COSC342: Lab 03

2D Transforms in C++

March 20th 2017

1 Introduction

In the last lab you saw how to use a basic Matrix library in C++and worked with OpenCV. In this lab you'll use these libraries to apply basic transforms in 2D. Online documentation for the code provided for this lab is available at `http://www.cs.otago.ac.nz/cosc342/docs/transforms2d/index.html`.

The lab uses OpenCV (Computer vision library) to draw and display points and lines. This library is not yet available on the lab machines under Linux, so you should work under OS X. You can get the starting code for this lab from `/home/cshome/coursework/342/pickup/labs/lab03-Transforms2D`

Use the command line and go to the directory where you are storing your COSC342 work and get the files for this lab:

```
cp -r /home/cshome/coursework/342/pickup/labs/lab03-Transforms2D .
```

The `-r` option copies files recursively, and the dot, `.`, at the end means copy to here.

2 Building Projects with CMake

Similar to the last lab, we will use CMake to prepare our development environment. Again there are two options supported in the lab. The two main options are to compile code with Makefiles from the command line or to use XCode.

In your `lab03-Transforms2D` directory there is again a file called CMakeLists.txt which describes the project that needs to be made. Lets take a look inside that:

```
project( transform2D )
...
add_executable( transform2D
transformMain.cpp
transforms.cpp
transforms.h
utility.h
```

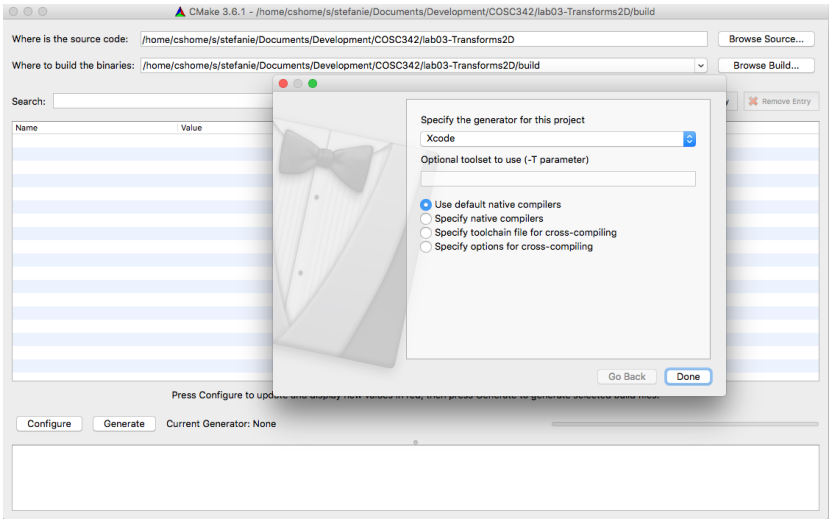
```
Vector.h
Vector.cpp
Colour.h
Display.h
Display.cpp
Matrix.h
Matrix.cpp )
```

This is a very simple CMake file, which declares that we're making a project called `transform2D`. This is also the name of the executable program we'll be building, and that executable depends on the listed `.cpp` and `.h` files.

To set up the project, run the program called `CMake`, which is in the Applications directory. If you can't find it press `command-space` and type "CMake" to search for it. CMake asks you two simple questions – "Where is the source code" and "Where to build the binaries". The source code is where you copied it to, so browse to that directory, something like
`/home/cshome/a/astudent/Documents/cosc342/lab03-Transforms2D`

Where to build the program is up to you, but the conventional thing to do is to add `/build` to the source code location, so something like
`/home/cshome/a/astudent/Documents/cosc342/lab03-Transforms2D/build`

Press the *Configure* button. You will probably get a prompt to create the build directory, click *Yes*.

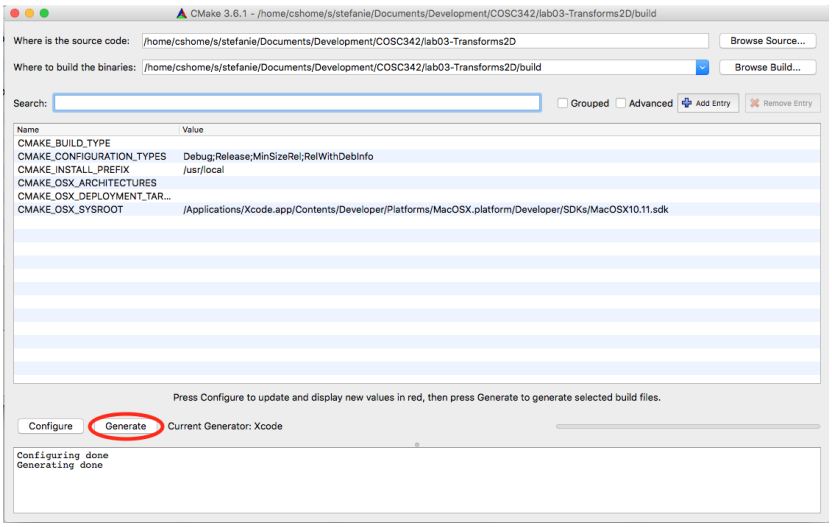


You'll next be prompted to choose a toolchain to build. You can use whatever is installed on the machine you are using, but for the labs we'll assume you're using either *Unix Makefiles* or *XCode*. Choose your preferred environment from the drop-down, leave the other selection as *Use default native compilers*, and press *Done*.

CMake looks for the relevant tools, and checks that they are available, then

updates its settings. You'll see a number of new settings added, which are highlighted in red. You can edit these if you need to, but you don't. So don't.

Finally hit the button Generate to generate your project files (either XCode or Makefile).



2.1 Building with Makefiles

If you are using Unix Makefiles, you'll need to go back to the terminal. Assuming you're in the `cosc342` directory you made earlier, you'd go to the build directory and compile the program with

```
cd build
make
```

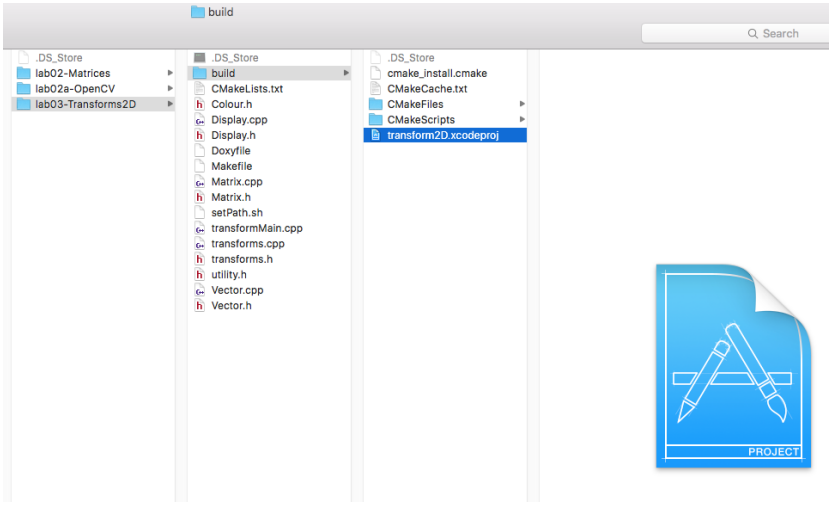
Then run the executable you just built:

```
./transform2D
```

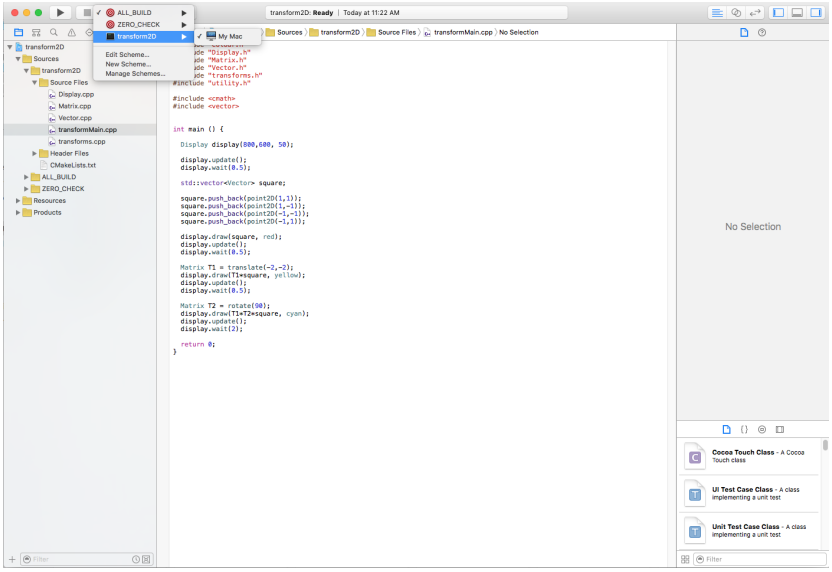
You should see a window appear with axes and a grid. Several coloured squares will be drawn, and then the window will close after a few seconds.

2.2 Building with XCode

Open the XCode project file from your build directory.



XCode will open and you can select the source files in the source file explorer on the left hand side. To run the application, select the **transform2D** scheme and press the run button.



You should see a window appear with axes and a grid. Several coloured squares will be drawn, and then the window will close after a few seconds.

2.3 Adding an Application to CMake file

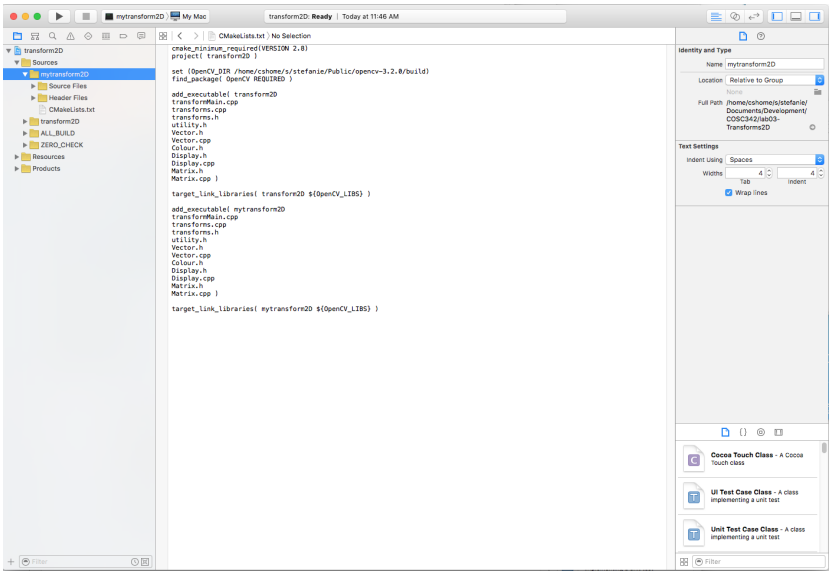
If you want to keep the existing project and create your own application, you need to add a new executable to the CMakeLists.txt file. In the following example we add an application called mytransform2D.

```
...
add_executable( mytransform2D
mytransformMain.cpp
transforms.cpp
transforms.h
utility.h
Vector.h
Vector.cpp
Colour.h
Display.h
Display.cpp
Matrix.h
Matrix.cpp )

target_link_libraries( mytransform2D ${OpenCV_LIBS} )
```

You will also need to create the file `mytransformMain.cpp` – a copy of `transformMain.cpp` is a good start for that.

After changing the CMakeLists.txt file you need to go back to CMake and press Configure again and afterwards press Generate again. Then go back and rebuild your project either with Make or XCode (depending on your selected configuration). In XCode you should see an extra scheme called mytransform2D and see the added files under sources.



3 Displaying Points and Lines

Lets have a look at `transformsMain.cpp`. The `main` routine begins by creating a `Display` object. You can have a look inside the code for this object if you're interested, but for the purposes of this lab it is just a convenient way to draw points and lines. A `Display` object is created by a constructor which takes 3 arguments:

- The width of the window in pixels
- The height of the window in pixels
- The size of each unit along an axis in pixels

So the code

```
Display display(800, 600, 50);
```

Creates an 800×600 window where every 50 pixels corresponds to one unit. Since the origin is in the middle of the window, the visible part of the X-axis runs from $-(800/2)/50 = -8$ to $(800/2)/50 = +8$. Similarly, the Y-axis runs from -6 to $+6$.

There are three main things you can do with a `Display` object. Firstly you can draw points, lines, and polygons with `draw()` methods. To draw a point, pass its co-ordinates (in homogeneous form as a `Vector` with 3 elements) and a `Colour`:

```
Vector point(3);
point(0) = 1;
point(1) = 2;
point(3) = 1;
display.draw(point, red);
```

Since defining Vectors by specifying their co-ordinates separately gets tedious, the `utility.h` header provides a function, `point2D`, to create a homogeneous `Vector` given x and y values:

```
Vector point = point2D(1,2);  // [1,2,1]^T
```

There are pre-defined `Colours` `red`, `green`, `blue`, `cyan`, `magneta`, `yellow`, and `white`, or you can create your own, such as

```
Colour purple(128, 0, 196); // red, green, blue values
```

Lines are drawn by passing two points and a colour:

```
display.draw(point1, point2, green);
```

Polygons are drawn by passing a `std::vector` of points and a colour, as illustrated in the sample code.

Drawing to the `Display` updates an internal buffer, so to view your changes you need to call `display.update()`. Finally, since the window will update quickly and close when your program ends, `display.wait()` will cause the program to pause for a time, given in seconds. If not time is given, the program pauses for 1 second. Pressing any key while paused will continue.

4 Working with Transforms

The rest of the code creates a square as a set of four points, draws it, then draws some transformed versions. The transforms are 3×3 matrices, and the file `transforms.h` defines some functions to make the basic transformation matrices discussed in lectures. It also provides a multiplication operator that allows you to apply a transformation matrix to a `std::vector` of `Vector`s as a single operation.

Since transformations are matrices, the order of operation is important. This is shown in the example code where the a translation by $(2, 1)$ and a scaling by $s = 0.5$ are applied. The transform `T1` applies the scaling first, while `T2` translates then scales.

5 Exercises

1. Change `transformMain.cpp` so that the following items are drawn:
   - A green point at $(-3, 2)$.
   - A line from $(-1, 2)$ to $(3, -1)$ drawn in orange (red = 255, green = 128, blue = 0).
   - A white triangle with corners $(2, 2)$, $(0, 0)$, and $(1, -1)$.
   - A sequence of lines running vertically between $(x, -1)$ and $(x, 1)$ for $x = -7, -6, \ldots, 6, 7$. The line's colour should depend on x and be (red $= 128 + 15x$, green $= 128 - 15x$, blue $= 0$).
2. Don't write the code yet, but we're going to transform the triangle from the previous exercise so that it is scaled by a factor of 2 and then translated left by 3 units.
   - Where do you expect its corners to end up after the scaling?
   - Write the code to apply the scaling, and check that it matches your expectations.
   - Where to you expect the triangle's corners to be after both the scaling and translation have been applied?
   - Again, apply this transformation in code and see if you agree with the result.
3. The rotation transform is not currently implemented, and `rotate(angle)` just returns the 3×3 identity matrix.
   - Update `transforms.cpp` so that the correct rotation matrix is returned. Note that `rotate()` expects an angle in *degrees*, but the C++ trigonometric functions expect angles in *radians*. The file `utility.h` provides functions to convert between degrees and radians.

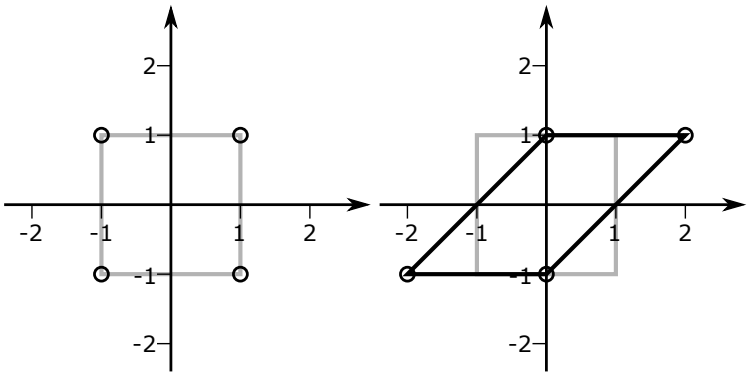
- Add a rotation of 45° to the triangle, after it has been scaled and shifted.
- Update your sequence of lines so that the line between $(x, -1)$ and $(x, 1)$ is rotated by $10x$ degrees about the point $(x, 0)$.

4. Shear is another basic transform and can be described as horizontal shear or vertical shear. A horizontal shear can be implemented with the following transformation matrix:

$$\mathrm{S}_x = \begin{bmatrix} 1 & m & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The equivalent transform for a shear in the vertical direction is

$$\mathrm{S}_y = \begin{bmatrix} 1 & 0 & 0 \\ n & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$



- Add two new transform functions, `shearX(m)` and `shearY(n)`, which implement the shear transformation described above. You will need to update `transforms.h` and `transforms.cpp`.
- Draw a square centred at the origin with corners at $(2, 2)$, $(-2, 2)$, $(-2, -2)$, and $(2, -2)$.
- Draw the results of shearing the square horizontally by 1 and vertically by 0.5.

5. It can be shown that rotation can be implemented as a sequence of shearing operations. The details are discussed at `https://www.ocf.berkeley.edu/~fricke/projects/israel/paeth/rotation_by_shearing.html`, but the end result is that `rotate(a)` can be implemented as

```
R = shearX(-tan(a/2))*shearY(sin(a))*shearX(-tan(a/2));
```

- Implement a rotation of the square from part (4) by 30° using your shearing operations.
- Check that the results of your sequence of shears are the same as a direct rotation.