

Implementation of the 2D Polygon Program

Robert Vallance

8263714

May 2016

Abstract

A program to input 2D polygons into a map and allow the user to edit or return information on them was built. An abstract base class was used with derived classes (in separate header files) for an isosceles triangle, rectangle, pentagon, hexagon and other polygon. The user could specify which shape to create and specify whether they wanted the shape to be regular (co-ordinates automatically constructed) or input their own co ordinates. Each type shape was given a text colour using the 'windows.h' library to help with identification. The user could then access each shape (with a name they assigned to it) and perform translations, enlargements and rotations about the shape centres, in addition to retrieving information such as the perimeter and area of the shape or re-ordering the co-ordinates. The program ensures user input is valid and successfully implements the shapes.

1 Introduction

The project chosen was to design a class hierarchy to store the co-ordinates of 2D polygons. The basic task was to enable the user to choose types of shapes to input and then enable them to transform the shapes. The minimum requirements were for an isosceles triangle, rectangle, pentagon and hexagon, though this was extended further to include general polygons. The user is prompted to specify which type of shape they would like (from a menu) and type in the corresponding shape parameters as well as giving the shape a custom name. Once the shapes are built, the user is then presented with a list of shape names they can alter. They can then stipulate the type of transformation they would like to perform (from another menu). If they choose translation, the user types in two parameters (x, y) that the shape is moved on. Should they choose rescaling, they provide the enlargement factor that works from the centre of the shape. If they choose rotation, they are asked to provide the clockwise angle in degrees the shape will be rotated by. A rotation matrix (vector of vector of doubles) is used with matrix multiplication to rotate the shape about its centre point. The user can also specify if they would like information about the shape which prints out the co-ordinates, shape name, perimeter and area in a nice form, as well as giving the total number of shapes by use of a static variable. Also, a function to arrange the co-ordinates in clockwise order was written so the user could ensure the shape did not intersect itself and that the function to calculate the shape area would be valid.

2 Code Design and Implementation

The abstract base class, Polygon, was created, which we can point to every time we refer to a shape. This was put inside a header file called polygon.h. It contains member variables common to all shapes including x co-ordinates, y co-ordinates, a shape name, the shape number of sides and a static variable for total number of polygons. The class was put inside a namespace called polygonSpace so we had to specify this when using the associated data. The reason was because the 'windows.h' library (used later) also contains a function called Polygon that could clash with our naming. The class also contains member functions to GET data such as the co-ordinates, number of shape sides, perimeter, area, etc., as well as member functions to SET data such as the co-ordinates, shape name and re-order the co-ordinates clockwise. Friend functions exist to overload

the << and >> operators (for outputting and inputting the shape co-ordinates) and functions to translate, rescale and rotate the shape are created. All the functions are defined in a separate .cpp file called polygon.cpp. Finally, a virtual destructor was declared to enable the object to be deleted. Derived classes were also constructed for the individual shapes (called IsoscelesTriangle, Rectangle, Pentagon, Hexagon and OtherPolygon), which inherit the base class member functions and variables (since they were set as 'protected'). Constructors for the shapes could be specified in these classes. A constructor for each shape was created to put in 2N values for the co-ordinates (where N is the number of sides) for all shapes apart from OtherPolygon. Another constructor enables the user to put in a vector of co-ordinates for all shapes. Also, in the case of Pentagon, Hexagon and Other-Polygon, constructors were set so the points would be automatically generated (assuming regular polygons) given the shape radius, centre-point and angle top point is from vertical. Circle geometry is employed to put the points at regular angles along the circle circumference:

```

1   for (int i{ 0 }; i < number_sides; i++)
2   {
3       x_coordinates.push_back(x_centre + radius*sin(2.0*pi*i/number_sides + angle*
4       pi/180.0));
5       y_coordinates.push_back(y_centre + radius*cos(2.0*pi*i/number_sides + angle*
6       pi/180.0));
7   }

```

Figure 1: General method for automatically generating the co-ordinates for the shape given its radius, initial offset and clockwise angle of top point from vertical in degrees.

Further member functions were over-ridden in the shapes classes for shape translation, rescaling and rotation. The former was a simple case of shifting each co-ordinate by a user specified x and y value, iterating over all points:

```

1   for (int i{ 0 }; i < number_sides; i++)
2   {
3       x_coordinates[i] += x_translation;
4       y_coordinates[i] += y_translation;
5   }

```

Figure 2: Code for performing translations.

For the other two transformations, it was necessary to determine the shape centre first. This was taken to be the arithmetic mean of the points which corresponds to the centre of mass of the shape. The distance from the centre-point to each vertex was calculated in x and y. For rescaling, this distance is multiplied by an enlargement factor, which the user inputs:

```

1   for (int i{ 0 }; i < number_sides; i++)
2   {
3       x_coordinates[i] = x_centre_of_mass + enlargement*(x_coordinates[i] -
4       x_centre_of_mass);
5       y_coordinates[i] = y_centre_of_mass + enlargement*(y_coordinates[i] -
6       y_centre_of_mass);
7   }

```

Figure 3: Rescaling code.

For rotation our shape, a rotation matrix was generated from a user-given clockwise angle from the vertical as in equation 1:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \quad (1)$$

A vector of vectors was produced for the rotation matrix and matrix multiplication was used by multiplying the rotation matrix elements by the corresponding co-ordinates, and looping over all the data points:

```

1  for (int i{ 0 }; i < number_sides; i++)
2  {
3      x_coordinates_temp.push_back( x_centre_of_mass + rotation_matrix[0][0] * (
4          x_coordinates[i] - x_centre_of_mass) + rotation_matrix[0][1] * (y_coordinates[i]
5          - y_centre_of_mass) );
6      y_coordinates_temp.push_back( y_centre_of_mass + rotation_matrix[1][0] * (
7          x_coordinates[i] - x_centre_of_mass) + rotation_matrix[1][1] * (y_coordinates[i]
8          - y_centre_of_mass) );
9      x_coordinates[i] = x_coordinates_temp[i];
10     y_coordinates[i] = y_coordinates_temp[i];
11 }

```

Figure 4: Rotation code.

As additional work, it was decided to create additional functions to calculate the perimeter and area of the shape as well as re-order the points clockwise. These were defined in the base class since they were the same for each shape. The perimeter involved iterating over all points (but ending back at the first point) and calculating the distance between them using $\sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$.

The area was calculated using the Shoelace formula [1]:

$$A = \frac{1}{2} \left| \sum_{i=1}^{n-1} x_i y_{i+1} + x_n y_1 - \sum_{i=1}^{n-1} x_{i+1} y_i - x_1 y_n \right| \quad (2)$$

Here, the two sums were implemented using for loops iterating over the number of sides minus one.

Finally, the function to re-order the points clockwise was created using the ‘algorithm’ library. The purpose of this is to ensure the shape no longer intersects itself and that the Shoelace formula is valid. The N co-ordinate pairs themselves were placed in a vector giving N elements. A function to retrieve the angles of the points from the vertical was produced and then the sort() function used to order them by comparing the N elements. The ordering starts from the bottom-left quadrant and goes around clockwise.

One advanced C++ feature used was to create a map to store our shapes, polygon_shapes_map. The key is a string for shape name and the type is the object itself. The purpose of the name is so we can then easily access this shape later on rather than referring to it with an integer.

Another feature was using the ‘windows.h’ library. A function was built that changed the text colour depending on the number of shape sides. It is green for 3 sides, red for 4 sides, cyan for 5 sides, magenta for 6 sides and yellow for > 6 sides. This helps the shapes to stand out and tells us the type of shape.

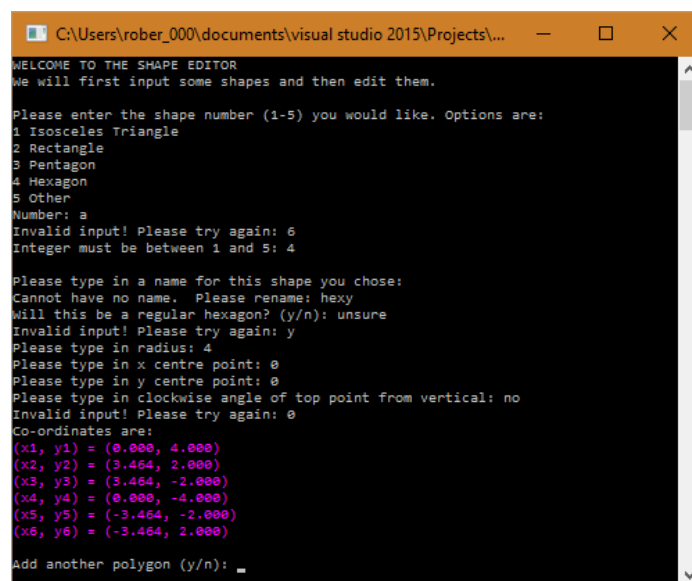
In the main code, a function is called that lets the user input a shape. This function is defined in ‘other_functions.cpp’ and first presents a menu of the shapes available where the user is prompted to enter a number 1-5. After selecting a shape type, a name is required for the shape (with a catch for if the user puts in no name or a name that already exists). Then they are asked for specific input regarding their shape. For the Pentagon, Hexagon and OtherPolygon, we can have the program create a regular polygon for us automatically or we can manually input the co-ordinates by employing the different constructors. For manual input, we use the overloaded >> operator: (cin >> *polygon_map[shape_name]). The co-ordinates are then outputted in the colour corresponding to the number of sides.

A further function to play with the shapes was also created that first produces a colour-coordinated list of the shapes and the user is asked to pick one of them to manipulate. They can then specify from a menu if they would like to transform it, reorder the co-ordinates or return information about the shape.

Since the user will be putting in lots of numbers and we want to check the input is valid, a function was defined, `input_and_test_number()`. This was made a template so it would accept both integers (such as in when the user selects a shape type) and doubles (e.g. inputting the co-ordinates). It ensures that the input are numbers between -999 and 999 and contains a while loop to force the user to put in valid input. A further function was produced called `test_yes_or_no()` that reads in a string. The program asks for a yes or no on several occasions and the function will accept many variations of 'y' such as 'Y', 'yes', 'Yes', 'YES' and similarly for 'n'.

3 Results

In starting the program, the user is greeted with a welcome message and asked to select the shape type from a menu. As an example, I will choose a hexagon (option 4) and call it 'hexy'. I say I want the shape to be regular and then give the radius (4), centre point (0, 0) and angle of top point from vertical (0 degrees). The constructor is employed and the co-ordinates are generated for me and printed in magenta in a nice form (see Figure 5).



```

C:\Users\rober_000\documents\visual studio 2015\Projects\...
WELCOME TO THE SHAPE EDITOR
We will first input some shapes and then edit them.

Please enter the shape number (1-5) you would like. Options are:
1 Isosceles Triangle
2 Rectangle
3 Pentagon
4 Hexagon
5 Other
Number: a
Invalid input! Please try again: 6
Integer must be between 1 and 5: 4

Please type in a name for this shape you chose:
Cannot have no name. Please rename: hexy
Will this be a regular hexagon? (y/n): unsure
Invalid input! Please try again: y
Please type in radius: 4
Please type in x centre point: 0
Please type in y centre point: 0
Please type in clockwise angle of top point from vertical: no
Invalid input! Please try again: 0
Co-ordinates are:
(x1, y1) = (0.000, 4.000)
(x2, y2) = (3.464, 2.000)
(x3, y3) = (3.464, -2.000)
(x4, y4) = (0.000, -4.000)
(x5, y5) = (-3.464, -2.000)
(x6, y6) = (-3.464, 2.000)

Add another polygon (y/n):

```

Figure 5: A demonstration of creating a regular hexagon of radius 4 and (0, 0) centre-point. The program generates the co-ordinates automatically and outputs them in a user-friendly way. The image shows the user putting in bad input for the shape numbers, for y/n, and initially no name for the shape to demonstrate the catchment of errors.

I can then specify if I want another shape, so I choose a Pentagon and manually input the co-ordinates as seen in Figure 6.

```

C:\Users\rober_000\documents\visual studio 2015\Projects\...
Please enter the shape number (1-5) you would like. Options are:
1 Isosceles Triangle
2 Rectangle
3 Pentagon
4 Hexagon
5 Other
Number: 3

Please type in a name for this shape you chose: hexy
Cannot have name clash. Please rename: penty
Will this be a regular pentagon? (y/n): no
Inputting co-ordinates manually in order lines will be drawn.
Please type in coordinate x1: 1
Please type in coordinate y1: 1
Please type in coordinate x2: 2
Please type in coordinate y2: 0
Please type in coordinate x3: a
Invalid input! Please try again: -2
Please type in coordinate y3: -2
Please type in coordinate x4: 2
Please type in coordinate y4: -2
Please type in coordinate x5: -3
Please type in coordinate y5: 4
Co-ordinates are:
(x1, y1) = (1.000, 1.000)
(x2, y2) = (2.000, 0.000)
(x3, y3) = (-2.000, -2.000)
(x4, y4) = (2.000, -2.000)
(x5, y5) = (-3.000, 4.000)
Add another polygon (y/n): _

```

Figure 6: User inputting a new shape, ‘penty’, and specifying the shape is not regular, so being prompted to input the co-ordinates themselves. Since there are five sides, the text colour is cyan. Note we also try calling this shape ‘hexy’ but a catch is in place to make sure shape names cannot be overwritten. We also try bad input on the co-ordinate, x3.

Once all the shapes are inputted, they are then printed on the screen.

Now, the user is presented with a colour-coordinated list of the shapes (created using an iterator over the map) which they can edit. First of all, I choose to edit ‘hexy’ and use the rescaling member function first as in Figure 7.

```

C:\Users\rober_000\documents\visual studio 2015\Projects\Obj...
Please type in the name of the shape you would like to edit.
Options are: hexy, penty
Name: hexy

Please type in (number 1-5) what you would like to do with hexy.
Options are:
1 Translate
2 Rescale
3 Rotate
4 Get info
5 Sort points clockwise
Number: 2

Please type in the rescaling factor:
Enlargement = 2

Rescaled by factor of 2.000.
Co-ordinates are:
(x1, y1) = (-0.000, 0.000)
(x2, y2) = (6.928, 4.000)
(x3, y3) = (6.928, -4.000)
(x4, y4) = (0.000, -8.000)
(x5, y5) = (-6.928, -4.000)
(x6, y6) = (-6.928, 4.000)

Would you like to edit the shapes again? (y/n) _

```

Figure 7: Rescaling ‘hexy’ by a factor of 2.

Figure 8 shows the user then rotating the shape by 180 degrees clockwise.

```
C:\Users\rober_000\documents\visual studio 201...
Would you like to edit the shapes again? (y/n) y
Please type in the name of the shape you would like to edit.
Options are: hexy, penty
Name: hexy

Please type in (number 1-5) what you would like to do with hexy.
Options are:
1 Translate
2 Rescale
3 Rotate
4 Get info
5 Sort points clockwise
Number: 3

Please type in the clockwise rotation angle (degrees):
Angle = 180

Rotated by angle 180.000 degrees.
Co-ordinates are:
(x1, y1) = (0.000, -8.000)
(x2, y2) = (-6.928, -4.000)
(x3, y3) = (-6.928, 4.000)
(x4, y4) = (-0.000, 8.000)
(x5, y5) = (6.928, 4.000)
(x6, y6) = (6.928, -4.000)

Would you like to edit the shapes again? (y/n) _
```

Figure 8: Rotating 'hexy' by 180 degrees.

Finally, we translate the shape by (2, 3) in Figure 9.

```
C:\Users\rober_000\documents\visual studio 201...
Would you like to edit the shapes again? (y/n) y
Please type in the name of the shape you would like to edit.
Options are: hexy, penty
Name: hexy

Please type in (number 1-5) what you would like to do with hexy.
Options are:
1 Translate
2 Rescale
3 Rotate
4 Get info
5 Sort points clockwise
Number: 1

Please type in the coordinates x, y you would like to translate by:
x translation = 2
y translation = 3

Translated by (2.000, 3.000).
Co-ordinates are:
(x1, y1) = (2.000, -5.000)
(x2, y2) = (-4.928, -1.000)
(x3, y3) = (-4.928, 7.000)
(x4, y4) = (2.000, 11.000)
(x5, y5) = (8.928, 7.000)
(x6, y6) = (8.928, -1.000)

Would you like to edit the shapes again? (y/n)
```

Figure 9: Translating 'hexy' by (2, 3).

Following this, we now edit 'penty'. From Figure 6, the co-ordinates $(x_1, y_1) \rightarrow (x_6, y_6)$ are not in clockwise order. For instance, (x_3, y_3) is in the bottom-left quadrant but (x_4, y_4) is in the bottom-right. I thus want to re-order these so choose option 5.

```
C:\Users\rober_000\documents\visual studio 2015\Proje...
Would you like to edit the shapes again? (y/n) y
Please type in the name of the shape you would like to edit.
Options are: hexy, penty
Name: aardvark
Please type in correct name: penty

Please type in (number 1-5) what you would like to do with penty.
Options are:
1 Translate
2 Rescale
3 Rotate
4 Get info
5 Sort points clockwise
Number: 5

Sorting points clockwise (from bottom left quadrant).
Co-ordinates are:
(x1, y1) = (-2.000, -2.000)
(x2, y2) = (-3.000, 4.000)
(x3, y3) = (1.000, 1.000)
(x4, y4) = (2.000, 0.000)
(x5, y5) = (2.000, -2.000)

Would you like to edit the shapes again? (y/n) _
```

Figure 10: Sorting the points of ‘penty’ clockwise - ordering from bottom-left quadrant. User also tries accessing a shape name that isn’t there and is asked to try again.

Finally, I now want to get some information on this shape, so I choose option 4. This gives the co-ordinates, perimeter, area and total number of shapes in our map using the static variable.

```
C:\Users\rober_000\documents\visual studio 2015\Proje...
Would you like to edit the shapes again? (y/n) y
Please type in the name of the shape you would like to edit.
Options are: hexy, penty
Name: penty

Please type in (number 1-5) what you would like to do with penty.
Options are:
1 Translate
2 Rescale
3 Rotate
4 Get info
5 Sort points clockwise
Number: 4

Shape name is: penty

Number sides is: 5
Co-ordinates are:
(x1, y1) = (-2.000, -2.000)
(x2, y2) = (-3.000, 4.000)
(x3, y3) = (1.000, 1.000)
(x4, y4) = (2.000, 0.000)
(x5, y5) = (2.000, -2.000)

Perimeter of shape is 18.497

Area of shape is 17.500
(area valid if not self-intersecting - use Sort Points Clockwise option otherwise)

This is 1 of 2 polygons

Would you like to edit the shapes again? (y/n) n
Stopped messing with shapes.

polygon_map has size 2
Destroying Hexagon
Destroying Polygon
Destroying Pentagon
Destroying Polygon
polygon_map now has size 0

THANK YOU FOR USING THE SHAPE EDITOR.

Press any key to continue . . . _
```

Figure 11: Getting info about ‘penty’. After this, the user has finished with the shapes and so the program clears up by iterating over the shapes and deleting them, calling the destructors. A goodbye message finally appears.

4 Discussion and Conclusions

The code successfully creates, transforms and returns information about the shapes. The colours help the user to differentiate the shapes apart and together with outputting the co-ordinates in a nice form and providing many error catches, the experience is user-friendly. A future way to further improve the visual appeal would be to draw out the shape. This could be done with ASCII characters on the console or even by creating a bitmap image output file where we can manipulate individual pixel colours. A separate class could be created to perform this.

The addition of an extra class, OtherPolygon, made the program much more general and the automatic creation of regular polygon co-ordinates saved a great deal of time. Other features that could be added include a function to create a random shape for us that uses a random number generator to select the shape type and parameters for the shapes. One could also test for overlap between the polygons by using the Separating Axis Theorem [2]. This uses the idea of creating axes that run along each shape side for all shapes. If there is shape overlapping along all the axes, then we have a collision. We could also extend the program so we are not restricted to polygons (e.g. we could create circles and ellipses) or even go on to create 3D shapes.

In terms of code design, putting the user input with error checking in functions significantly reduced the amount of code repetition. Using polymorphism with an abstract base class and derived classes for the individual shapes allowed simple creation of shapes and a map of base class pointers to be used. The map enabled the user to choose the shape to access via its given name. However, in making the translation, rescaling and rotation functions pure virtual in the base class, we required them to be over-ridden in the derived classes. These functions can easily be generalised to iterate over the number of shape sides and so could be defined in the base class as non-virtual and hence would reduce code repetition and decrease the chances of an error in the code.

References

- [1] Shoelace Formula. Taken from https://en.wikipedia.org/wiki/Shoelace_formula. Accessed 11/05/16.
- [2] Separating Axis Theorem. Taken from <http://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169>. Accessed 11/05/16.

The number of words in this document is 2364.