** (keyboard shortcut **P**): Use x-motion of the mouse to:
  - **LMB**: Change the FOV over the range of 5 to 160 degrees.
  - **MMB**: Translate the near plane along z.
  - **RMB**: Translate the far plane along z.

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

Model Interaction Modes

The following model interaction modes should be supported:

- **Rotate** (keyboard shortcut **R**): Use x-motion of the mouse to:
  - **LMB**: Rotate box about its local x axis.
  - **MMB**: Rotate box about its local y axis.
  - **RMB**: Rotate box about its local z axis.
- **Translate** (keyboard shortcut **T**): Use x-motion of the mouse to:
  - **LMB**: Translate box along its local x axis.
  - **MMB**: Translate box along its local y axis.
  - **RMB**: Translate box along its local z axis.
- **Scale** (keyboard shortcut **S**): Use x-motion of the mouse to:

- **LMB**: Scale box along its local x axis.
- **MMB**: Scale box along its local y axis.
- **RMB**: 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 by 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 butto is pressed at the same itme; all the motion events should be processed simultaneously, as specified above, although individual "incremental" transformations can be compused in a fixed order.

You should use appropriate scaling factors to map the relative mouse motions 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; ie. there should be no restriction on the cumulative amount of rotation or translation applied to an object, or to the number of sequential transformations.

Viewport Mode

The viewport mode allows the user to change the viewport. Assume you have a square window into the world, and that this window is mapped to the (possibly non-square) viewport. The window-to-viewport mapping has been described in lecture and in the optional course textbook; if the aspect ratio of the viewort doesn't match the aspect ratio of the window (ie. the viewport is not square), then the objects appearing in the viewport will be stretched. Further, when you change the viewport, you will see the same objects in the new viewport (possibly scaled and stretched) that you saw in the old viewport.

You should draw the outline of the viewport so 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 on 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 viewort to the visible part of the window.

The initial viewport should fill 90% of the full window size, with a 5% margin arond the top, buttom, left and right edges. This is important so we can verify that your viewoprt 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 larger 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 viewort mode should be **V**.

Projective Transformation

You will need to implement a projective transformation. This will make the cube look three-dimensional, with the perspective foreshortening distinguishing front and back. You may use a projective transformation matrix if you wishs. 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.

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 full marks. You may also want to implement an orthographic view first, and do your projective transformations last.

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.

Donated Code

You have been provided with the following files:

- `main.cpp` - The main point of entry into the application.
- `draw.cpp, draw.h` - The drawing routines.
- `window.cpp, window.h` - The application window, to which you may add widgets.
- `renderer.cpp, renderer.h` - A widget that will display your rendering. This is where the core part of your code will go.
- `algebra.cpp, algebra.h` - Routines for geometry, points, etc.
- `a2.cpp, a2.h` - Matrix routines you should implement and use.

To compile and run the provided program, execute:

```
qmake -project QT+=widgets
qmake
make
./a2
```

The provided code creates the UI for your simple box drawing program. As a test, it draws four lines, cross through the center of the screen, and in the top left corner. You need to modify this code to draw your visible viewport, render a box and the two gnomons, and perform the requested transformations and clipping. A suggested to-do list, in an order that will help you is as follows:

- Draw the outline of the default visible viewport.
- Consider an orthographic projection and define the coordinates necessary for drawing a box. Transform each 3D line segment into your 2D screen coordinates. You may wish to create a helper function that takes an arbitrary line segment, and draws it on the 2D screen. If this goes well, work directly with a perspective projection, otherwise leave it till later.
- Extend your viewport to support resizing, and implement clipping.
- Incorporate model transformations. Then incorporate view transformations. Make sure to leave sufficient time to debug the composition of matrices as you generate your final MVP transformation.
- Wrap up the necessary UI controls and if not already done, perspective projection including near/far/FOV controls.

You may wish to create helper methods, helper classes or use data structures such as queues or stacks, depending on your design. C++ standard template library offers stacks, queues and other collection mechanisms which may be used.

Bonus (20 Marks)

The core of this assignment is fundamentally a decent amount of work. However, if you have time, and the creative inclination, there are a lot of ways this simple graphics pipeline can be made much more interesting. You are encouraged to experiment with the code to implement these sorts of changes, as long as you have already met the assignment's basic objects. The maximum additional marks from bonuses is 20. It is at the discretion of the TA to determine coolness factors in awarding bonus points.

- A fancy way of drawing the viewport. (up to 5 marks)

- Additional geometric shapes; 3 marks per shape. (up to 10 marks)
- Support for OBJ file loading. (5 marks)
- Support and GUI support for adding, transforming and removing multiple 3D shapes on the screen. (10 marks)
- Support face drawing (rasterization), including hidden surface and z-buffer algorithms. (10 marks)

If you make an amazing modification (an actual feature, not an unintended bug...), document it in your `README` and it will be considered and graded at the discretion of the TA for a maximum of 10 marks.

If you make extensive changes, additionally offer a "compatibility mode" by default. You should support at least the user interface required by the assignment. You can activate your extensions either with a special command line argument or a menu item. Document this in your `README` file.

Non-functional Requirements (20 Marks)

Documentation

1. You must provide a **README** file. A sample one has already been provided.
2. Your README file should contain:
   (a) Your name and UCID.
   (b) Short description of algorithms you implemented to complete the program. This includes how you set up the view volume clips and what you called the function that implements these clips.
   (c) A brief description of the data-structures you used to implement the assignment. This includes what matrices you chose to store in the renderer.cpp and their purpose.
3. You must provide at least one screenshot of your assignment demonstrating its capabilities. Additional screenshots are necessary to demonstrate any implemented bonuses. Screenshots should be named 'firstname.lastname_a2_#.png' where # is 1-n for n screenshot images.

Source Code

1. All your source code must be written in **C/C++** and properly commented. All graphics rendering must be done using **OpenGL**. All event handling and windowing must be performed via **Qt**. Your source code must compile on the lab machines in MS 239 without any special modifications. Your source code must be clear and well commented.

2. You will lose marks for inefficient and slow implementations.

3. You may reuse source code:

   (a) which has been provided by the instructor for use in the course,

   (b) which has been written by you which implements basic data structures, such as linked lists or arrays,

   (c) which you have received permission from the instructor or one of the TAs of CPSC 453 prior to handing-in your assignment,

4. Any instances of code reuse by you for this assignment must be explicitly mentioned within the README file. Failure to do so will result in a zero in the assignment. Please read the University of Calgary regulations regarding plagiarism `http://www.ucalgary.ca/honesty/plagiarism`.

Functional Requirements (80 Marks Total)

Transformations (30 Marks)

1. Model transformations performed with respect to the box's local origin. (8)

2. Model's gnomon undergoes all transformations except scaling. (4)

3. Viewing transformations work as specified. (8)

4. Rotation, translation and scale can occur in any order, at any time, with no restrictions on model or view transformation ordering. Regardless of the sequence, the box never distorts so that edges fail to meet at right angles (in 3D). (10)

Viewing and Clipping (30 Marks)

1. Perspective transformation correctly implemented. (8)

2. Viewport mapping works as specified. Defaults to a centered square with 90% maximum size. (8)

3. Lines are clipped to the near and far planes. (7)

4. Lines are clipped to the sides of the viewing volume. (7)

UI Interaction (20 Marks)

The user should be able to:

1. Access pull down menus for controls as specified. (5)
2. On-screen feedback for current mode as specified. (5)
3. Mouse controls for model, view and viewport as specified. (5)
4. Perform transformations smoothly while mouse is moving. Pressing two buttons simultaneously results in two transformations performed together. (5)

Lost Marks

Possible areas for losing marks:

- Lines drawn outside the viewport region.
- Gnomon does not follow box model.
- Gnomon size scales.
- Unable to see box and gnomons on application start up.
- Poor choice of mapping from mouse x values to transformations (eg. cause excessively rapid or excessively slow transformations). Ideally the motion follows the mouse movement.
- Poor cumulation of transformations. eg. As the mouse moves uniformly across the screen, the box translates faster and faster.
- Application redraw slows down over time, as more transformations are applied.
- Draw updating is not live with mouse movements.

Demo

You are required to give an approximately 5 minute live demo to your TA. Failure to show up at the presentation will result in a zero in the assignment. You will need to schedule your demo with your TA - they will have details about how your tutorial section demos will run.