# CS 3258/5258 Spring 2021 Assignment 4
# 3D Transformations

March 18, 2021

## 1 Purpose

In this assignment you will construct a module to the CLI of the last project to manipulate and view 3D graphical objects. The objects you will create are line drawings in white, on a black background (i.e., no color). The transformations used to manipulate the scenes require you to implement a matrix stack, and translate, rotate, scale, and view an object via both orthographic and perspective projections.

## 2 Due Date

The assignment is due by midnight on Tuesday, April 6, 2021.

## 3 3D Transformations

### 3.1 Command Format

Implement nine additional commands in your CLI, as follows. The format is:

| Command | Parameters | Description |
|---|---|---|
| Push | none | push $C$ on the matrix stack. |
| Pop | none | pop the top of the matrix stack. |
| Translate | $x, y, z$ | translate the origin by the vector $(x, y, z)$ |
| Scale | $s_x, s_y, s_z$ | scale in $x$ by $s_x$, etc. |
| Rotate | $\theta, a_x, a_y, a_z$ | rotate $\theta$ degrees about the axis $(a_x, a_y, a_z)$ |
| Ortho | $l, r, b, t, n, f$ | implement an orthographic projection. |
| Perspective | $f, a, n, f$ | implement a perspective transform. |
| Lookat | $f_x, f_y, f_z, a_x, a_y, a_z, u_x, u_y, u_z$ | Change the point of view. |
| Vertex | $x, y, z$ | Draw a line between two points |
| Reset | Clears the screen and the matrix strack | |

All these commands take floating point arguments.

### 3.2 Implementation Details

You will maintain a $4 \times 4$ homogeneous coordinate transformation matrix called the *current transformation matrix* $C$. Each time you draw a line, it is implicitly transformed by $C$. For example, the transformation of one point to another

is given by

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ w_n \end{bmatrix} = C \begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}$$

Usually, $C$ is the product of rotations, translations, and scalings, with either a perspective projection or orthographic projection (rarely, some other type). If $T$ is a transformation, such as a rotation, it effects the current transformation matrix via

$$C \longleftarrow CT$$

that is, via premultiplications rather than postmultiplication.

You may wonder why the transformation is a pre-multiplication rather than a post-multiplication, since it seems like the post-multiplication is the right thing to do. As we discussed in class, the data structure we are building is a *scene graph* and thus the reason is that a collection of objects and sub-objects, etc., (like a car and its tires) is thought of as a tree-like structure. Rendering this scene is traversing the tree in top-down order. The "more" global a transformation is, the sooner it is encountered, but they must be multiplied into $C$ as they are encountered. The result of this is that the transformations seem to be applied in the reverse order, but in fact they are applied to a point in the reverse order in which they were multiplied into $C$. If you're into linear algebra and such things, you can think of it as the transformations are applied in the same order that they're specified in, but the coordinate system is transforming with the object as each transformation is applied. This explanation may help motivate the push and pop commands below: at each node on the tree, the push and pop allow you to save and restore the current contents of $C$. See section 12.2 of the book for more information.

### 3.2.1 Push,Pop

These two commands push and pop $C$ off the stack. When initialized, the stack should have an identity matrix as its first element. It is an error to attempt to pop this off the stack.

"Push" pushes all matrices in the current stack down one level. The topmost matrix is copied, so its contents are duplicated in both the top and second from the top matrix. If too many matrices are pushed, generate a diagnostic.

"Pop" pops the top matrix off the stack, destroying its contents. What was second-from-the-top matrix becomes the top matrix. If the stack contains a single matrix, generate a diagnostic.

### 3.2.2 Translate

"Translate $x$ $y$ $z$" premultiplies $C$ by an elementary transformation matrix

$$C \longleftarrow C \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

### 3.2.3 Scale

"Scale $s_x$ $s_y$ $s_z$" transforms C via the scaling matrix

$$C \longleftarrow C \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

### 3.2.4 Rotate

"Rotate $\theta$ $a_x$ $a_y$ $a_z$" multiplies the current transformation matrix by a matrix that specifies a rotation of $\theta$ *degrees*, counterclockwise, as one looks from the point $(a_x, a_y, a_z)$ toward the origin. We discussed this formulation in class: it's more complicated than can be easily written down here, but here's what "Rotate 52 1 0 0" would be implemented as

$$C \longleftarrow C \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(52) & -\sin(52) & 0 \\ 0 & \sin(52) & \cos(52) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

noting that 52 is 52 degrees, not 52 radians!!

### 3.2.5 Ortho

This specifies that an *orthographic* or *parallel* projection be performed on the vertices. The direction of projection is assumed to be along the $z$-axis. The six parameters specify the dimensions of a box to which all lines will be clipped: $l$ and $r$ are the minimum and maximum $x$ values that are mapped to the left and right edges of the window; $b$ and $t$ specify the min and max $y$ values that are mapped to the top and bottom of the window; $n$ and $f$ specify the closest and farthest $z$ values that are drawn. The camera faces the negative $z$-axis, so the "near" and "far" values define clipping planes along negative $z$. However, the code assumes $n$ and $f$ are the other way around, so simply negate them in your routines.

### 3.2.6 Perspective

This specifies that a perspective projection be performed. The center of projection is the origin, and as in the parallel case, the viewing direction is along the negative $z$-axis. The parameter $f$ is an angle in degrees that describes the verticle field of view; $a$ describes the aspect ratio, that is, the ratio of the viewing frustum's width to its height. $n$ and $f$ specify locations along the negative $z$-axis at which to perform near and far clipping (as in the ortho case).

When we discussed the perspective projection in class, we said it looking something like

$$C \longleftarrow C \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & q & s \\ 0 & 0 & -t & 0 \end{bmatrix}$$

where where the precise values of those parameters meant something. However, the code I provided you does a $z$-divide already, so this form of the perspective projection isn't used. The code I provide does the z-divide for ou, so the effect is the same, however. Thus the major part of the perspective transform in this assignment is figuring out the windowing transform to take it to the proper coordinates with code doing a $z$-divide.

### 3.2.7 Lookat,Vertex

The lookat command specifies a viewing transform, i.e., locates where and how the camera (or eye) looks at points. I'll supply skeleton code for this, which you may have to make minor modifications to, to fit into your code, but basically it will do all the work for you. The vertex command is going to be our line-drawing command. It will take pairs of points and draw a line between them. I'll supply the code for this as well. *For proper viewing, an ortho or persp command must be executed before any vertex commands are done.*

### 3.2.8 Reset

A reasonably fast way to clear the screen in the reset command is:

```
glClear(GL_COLOR_BUFFER_BIT);
glRasterPos2i(0,0);
glFlush();
```

The reset command should clear the stack and set the current transformation matrix $C$ to the identity.

## 3.3 Provided Code

In addition to skeleton code for the lookat and vertex commands, I supply a clipping routine, `near_far_clip`, which clips the line between $(x_0, y_0, z_0)$ and $(x_1, y_1, z_1)$ to specified "near" and "far" distances along the z-axis. See the code for the prototype. I also provide a `draw_line` routine which draws lines onto the image array, although my code uses both of these and I don't think you'll need to. The provided code is located in the github distribution as `3D.c`.

The code that does the rendering is contained in the supplied code, namely, the `gtLookAt` and `gtVertex3f` routines. Notice that you need to implement a stack of matrices. In the supplied code, that is associated with a global variable "stack." The top of the stack is where the current transformation matrix is. The code does all the necessary line drawing and so on based on what's in this matrix.

The push command copies the top of the stack to the next level down, i.e. stack[top+1] = stack[top]. If you think about things this way, it's also clear that the top of the stack should be the result of a translate, scale, or rotate command. Thus, the top of the stack IS the current transformation matrix. The perspective and ortho commands should copy their results to the perspect and ortho matrices, respectively, but do not affect the stack.

You do NOT modify the global `checkImage` to display in this program, but only to do the extra credit discussed below. Also, you can't reshape or close the window without the screen going black.

## 3.4 Requirements

I will provide some cli scripts which draw things, and tif images of what they are supposed to draw. Get them all right and get an A. The CLI scripts will use the commands above to draw things. They also have comments in them so you can see what's going on (read: if you didn't get comments working yet, do so now). The provided scripts have been uploaded to github. The provided images are located in the images directory of the CS3258 distribution archive (also available on github) in `images/p4_*.tif`.

## 3.5 Implementation Suggestions

Get the stack commands working first, and then implement a 4x4 matrix multiply. Write skeletons for the translate, scale, rotate, etc. commands, and then implement them. The rotate command is the trickiest. Try to get the lookat and vertex code working and tested as soon as it's supplied. When you're debugging your code, you may get a black screen. It will be important to break things down into steps and test each step one at a time to see what's not working. For final testing and so on, you may want to implement a "pause" command that holds the screen still so that you can see what's drawn there.

For this assignment, you may find it useful to get the size of the window you are working in. Here are the OpenGL commands that do that:

```
GLint viewport[4];
glGetIntegerv(GL_VIEWPORT,viewport);
width = abs(viewport[2]-viewport[0]);
height = abs(viewport[3]-viewport[1]);
```

## 3.6 Extra Credit

For extra credit, you can implement the additional feature:

- Implement a command `orient` that takes 9 parameters, $a$ through $i$, and implements a transformation of C given by:

$$C \longleftarrow C \begin{bmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  Experiment with some different values of parameters and structures of matrices to see what types of transformations you can invent. Do they have a reasonable interpretation? *Hint*: examine skewing and shearing transformations from your book. Provide a written description of your findings (10pts).[1]

- Make the drawing that is displayed on the screen tiffwriteable (5pts).

- Implement a command "pause" that pauses for a small amount of time (0.1-0.3s, the exact amount is not really important). Then write a script or set of scripts that generates an animated motion using a geometric object of your construction. For example, implement a cube tumbling down, or a robot arm moving, and those of kinds of things. Provide the script and a README of what it's supposed to look like (10pts).

[1] This "Extra Credit" is mandatory for graduate students taking CS258 as a gateway class.