CS4815 Substitute Project: Camera System Programming Assignment

DueDate: 12.00, Fri, ~~Week14 (8th May)~~ Week15 (15th May).

Objective: The objective of this project is to demonstrate understanding of, and competence using, graphical algorithms and techniques as discussed in lectures. Given the specifications of a camera system you should demonstrate that you know how to perform the transformations of the 3-D pipeline by transforming world co-ordinates into normalised device coordinates. You will then open a simple viewport using OpenGL and display a polygon that was placed in the world.

I will run your program on two scenarios (camera system specifications). The assignment counts for 75% of your semester grade so it will be marked out of 75. For reference the marking scheme is given below.

Category	Query	Marks
File submitted		5
Clean compilation		10
What is z_v vector?	1	10
What is y_v vector?	2	10
What are 8 corners of frustum in VC?	3	10
Given p_1 in WC what is it in VC?	4	6
Given p_2 in WC what is it in VC?	5	2
Given p_3 in WC what is it in NDC?	6	6
Draw a given triangle in a given viewport	7	16
Total		75

Table 1: Marking scheme for assignment.

Your job: Following the transformation matrices given in the lectures implement a *perspective* mapping. You should put all of your code in a single file (tut, tut!) called `camsys.cc`. As this is a 3-D camera system you will be working with 4-D vectors and matrices. You will need to be able to compute dot product (a.k.a. inner product) and cross product of vectors, normalize vectors, and multiply matrices of these sizes.

To do these matrix calculations you could either write your own or you could rely on the very cool `eigen` library. (See here for initial details, especially the section "Compiling and running your first program".) As described there, when compiling your program I will add

in an include path to find the eigen files

```
g++ -I /path/to/eigen/ camsys.cc -o camsys
```

so you should follow the template provided there.

If you don't want to use this don't worry about the above.

Input Format: All input to your program will come from standard input, the keyboard. The input will be given entirely as numbers, either as `double`s for distances, triples of `double`s for points/vectors, or `int`s for window dimensions or for the query number. In order to set up the camera system the input you will need will be

1. camera position;
2. camera aim point;
3. the up-vector, a hint as to where the camera's y-axis should point;
4. two numbers that represent, respectively, the distance to the near and far planes of the frustum from the apex of the pyramid;
5. the width and height of the clipping window on the near plane.

The camera system is now fully specified and at this point your program, `camsys`, should read a single `int` that represents a query, respond to that query, and exit. One run of the program, one query. Depending on the query given, you will need to read some more input data. Table 2 below details the exact inputs you will need to read to respond to a given query.

Some description of each query:

1. There is no additional input required for this since it just works off the camera specs previously given. Please give the vector as it appears in *world* coordinates since the vector in VC is obviously $(0, 0, 1)^T$! The vector should be normalized (unit-length);
2. Same applies here;
3. Output the eight corners of the frustum in the order blf, tlf, trf, brf, blr, tlr, trr, brr, where 'b', 't', 'l', 'r', 'f' and 'r' stands for, respectively, bottom, top, left, right, front and rear. That is, output the front (near) plane then the rear (back, far) plane in counter-clockwise order starting with the bottom-left corner of each;
4. After reading the point as three `double`s output its representation in VC;
5. ditto;
6. This time apply the additional transformations to find, for the point you read, its representation in normalized device coordinates. Assume, as OpenGL does that the NDC cube extends from $(-1, -1, -1)$ to $(+1, +1, +1)$;

7. In order to respond to this query you will need to read three points from stdin that make up a triangular face of some object in the world and followed by two `int`s that are the width and height of a viewport in pixels. Create an OpenGL window of those dimensions and, draw the camera's view of the face in this viewport.

Query	Number	Required Inputs
What is z_v vector?	1	-
What is y_v vector?	2	-
What are 8 corners of frustum in VC?	3	-
Given p_1 in WC what is it in VC?	4	3 `double`s
Given p_2 in WC what is it in VC?	5	3 `double`s
Given p_3 in WC what is it in NDC?	6	3 `double`s
Draw a given triangle in a given viewport	7	9 `double`s, 2 `int`s

Table 2: Inputs required per query.

A typical input to your program would be:

```
(6.0,5.0,4.0)
(9.9,8.8,7.7)
(6.0,5.0,4.0)
2.0 6.0
6 8
7
(1.0,0.0,0.0)
(0.0,1.0,0.0)
(0.0,0.0,1.0)
400
300
```

That was for query 7. For query 4, assuming the same camera parameters, it would be

```
(6.0,5.0,4.0)
(9.9,8.8,7.7)
(6.0,5.0,4.0)
2.0 6.0
6 8
4
(9.9,8.8,7.7)
```

Output Format:

Each point or vector should be output as a triple of numbers separated by commas with enclosing parentheses. All numbers output should be in floating point format and should be rounded to 6 decimal places. Output one "thing" per line, with no extraneous text. The expected response to a query that asked for a vector or point would be:

`(-1.234567,6.000000,1.543210)`

Useful Resources: Please email me if there are parts you do not understand. Have a look at the `eigen` library though this is not compulsory. You will have learnt a little Matlab in first year. Octave, its open source equivalent or itself is very handy for verifying your matrix calcs. For example to perform a cross product in `octave` and then to calculate the normalized version of a vector this is how it would look.

```
octave:1> u=[1.1,2.2,3.3]
u =
   1.1000   2.2000   3.3000
octave:2> v=[4.4,5.5,6.6]
v =
   4.4000   5.5000   6.6000
octave:3> cross(u,v)
ans =
  -3.6300   7.2600  -3.6300
octave:4> cross(v,u)
ans =
   3.6300  -7.2600   3.6300
octave:5> U=u/norm(u)
U =
   0.26726   0.53452   0.80178
```

Good luck.