Assignment 2: Camera

Comp175: Introduction to Computer Graphics – Spring 2016

Algorithm due: **Wednesday Feb 24th** at 11:59pm
Project due: **Wednesday March 2nd** at 11:59pm

```
Your Names: _________Holt Spalding_______

            _________Winston Tan_________

Your CS Logins: _____hspald01____________

                _____wtan02______________
```

Instructions

Complete this assignment only with your teammate. You may use a calculator or computer algebra system. All your answers should be given in simplest form. When a numerical answer is required, provide a reduced fraction (i.e. 1/3) or at least three decimal places (i.e. 0.333). Show all work; write your answers on this sheet. This algorithm handout is worth 3% of your final grade for the class.

Axis-Aligned Rotation

Even though you won't have to implement Algebra.h, you will need to know (in principle) how to do these operations. For example, you should know how to create a rotation matrix...

```
Matrix rotX_mat(const double radians)
Matrix rotY_mat(const double radians)
Matrix rotZ_mat(const double radians)
```

Each of these functions returns the rotation matrix about the x, y, or z axis (respectively) by the specified angle.

[1 point] Using a sample data point (e.g., (1, 1, 1)), find out what happens to this point when it is rotated by 45 *degrees* using each of these matrices.

We determined this using a testfile

$$\text{rotX\_mat: } \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(45) & -\sin(45) \\ 0 & \sin(45) & \cos(45) \end{bmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \sqrt{2} \end{pmatrix}$$

$$\text{rotY\_mat: } \begin{bmatrix} \cos(45) & 0 & \sin(45) \\ 0 & 1 & 0 \\ -\sin(45) & 0 & \cos(45) \end{bmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} \sqrt{2} \\ 1 \\ 0 \end{pmatrix}$$

$$\text{rotZ\_mat: } \begin{bmatrix} \cos(45) & -\sin(45) & 0 \\ \sin(45) & \cos(45) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ \sqrt{2} \\ 1 \end{pmatrix}$$

Arbitrary Rotation

Performing a rotation about an arbitrary axis can be seen as a composition of rotations around the bases. If you can rotate all of space around the bases until you align the arbitrary axis with the x axis, then you can treat the rotation about the arbitrary axis as a rotation around x. Once you have transformed space with that rotation around x, then you need to rotate space back so that the arbitrary axis is in its original orientation. We can do this because rotation is a rigid-body transformation.

To rotate a point p, this series of rotations looks like:

$$p' = M_1^{-1} \cdot M_2^{-1} \cdot M_3 \cdot M_2 \cdot M_1 \cdot p$$

where M_1 rotates the arbitrary axis about the y axis, M_2 rotates the arbitrary axis to lie on the x axis, and M_3 rotates the desired amount about the x axis.

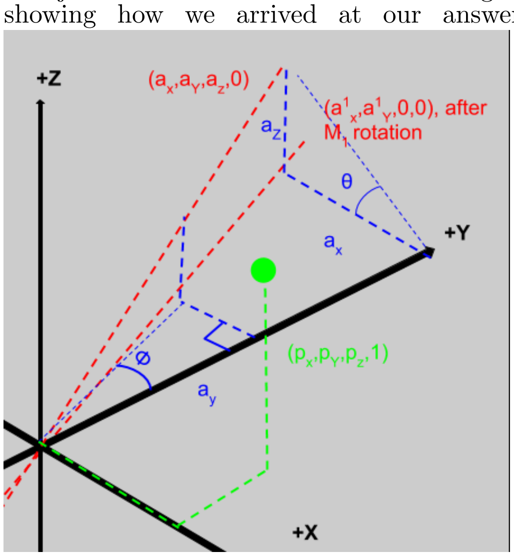
Rotation about the origin

[1.5 points] Assuming we want to rotate $p = (p_x, p_y, p_z, 1)$ about a vector

$a = \langle a_x, a_y, a_z, 0 \rangle$ by λ radians, how would you calculate M_1, M_2, and M_3 in terms of the functions `rotX_mat`, `rotY_mat`, and `rotZ_mat`?

Hint: This is tough! You can (and should) compute angles to use along the way. You can accomplish arbitrary rotation with `arctan` *and/or* `arcos`*, for instance. Approach this problem step by step, it's not extremely math heavy. You should need only one* `sqrt` *call.*

Since the fourth element of a is 0, we can assume it passes through the origin, making our job much easier. Below is a diagram showing how we arrived at our answers:



$M_1 = rotY\_mat(\theta)$ where $\theta = \arctan(\frac{|a_z|}{|a_x|})$

$M_2 = rotZ\_mat(-\phi - \frac{\pi}{2})$ where

$\phi = arccos(\frac{|a_y|}{\sqrt{a_x^2+a_y^2+a_z^2}})$

$M_3 = rotX\_mat(\lambda)$

Rotation about an arbitrary point

[1 point] The equation you just derived rotates a point p about the origin. How can you make this operation rotate about an arbitrary point h?

Translate p and h so that h is in line with the origin, rotate p about the origin, then translate both point again so h is back in it's original location.

Camera Transformation

To transform a point p from world space to screen space we use the *normalizing transformation*. The normalizing transformation is composed of four matrices[1], as shown here:

$$p' = M_1 \cdot M_2 \cdot M_3 \cdot M_4 \cdot p$$

Each M_i corresponds to a step described in lecture. Here, p is a point in world space, and we would like to construct a point p' relative to the camera's coordinate system, so that p' is its resulting position on the screen (with its z coordinate holding the depth buffer information). You can assume that the camera is positioned at $(x, y, z, 1)$, it has look vector $look$ and up vector up, height angle θ_h, width angle θ_w, near plane $near$ and far plane far.

[1/2 pt. each] Briefly write out what each matrix is responsible for doing. Then write what values they have. Make sure to get the order correct (that is, matrix M_4 corresponds to only one of the steps described in lecture.)

M_4 : Translate view volume to the origin

$$\begin{pmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Let $w = \frac{look}{|look|}, u = w \times \frac{up}{|up|}, v = w \times u$

$$M_3 : \text{Rotation} \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Let $w_s = \frac{1}{tan(\frac{\theta_w}{2}) \cdot far}, h_s = \frac{1}{tan(\frac{\theta_h}{2}) \cdot far}$

$$M_2 : \text{Scaling} \begin{pmatrix} w_s & 0 & 0 & 0 \\ 0 & h_s & 0 & 0 \\ 0 & 0 & \frac{1}{far} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

[1] Technically, there are five matrices. The last step is the 2D viewport transform, which is omitted here because OpenGL takes care of this step for us.

Let $c = \frac{-near}{far}$

$$M_1 : \text{Unhinging Transform} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-1}{c+1} & \frac{c}{c+1} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

For the assignment you need to perform rotation operations on the camera in its own virtual *uvw* coordinate system, e.g. spinning the camera about its v-axis. Additionally, you will need to perform translation operations on the camera in world space.

[1/2 pt. each] How (mathematically) will you translate the camera's eye point E:

1. One unit right? $E' = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times E$
2. One unit down? $E' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times E$
3. One unit forward? $E' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times E$
   (Assuming you mean away from the user and closer to the object the camera faces)

You can either move in and out of the camera coordinate space to perform these transformations, or you can do arbitrary rotations in world space. Both answers are acceptable.

[1/2 pt. each] How (mathematically) will you use the u, v, and w vectors, in conjunction with a rotation angle θ, to get new u, v, and w vectors when:

1. Adjusting the "spin" in a clockwise direction by θ radians?
   This would be a rotation about the forward Z-axis. We would perform this rotation in terms of $-\theta$ because we want the rotation to be clockwise. To get our new rotated vectors, we would multiply our original u, v, and w by the rotation matrix: $\begin{bmatrix} \cos(-\theta) & -\sin(-\theta) & 0 & 0 \\ \sin(-\theta) & \cos(-\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
2. Adjusting (rotating) the "pitch" to face upwards by θ radians?
   This would be a rotation about the left X-axis. We would perform this rotation in terms of θ. To get our new rotated vectors, we would multiply our original u, v, and w by the rotation matrix: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
3. Adjusting the "yaw" to face right by θ radians?
   This would be a rotation about the upward Y-axis. We would perform this rotation in terms of $-\theta$. To get our new rotated vectors, we would multiply our original u, v, and w by the rotation matrix: $\begin{bmatrix} \cos(-\theta) & 0 & \sin(-\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(-\theta) & 0 & \cos(-\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Inverting Transformations

[1 point] Computing a full matrix inverse isn't always the most efficient way of inverting a transformation if you know all the components. Write out fully (i.e., write out all the elements) a 4x4 translation matrix M_T and its inverse, M_T^{-1}. Our claim should become obvious to you.

$$M_T = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$M_T^{-1} = \begin{pmatrix} 1 & 0 & 0 & -dx \\ 0 & 1 & 0 & -dy \\ 0 & 0 & 1 & -dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

You can check these two matrices are in fact inverses by showing their product is a 4x4 identity matrix.

Extra credit: Will this same technique work for scale and rotation matrices? If so, write out the inverse scale and inverse rotation matrices. If not, explain why not. What about the perspective "unhinging" transformation? Will this technique be as efficient? Explain.

The same technique works for scale and rotation matrices.
For rotation matrices (the inverse being the transpose):

$$M_R = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$M_R^{-1} = \begin{pmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For scale matrices:

$$M_S = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$M_S^{-1} = \begin{pmatrix} \frac{1}{S_x} & 0 & 0 & 0 \\ 0 & \frac{1}{S_y} & 0 & 0 \\ 0 & 0 & \frac{1}{S_z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The same technique will not be as efficient in the "unhinging" transformation. While it is still possible to transform the points back to perspective plane, given that relative depth is preserved and data required to map back to the original plane has not been lost. However, the transformation is non-affine and is thus harder to compute; as it does not perform a similar operation to any done previously unlike the translation, scaling and rotation matrices. The inverse of the unhinging transform is as follows:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{c+1}{c} & \frac{-1}{c} \end{pmatrix}$$

How to Submit

Hand in a PDF version of your solutions using the following command:

```
provide comp175 a2-alg
```