COMP 3490 Assignment 3
Fall 2024

Due date and submission instructions. Submit **one zip file** containing **all** of the pde files required to run your submission. Name your zip file in the style of `VaughanJenniferA3.zip`, but with your name instead of my name. All submissions should be uploaded to the Assignment 3 folder on UM Learn by 11:59pm on **Monday November 18**.

Getting started. Read these instructions all the way through. Then read the supplied template code. There are several drawing modes that you need to implement, and function stubs that you need to complete.

The purpose of this assignment is for you to implement your own version of the transformation pipeline from object space to viewport space. You may not use any of Processing's built-in projection or transformation commands such as `ortho()`, `rotate()`, `translate()`, `scale()`, `pushMatrix()`, `popMatrix()`, etc. You must write your own versions of all of these functions. You can only use `vertex()` in the one place mentioned in the code (see Question 1).

You will also use hierarchical modeling to draw a scene. For your drawing, you will use Processing's built-in `beginShape()`/`endShape()` functions, together with your `myVertex()` command and the transformation pipeline that you built.

Vectors and Matrices. You will use the `PVector` class (see here) and the `PMatrix2D` and `PMatrix3D` classes (see here for 2D and here for 3D). All of the `PVector` methods are fine to use, though you should not need very many of them. `PMatrix` methods that implement basic arithmetic or allow you to make copies are also fine to use: my solution code uses `apply()`, `get()`, `mult()`, and `reset()`. You may **NOT** use `invert()`.

Questions 1 - 3 take place in 2D, and Question 4 uses 3D. For the first three questions, all of your transformation matrices should be `PMatrix2D` objects, and all of your points should be `PVector` objects. When you get to Question 4, you will need a `PMatrix3D` to represent the 3D perspective projection matrix.

Processing warning! Suppove `v` is a `PVector` and `M` is a `PMatrix`. Suppose you want to calculate `Mv`, and store the result in `v`. The command `M.mult(v, v)` will **NOT** do this correctly! You need to do `v = M.mult(v, null)`.

Coding standards

Your code will be read, and you should write it with that in mind. Use reasonable variable and function names, and avoid magic numbers. Poor style may result in lost marks.

Code will be tested with Processing 4.3. Code that does not compile, that does not run, that runs but produces no image, or that crashes often enough that it is difficult to test will receive a score of 0.

You must complete the function stubs provided in the template. Your code must implement all of the hotkeys set up in the template. You can and should add new functions and global variables as needed. Some global variables are explicitly suggested in the text, but this is not a complete list; you will probably find that you need others. You can add new classes if you wish, particularly for use in drawing your scene.

Questions

1. **Implement the Transformation Pipeline (13 marks)**

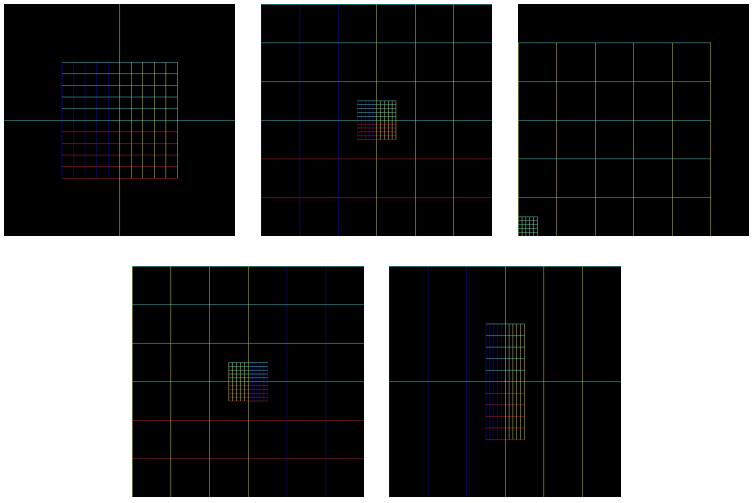
   You will implement the full transformation pipeline, beginning with a vertex coordinate in object space and ending with its corresponding coordinate in viewport space. To mimic OpenGL Immediate mode, you will make the following global, stateful matrices to carry out the different transformations.

   - `Vp` - the viewport matrix. Transform from NDC to the Processing viewport. Remember that the Processing viewport coordinates are set up so that the origin is at the upper left corner of the canvas, the $+x$-direction is to the right, and the $+y$-direction is downward.

     Use the function `getViewport()` to create and return the viewport matrix. Set up `Vp` once at the beginning of your program, and don't touch it thereafter.

   - `Pr` - the projection matrix. Transform from camera coordinates to NDC. Specifically, set up an orthographic projection of the region in camera coordinates that is bounded by the parameters `left`, `right`, `bottom`, and `top`.

     Use the function `getOrtho()` to create and return the projection matrix. You will call this function whenever you need to set up or change the projection. Several projection modes are given in the template. Make sure you implement and test them all.

     Top row: `IDENTITY`, `CENTER600`, `TOPRIGHT600`.

     Bottom row: `FLIPX`, `ASPECT`

   - `V` - the view matrix. Transform from world coordinates to camera coordinates.

     Use the function `getCamera()` to create and return the view matrix. You will call this function any time a camera parameter changes.

     By default, set the camera so that its center is at $(0, 0)$, its up vector is in the direction of the y-axis, and its perp vector is in the direction of the x-axis. The `zoom` parameter will be discussed in Question 2; to begin with, set it to 1.

- `M` - the model matrix. Transform from model coordinates to world coordinates.

  For now, set this matrix to the identity. It will only be modified by the transformation functions that you will write in Question 3.

Each of the functions listed above can be found in the file `Transforms2D.pde`. The return type in each case is `PMatrix2D`. The only way that you should modify one of the global matrices `Vp`, `Pr` or `V` is by assigning it the return value from one of these function calls. Don't write code within `getOrtho()`, `getCamera()` etc. that uses or modifies a global matrix.

Now complete the `myVertex()` function that is set up in `Transforms2D.pde`. Given a vertex `v` in object coordinates, `myVertex(v)` computes

$$(\texttt{Vp})(\texttt{Pr})(\texttt{V})(\texttt{M})\texttt{v}$$

using the global matrices, and plots the result using the Processing `vertex()` command. Use a `PMatrix2D` method for matrix multiplication. Read the API carefully and make sure that you are multiplying on the correct side.

Comments.

- The key to this question is to come up with a reliable debugging strategy. Once you have `myVertex()` implemented, start by initializing all of the transformation matrices to the identity. Draw a test pattern. Do the points appear where you expect them?

  Then incorporate your calculation of `Vp`. This should allow you to give drawing commands in normalized device coordinates.

  Finally, include `Pr`. (`V` and `M` don't do anything yet.) Make sure you check that all of the projection modes work as expected.

- For this and future questions, it will probably be helpful to write functions that construct and return the matrices corresponding to different transformations. For example,

  ```
  PMatrix2D scaleMatrix(float sx, float sy)
  ```

  returns the matrix that scales the x and y directions by the given parameters.

- **Make sure that each function is doing its job, and only its job!** You may find that your problems are easier to solve when you remove extraneous operations from functions like `getCamera()` and `getOrtho()`.

(2 marks for viewport matrix; 2 marks for projection matrix; 2 marks for camera matrix; 2 marks for myVertex and transform pipeline; 5 marks for the test pattern rendering correctly in all 5 ortho modes, 1 mark each)

2. **Advanced Camera Models (14 marks)**

Now you will introduce the ability to change the camera model. The file `DrawingModesA3.pde` contains the hotkeys that you need to implement. Add the appropriate code to `keyPressed()`.

For this question, you should add global variables that track the center of the camera, its up and perp vectors, and the current zoom setting. (Although you can calculate the perp vector from the up vector, you may find it convenient to save both.)

Your `getCamera()` function receives the center, up and perp vectors and the zoom value as parameters, and calculates and returns the view matrix `V`. Implement each of the following effects by changing one or more of these parameters, then calling `getCamera()` to update `V`.

- Zoom in/out. Modify how much the camera scales the image. Zooming in should make the image appear larger, and zooming out should make it appear smaller.

  When you change the zoom, the effect should be centered on the origin in *camera* coordinates. (This choice is intended to make your task easier, not harder.)

  Make the changes to the zoom value multiplicative, rather than additive. That is, when you zoom in or out, multiply or divide the global zoom variable by a fixed constant, rather than adding or subtracting.

- Pan the camera by clicking and dragging the mouse.

  Complete the `mouseDragged()` function so that it changes the camera's center location. Note that the variables `mouseX`, `mouseY`, `pMouseX` and `pmouseY` are all measured in viewport coordinates, whereas the center of the camera is measured in world coordinates. You will need to do some coordinate system conversions, but do *not* use the `PMatrix2D invert()` method. This calculation can be done using proportions.

  Make sure that panning and zooming work correctly together, and that panning works correctly in all of the projection modes. When you click and drag, the scene should move at just the right speed to keep up with the mouse movements, and this should remain true even when you zoom in or out. Your conversion from viewport to world measurements will depend on the projection mode and the current zoom value.

- Rotate the camera. Add another global variable to track the angle of rotation, and use it to modify the camera's up and perp vectors. The key `KEY_ROTATE_CW` should rotate the **camera** clockwise (left-handed), which means that the **scene** rotates counterclockwise (right-handed).

  Now make rotating and panning work correctly together. Once you rotate the camera, the y direction in viewport coordinates no longer corresponds to the y direction in world coordinates. What direction should it be instead? (Hint: you should already have a vector that points in this direction!)

(3 marks for zoom; 5 marks for panning; 3 marks for rotation; 3 mark for panning and rotation working correctly together)

3. **Hierarchical Modeling (13 marks)**

Implement a global stack for the model matrix. Complete the following functions, which can all be found in `Transforms2D.pde`.

- `void myPush()` and `void myPop()` back up and restore the model matrix. Careful of deep versus shallow copy.

- Each of the functions below creates the appropriate transformation matrix, and post-multiplies it into the global model matrix `M`. That is, if `M` is the current model matrix and `T` is the matrix representing the transformation, then the effect of calling the function is to replace `M` by `MT`.
  - `void myRotate(float theta)`
  - `void myScale(float sx, float sy)`
  - `void myTranslate(float tx, float ty)`

  These functions do modify a global matrix, unlike the functions you wrote in Question 1.

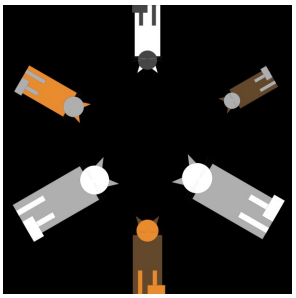
Now use the matrix stack and your transformation functions to create a complex scene. You are allowed to use Processing's built-in functions `fill()`, `stroke()`, and `strokeWeight()` for setting the colors of objects and lines. Your scene must have the following properties.

- Each draw function creates visual objects roughly within a 2×2 object space, with coordinates ranging from $(-1, -1)$ to $(1, 1)$.

  You can pass properties such as ratios or proportions, colors, number of subcomponents, etc. to your draw functions, but do not pass scales, angles, or positions as parameters. Instead, the code that calls a particular draw function should use transforms to resize, rotate and position it. Use `myPush()` and `myPop()` to support hierarchical modeling.

- Your scene needs to be at least three nested calls deep: one draw function applies one or more transformations, then calls a second draw functions, which applies further transformations and calls a third (or more).

- Your end result must include some complex, multi-stage object that is drawn at different positions, angles and scales. For example, here is how I met that standard with my cat picture.

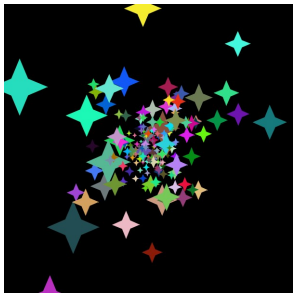


Be careful of accidentally using Processing's `scale()`, `translate()`, `rotate()`, `pushMatrix()` or `popMatrix()` functions. They will not work at all with the coordinate systems you are using.

(5 marks for matrix stack and transformation functions; 8 marks for scene, including sufficient hierarchy levels, all transformations demonstrated, and all camera controls working correctly)

4. **3D Mode – Flying Shapes (8 marks)**

Now you will build a 3D image in which a field of shapes is flying past you in space. A key point: you will be using a **perspective** projection now, not an orthographic one.



I created four-pointed stars. You can choose any shape you like. Each shape will be drawn as follows.

```
beginShape();

// a sequence of myVertex() calls

endShape();
```

In this drawing mode, reset `M`, `V` and `Pr` to the identity matrix. Use the same `Vp` matrix as in the rest of the assignment. The net result is that the transformation pipeline that you already implemented expects to receive coordinates in NDC. ?????

Consider a 3D view frustum in camera coordinates that is bounded by the parameters `left`, `right`, `bottom`, `top`, `near` and `far`. In the file `FlyingShapes.pde`, complete the function `getViewFrustum()` so that it creates and returns the 4×4 homogeneous matrix that maps this volume into 3D NDC coordinates. Use the conventions described in the Unit 2.8 notes: `left`, `right`, `bottom` and `top` are *coordinates* in 3D camera space, but `near` and `far` are *distances*. The view frustum itself is located in $-z$ space.

You will also create functions to perform the following tasks.

- Randomly generate a collection of your chosen shapes in 3D space. Each shape should have its own randomly generated position and size. All of the x and y coordinates of the vertices should be between -300 and 300, and the z coordinates should be between -500 and 0. Generate all of these random values once at the start of the program.

- In each frame, move each of the shapes by modifying the z coordinate of each of its vertices.

  Your flying shapes can fly either forward or backward. They can have all the same speed, or they can have varying speeds. The colors can be uniform or randomized. Experiment to find an effect that you like.

  If a shape crosses either of the boundaries $z = 0$ or $z = -500$, make it loop around to the opposite z boundary.

- In each frame, draw the shapes by projecting each vertex into 3D NDC using the view frustum matrix that you created earlier. Remember, we are doing perspective projection now, which means that the homogeneous w coordinate is playing a nontrivial role! Divide out by w to convert to standard NDC coordinates.

  Coordinate system check: once the above calculation is complete, the z coordinate has served its purpose, and we don't need it anymore. The (x, y) coordinates have been converted to NDC, and we can feed these coordinates into our 2D pipeline by calling `myVertex()`.

Choose reasonable boundaries for your view frustum. Setting `(left, right)` and `(bottom, top)` to $(-300, 300)$ is probably a good choice. You can set `far` to 500, but remember that `near` should *not* be 0.

An annoyance: when a `PVector` is multiplied by a `PMatrix3D`, Processing will throw out the w component unless you force it not to. To get around this, store your initial vertex coordinates in a multi-dimensional array, so that the homogeneous coordinates of each vertex take the form of an array of length 4. Then, when you multiply your frustum view matrix by this vector, tell Processing to store the result in another array of length 4. Here is a simplified example:

```
float[] vertex = new float[4];

// store homogeneous coordinates in vertex

float[] projectedVertex = new float[4]; // vector to hold the answer
frustumView.mult(vertex, projectedVertex);
```

After the above code, `projectedVertex` will contain the result of the matrix multiplication, including the w coordinate that you need to complete the conversion to NDC.

(4 marks for view frustum transform; 4 marks for creating, projecting and drawing the flying shapes)