Python: Introduction to Computational Thinking
Hatching Polygons

1. Introduction

The objective of the mini project is for you to produce a program of a moderate size and depth that will require you to utilise what you have learned in this course, and a bit more, to do something useful and interesting. Through this, you would learn to design, manage and execute a sizable program.

2. The Project – Art with geometry and engineering maths

The project gets you to work with geometry in the guise of polygons and engineering mathematics, in using matrix algebra for shape transformation. You are to create polygonal shapes which contain straight and curved edges and produce systematic patterns containing multiple copies of these shapes, each copy a transformation of the parent shape. Figure 1 shows two examples.

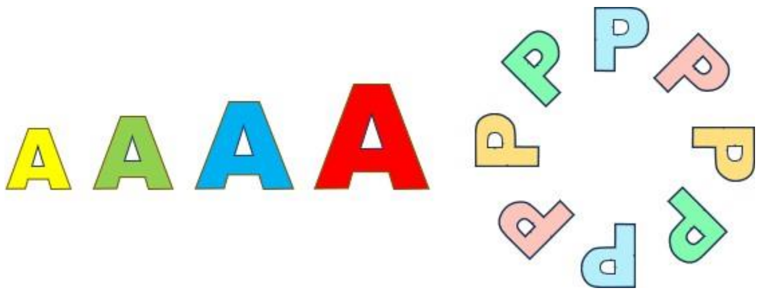

Figure 1: Two patterns of shapes

2.1 Your Tasks

Your task is to write a program that inputs one or more polygons and generates patterns with them that contains transformed copies of the same polygons. A polygon is defined by its ordered list of vertices. Vertices are defined by their (x, y) coordinates. Two adjacent vertices uniquely define a straight edge. Where a curved edge exists between two vertices, then more data is required for the curve. A method for creating a curve is given in Appendix 1. There are different types of transformations, and they are defined in Appendix 2, together with the mathematical formulae for performing the transformations.

You are to write the program in Python using what you have learned in MA1008. Program quality will be assessed according to the design of the data structures, the quality of the algorithms, suitable use of programming constructs, and the logical flow and readability of the program.

The graphical components of the project are to be developed using only the built-in graphics module in Python called *turtle*. Section 2.2.5 provides more information about *turtle*.

You are also required to produce a brief report explaining the functionality of your program. The report is expected to highlight the features that are included in the program. You need not include code snippets or explain them in detail. These should be provided as comments in your code.

2.2 Actions you or your program need to take

To help you in designing your program, this section provides a list of actions that you or your program need to take. You may, of course, add to the list.

2.2.1 Program Capabilities

Firstly, here is the list of capabilities that your program needs to provide:

a. Define and display a polygon with straight edges.
b. Define and display a polygon with a mix of straight and curved edges.
c. Perform transformation(s) on a polygon and display the outcome.
d. Generate systematic patterns with a polygon or polygons.

e. Design and implement the data structure, the inputs and user interactions for the above steps. Cater to two modes of input:
    - interactive input by typing in the data using the keyboard or mouse clicks
    - file input

At the most basic, you need to get the data structure and one of the input modes in Item e plus the first two items working. Then, extend the program by adding the other capabilities one at a time.

2.2.2 Program Design

a. A program needs to be designed before you start any coding. Work out what modules are required in your program, and how the modules link with each other.
b. Most modules can and should be coded as functions, and your main program should be fairly short, consisting mainly of calls to these functions.
c. Where a set of statements is repeated in different places in the program, consider putting it into a function, and call the function where the repetition occurs.
d. When a repetition occurs but with different variables and outputs, then the variables probably should be parameters to your function, and the outputs should be the returned values.
e. At the outset, you need to decide on how your polygon is to be stored (i.e. design a data structure for the polygon). Remember, you need to cater to polygons with curved sides, and decide how to accommodate the curve in among the straight edges of the polygon. Some polygons also have another polygon inside (such as the A in Figure 1). The bulk of your algorithms and program depends on this decision. So, you need to be sure of the decision very early.

2.2.3 Inputs

a. You need to provide facilities for the user to define a polygon (with or without curved edges, and with or without holes).
b. Allow for inputting the polygon by inputting its vertices, and store these vertices in the data structure you have designed.
c. You need to allow the user to select between interactive and file input modes. You may input the coordinates by typing in their values. However, for interactive input, you may consider inputting the vertices by mouse clicks.
d. Allow for inputting the specifications for the transformations and the patterns. A pattern should be controlled by an algorithm, not by you manually placing copies of a polygon.
e. Allow the user to save the data of both the polygon, the transformations and the pattern to a data file. You would need to design the file format.
f. Allow the user to retrieve inputs from a data file (same format as above).

2.2.4 Computations

a. Specifying a straight-edge polygon requires only the vertices. A curved edge needs more. Appendix 1 gives the specifications of curved edges and how to connect them to the other edges of the polygon.
b. The transformations involve mainly matrix multiplications. Appendix 2 gives the full mathematical specifications.
c. You may choose the pattern to generate and this is to be controlled by an algorithm, which basically involves a simple mathematical distribution which you have to device yourself.

2.2.5 Graphics

The polygons are to be drawn using the graphics library called Turtle. Turtle capabilities and functions are described in Section 24.1 of the Python documentation, which you can access via the IDLE interactive shell by clicking Help > Python Docs and then search for “turtle”.

In doing the drawing, you need to consider a few basic things.

a. Turtle provides you with many functions for doing display, but you only need to use a few of them for the purpose of this project. So, determine the graphics functions you need before reading the Turtle documentation. That would save you some time.

b. When you use turtle functions, it opens a display window with a default size of about (-300, -300) to (300, 300) in terms of pixels. You may work within this default window size. However, you may also reset the window size according to your need by calling the appropriate turtle function.
c. To draw a line between two points, which is basically what you need to do when drawing the straight and curved edges of the polygon, you should consider using the `goto(x,y)` function where `(x,y)` is the coordinates of the point you are drawing to from the current position (which is the position you last drew to). Avoid using the directional functions such as `forward()`, `left()`, `right()` because they are cumbersome to use when you are not drawing in a specific direction.
d. When inviting input from the user, one may use the usual `input()` function. The dialogue of this function is done in the IDLE shell, which may be inconvenient when you are working in a graphics window also. You may consider using the turtle functions `textinput()` and `numinput()` which allows you to input via a pop-up dialogue box.
e. You may consider beautifying your display using colours and lines of different thicknesses or styles to add interest to your display.
f. By default, Turtle draws slowly to allow you to see how the drawing is done. However, for this project, you should draw quickly. So make sure that you set the drawing speed to the fastest possible, again doable by calling the appropriate Turtle function with the appropriate parameter value. You should hide the turtle too.

2.2.6 Program Flow and Control

a. It should be clear at every stage of running the program what the user needs to do.
b. Therefore, you need to ensure that the information presented on the screen, such as prompts for user inputs, is clear and unambiguous. Be short and sharp too.
c. When asking the user to make a selection, always require the minimal input from the user to reduce burden and error.
d. Allow the user to select between ending the program and returning to input for a new polygon.
e. Your program should trap and handle errors appropriately. For example, in the case of an input error, your program should tell the user what the error is and allow the user to re-input.
f. Your program should be easy to read and understand. So, make sure that it is well commented, well modularized and uses meaningful (but not overly long) variable and function names.

3. Prohibitions

Apart from the graphics library, every programming need for this project can be met with what you have learnt in this course. To allow you to exercise what you have learnt and to prevent you from veering wildly beyond the scope of this project, your program should:

i. Not use the *class* construct to define objects you require in your program
ii. Not use *tkinter, matplotlib, plotly* or similar for graphics. Only the *Turtle* library is allowed.
iii. Not use the libraries *numpy, pygame* or *json*.
iv. Not use the *lambda* construct. This is a very useful construct and is used in many programs online. But many past students simply copied those programs without any idea what *lambda* is or does.
v. Not use the *eval()* function in Python. It is a powerful function but does too much for you. You should do things yourself to demonstrate your ability.

You may use standard Python libraries like *math*. If you are unsure whether a certain library or construct is allowed, please consult your tutor.

4. Conduct of the Project and the Submission

The duration of this project is from the Monday of Week 10 to the Sunday of Week 14, spanning four weeks. The class hours in Weeks 11-13 are dedicated to the project, apart from CA4 in Week 11 for one hour. Your tutors will be in the class to offer you help. Make full use of them.

In the classes between Weeks 11 to 13, your tutor will check your progress, during which you need to show and explain your program thus far. **This interim progress is worth 15%.**

The project deadline is at **23:59 hours, Sunday 19 November 2023.** You need to submit

i. **A working Python program** that gets the relevant inputs from the user or from a file and displays the polygons and their transformations to generate the patterns.
ii. **A data file** containing the required data for at least three different patterns of different polygons with different transformations. Your grader will be using these data to test run your program.
iii. **A report in a Word or PDF file**, providing
   - A guide on how to run your program
   - A list of the capabilities your program provides, including the input capabilities, either interactively or from file, and the display capabilities.
   - Your data file format.
   - Three different sets of diagrams captured from your graphics window of three different patterns of different polygons. Your pictures should
   - Highlights of the **key strengths and limitations** of the program.
   - Highlights of features you consider to be worth some bonus marks.

Please submit all your files through the course site at the same item where you fetched the project file. The title of the item is a link to the submission page, which has a link for uploading your files.

5. Grading Rubrics
Here are what the graders will be looking for when grading:
i. The ease in interacting with the program, which includes specifying the input data and working with the input files.
ii. The quality and correctness of your program, which includes:
   - How easy it is to read and understand your program. This means your program should be adequately commented with good choice of variable and function names.
   - Logical program flow.
   - Modularisation of the program including appropriate use of functions, the design of the functions which include the appropriateness of parameters in the functions.
iii. The quality and correctness of the outputs, especially the graphics display.
iv. Due consideration will be given to thoughtful designs of patterns, with some form of artistic quality.
v. Full mark is 100%, including 10% for homogeneous coordinates and 10% for curves.
vi. There are bonus marks, maximum 15%, for nice features that are beyond the requirements. Hence you can potentially score a maximum of 115%. If your total is beyond 100%, the extra marks will go to top up your CA1 or CA3.

6. Epilogue
You should start working on the program immediately. Do not procrastinate. What you produce depends on the time you spend on the project and your ability in creating good algorithms and writing good code.

One of the major problems of past projects for the tutors is that there is no way to tell how to run a program. The program must offer help to the user directly (using prompts and on-screen instructions) on what input is required at every stage. Quite often, the input is very cumbersome, such as requiring the user to type a long string exactly. You need to help your user by making your input requirements as obvious and as simple as possible. For example, it is better to ask the user to “Enter b for black and w for white”, instead of asking for the words “Black” and “White”.

This is an individual project. You may consult the tutors and discuss with your classmates, but everyone must write their own program. **Any programs deemed to have been copies of each other will be penalised heavily, regardless of who is doing the copying.**

If in doubt or in difficulty, always ask. And ask early!

Appendix 1: Cubic Bezier Curves

A cubic Bezier curve is defined by four points, and its equation is of the form:

$$\mathbf{P} = (1-t)^3\mathbf{P_0} + 3t(1-t)^2\mathbf{P_1} + 3t^2(1-t)\mathbf{P_2} + t^3\mathbf{P_3}, \qquad 0 \le t \le 1.$$

where $\mathbf{P} = (P_x, P_y)$ is the point on the curve at parameter value t, $\mathbf{P_0}$, $\mathbf{P_1}$, $\mathbf{P_2}$, and $\mathbf{P_3}$ are the four given points, called control points. Bold characters signify vectors. This is a vector equation, which means there are actually two separate equations:

$$P_x = (1-t)^3P_{0x} + 3t(1-t)^2P_{1x} + 3t^2(1-t)P_{2x} + t^3P_{3x}$$
$$P_y = (1-t)^3P_{0y} + 3t(1-t)^2P_{1y} + 3t^2(1-t)P_{2y} + t^3P_{3y}$$

The equation is cubic in t. t going from 0 to 1 traces out the trajectory of the curve. Below are four examples of cubic Bezier curves.

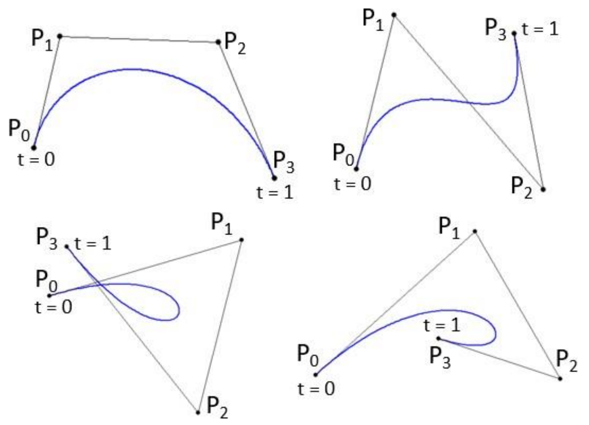

Figure A1: Example Bezier curves (in blue) and their control polygons

The four control points, when connected by straight lines, form the control polygon. This term "control polygon" is a misnomer because the "polygon" is not closed. However, it is commonly used in the literature, so we will stick to it here. There are several useful properties relating the curve and the control polygon. We will mention only two which are relevant to this project:

1. The curve starts from the first point (**P₀**) of the control polygon and ends at the last point (**P₃**).
2. At the start point, the curve is tangential to the first segment of the control polygon, and at the end point, the curve is tangential to the last segment of the control polygon.

You can observe these properties in Figure A1. You can also see that the curve follows the "flow" of the control polygon, and you can manipulate the shape of the curve by moving the control points.

You will need to include Bezier curves to define the curved sides of your polygon. Often, you require a curve to be tangential to an edge of your polygon, and Property 2 above comes in useful because then you simply have to make sure that the first or last edge of your control polygon is collinear with the polygon edge the curve is to be tangential to. See Example below. When drawing Bezier curves, you need only draw the curve without the control polygon.

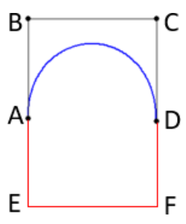

Figure A2: ABCD is the control polygon to the blue Bezier curve, which forms a closed shape with the bottom red edges. To ensure that the curve is tangential to AE at A, E must be collinear with the segment AB of the polygon, and to ensure tangency at D, F must be collinear with CD.

There are other types of curves, such as circular arcs and sinusoids. You are welcome to use them if you know how to handle them. The reason we are using Bezier curves is because they are defined purely by points, the control points. The transformation of a curve is a straightforward transformation of its control points, and therefore does not add much complication.

Appendix 2: Shape transformation

A shape is transformed by uniformly transforming the vertices that form the shape. For example, to translate the shape in Figure A2 by the vector (a, b), we add (a, b) to the coordinates of every point in the shape, as Figure A3 illustrates. Hence, the mathematics of the transformation of shapes starts at the transformation of a point.

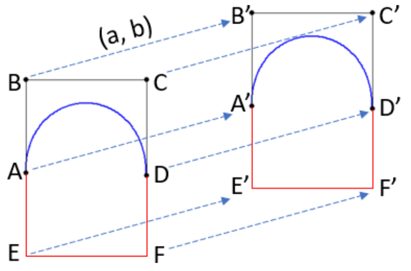

Figure A3: Translation of a shape: A' = A + (a, b), B' = B + (a, b) … F' = F + (a, b)

We will deal with five different transformations: translation, rotation, scaling, shearing and reflection.

1. **Translation**: This is moving a point linearly by a given vector (a, b). If the point is P = (p_x, p_y), then the result of the translation is P' = (p_x, p_y) + (a, b) = (p_x+a, p_y+b).

2. **Rotation**: This is the rotation of a point (p_x, p_y) about a given point (a, b) by a given angle θ. If (a, b) is the origin (0, 0), then the rotated point (q_x, q_y) is (see Figure A4):

   $q_x = r\cos(\beta + \Theta) = r\cos\beta\cos\Theta - r\sin\beta\sin\Theta$, and
   $q_y = r\sin(\beta + \Theta) = r\cos\beta\sin\Theta + r\sin\beta\cos\Theta$

   But, $p_x = r\cos\beta$ and $p_y = r\sin\beta$, so

   $q_x = p_x\cos\Theta - p_y\sin\Theta$, and
   $q_y = p_x\sin\Theta + p_y\cos\Theta$

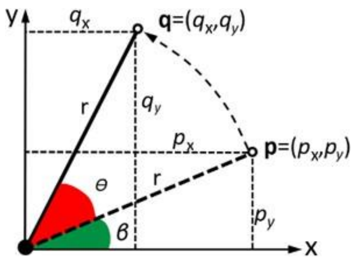

Figure A4: Rotation of a point.

   If (a, b) is not at the origin, then the rotation has three steps:

   i. Translate (a, b) to the origin, effectively moving (p_x, p_y) to (p_x-a, p_y-b).
   ii. Rotate this new (p_x-a, p_y-b) about the origin by the required angle, producing (q_x, q_y).
   iii. Translate (q_x, q_y) back to the real position by doing ((q_x+a, q_{y+}+b).

3. **Scaling**: This is to enlarge or reduce the size of an object, which can be equal or unequal in the x and y directions. If the scaling factor in the x direction is S_x and the scaling factor in the y direction is S_y, then the point (p_x, p_y) becomes (S_xp_x, S_yp_y). One may ask what the meaning of scaling a point is, as a point has no size. However, if the point is a part of an object, then it is meaningful. Take the line from the origin to (p_x, p_y). If $S_x = S_y = 2$, then upon scaling, the two end points of the line are the origin again and ($2p_x$, $2p_y$), which is twice the length of the original line.

   It is necessary to take note of the anchor point on the object when scaling. The anchor point is the point which does not move in the scaling. A square with centre at (a, b) when scaled by 2 in both the x and y directions will have the new center at (2a, 2b). Is that what you want?

   If you want the centre (or any point you choose) to be the anchor point, then you need to take three steps, just like in the rotation above: (i) move the anchor point to the origin which requires moving all the points in the object by the same amount, (ii) apply the scaling, (iii) move the anchor point and all the points in the object back to their real positions.

4. **Shearing**: The shear transformation slants the shape of an object: a rectangle would become a parallelogram upon shearing. There are two shear transformations: x-shear which shifts x coordinates values and y-shear which shifts y coordinate values. Each case only changes one dimension's values

and preserves the other. Hence, given the point (p_x, p_y), and an x-shear factor of s_x for shearing in the x direction, the new point is ($p_x + s_x p_y$, p_y); for y-shear, it is (p_x, $p_y + s_y p_x$). To have both x and y-shear requires both the factors. Figure A5 illustrates the application of shear in the x and y directions.

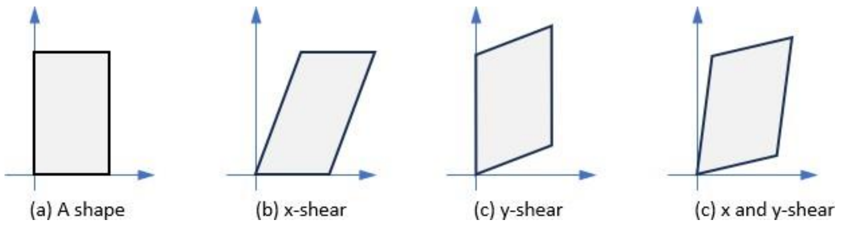

Figure A5: Shearing

5. **Reflection**: This is to produce the mirror image of an object given a "mirror". In 2D, a mirror is merely a straight line. For a point (p_x, p_y), its mirror image about the x-axis is simply (p_x, $-p_y$), and its mirror image about the y-axis is ($-p_x$, p_y). The reflection about a line that is not the x or y axis is more complicated. It involves transformations to first bring that mirror line into either the x or the y-axis, then do the reflection, and then invert the earlier transformations to take the line back to its original position. What the transformations are depends on the position of the line.

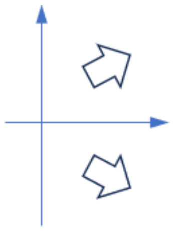

Figure A6: The bottom arrow is the mirror image of the top arrow above the x-axis.

Homogeneous Coordinates

All the above transformations are to be applied individually and separately to every point one wishes to transform. This works but can be painful when one wants to transform an object multiple ways and multiple times. Take the rotation about a point that is not at the origin, we saw that three are three operations: translation, rotation, translation. One may wish to add in scaling, shearing, and reflection too, not to mention further rotations and translations. When these are to be done over an object with, say, 50 points, you can see the hard work there. Granted it is not you but the computer that is to do the work, but think about real life objects that are defined by hundreds or even thousands of points. The computational load is enormous. (In 3D, the number of points can be in the millions. Seen movies with digital human figures? There are millions of points there.) Homogeneous coordinates come to the rescue.

In homogeneous coordinates, a 2D vector (x, y) is represented as a 3D vector (x, y, 1). When we only want the coordinates, we only use the x and y and ignore the third element which has the value 1. I will deal with the third element later. For now, just take its value to be 1.

With homogenous coordinates, all the above transformations can be encoded in a 3x3 matrix.

Translation of (p_x, p_y) by (a, b): $\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$

The point vector is written as a column vector and placed on the right of the expression. Perform the matrix multiplication and you get (p_x+a, p_y+b, 1), which is the resulting point. Note that translation cannot be done using a matrix if we remain purely in 2D.

Rotation of (p_x, p_y) by angle θ: You can see from the rotation equations that the matrix is

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

Scaling: The appropriate scaling factors are to be placed in the x and y diagonal elements.

$$\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

Performing the matrix multiplication gives ($S_x p_x$, $S_y p_y$, 1). If only scaling one of the axes, then the unscaled value should be set to 1.

Shearing: The shearing factors are to be placed in the off-diagonal positions.

$$\begin{pmatrix} 1 & s_x & 0 \\ s_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

Performing the matrix multiplication gives ($p_x + s_x p_y$, $p_y + s_y p_x$, 1). If applying shear only in one direction, then the unused factor should be set to 0.

Reflection: This requires negating the diagonal x or y values. For reflection about the x-axis:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

Performing the matrix multiplication gives (p_x, $-p_y$, 1).

The power of homogeneous coordinates

Let's say we wish to translate a point, then rotate it, then translate it again, and then scale it. Below is the sequence of matrix operations, written as one expression in the order the operations are applied, with the matrices applied right to left:

$$\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & a_2 \\ 0 & 1 & b_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & a_1 \\ 0 & 1 & b_1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

Assume that all the variables carry appropriate values. There are four matrix multiplications there. Then consider that the object to be transformed has 1000 points. Then there are 4000 matrix multiplications, which is a lot of work.

Because matrix multiplications are associative, i.e. (AB)C = A(BC), where A, B and C are matrices, we can write the above expression as

$$\left(\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & a_2 \\ 0 & 1 & b_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & a_1 \\ 0 & 1 & b_1 \\ 0 & 0 & 1 \end{pmatrix}\right) \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

This means the four 3x3 matrices could be multiplied first to produce one 3x3 matrix. So let

$$(M) = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & a_2 \\ 0 & 1 & b_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & a_1 \\ 0 & 1 & b_1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}.$$

Then we get $(M)\begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$ as the resulting expression, which is one matrix multiplied to one column vector.

Now, if you transform an object of 1000 points, you have 1000 matrix multiplications. Together with the three matrix multiplications to form (M), the total is 1003. 1003 against 4000, the advantage is obvious.

This saving far outweighs the small extra work to deal with the third element in a homogenous coordinate vector.

Now, back to the third element where the value so far is always 1. In general, a homogeneous coordinate point is not (x, y, 1) but (x, y, w) where w is a non-zero value. The true 2D coordinates of the vector (x, y, w) is (x/w, y/w, w/w) = (x/w, y/w, 1). So, when we are given (x, y, w), we recognise its 2D coordinate as (x/w, y/w).

If you observe the 3x3 matrices we have used so far, the last row is always (0, 0, 1). This isn't always the case. I will only deal with one situation involving this row, and that is when the value 1 is replaced by a positive value S (and S ≠ 0). Now, take the following matrix multiplication:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & S \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

The result is (p_x, p_y, S) in homogeneous coordinates. The 2D coordinates of this point is (p_x/S, p_y/S), which is essentially a scaling of both p_x and p_y by the same factor 1/S. Hence, S serves as the uniform scaling factor for both the x and y values, as opposed to the individual scaling factors we had earlier. Do note that the factor here is 1/S, which means a value of S more than 1 reduces the size of the object and a value less than 1 increases the size of the object.

The first two positions that currently carry 0 in the last row have very interesting applications as well. But that will be a story for another day.

Final note: In this project, we prefer that you use homogeneous coordinates for the transformations because it is the right thing to do, and it makes for a better program. Using of homogeneous coordinates carries 10 marks.