Manipulation of polygons in two dimensional space

Joseph Kellett

83458200

School of Physics and Astronomy

The University of Manchester

Computing report

May 2015

Abstract

An abstract base class was used to create a common interface for the derived classes that store vertices of different example polygons. Additionally these derived classes contained overridden functions that were specific to each polygon while maintaining the common interface of the base class. A vector of base class pointers was used to store a list of user inputted polygons, which then allowed them to be printed to the console output, further edited by the user and then saved in a text file, or a MATLAB script for plotting.

1 Introduction

The project specification for this programme required that it allows users to create and store polygons in 2D space. The user had to be able to then manipulate the polygons by rotating, translating and scaling them. Finally the programme had to output the details of the polygons in an appropriate format.

2 Code design and implementation

2.1. File structure

The code was split into five different files: 'polygonsMain.cpp', 'polygonsClasses.h', 'polygonsClasses.cpp', 'functionsAndSubroutines.h' and 'functionsAndSubroutines.h'.

The reasons for splitting the code were two fold. Firstly this reduced compiler time as only the file that had been edited needed to be recompiled, making code editing and debugging much faster.

Secondly the physical separation of declaration and implementation makes reading the source code easier for outside parties. They need only look in the 'polygonsMain.cpp' and header files to see what the programme does, and then can check the other source files if they wish to know how the solution was implemented.

As 'polygonClasses.h' had to be included in 'polygonsMain.cpp' and 'functionsAndSubroutines.h' header guards were required to prevent redefinition.

2.2. Class hierarchy

An abstract base class was used to produce a common interface for the different polygons. Pure virtual functions were used to make the base class abstract and enforce that these functions were overloaded. These functions included a default constructor, a destructor, a function to rotate the polygon by an angle in radians about a point on the *x*-*y* plane, a function to scale the shape, and a function to translate the shape.

A parametrised constructor was not implemented in the base class as each of the derived polygons took different input parameters for their constructors. Likewise the vertices of each polygon were stored as arrays of doubles in the derived classes while the position of the centre of the polygon (common to all polygons) was stored in the base class.

2.3. Implementation of class functions

The functions of each derived class are largely the same and so only one example for each overloaded function will be discussed. The exception to this is the parametrised constructors, for which the general method differs greatly for the hexagon and pentagon classes compared with the rectangle and pentagon classes.

It should be noted that where possible the variables are passed by reference instead of value to minimise system resources used by the programme without compromising functionality.

2.3.1 Rotating and translating polygons

This programme has the ability to rotate a polygon about any point in 2D space by a set angle. In practice the user is only given the option to rotate about the centre of the polygon, however, the general case was maintained to allow future edits to add more functionality to the user.

The functions to rotate each polygon simply call another rotate function for each vertex of the shape and the centre point (in case of rotation about a point that is not the centre).

```
void rotatePoint(double point[2], const double &angle, const double(&about)[2]){
	// Translate the point to the origin so we can then rotate about the origin.
	point[0] = point[0] - about[0]; point[1] = point[1] - about[1];

	// Rotate new point about the origin using explicit form of matrix multiplication:
	// (x') - (cos(theta)  -sin(theta) ) (x)
	// (y') - (sin(theta)   cos(theta) ) (y)
	double rx = point[0] * cos(angle) - point[1] * sin(angle);
	double ry = point[0] * sin(angle) + point[1] * cos(angle);

	// Move the point back by the same amount.
	point[0] = rx + about[0];
	point[1] = ry + about[1];
}
```

Code 1. The rotatePoint() function, called by the rotateShape() functions for each derived class.

As can be seen by Code 1. the rotate function implements an explicit matrix multiplication for each point after moving the point so that it is being rotated about the origin. The point is then moved back by the same distance.

The polygons were translated using a similar method as that for rotating the polygons. A translate function is called for each point and using user inputted arguments the *x* and *y* coordinates of each point are adjusted.

2.3.2 Scaling polygons

The polygons were scaled by finding the displacement between each vertex and the centre of the polygon and multiplying these values by the inputted scale factor. The new vertices were found by applying the newly scaled displacement to the centre of the polygon.

```
void isosolesTriangle::scale(const double &scaleFactor){
	//Find the displacement between vertices and the centre of the polygon and then
	scale these vales
	double deltaA[2] = { (A[0] - polygonCentre[0])*scaleFactor, (A[1] -
	polygonCentre[1])*scaleFactor };

	double deltaB[2] = { (B[0] - polygonCentre[0])*scaleFactor, (B[1] -
	polygonCentre[1])*scaleFactor };

	double deltaC[2] = { (C[0] - polygonCentre[0])*scaleFactor, (C[1] -
	polygonCentre[1])*scaleFactor };

	//Find new vertices by adding the displacements to the position of the polygon
	centre.
	A[0] = deltaA[0] + polygonCentre[0]; A[1] = deltaA[1] + polygonCentre[1];
	B[0] = deltaB[0] + polygonCentre[0]; B[1] = deltaB[1] + polygonCentre[1];
	C[0] = deltaC[0] + polygonCentre[0]; C[1] = deltaC[1] + polygonCentre[1];
}
```

Code 2. The scale function for the isosceles triangle class. The vertices A, B and C are scaled about the centre.

2.3.3 Outputting polygon vertices

Two functions were created to output the polygon vertices. Each outputs them in a slightly different format as they are used for different purposes.

The first, *info()*, outputs the type of polygon to the console and then the vertices in bracket form, (x-coordinate, y-coordinate). A typical output for this function is as follows:

```
Isosceles triangle with vertices at
(0.2518, 4.765), (3.834, 1.276), (4.914, 2.959)
```

The precision of the vertices is limited to four to improve presentability and ensure that the higher order polygons do not take up an inordinate amount of space in the console output.

The second function, *stringVertices()*, creates an output string consisting of the vertices of the polygon. This is ideal for writing the details of the shapes to files, however, it is not as useful to the user if outputted to the console as it lacks a description of the shape and is not formatted to accommodate the limited width of the console.

2.3.4 Class constructors

To construct the rectangle and isosceles triangle the size of their base, the length of the other set of sides, the angle of orientation and the position of the centre of the polygon were required. In the case of the regular hexagon and pentagon only the size of one side was needed as these shapes have no easily defined 'base'.

The hexagon and pentagon were constructed by first defining a polygon of unit radius, and then adjusting the polygon to the user's specification using the rotation, scaling and translation functions described above.

```
pentagon::pentagon(double const &side, double const &angle, double const (&posOfCentre)[2])
{
	//Make a pentagon of circumradius 1:
	A[0] = 0, A[1] = 1;
	B[0] = sin(2 * pi / 5); B[1] = cos(2 * pi / 5);
	C[0] = sin(4 * pi / 5); C[1] = -cos(pi / 5);
	D[0] = -sin(4 * pi / 5); D[1] = -cos(pi / 5);
	E[0] = -sin(2 * pi / 5); E[1] = cos(2 * pi / 5);
	polygonCentre[0] = polygonCentre[1] = 0;

	//Resize and reposition pentagon according to user specifications.
	double s = 2 * sin(pi / 5); //Length of side of this unit pentagon
	scale(side / s); //Scale shape up to user specification
	rotateShape(angle, polygonCentre);
	translateShape(posOfCentre[0], posOfCentre[1]);
}
```

Code 3. The parametrised constructor of the pentagon class.

Conversely, as both the rectangle and isosceles triangle are not regular shapes different methods were used to construct them. In both cases basic geometry was used to construct the triangle or rectangle to the size and shape of the user specifications, and then the respective rotate and translate functions were called to position the shape as desired in 2D space.

2.4. Implementation of functions outside classes

Many of the functions are only called once, have no return type (void) and few input arguments; these functions may be considered subroutines. The code was implemented with this structure for two reasons: Firstly it keeps the main function uncluttered so as to improve readability. Secondly, it makes keeping track of variables easier, as variables declared within the subroutines are destroyed as the subroutine goes out of scope and variable names can be reused without defining another namespace, improving the readability of the code.

2.4.1 Obtaining and verifying user input

The functions *giveYN()* and *giveNumber()* were created to verify that the user had correctly answered a yes/no question and entered a number respectively.

```
double giveNumber(bool const &checkNegative){
    double dNumber;
    bool bInputCorrect(false);
    do{
        cin.ignore(numeric_limits<streamsize>::max(), '\n'); //Ignore the rest of the
string stream, ie delete it.
        cin.clear(); //Resets any fail flags
        cin >> dNumber;
        if (cin.fail()){
            cout << "Input incorrect, please enter correctly." << endl;
            cin.ignore(1000, '\n');
            cin.clear();
        }
        //Stops user inputting negative value if checkNegative is true.
        else if (checkNegative == true && dNumber < 0){ cout << "Please enter a
positive value." << endl; }

        else{ bInputCorrect = true; }

    } while (bInputCorrect == false);

    return dNumber;
}
```

Code 4. The giveNumber() function.

Code 4. demonstrates the basic verification implemented. The *cin.fail()* function automatically checks for many mistakes such as entering a letter instead of a number, while if the boolean *checkNegative* is true the function will stop the user entering a negative value.

If the user input is invalid these functions will loop until the user input is acceptable, while outputting messages describing what they did wrong if possible.

2.4.2 Creating and storing user entered polygons

The function *userInputPolygons()* was used to create multiple polygons as determined by the user.

```
vector<polygon*> userInputPolygons(){
do{
                          Ask for user char cPolygonChoice
                                       ...

    //Switch block performs different things for different cases. Equivalent to many if's
and else if's.
switch (cPolygonChoice){
case 't':
    cout << "Please enter the size of the base (unique side) of the isosoles triangle: ";
    base = giveNumber(true);

    cout << "Please enter the size of the other side(s) of the triangle: ";
    side = giveNumber(true);

    cout << "Please enter the angle in radians you wish to orientate the triangle by: ";
    angle = giveNumber(false);

    cout << "Please enter where you want the the centre of the triangle to be: x= ";
    posOfCentre[0] = giveNumber(false);
    cout << "y= ";
    posOfCentre[1] = giveNumber(false);

//Create the triangle in storage vector using the values the user entered.
    storedPolygons.push_back(new isosolesTriangle(base, side, angle, posOfCentre));

     break;

                             Repeat for other polygons
                                       ...

     cout << "Do you wish to add another polygon? y/n " << endl;
} while (giveYN()==true);

    return storedPolygons;
}
```

Code 5. The function used to create a vector of base class pointers, which point to objects of the derived polygon classes. Portions of the code have been removed to improve readability.

The user is asked to enter what type of polygon they wish to create by entering a character, *t* for an isosceles triangle, *r* for a rectangle, *p* for a pentagon and *h* for a hexagon. A switch block is then used to output information specific to what polygon the user inputted, and to request the required information to construct the polygon.

As the variables *angle*, *side*, *posOfCentre* and *base* are used (and likely reused) for the construction of most of the polygons, they are declared outside of the switch block. Using the variables entered by the user, the relevant parametrised constructor is then called.

2.4.3 Printing and editing the stored polygons

Using the base class vector of user entered polygons created by *userInputPolygons()* the subroutines *printAll()* and *userEditPolygons()* can be called with the vector as an input argument.

```
vector<polygon*> userEditPolygons(vector<polygon*> &storedPolygons){
       double dMove[2], dScaleFactor, angle;

       for (vector<polygon*>::iterator iter = storedPolygons.begin(); iter <
storedPolygons.end(); ++iter){
              cout << "Do you wish to edit this polygon?  y/n  " << endl;
              (*iter)->info();

              if (giveYN() == true){
                     cout << "Enter the angle (radians) that you wish to rotate the polygon
by: ";
                     angle = giveNumber(false);

                     cout << "Enter how much you wish to translate the shape by in the x
direction: ";
                     dMove[0] = giveNumber(false);
                     cout << "Enter how much you wish to translate the shape by in the y
direction: ";
                     dMove[1] = giveNumber(false);

                     cout << "Enter the factor you wish to scale the shape by: ";
                     dScaleFactor = giveNumber(true);
              }
       }
       return storedPolygons;
}
```

Code 6. The subroutine called if the user wishes to edit any of the created polygons.

The vector is iterated using iterators through until the user chooses a polygon to edit. The subroutine then asks for variables and calls the relevant functions (described in section 2.3).

The *printAll()* and *writeAll()* subroutines are similar, yet simpler and use constant iterators, as their input argument is a constant vector (guaranteeing that the subroutine cannot change the vector). The *printAll()* subroutine uses the *info()* class function and outputs to the console while *writeAll()* uses the *stringVertices()* function and writes to a text file in the programme's directory.

2.4.4 Writing the polygons to a MATLAB plotting script

This programme has the functionality to produce a MATLAB script. As MATLAB script file type (*.m*) is essentially a text file the normal C++ insertion operators could be used to write the script, in combination with the *stringVertices()* function described in section 2.3.3. The file is saved in the default directory, the same folder as the source files or executable file, depending on whether the user has compiled the executable or has simply received it from somewhere else.

```
void plotMatlab(const vector<polygon*>  &storedPolygons){
	ofstream myfile;
	myfile.open("polygonsplot.m"); //Open file for writing inside the polygons solution 
folder.

	//Write a matlab script by directly inputting strings to the file.
	myfile << "%This is an automatically generated script by polygons.exe\nhold on \n";

	for (vector<polygon*>::const_iterator iter = storedPolygons.begin(); iter < 
storedPolygons.end(); ++iter){
		myfile << "polygon = " << (*iter)->stringVertices() << "; \n";
		myfile << "patch(polygon(:, 1), polygon(:, 2), 'r', 'FaceColor',[rand() 
rand() rand()]); \n";
	}

	myfile << "axis equal\n";//Make x and y axis have same scale so regular polygons 
look regular!

	myfile.close();

#if defined (_WIN32)
	ShellExecute(0, 0, L"polygonsplot.m", 0, 0, SW_SHOW); //opens the file with the 
default programme defined by windows.
#else
	cout<<"A matlab .m file has been saved to this directory"<<endl;
#endif

	cout << "Matlab script created. You will have to run it yourself though!" << endl;
}
}
```

Code 7. The subroutine used to create a MATLAB script to plot the polygons

After the script is created the subroutine will open the script in MATLAB if the programme is running on a Windows based computer. This functionality is implemented using preprocessor macros and if-statements. A similar preprocessor if-statement is used to determine whether to include the 'Windows.h' header file, which is only available on Windows based computers, and is required to use the *ShellExecute* function.

The *ShellExecute* function uses to file associations defined in the user's Windows operating system to open the script with MATLAB.[1] If no such association exists, for example if the user does not have MATLAB installed on their computer, Windows will attempt to find a programme to open it. The script can easily be read in notepad if the user chooses to do so.

3 Results

The user can store their polygon vertices in a text file with vertices delimited by semicolons:

```
[1.413307 4.018804; 3.253044 1.649127; 4.333649 3.332069]
[-3.000000 0.000000; -1.500000 2.598076; 1.500000 2.598076; 3.000000 0.000000; 1.500000 -2.598076; -1.500000 -2.598076]
```

An example text file output. The first row shows the vertices of a stored triangle while the second shows the vertices of a hexagon.

On a Windows based computer with a MATLAB file association for *.m* files, a plotting script is automatically opened for the users. Fig 1. (below) is an example of the output of such a MATLAB script.

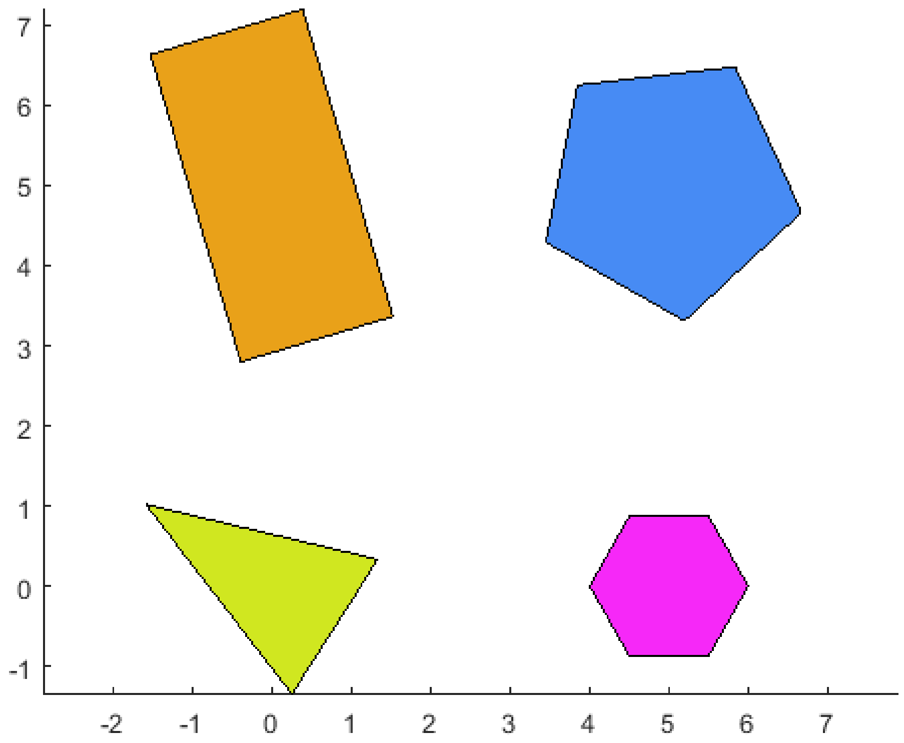

Fig 1. An example of a ***MATLAB*** *plot after the user entered the details for four different polygons.*

If the user compiles the code on a non-Windows based computer the programme should only create the *.m* file but not attempt to open it, however, this has not been tested.

4 Discussion

The programme was designed to be compatible when compiled on different operating systems and compilers. As such the Windows specific header, 'Windows.h', and all the functions from it are only included when the code was compiled on a Windows based computer. Additionally compiler specific headers and functions were avoided. The programme could be improved by adding the ability to automatically open MATLAB (if installed) on operating systems other than Windows, this would require specific code for each operating system family.

There were multiple ways the programme could have used MATLAB functionality to plot the polygons on a Windows based computer. MATLAB functions can be called directly from a C++ file, however, they rely on the header file 'engine.h' being able to be included in the source code. The full implementation differs for 32 bit and 64 bit versions of MATLAB, and thus could not be done while maintaining the ability to run the code for different versions of MATLAB. [2]

Another alternative was to to cause the programme to access the Windows command prompt and run the file with MATLAB engine with extra conditions:

```
%matlabpath%\matlab.exe" -nodisplay -nosplash -nodesktop -r "run(%filepath%\plotscript.m');
```

This could not be implemented without more extensive programming to find the path in which the MATLAB executable could be found. Additionally the programme would fail if the user did not have a version of MATLAB installed, where as the current implementation searches for an alternative programme to open the file with.

MATLAB is not a free programme and despite its wide spread use within the academic and engineering communities it can not be assumed that every user will have access to it. Additionally it is costly in terms of system resources to open MATLAB to simply plot the polygons. Therefore it might be wise to investigate other ways to present the polygons to the user, for example with additional C++ libraries or gnuplot.

The programme could be improved by including the ability to create N-sided polygons. This would require the vertices of this polygon to be stored in a two dimensional dynamic array or alternatively a vector or list of arrays of doubles. Using dynamic arrays and vectors will increase the amount of memory required by the programme to store polygons, however, as this is a very lightweight programme limitations from system specifications need not be a strong consideration.

Currently the interface for editing user entered polygons is not ideal. As the user has no visual feedback for their edits it is difficult for them to make an informed decision about what they want to change (they only have access to the vertices' coordinates). While it would be feasible to produce a MATLAB figures for each edit, it would be inconvenient for the user to continuously switch between programme windows.

This programme was created using C++, a programming language whose strength lies in its flexibility and the efficiency of its compiled programmes. If this programme remains solely used by a user to enter and manipulate polygons then these qualities are not required. In this case transferring the programme's logic into a higher level programming language with graph plotting tools built in may solve some of the problems described above. However, if this programme's class structure and functions were adapted to be used as part of a larger, more complex programme that would require large numbers of polygons created and stored in a small amount of time with limited resources, then C++ remains one of the best choices of language to implement it in.

5 Conclusion

The programme fulfilled all objectives set out in section 1. While it was specified that the programme had to output the polygons in a suitable format, the programme outputs the polygon details in three ways: to the console, a text file and a figure using a MATLAB script. This functionally becomes more limited for other operating systems or users without MATLAB installed, as a MATLAB cannot be automatically opened to run the written script.

References

[1] Windows Dev Center, *Microsoft* (Accessed: 02/05/15)

[2] MATLAB User documentation, *Mathworks* (Accessed: 02/05/15)