CS

B411324

An Application Program Designed For Curve Learning

by

Jintian Zhu

Supervisor: Dr H. Bez

June 2015

Abstract

In this project, an application with graphical user interface is developed using Windows Presentation Foundation (WPF) to help learn plane curves. One page (actually UserControl) is used for demonstrating each curve, it contains literal description, equation, associated curves and representation of the curve. To facilitate understanding, this application also applies a timer to implement animation for the curve drawing process, and some auxiliary shapes or texts are available for the user, who can choose to see these auxiliary objects or not during animation. To further strengthen interactivity, the user can change curves' shapes by setting different parameters, and a slider is created to adjust speed of animation. Moreover, the application applies customised validation mechanism inform the user of invalid parameter inputting.

Some efforts are made to make the user interface friendlier. The window is redesigned and is quite different from its original style. The scroll bar is designed to be hidden but appears when mouse is over. Buttons are beautified and defined with some animation.

In general this application functions well, while it still has some flaws like the animation might be slower than expected, and the application can be improved in many aspects.

Content

1. Introduction

This project aim at making a tool that provides an easier way for geometry learning. The achievement of this project is an illustrated handbook like application that introduces some typical planar curves. This application is developed to help geometry learners more intuitively understand how different curves are generated and how they relate to other curves. In this application 24 curves are introduced. And for each curve, the user can not only know its history and equation, but also change parameters to see how the shape of the curve changes. Also, this application allows user to see animation of the whole process of drawing the curve, and provide easy access to its associated curves to help with understanding.

This report first gives some mathematic background knowledge that is useful in this project, next discusses the design of the program which includes its sub-objectives and the methods designed for realizing them, followed by the structure of this application, then introduces in detail how the project is implemented, finally evaluates it and also discussed some extensions.

It is an individual completing project that there are no human participants other than undertaker and supervisor.

2. Background

2.1 Mathematic representations

In algebraic geometry a plane curve can be defined using explicit equation, implicit equation or parametric equation.

In mathematics, an explicit equation has one dependent variable on the left-hand side, and all the independent variables and constants on the right-hand side of the equation. For example, the explicit equation of a line is: $y = mx + b$. Where m is the slope and b is the y -intercept. Explicit functions generate y values from x values.

An implicit equation is a relation of the form $R(x1, \ldots, xn) = 0$, where R is a function of several variables (often a polynomial). For example, the implicit

equation of the unit circle is: $x^2 + y^2 - 1 = 0$. Moving all variables and constants on the right-hand side of any explicit equation to the left makes it implicit.

Parametric equations of a curve express the coordinates of the points of the curve as functions of a variable, called a parameter. For example: $x = \cos\theta, y = \sin\theta,$ are parametric equations for the unit circle, where θ is the parameter. Together, these equations are called a parametric representation of the curve.

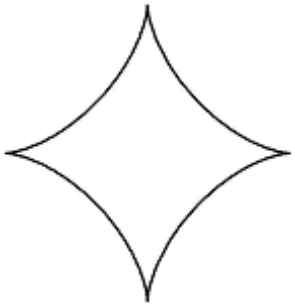
It is easy to convert explicit equations into implicit equations, but for some implicit equations it is difficult to convert them conversely. Thus curves that can be represented using polynomial equations are usually defined in implicit equations for unifying. The implicit and parametric equations are mostly used to represent curves. Generally, implicit representations are used in number theory, and are preferred for problems such as point classification, while parametric representations are normally preferred for curve drawing purposes (Wetzel).

More concretely, in this application curves introduced are represented in three ways: implicit polynomial equations, parametric equations using the angle as parameter, and explicit equations in polar coordinate system. It is easy to convert the third representations into parametric equations in Cartesian coordinate system using the equations: $x = r\cos\theta,\ y = r\sin\theta$. Further, usually using angle as variable (which increases from 0 to the maximum angle) in generating the curve makes the whole drawing process more intuitional, so the second and third ways above are preferred in designing curve drawing functions, while all possible rational representations for each curve is introduced in the application.

2.2 Curve index of this application

The following enumerates briefly all the curves introduced in this application.

2.2.1 Astroid



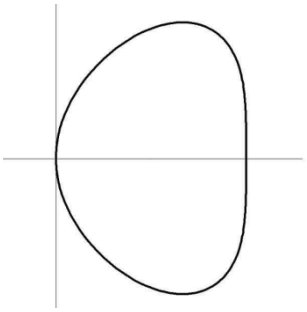
A four-cusped hypocycloid. Equations:

$(i)(x, y) = (\cos^3\theta, \sin^3\theta)$,

$(ii)(x, y) = \frac{3}{4}(\cos\theta, \sin\theta) + \frac{1}{4}(\cos 3\theta, -\sin 3\theta)$,

$(iii) x^{\frac{2}{3}} + y^{\frac{2}{3}} - 1 = 0$.

2.2.2 Bean curve

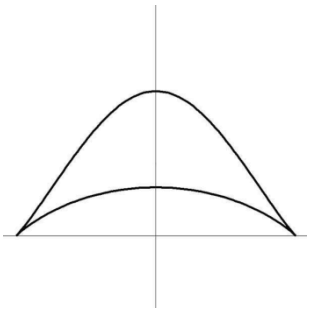


The bean curve is a quartic plane curve. Equations:

$(i) x^4 - x^3 + x^2y^2 - xy^2 + y^4 = 0$,

$(ii) r = \frac{4\cos\theta}{3+2\cos^2\theta}$.

2.2.3 Bicorn curve

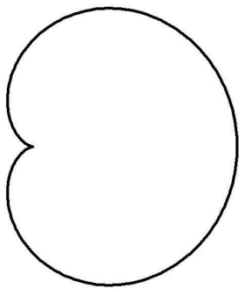


The bicorn is a rational quartic curve. It has two cusps and is symmetric about the y-axis. Equations:

$(i)(x, y) = a(\sin\theta, \frac{\cos^2\theta(2+\cos\theta)}{3+\sin^2\theta})$,

$(ii) y^2(a^2 - x^2) - (x^2 + 2ay - a^2)^2 = 0$.

2.2.4 Cardioid

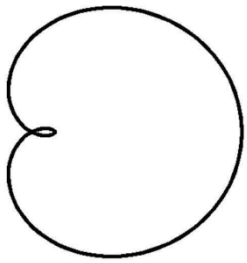


A cardioid is a plane curve traced by a point on the perimeter of a circle that is rolling around a fixed circle of the same radius. It is therefore can also be defined as an epicycloid having a single cusp. Equations:

$(i) r = 1 + \cos\theta$,

$(ii)(x, y) = (\cos\theta, \sin\theta) + (\cos 2\theta, \sin 2\theta)$.

2.2.5 Cayley's Sextic

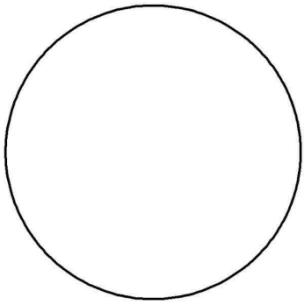


A Cayley's Sextic is the pedal of a cardioid with respect to its cusp. Equations:

$(i) r = \cos^3\left(\frac{\theta}{3}\right)$,

$(ii) 4(x^2+y^2 - ax)^3 - 27a^2(x^2 + y^2)^2 = 0.$

2.2.6 Circle

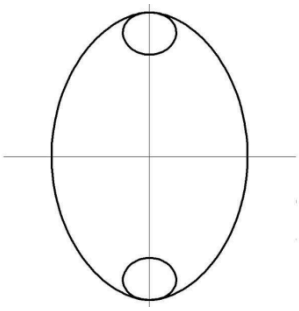


A curve of constant distance from a centre. Equations:

$(i) r = \text{a constant}$,

$(ii) x^2 + y^2 - a^2 = 0.$

2.2.7 Cornoid

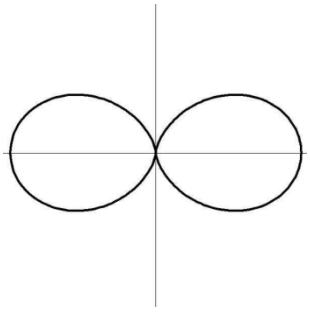


The cornoid is a sextic curve. Equations:

$(i) x^6+y^6 + 3x^4y^2 + 3x^2y^4 + 3x^4 - 5y^4 + 8y^2 - 4 = 0$,

$(ii)(x, y) = a\big(\cos\theta(1 - 2\sin^2\theta), \sin\theta(1 + 2\cos^2\theta)\big).$

2.2.8 Double Egg

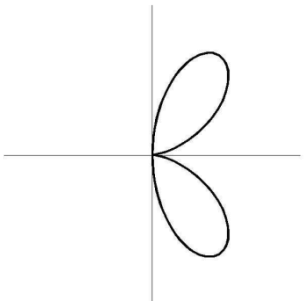


The double egg is also a conchoid of the four-leaved rhodonea curve. Equations:

$(i) r = a\cos^2\theta$,

$(ii)(x^2 + y^2)^3 - a^2x^4 = 0.$

2.2.9 Double folium

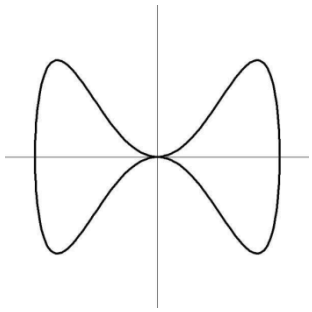


A two-lobed case of the general folium curves.
Equations:

$(i) r = 4a\cos\theta\sin^2\theta$,

$(ii)(x^2 + y^2)^2 - 4axy^2 = 0$.

2.2.10 Dumbbell curve

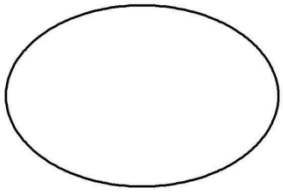


The dumbbell curve is the sextic curve. Equations:

$a^2(x^4 - x^6) - a^4y^2 = 0$, where a is a constant.

It has area $A = \frac{1}{4}\pi a^2$.

2.2.11 Ellipse

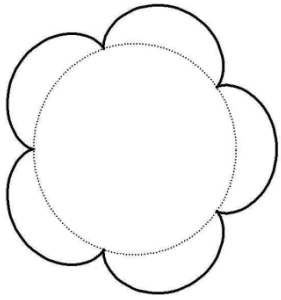


A curve such that the sum of the distances of from any point on the curve to two foci is constant. Equations:

$(i) r = \frac{1}{1+\epsilon\cos\theta}$, where $\epsilon \leq 0$ is a constant,

$(ii) \frac{x^2}{a^2} + \frac{y^2}{b^2} - 1 = 0$, where a and b are constants.

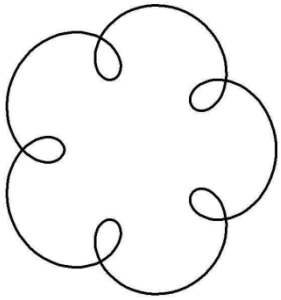
2.2.12 Epicycloid



An epicycloid is a plane curve produced by tracing the path of a chosen point of a circle which rolls without slipping around a fixed circle. Equations:

$(x, y) = (a + b)(\cos\theta, \sin\theta) + b\left(\cos\left(\frac{a+b}{b}\right)\theta, \sin\left(\frac{a+b}{b}\right)\theta\right)$.

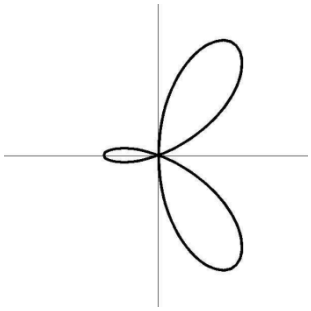
2.2.13 Epitrochoid



An epitrochoid is the locus of a point on a fixed radius at a fixed distance from the centre of a circle as that circle rolls without slipping around the outside of another circle. Equations:

$$(x,y) = (a+b)(\cos\theta, \sin\theta) + c\left(\cos\left(\frac{a+b}{b}\right)\theta, \sin\left(\frac{a+b}{b}\right)\theta\right).$$

2.2.14 Folium

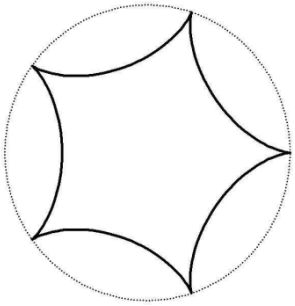


A class of leaf-shaped curves where the constants a and b determine the relative sizes and shapes of the individual leaves. Equations:

$$(i) r = -b\cos\theta + 4a\cos\theta\sin^2\theta,$$

$$(ii)(x^2+y^2)\left(y^2 + x(x+b)\right) - 4axy^2 = 0.$$

2.2.15 Hypocycloid

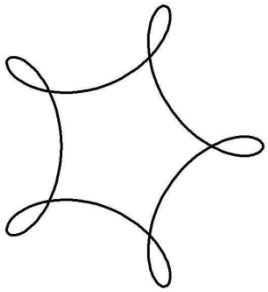


A hypocycloid is a special plane curve generated by the trace of a fixed point on a small circle that rolls within a larger circle. Equations:

$$(x,y) = (a-b)(\cos\theta, \sin\theta) + b\left(\cos\left(\frac{b-a}{b}\right)\theta, \sin\left(\frac{b-a}{b}\right)\theta\right),$$

Where $b < a$.

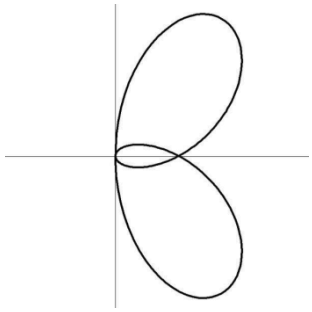
2.2.16 Hypotrochoid



A hypotrochoid is the locus of a point on a fixed radius at a fixed distance from the centre of a circle as that circle rolls without slipping around the inside of another circle. Equations:

$$(x,y) = (a-b)(\cos\theta, \sin\theta) + c\left(\cos\left(\frac{b-a}{b}\right)\theta, \sin\left(\frac{b-a}{b}\right)\theta\right).$$

2.2.17 Links curve

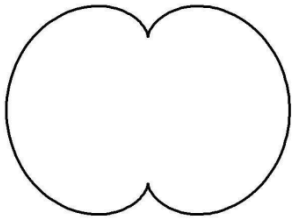


The links curve is a quartic curve. The origin of the curve is a tacnode. Equations:

$(i)(x^2 + y^2 - 3x)^2 + 4x^2(x - 2) = 0,$

$(ii)(x, y) = (\frac{6\cos^2\theta - 2\cos^4\theta + 4\sqrt{2}\sin\theta\cos^2\theta}{(1+\cos^2\theta)^2})(1, \frac{\sin\theta}{\sqrt{2}\cos\theta}).$

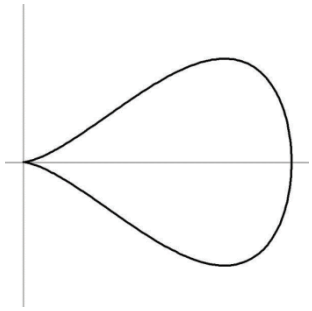
2.2.18 Nephroid



A nephroid is the 2-cusped epicycloid formed by a circle of radius a rolling externally on a fixed circle of radius $2a$. Equations:

$(x, y) = 3(\cos\theta, \sin\theta) + (\cos 3\theta, \sin 3\theta)$

2.2.19 Piriform

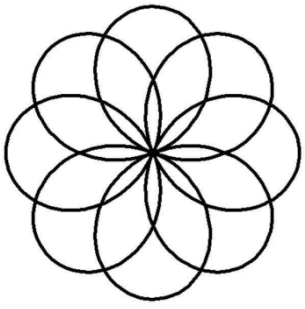


A quartic algebraic curve also called the peg-top curve. Equations:

$(i)bx^3(2a - x) - a^4y^2 = 0,$

$(ii)(x, y) = (1 + \sin\theta)(a, b\cos\theta);\ -\frac{\pi}{2} < \theta < \frac{3\pi}{2}.$

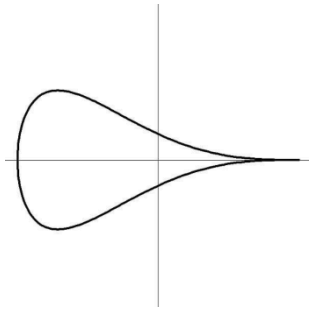
2.2.20 Rhodonea curves



Because of its flower-like form, the rhodonea curve is also called the rose or the rosette. It is a hypocycloid for which the amplitudes of the two terms of the parameter equation are equal. Equations:

$r = a\sin(b\theta)$, or $r = a\cos(b\theta)$.

2.2.21 Teardrop curve

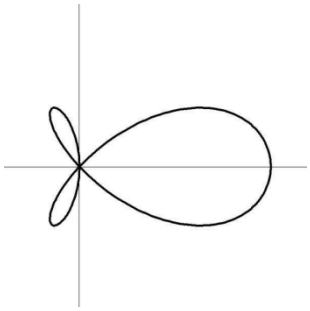


The curve is a generalization of the quartic piriform. A piriform is the curve that results when the parameter m has the value 3. Equations:

$(i)(x, y) = \left(\cos\theta, \sin\theta\cos^m\left(\frac{\theta}{2}\right)\right)$,

$(ii)(a + x)(a - x)^{m-1} - (2a)^m y^2 = 0$.

2.2.22 Trefoil curve

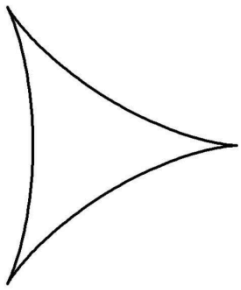


A trefoil curve is a three-lobed quartic curve. It is also simply a modification of the folium with a=1 and b=2. Equations:

$(i)x^4 + x^2y^2 + y^4 = x(x^2 - y^2)$,

$(ii)r = \frac{4\cos\theta\cos 2\theta}{3+\cos^2 2\theta}$.

2.2.23 Tricuspoid

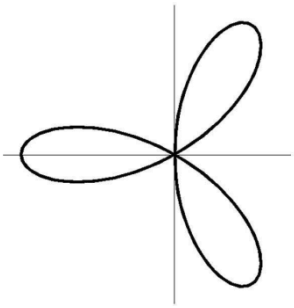


A tricuspoid, also called deltoid, is a three-cusped hypocycloid. Equations:

$(i)(x, y) = 2(\cos\theta, \sin\theta) + (\cos 2\,\theta, -\sin 2\,\theta)$,

$(ii)(x^2 + y^2 + 12ax + 9a^2)^2 - 4a(2x + 3a)^3 = 0$.

2.2.24 Trifolium



The trifolium is the three-lobed folium with $b = a$, i.e., the 3-petalled rose. Equations:

$(i)r = -a\cos 3\theta$,

$(ii)(x^2 + y^2)\left(y^2 + x(x + a)\right) - 4axy^2 = 0$.

3. Design

The program should solve the following problems to make the application useful and interactive. First is the design of the graphical user interface, this includes what content the main menu and each page will contain. Second is the method to draw curves so they can be shown on screen. This application may also be able to show the whole process of plotting, thus the animation method is required in designing. Then, some small designs are needed to make the application look nice. Finally, this application need apply validation in parameter inputting. The following specifies these separately.

3.1 GUI design

In this application one page is defined for each curve, and it need include the following information: introduction that contains its history, its type and how it can be converted to other curves; equations; associated curves; Representation of the curve. Furthermore, in the associated curves section, it is better to have links to other curves, and there should be an interaction section for user to customize curve shape and control the progress when the curve is being drawn. Thus, each page is designed to be divided into two parts, the left part contains the representation and interaction section, and the rest are put at the right part.

In addition to curve pages, there should be a main page that is initially loaded when the application runs, and the user can access to every curve through this page. Because now the application has several pages, there also need to be some controls (e.g. button) that are independent of pages so the user can easily switch between different pages.

3.2 Curve drawing method

Drawing an ideal arc requires infinite points and it is impossible to achieve. As substitution, each curve is designed to be formed by short straight lines, if the number of lines is considerably big and each line is short enough, then human eye will not recognise edges.

Moreover, since the screen is constituted of finite pixels, it cannot perfectly show curving lines, but by choosing the adjacent pixels to represent the path, human eyes are not sharp enough to see the difference. Still, this requires algorithms to choose pixels from given end coordinates of straight lines. Also, because the thickness of curves is larger than 1 pixel, some algorithm that figures out how to change depth of marginal pixels to make the rendered path looks natural to the user is required as well. Luckily, most GUI toolkits have already done these jobs for programmers.

3.3 Animation realization

Some toolkits have already been integrated with some animation functions for programmers to use. However, the objective of animation in this application is to draw each curve line by line from nothing, while almost all animation function that are already to use can only change the form or position of an existing object. Thus new functions need be defined to implement the animation of drawing. The basic idea for achieving this is greatly shower the drawing method. To be specific, this application applies a timer and sets its interval. Each time the timer ticks, the drawing function plots another straight line, until the whole curve is rendered. Because the interval is very short, and it takes much less time to draw a line, so to the user the process seems to be consecutive.

3.4 Beautifying

As an application with graphical user interface, it is necessary to use some skills to beautify its appearance because people are more willing to try attractive programs, and for curves learners a good looking application may let them study with better mood, or hopefully study longer. This application abandons the traditional window form and cancels title bar, but redesign the window to make it friendlier. Furthermore, scroll bar is designed to only appear when mouse is over or when the user scrolls mouse wheel. Also since this application may contain many buttons, their design need to be more scrupulous. Not only customised styles are designed for the buttons, but the buttons are defined to be ‘active’. Their appearances changes when mouse

moves over them, and when mouse leaves their appearances change back again. Their size will change when mouse clicks.

3.5 Validation

To make sure the parameters that the user input are legal and will not cause the generated curve exceed the coordinate system, this application need be integrated with verification mechanism that checks the inputted numbers. The system will not draw curve if there is number considered as illegal but inform the user to input another acceptable number by a message (e.g. in a tooltip). Because curves differ in shape, so their parametric value ranges differ too. Thus multiple validation rules are needed.

4. Structure

4.1 User Interface

Based on the design, the graphical user interface contains only one window and the structure of the main window is shown in Figure 1. The global controls (i.e. global buttons) are defined straightly in the main window and occupy some space, and another large partition of the window is used to hold switchable pages.

Each curve page can also be separated into two parts. The lift side part has a rectangular coordinate system with curve’s shape on it and some controls for user to input parameters, choose to see the curve immediately or slowly in animation, change drawing speed, and select to see some auxiliary line and shape or not. The right side part is more descriptive and contain introduction, equations and associated curves list.

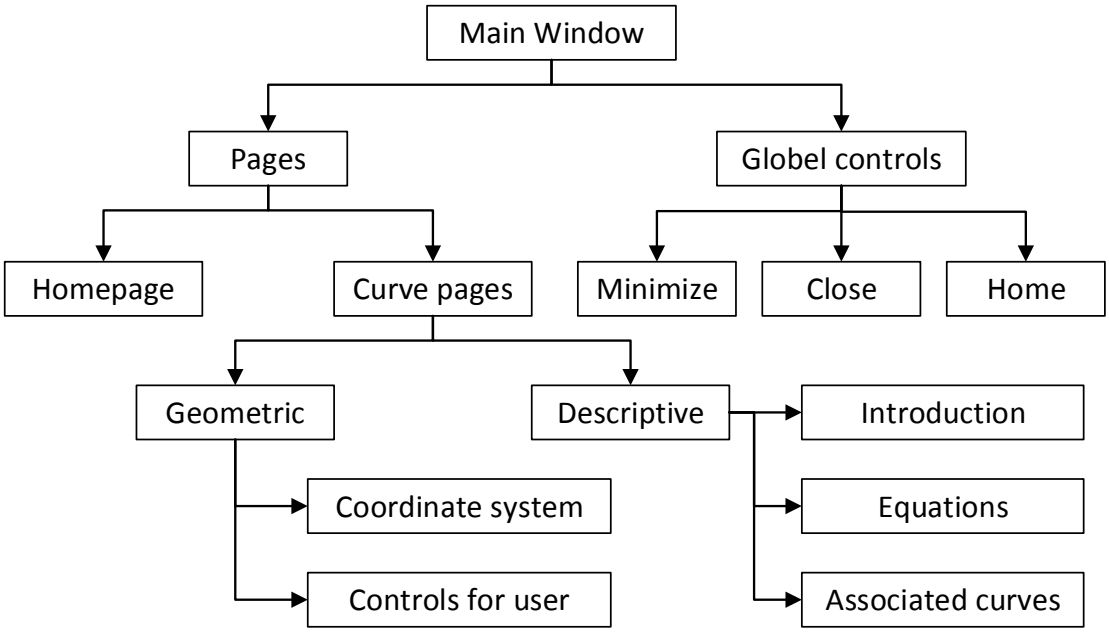


Figure 1 Main Window Structure

There is also a special ‘dictionary’ that contains styles of different widgets like scroll bars and buttons, it describes how these styles are designed. This dictionary together with the main window constitute the graphical user interface of the application, as Figure 2 shows.

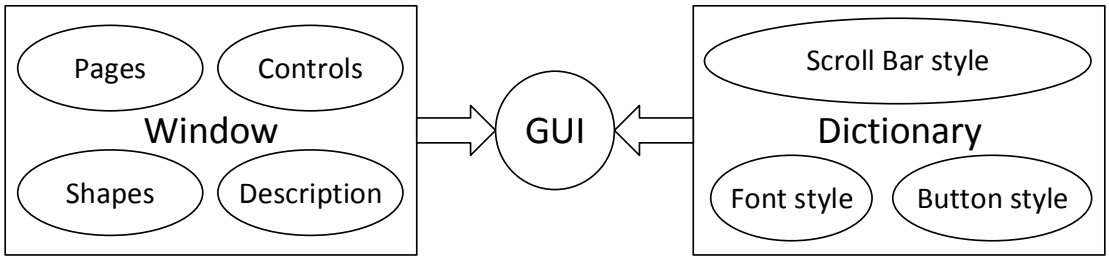


Figure 2 GUI Constitution

4.2 Functions

Figure 3 lists all the functions or methods used to implement the application program.

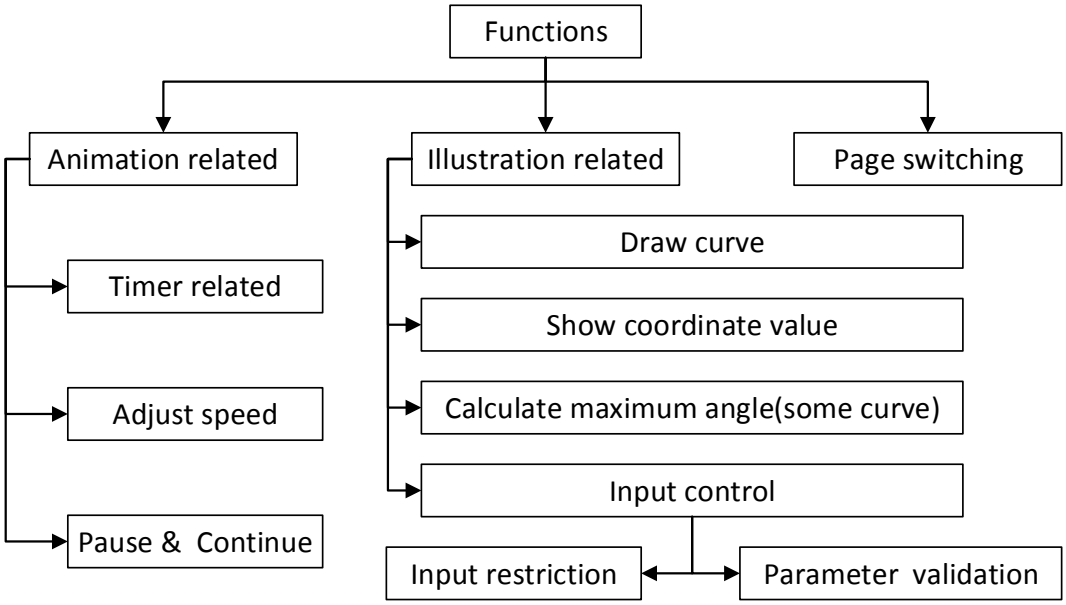


Figure 3 All Functions

The function that calculates the maximum angle is required for drawing some curves. For example, the equation for Rhodonea curves is $r = a\cos(b\theta)$, here a determines curve's size while b determines curve's shape, and in this application the user is free to set different a and b values. If b is an even integer, the maximum angle $\theta_{max} = 2\pi$, if b is odd, the maximum angle $\theta_{max} = \pi$. If b is not an integer, the calculation will be more complex, and this function will be discussed in detail in the implementation chapter. Thus, θ_{max} is not invariant with different b value, so the program need decide θ_{max} according to current b value.

Functions of input control consists of two parts. The input restriction function only allows the user to input numbers and decimal point, and the parameter validation function checks if the number that user inputs is valid, if not the function will send a message to the user.

Other functions in Figure 3 are quite straightforward, so it seems unnecessary to introduce their uses one by one.

5. Implementation

This chapter explains in detail on how the application is actually accomplished, including the programming framework and language used and the realisation of the user interface and its functionality. The order of

introduction in this chapter follows the figure structures of the previous chapter to make it clear.

5.1 Introduction to WPF

This application is implemented using the UI framework as known as Windows Presentation Foundation. ‘Windows Presentation Foundation (WPF) is a next-generation presentation system for building Windows client applications with visually stunning user experiences.

The core of WPF is a resolution-independent and vector-based rendering engine that is built to take advantage of modern graphics hardware. WPF extends the core with a comprehensive set of application-development features that include Extensible Application Markup Language (XAML), controls, data binding, layout, 2-D and 3-D graphics, animation, styles, templates, documents, media, text, and typography’ (Microsoft Developer Network). WPF is included in the Microsoft .NET Framework, thus applications built in WPF can incorporate other elements of the .NET Framework class library.

Similar to ASP.NET, a WPF application consist of both markup and code-behind. The markup language is called Extensible Application Markup Language (XAML), normally the markup language is used to design the appearance of an application: ‘XAML is an XML-based markup language that is used to implement an application's appearance declaratively. It is typically used to create windows, dialog boxes, pages, and user controls, and to fill them with controls, shapes, and graphics’ (Microsoft Developer Network).

WPF supports C# and Visual Basic as managed programming languages that is associated with markup, these languages in WPF are also called code-behind, and normally code-behind implements functionality of an application, like handling events triggered by the user and doing logical reasoning work. This application is written in C#.

5.2 GUI implementation

5.2.1 Layout of main window

In this application the main Window, also the only window is named ‘MainWindow’. As Figure 4 shows, this application applies one way to design customised window. First of all, the default window style is canceled, then a ‘Canvas’ control with length and width 800*600 is defined and all other objects of the window are put inside the Canvas. It looks like the original window is replaced by a canvas. Within the Canvas, a ‘Rectangle’ shape of the same size as the Canvas is inserted as the frame of the window. It is filled in self-defined grey, and its stroke is set black. WPF also provides developers with properties named ‘RadiusX’ and ‘RadiusY’ that set x-axis and y-axis radius of an ellipse that is used to round the corners of the rectangle, and these are used in this Rectangle.

Figure 4 Layout of MainWindow

Another features of the MainWindow is that its ‘MouseLeftButtonDown’ event is handled and the ‘DragMove()’ function is called when mouse left button is down. As a result, the window is allowed to be dragged by a mouse with its left button down over an exposed area of the window's client area. Also, its background is allowed to be transparent, while its transparency is set to be pretty low so the window is still clear to see.

Because the MainWindow does not apply the default style, so the title bar is gone as well. For this application a title bar is really not necessary, however,

the control buttons at the top right corner are required in almost every application. Thus buttons such as close and minimize need be defined manually in the markup.

The main window is divided into three parts by a ‘Grid’ control. The top row contains a ‘Textblock’ whose content is ‘Curves’ and two ‘Button’ controls named ‘minimizeButton’ and ‘closeButton’. The uses of the two buttons are minimizing and closing the application respectively. When the minimizeButton is clicked, the 'State’ property of the window is set to be ‘WindowState.Minimized’, and when closeButton is clicked, the window calls its ‘Close()’ function that triggers closing event.

The bottom row is arranged to be inserted with a ‘ContentControl’, which is named ‘changeableControl’. A ContentControl can only contain one object as its content, but the content can be any type of common language runtime object or a UIElement object. To be specific, the content of a ContentControl can be an object such as a ‘Rectangle’, or a ‘Button’ or even a ‘Window’. Indeed most of controls in WPF are inherited from the ContentControl class, and for these controls setting different objects as their contents improves their appearances or enriches their functionality. Because normally the content of a control is also a control, and it can contain its own content, this allows the graphic user interfaces programmed in WPF to be complex and powerful. But ContentControl is not the only control model in WPF. The framework also provides other elements. For example, an ‘ItemsControl’ and its derivative classes (like ‘Menu’ and ‘TreeView’) can contain multiple contents. As the base class of many WPF controls, ContentControl is hardly used because the variety of its derivative classes is abundant enough. While the changeableControl here in the MainWindow is merely used as a vessel that provides room for a ‘UserControl’, and the UserControl it contains can be changed with the help of code-behind. UserControls are used to contain either the main menu or Introduction page for different curves.

The second row only contains a Button, this is a navigation button defined in the application for the user to go back to the home page, and is referred to as

‘homeButton’ in the context. The mechanism of navigating to different pages (UserControls) is presented specifically in 5.3.

5.2.2 Layout of UserControls

Before introducing the navigation mechanism that implementing switching between UserControls, it is better to introduce the UserControls.

A UserControl also consists of both a XAML and a code-behind file. It is a reusable component that can be added with multiple existing controls. For instance, a UserControl can contain other controls like resources, and animation timelines. UserControl is similar to Window, but still their different root elements. Unlike Windows, UserControls do not have style. Thus normally UserControls are used as extensions in customising and designing functionality and are put inside a Window.

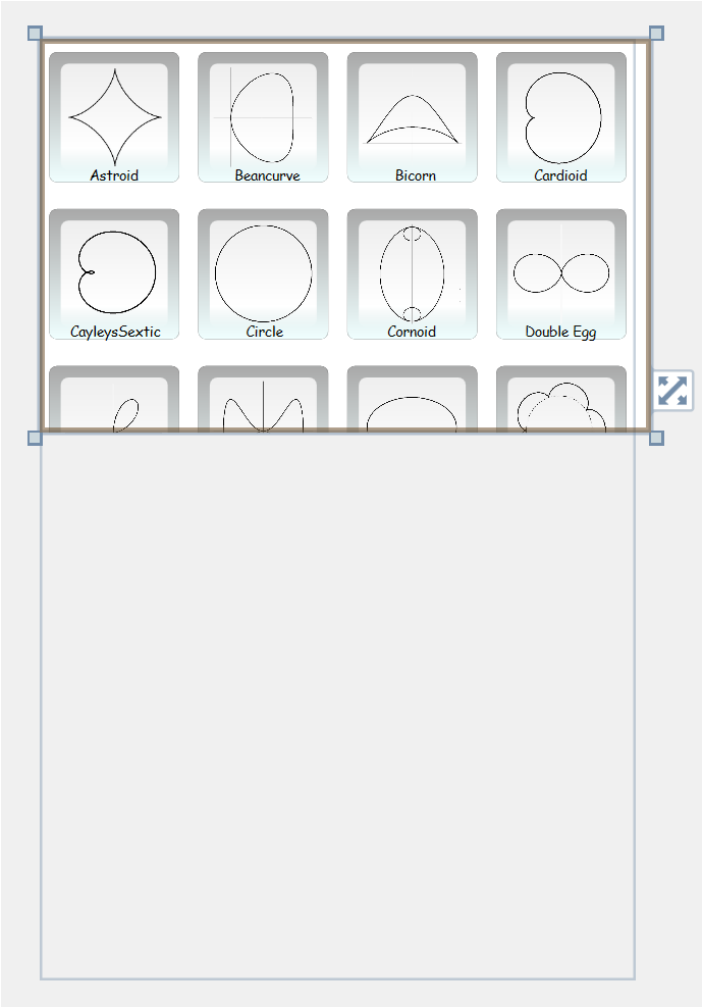


Figure 5 Homepage UserControl

And in this application, a UserControl can be simply regarded as a page. The Figure 5 below shows the layout of the default UserControl that system loads into changeableControl when the application is opened. This UserControl provides a menu of all the curves covered in the application and is composed of 24 buttons. Because the UserControl space is not sufficient to display all the buttons in the meantime, these buttons are placed inside a 'ScrollViewer' control.

A ScrollViewer encapsulates horizontal and vertical 'ScrollBar' elements and a content container (here is a 'Grid' that contains buttons in order) so as to display other visible elements in a scrollable area. 'The ScrollViewer control responds to both mouse and keyboard commands, and defines numerous methods with which to scroll content by predetermined increments' (Microsoft Developer Network). Comparing to ScrollBar, ScrollViewer is easier to use because the developer need not build a custom object for content scrolling as a ScrollViewer can automatically measure the size of its content container and adjust the ScrollBar it encapsulates.

The layout of UserControls for each curve is very similar. Here the UserControls of Teardrop curve is used in illustration, as shown in Figure 6.

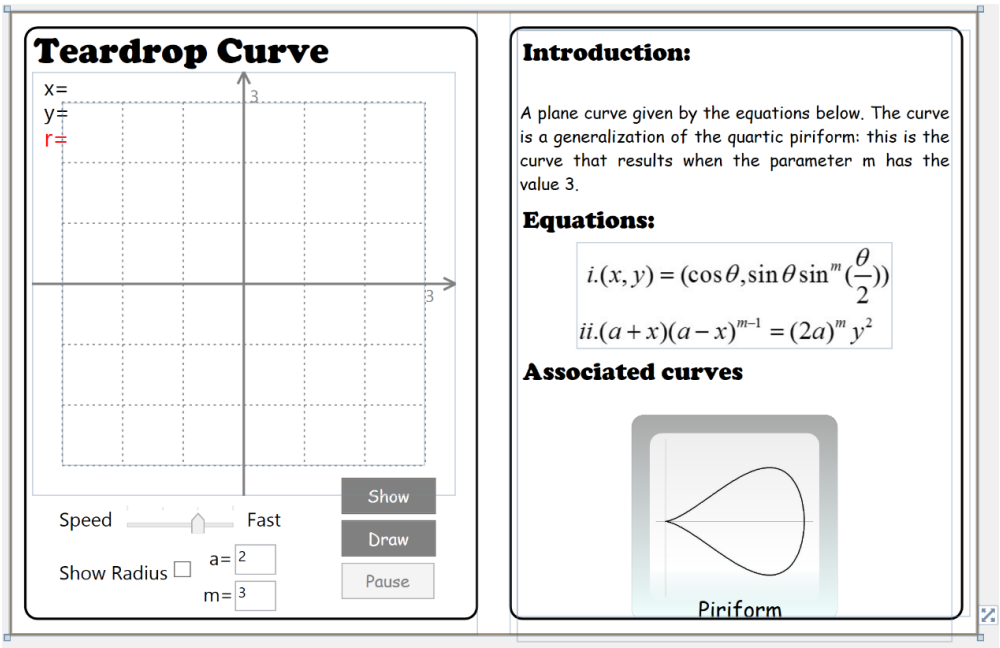


Figure 6 Curve UserControl example (Teardrop)

The UserControl on the whole is divided into three columns, while the middle column is narrow and is defied simply to separate the left and right side. The right side is arranged using a ‘StackPanel’ inside a ScrollViewer. A StackPanel arranges child elements into a single line that can be oriented horizontally or vertically, and here uses the default orientation which is "Vertical". The words in the right side is put in TextBlocks. The equations are presented as a picture, and some buttons are put after the ‘Associated curves’ tag, their style is the same as the buttons’ in the homepage.

Below the name tag at the top of the left side, a coordinate system is drawn. The coordinate system is implemented in a Canvas, which contains many ‘Line’ shapes. There are three TextBlocks at top left of the coordinate system, there uses are showing the current x value, y value and radius value when the curve is being drawn. Thus their contents need change continually during the process. To achieve this, their contents need be set by the drawing function of code-behind every time a new line is drawn. Initially these three TextBlocks are hidden, the ‘x=’ TextBlock and ‘y=’ TextBlock are only visible when the drawing process is on, and the ‘r=’ TextBlock is only visible when the ‘Show Radius’ Checkbox at bottom left corner of the UserControl is checked. There is another Canvas defined within the coordinate system Canvas, this is the drawing board for generating curve dynamically.

Under the coordinate system there are some widgets for user include three Buttons ‘Show’, ‘Draw’ and ‘Pause’, a Slider that set drawing speed and its stating TextBlocks, TextBoxes for user to input parameters, and ‘Show Radius’ Checkbox mentioned above. The Draw button starts the animation of drawing the curve, while when the Show button is clicked, the curve is displayed instantly at the canvas. The Pause button is valid only during the drawing process and its use is surely pausing the animation. When clicked, its label (in fact is its ‘Content’ property) changes to ‘Continue’, when it is clicked again its label return to ‘Pause’ and the animation continues. The implementation and logic of their click events will be specified at 5.3. For other widgets, they provide different methods of interaction with the user, and their properties change when the user input different number into the TextBox or

drag the Slider to different position. These properties can be read in code-behind as parameters for different functions.

5.2.3 Design of buttons

It is a complicated work to make a button nice and interactive. In this application some attempts are tried. In WPF, each control has a 'ControlTemplate' that 'specifies the visual structure and behavioral aspects of a Control that can be shared across multiple instances of the control' (Microsoft Developer Network), so does Button control. A ControlTemplate contains two major properties: VisualTree and Triggers. The VisualTree describes the appearence of the control, and the Triggers property contains a collection of TriggerBase objects that apply property changes or perform actions based on specified conditions. ControlTemplates can be rewritten to customize controls. The introduction begins with the easiest ones.

The left sub-figure of Figure 7 shows the appearances of minimizeButton and closeButton. Their VisualTree contains a Canvas. Both buttons have a rectangle with corners rounded in their canvases, the rectangles are filled in white and have black stroke. Further the minimizeButton contains a transverse line in the middle of its canvas, while the closeButton contains two line to shape a cross. Only one trigger is defined in these two buttons, that is when mouse is over, the 'Fill' property of the rectangles are changed to other colors. For minimizeButton the filling color of rectangle changes to gray and for closeButton red, as shown in the middle and right sub-figure of Figure 7.

Figure 7 minimizeButton and closeButton

The design of homeButton is a bit gaudier. Its size is set 50*50 and its VisualTree contains a Grid to hold other shapes. The house-like shape is defined manually using a 'Polyline' shape. A Polyline is used to draw a custom polygon, it connects every pair of neighbouring points defined in the 'Points' property string with a line. Points property is a string that consists of x and y coordinates separated by blank space. Coordinate pair starts with 0 at top left and reaches the maximum value at bottom right with width and height

size of the button. The coordinate values of this polygon is "18,12 25,5 45,25 40,25 40,45 29,45 29,30 21,30 21,45 10,45 10,25 5,25 12,18 12,10 18,10 18,12". A Rectangle that has the same size as the button is also added into the Grid. The aim of adding this Rectangle is to enlarge the available area of the button, thus the Rectangle is transparent.

homeButton contains a ToolTip, it shows up when mouse is over. This ToolTip in the homeButton only includes a TextBlock with ‘Return to homepage’. Other actions performer when button property changes are designed in Triggers. Rather than simply changing Fill property to another color, here a LinearGradientBrush is applied to fill the Polyline area with a linear gradient from transparent to skyblue. Also, this process is animated using ColorAnimation. ColorAnimation ‘animates the value of a Color property between two target values using linear interpolation over a specified Duration’ (Microsoft Developer Network). Because for ‘IsMouseOver’ property of homeButton animations are applied, both its ‘EnterActions’ and ‘ExitActions’ need be defined using different ‘Storyboards’, which contains the applied animations that WPF provides. As a result, when mouse is over the homeButton, its filling color is gradually interpolated from left sub-figure of Figure 8 to its middle sub-figure. When mouse leaves homeButton, its filling color is gradually altered back to white.

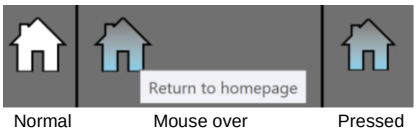


Figure 8 homeButton

Another designed trigger of homeButton checks its ‘IsPressed’ property. When the property is ‘true’, homeButton’scale is transformed to 95%, as the right sub-figure of Figure 8 shows.

The templates of curve buttons is defined to be a bit more complex. Instead of being defined straightly inside the button in the markup of MainWindow, these templates are defined in a dictionary file and can be applied by UserControls as static resources. However, the techniques used are quite similar as

mentioned above. In Figure 9 the button of Astroid is used as example to show how the color or size is changed when mouse is over or it is pressed.

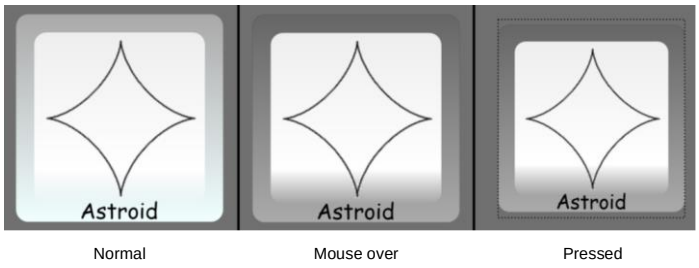


Figure 9 Curve Button (Astroid)

To ensure the curve button is unaffected by the transparent backgurand, a Border that contains a black Rectangle is deined as the background of the button. In addition to the border, a curve button consists of two Rectangles, a Label and an Image. Both Rectangles applies LinearGradientBrush.

5.2.4 Design of ScrollViewer

To be honest, the ScrollViewer style in the application is referenced from the Internet (CSDN.net) and is modified to better suit this application. Because ScrollViewer encapsulates ScrollBar and its appearance relies on its ScrollBar's appearance, thus changing the style of the ScrollViewer is equal to changeing style of its ScrollBar, in which the ControlTemplate of ScrollBar need be defined. So it is clearer to first design a style for ScrollBar and apply this style in designing the ScrollViewer's style. While ScrollBar of ScrollViewer also contains a 'Track' control that consists of the following visible elements: Thumb, RepeatButton (including vertical page button and horizontal page button). Thus it is clearer to define its components' style separately and apply them in designing the ScrollBar's style. Figure 10 shows the employing relationship of styles defined eventually for ScrollViewer. All of these styles are put in the dictionary file as well.

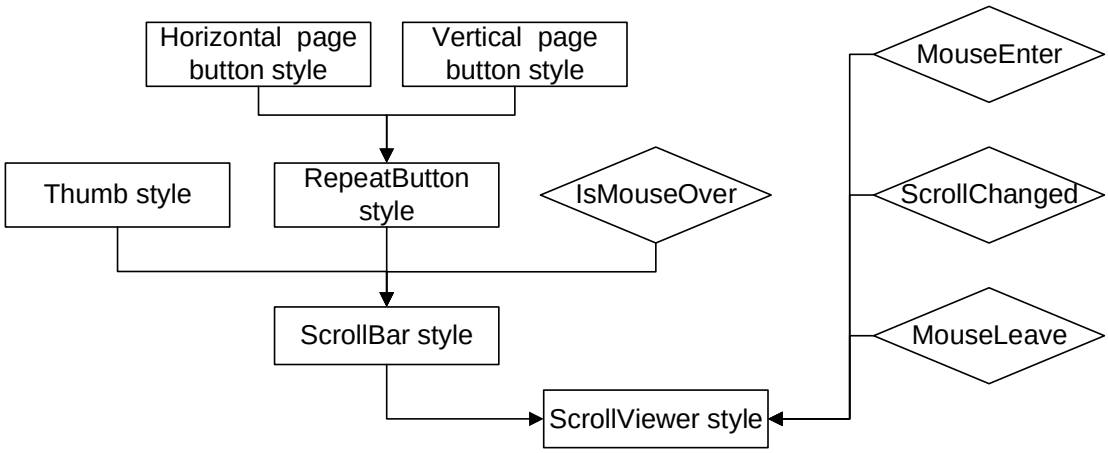


Figure 10 ScrollViewer style

Figure 10 also presents triggers applied in ControlTemplates of ScrollViewer's style and ScrollBar's style in diamond borders. In the ScrollBar style, the Track of the ScrollBar is defined to be enabled only when mouse is over, while in the ScrollViewer style, animations are defined when mouse enters, leaves and the scroll is changed. As a result, the ScrollBar in the application is initially hidden, but slowly appears when either mouse is over it or the mouse is wheeled causing scroll to be changed. This is shown in Figure 11. When mouse leaves the ScrollBar or mouse wheel stops being scrolled, the ScrollBar slowly disappears again.

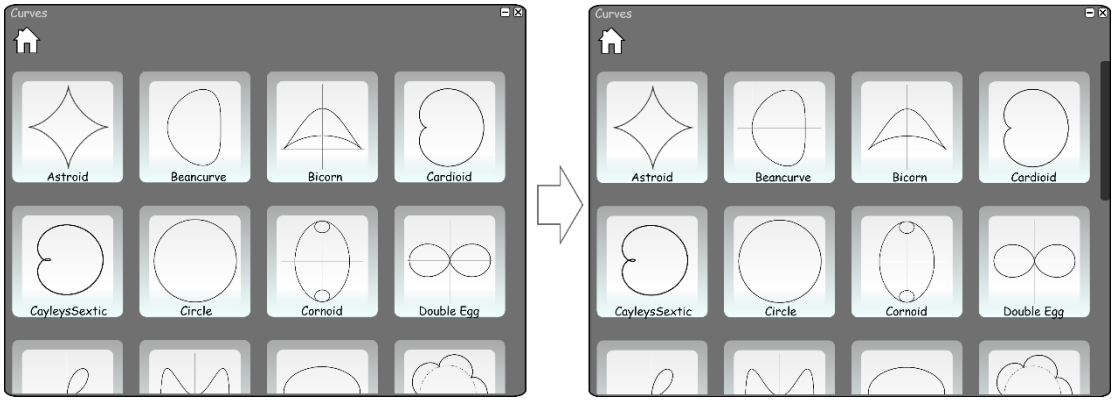


Figure 11 ScrollBar appears

In this application the width of MainWindow and all UserControls is fixed, so no horizontal ScrollBar has ever been generated by ScrollViewers. Still style of horizontal ScrollBar is defined, because the designed style for ScrollViewer is regarded as a resource, thus its portability should be guaranteed.

5.3 Functions implementation

This section specifies how code-behind is written to realize the functions of the application. Each following subtitle may unnecessarily on behalf of one specific function body, because usually it require multiple functions that work together to realise one objective.

5.3.1 Switching between UserControls

The easiest way to realise the switchover within different pages is implementing the main window as a ‘NavigationWindow’, which can hold one page at a time. While pages are defined separately. Thus, by simply calling the function ‘NavigationService.Navigate()’, the NavigationWindow changes the page it holds. Another advantage of applying NavigationWindow is that it is easy to set transition effect, such as fade in and fade out.

However, this method seems to be incompatible with the timer that is used in curve drawing, and the navigate action change content of the whole window, this may also cause inconvenience in defining global controls. Thus, still the commonest window ‘WPF Window’ is used in this application to hold everything, and each curve is introduced in a ‘WPF Usercontrol’ rather that a ‘WPF Page’ that is introduced above, though this method is less smart.

In fact all the UserControls are declared with an object (or instance) in the MainWindow code-behind, they are initialized when corresponding button is clicked. This application applies delegation to handle button click events. To put it simply, the code-behind of the MainWindow has stated what function to run when which button is clicked, and all buttons of all UserControls are instructed. This allows the user to switch between different UserControls when using the application. While in order to make the code-behind easy to read, these instructions are arranged separately.

When the MainWindow is loaded, it initializes a homepage UserControl and set it as changeableControl’s content. Thus the instructions of buttons homepage UserControl contains are written straight in the main function.

Other UserControls can only be accessed through clicking corresponding buttons. When a curve button is clicked, its event handling function embodies a UserControl and set it as changeableControl's content, then it delegates all the click events of the buttons this UserControl contains to their corresponding event handling functions. Here instructions are arranged according to curves, an event handling function for a curve only contains instructions of the buttons this curve's UserUserControl contains.

For the home shaped global button, its button click event is easy to handle. Just setting changeableControl's content to be the homepage UserControl and this button will navigate to the home page regardless of what current UserControl is displayed.

There may remain a question: a new UserControl is created every time a curve button is clicked, will too many accumulated UserControls slow down system operating and lower the performance of the application? The answer is certainly no, because the framework automatically recycles the resource for the former UserControl when the content of the changeableControl has been changed.

5.3.2 Timer and UserControl variants

UserControl variants(or objects) are declared within the UserControl class but outside the functions defined with the UserControl, thus their values can be read and modified by all functions of the UserControl. Because in this application a UserControl is used for only one type of curve, it is pretty safe to define many UserControl variants, and they are convenient to use. Table1 lists the UserControl variants appear in curve UserControls.

Table 1 All UserControl variants and objects

Name	**Type**	**Usage**
timer	DispatcherTimer	A timer that interval between timer ticks and whether it is running can be set.
t	double	The duration time of animation
a, b, c, m	double	Parameters of curve Equations
r	double	Length of radius
angle	int	Current angle value
maxAngle	int	Maximum angle value
pauseControl	bool	Its value is 'true' if drawing animation is paused, or its value is 'false'
radiusIsEnabled	bool	Its value is 'true' only when curve animation is in progress
circleIsEnabled(Epicycloid, Epitrochoid, Hypocycloid, Hypotrochoid, Tricuspoid only)	bool	Its value is 'true' only when curve animation is in progress
fociIsEnabled(Ellipse only)	bool	Its value is 'true' only when curve animation is in progress
radius	Line	Radius of the current angle
gLine, bLine (Ellipse only)	Line	Two foci of the current angle
circleA, circleB (Epicycloid, Epitrochoid, Hypocycloid, Hypotrochoid only)	Ellipse	Two circles that generate the curve
radiusB (Epicycloid, Hypocycloid, Tricuspoid only)	Line	A Line between the fixed point on the smaller circle and its centre
lengthC (Epitrochoid and Hypotrochoid only)	Line	A Line between the fixed point attached to the smaller circle and its centre
source	aInfoWithValidation	An IDataErrorInfo object that implements data validation

The 'aInfoWithValidation' is a self-defined class that inherits from IDataErrorInfo. IDataErrorInfo provides custom validation support for the members defined in its object. The utilization of IDataErrorInfo will be further discussed in 5.3.9.

WPF has provided plenty of easy ways to create animation for the user interface in 'System.Windows.Media.Animation' namespace, there can be classified into three types: timelines, storyboards, and key frames. However it seems that all these methods make animation by changing some property of an object, like a Shape or a Control, this pattern might not be usable in this application because a Canvas may contain many Shapes, and the Shapes are added to the canvas as its children, while some attempts to change the canvas's Children in the definition an DoubleAnimation instance had failed.

This application instead uses DispatcherTimer to create duration. A ‘DispatcherTimer’ is a timer that ticks at a specified interval of time and is integrated into the Dispatcher queue, DispatcherTimers can also be specified at different priority. When a timer is enabled, it causes a ‘Tick’ event once at a set interval. This event is handled in the application by delegating it to a curve generating function, so the function is called every time a new Tick event is created. The major job of this function is to add a new Line into the canvas, then check if the whole curve has been generated. If it is true, the function disables the timer, if not, the function updates UserControl variants that are used to get coordinate value. The logic of this curve drawing function is specified in 5.3.3. Figure 12 shows some phases in drawing an ellipse as example.

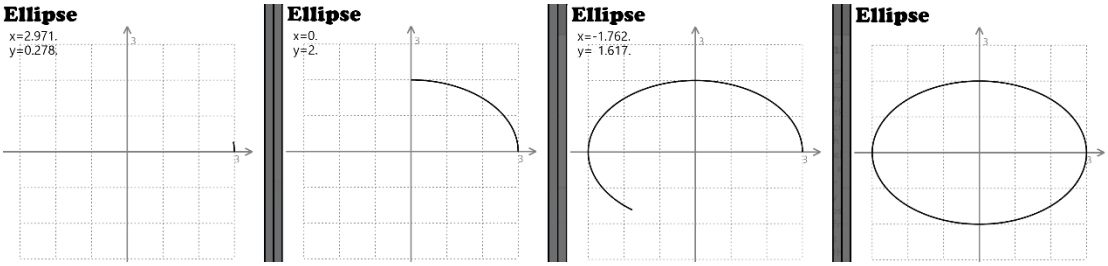


Figure 12 Draw an ellipse

Thus it is obvious that the speed of animation can be adjusted by modifying the timer’s Interval property, the longer its interval is the slower the animation will be. The variant ‘t’ in each UserControl stores how many seconds the animation will last and its value can be adjusted by the slider, so the interval can be set as $t \times \frac{\text{angle increment}}{\text{maximum angle}}$, here angle increment is the number the angle adds after drawing a line. When ‘Draw’ button is clicked, its handling function calculates the required interval in this way and sets is as the timer’s Interval property, then enables the timer. Also, once the speed slide is altered, its handling function changes t and then refreshs timer’s Interval.

In each UserControl the timer is initialized and enabled when the canvas that contains coordinate system is loaded, and its Interval is initially set 0. This means once the UserControl is opened, its timer constantly produces Tick events, thus its curve drawing function is called repeatedly without gap, until the curve totally displayed and the timer is disenabled. In a very short period

of time the curve is generated and the timer is disabled. To the user it just looks like the curve is displayed simultaneously with the page. Similarly, when the show button is clicked, the timer's Interval is also set 0 before it is enabled so the user can instantly get the curve. By simply change timer's Interval to different values, a Tick event handling function is enough to use both in drawing curve and showing curve. So there is no need to define another function that calculates all coordinates used and defines all the lines in its logic, then renders the curve at once.

5.3.3 Curve drawing function

The curve drawing functions responds to Tick events. This mechanism is stated when the UserControl is loaded. In the application this function is named 'AnimatedDraw' and is different in every curve's UserControl as the equations are different.

Moreover, different curve UserControls may contains their unique UserControl objects. These objects are normally graphics and are generated in this application to helps illustrate the features of each curve. Table 2 is a subset of Table 1, These shapes listed are defined in the code-behind for this purpose.

Table 2 UserControl shapes

Name	**Type**	**Usage**
radius	Line	Radius of the current angle
gLine, bLine (Ellipse only)	Line	Two foci of the current angle
circleA, circleB (Epicycloid, Epitrochoid, Hypocycloid and Hypotrochoid only)	Ellipse	Two circles that generate the curve
radiusB (Epicycloid, Hypocycloid, Tricuspoid only)	Line	A Line between the fixed point on the smaller circle and its centre
lengthC (Epitrochoid and Hypotrochoid only)	Line	A Line between the fixed point attached to the smaller circle and its centre

In addition to some UserControl objects, some TextBlocks are defined to show the current coordinate value or radius value, and they are defined in the markup rather than code-behind. These Shapes and TextBlocks together are regarded as auxiliary objects in the context. To improve flexibility, the visibility of these auxiliary objects is optional, and the user can choose whether to see these objects or not by controlling the CheckBox at the bottom left of each

UserControl. Here UserControls of Ellipse and Hypotrochoid are used as example, as shown in Figure 13.

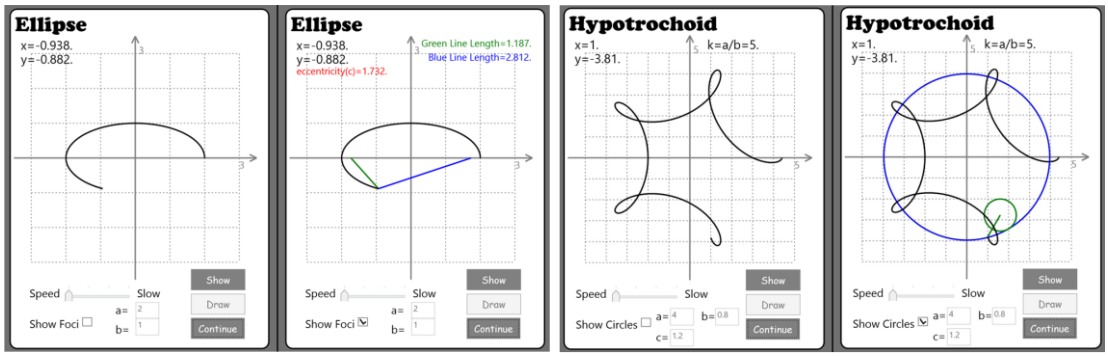


Figure 13 auxiliary objects and their visibility

During the process of generating the curve, the properties of some auxiliary objects like positions, values or shape also need be changed, and the easiest way to achieve this is to set altered properties of auxiliary objects in the AnimatedDraw function as well. However, because when ‘Show’ button is clicked a new curve will be displayed in a flash and there remains no time for the user to see auxiliary objects in this method, so auxiliary objects must not be available in showing curve. Thus, apart from the difference in setting timer interval, there is another major difference between drawing curve and showing curve. To eliminate potential confusion, a Boolean variant is used. In most UserControls this variant is named ‘radiusIsEnabled’, while for Ellipse it is named ‘fociIsEnabled’ and for Epicycloid it is named ‘circleIsEnabled’. To unify, here uses ‘objEnabled’ to represent all this kind of variants. The value of objEnabled of a UserControl is ‘true’ only after Draw button is clicked and before timer has been disabled as the curve is fully rendered. When objEnabled remains ‘false’, AnimatedDraw will not add any auxiliary object into the canvas even if the CheckBox is checked. Figure 14 is a universal chart of the fundamental logical structure of AnimatedDraw function.

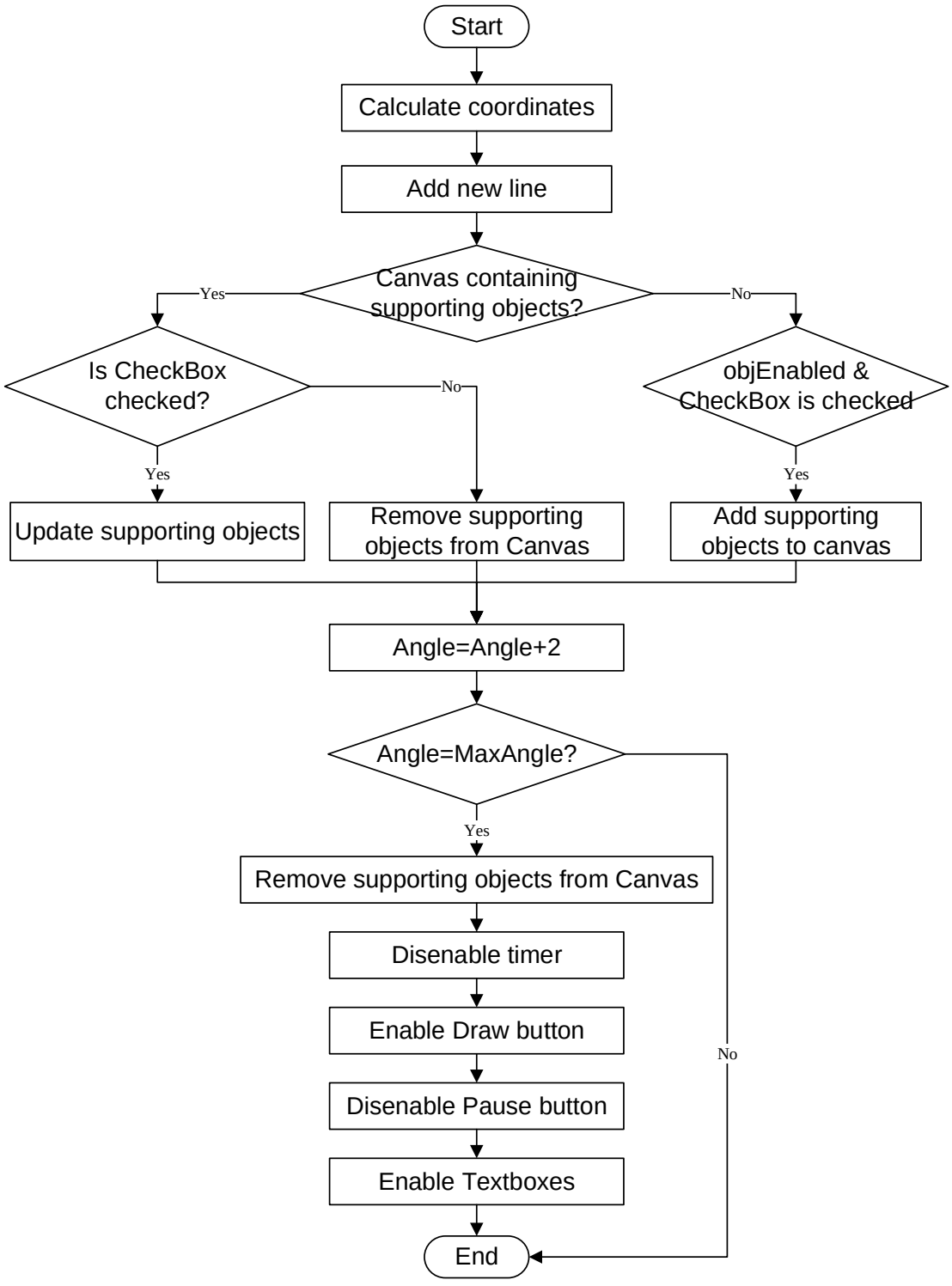


Figure 14 Flow chart of AnimatedDraw

In the function the angle increment is set at 2 instead of 1 to improve efficiency, and after testing it is confirmed that though the number of lines is decreased by half, the generated curves do not seem to have recognisable change, but still look nice.

5.3.4 Speed slider and explaining text

In each curve UserControl, the Slider is used to adjust speed of animation. Its value is defined to be capable to from 0 to 4 with 1 as small change. When its value is changed, a code-behind function is called to change t's value according the Slider's current position. To real-timely respond to Slider changes, both 'MouseLeftButtonDownEvent' and 'MouseLeftButtonUpEvent' are handled in the function, thus the value of t is updated by clicking the Slider or dragging the sliding block.

Further, a TextBlock is used to explain the current position of the slider automatically. The content of this TextBlock changes from 'Very fast' to 'Slow' when the sliding block moves from right to left. This is shown in Figure 15.

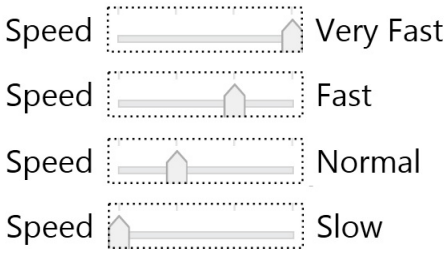


Figure 15 Respond to Slider changes

So overall the Slider event handling function does the follow job. When it is called, it reads Slider's value, then sets t's value and TextBlock's content corresponding to the Slider's value, finally updates timer's Interval using new t value.

5.3.5 Show coordinate values

A function called 'showValue' is defined to work together with AnimatedDraw. When AnimatedDraw has calculated new x and y coordinates, it sends this two values to showValue function. The showValue function has two major tasks. The first task is to adjust precision of coordinates by changing the number of significant figures so they will not be too long. The second task to use the adjusted values to update 'x=', 'y=' and 'r=' TextBlocks of the UserControl. Figure 16 is the flow chart of its logic.

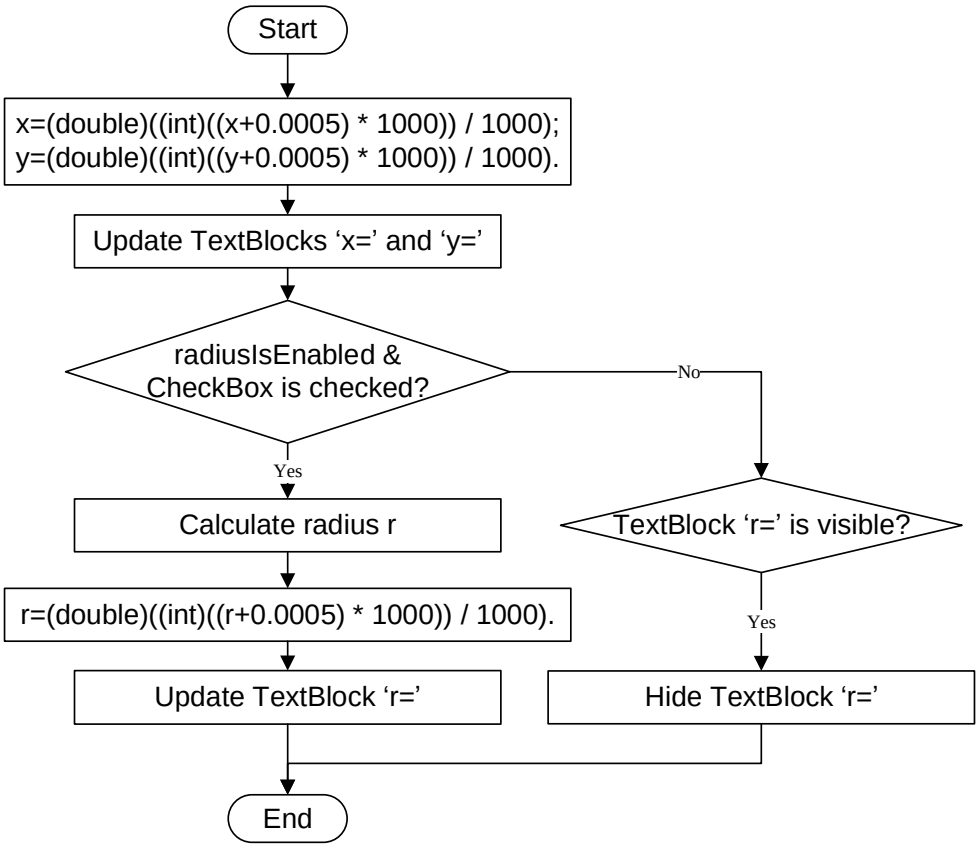


Figure 16 Flow chart of showValue

Here three decimal places is allowed to show, so the third decimal place is rounded. This function eliminates surplus decimal places by multiplying coordinate value by 1000, then change the value type from double to int. But before multiplying it is needed to add 0.0005 to the value to rounds the third decimal place. Thus only adopted decimals are put into an integer variant because when a double type variant is conversed to integer only digits before decimal point are remained. Then this integer is conversed back to double and divided by 1000. For example, the algorithm for adjusting x-axis coordinate is: $(\mathrm{double})((\mathrm{int})((\mathrm{x} + 0.0005) * 1000)) / 1000)$.

As shown in the chart, showValue only calculates radius value and updates ‘r=’ TextBlock when during drawing process of curve and the CheckBox is checked. If CheckBox is unchecked during the process, the function can hide ‘r=’ TextBlock in time. Because an Ellipse has two radiuses, so the the showValue of its UserControl is a bit different. While showing radius is meaningless for some curves thus in these curve UserControls radius is not calculated in showValue function and ‘r=’ TextBlocks is withdrawn.

5.3.6 Get maximum angle

For some curves, the maximum angle to draw a closed path is not constant. Further, ways to decide their maximum angle can be different. For the curves whose maximum angles need be calculated, a function named ‘getMaxAngle’ is defined in their UserControls. This function is called when Show button or Draw button is clicked. It receives parameter values from the TextBoxes, and after applying some algorithms it sets ‘maxAngle’ variant using calculation result. The algorithms shall be defined on the basis of the curve’s feature. This getMaxAngle function is very important because without calculating and setting maximum angle the drawing process would never ends, or if the maximum angle is set to a fixed value the drawing process might ends before a closed curve is created. Here Epicycloid and Rhodonea curve are used as representative examples of getMaxAngle function.

The form of an epicycloid can be described as tracing the path of a chosen point of a circle(whose radius is b) which rolls without slipping around a fixed circle(whose radius is a). The circle with radius b is called epicycle. At 2.2.12 its parametric equation is given and is shown again below. To explain the parameter θ in geometric way, it is the angle of a line from origin to the fixed point.

$$(x, y) = (a + b)(\cos\theta, \sin\theta) + b\left(\cos\left(\frac{a+b}{b}\right)\theta, \sin\left(\frac{a+b}{b}\right)\theta\right).$$

Thus the maximum angle depends on the relation of the perimeter of the two circles. If the perimeter of the fixed circle is an intact time as the perimeter of the epicycle, the chosen point returns to its initial position when angle θ reaches 2π. In this case the maximum angle is 360 degree. But if not, the epicycle continues rolling another 360 degree. Because the formula of calculating perimeter is $2\pi r$, thus perimeter ratio of this two circle is $\frac{2\pi a}{2\pi b}$, which equals to $\frac{a}{b}$. If $\frac{a}{b}$ is not an integer, the epicycle rolls another 360 degree, and the ratio becomes $\frac{a}{b} + \frac{a}{b} = 2\frac{a}{b}$. If $2\frac{a}{b}$ is still not an integer, the epicycle continues to roll 360 degree, and the ratio becomes $3\frac{a}{b}$. The rest can be done

in the same manner. In epicycloid's UserControl, the getMaxAngle function works following the procedures as shown in Figure 17.

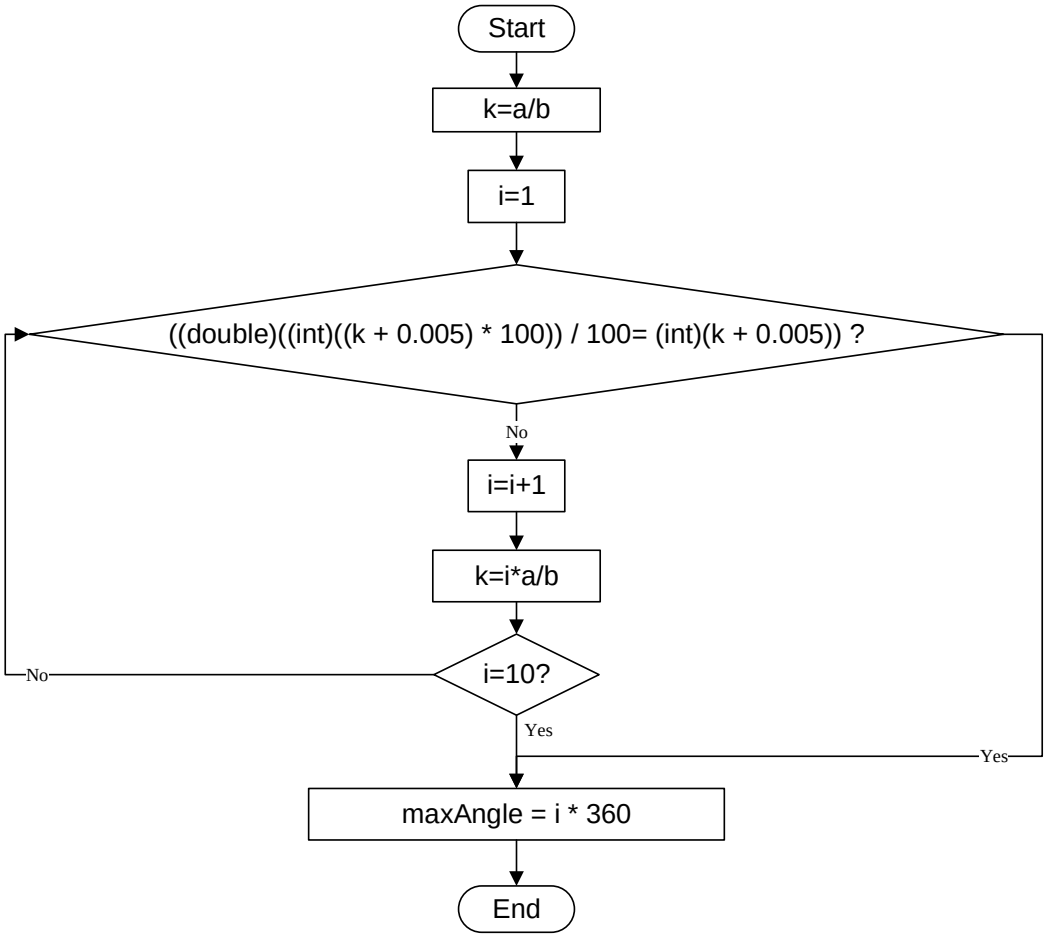


Figure 17 getMaxAngle function of epicycloid

This getMaxAngle function allow error within two decimal places. For example, if at a time k equals to 3.0049, then k is regarded 3 in the function. Similar to the instruction used in showValue, '((double)((int)((k + 0.005) * 100)) / 100' rounds k to two decimal places and is compared with the number of k before decimal point, in this way k is checked if it is an integer or not. Also, the function sets an upper limit of i, which is 10, so the maximum angle will not exceed 3600 degree. In fact without this limit the drawing process will end within 100*360 degree since error is allowed in the function, but still it is a lot of calculating and plotting. When lines are too dense, all the user can see is a block of black color. Even worse, the system makes mistakes in simply adding two double variants. Thus setting a limit is needed. In Figure 18, a is

set at 3.0 and b is set at 0.7, the curve is fully rendered and the drawing process ends automatically when angle reaches 2520 (7×360) degree.

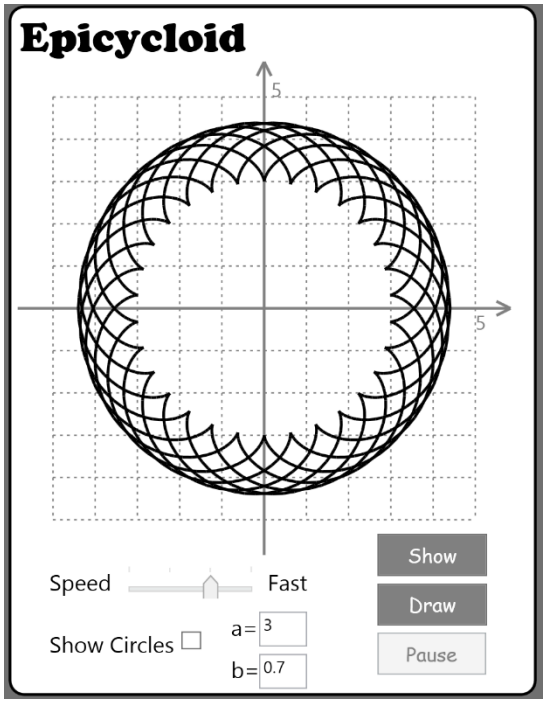


Figure 18 epicycloid with $a = 3$ and $b = 0.7$

The principle of getting maximum angle for a Rhodonea curve is similar. As mentioned in 4.2, the equation for Rhodonea curves is:

$$r = a\cos(b\theta).$$

In this equation a determines curve's size and b determines curve's shape. If b is an even integer, the maximum angle $\theta_{max} = 2\pi$, if b is odd, the maximum angle $\theta_{max} = \pi$. If b is not an integer, the getMaxAngle function use similar way as epicycloid's to get maximum angle. So all these three situations need be considered in defining the getMaxAngle function. As Figure 19 shows how it works.

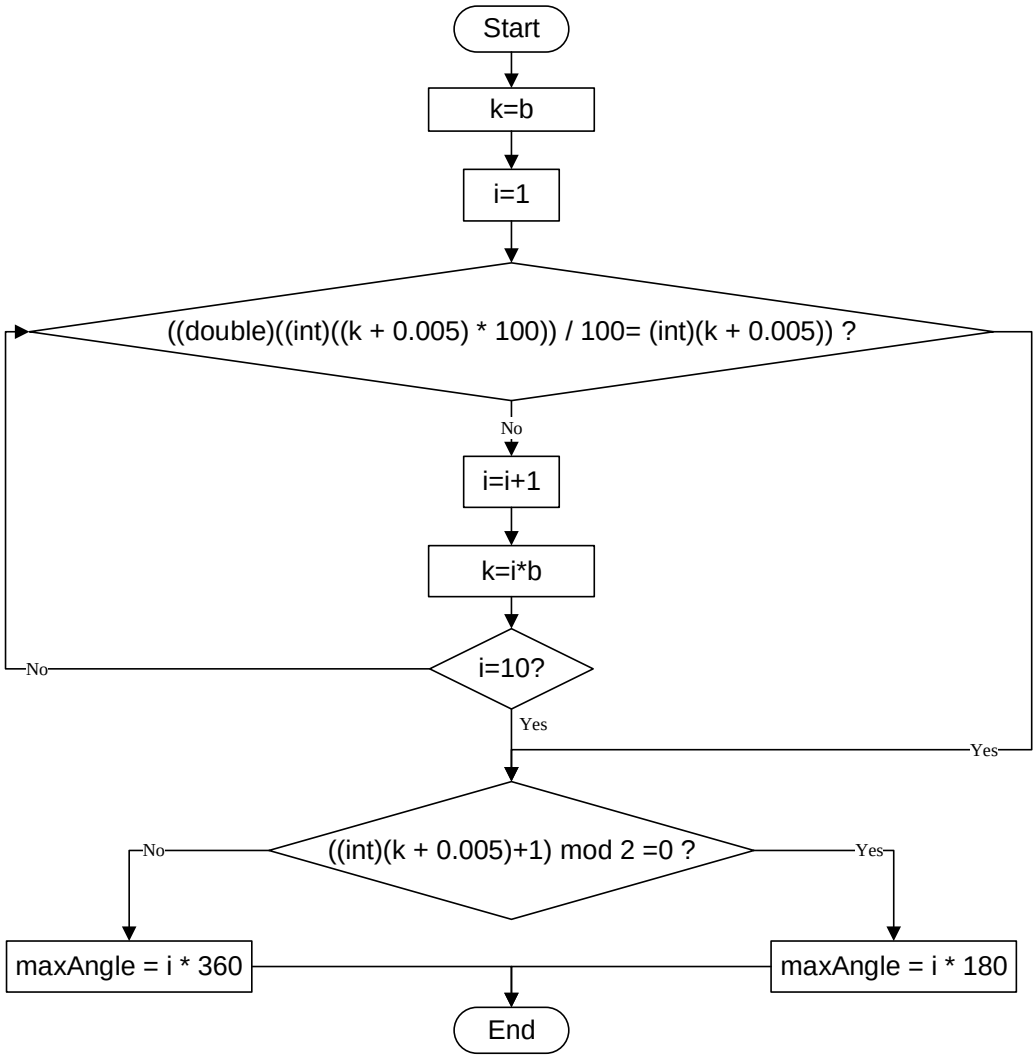


Figure 19 getMaxAngle function of Rhodonea curve

5.3.7 Pause and Continue button

Initially the Pause button is dieenabled, and its initial content is ‘Pause’. Only when Draw button is clicked it is enabled, and it is disabled again once the angle reaches maximum and the drawing ends. If during the animation the Show button is clicked, this button will also be disabled together with a curve instantly shown on the canvas replacing the unfinished one, and the content of Pause button will be set as ‘Pause’. A Boolean variant ‘pauseControl’ is defined to help in coding. Initially its value is 'true’. The realising of pausing and resuming is very simple in this application. This is shown in Figure 20.

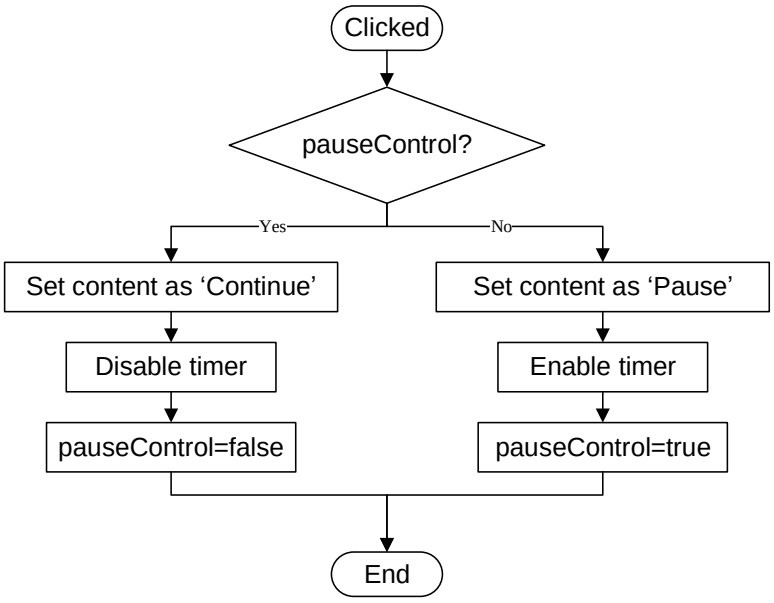


Figure 20 Pause button click event

When Pause button is clicked, its event handling function checks pauseControl.If it is true, the function changes button content as ‘Continue’, disables timer and set pauseControl as false. If not , the function does opposite settings. When Show button is clicked when the drawing process is paused(i.e. pauseControl is false), timer is enabled, and pauseControl is set as true and Pause button content is set as ‘Pause’.

5.3.8 Input restriction

To only allow numbers and decimal point be input into Textbox, ‘KeyDown’ event if the Textbox is handled in a defined function. ‘KeyDown’ event is a keyboard event, it is triggered when a key is pressed down. The basic principle of this handling function is to check which key has been pressed down, if it is neither a key from NumPad0 to NumPad9 nor a key from D0 to D9, and it is neither the Decimal nor the OemPeriod, it is not handled by the Textbox. In other words, it is blocked by the Textbox. The application shields illegal keys in this way.

5.3.9 Parameter validation

The ‘IDataErrorInfo’ Interface provides the functionality to offer custom error information that a user interface can bind to. An object (instance) of IDataErrorInfo contains two properties: Error and Item. ‘Error gets a message

that describes any validation errors for the object. Item gets a message that describes any validation errors for the specified property or column name' (Microsoft Developer Network). The Error property is not utilized in this property as validation error for a whole IDataErrorInfo object is not useful in this application. New properties can be added into an IDataErrorInfo object, and their validation rules and relevant error messages can be specified in Time's 'get' property, which returns a message. In this way Item gets a message corresponding to the property that has validation error. And the customized properties can be bound to some properties of some user interface objects. In WPF, binding is just like synchronising two properties, the two sides of binding respectively are source and object. There are 4 ways of binding, while here only 'TwoWay' binding is introduced because this way is used in the application. In 'TwoWay' binding, if the source is changed, the object will automatically be updated its value to source's new value, and if the object is changed, the source will be updated too.

For instance, an object of IDataErrorInfo is defined and named 'aInfoWithValidation'. It contains a double type property named 'aValue', which is defined to have validation error if its value is greater than 3 or smaller than 0. Item is set to get message 'Please set between 0.0 to 3.0' if aValue is erring. And aValue is 'TwoWay' bound to the 'Text' property of a TextBox. Thus initially this TextBox shows aValue's default value, and if the content of the TextBox is changed, so will aValue's value. If the user input a number greater than 3.0 into the TextBox, there will be a validation error for TextBox because aValue has a validation error.

To show the error message, the style of the TextBox need be supplemented with a Trigger that triggers when the TextBox has validation error. In this application, the message is put into a ToolTip. To strength visual effect, the Trigger sets background of the TextBox at gray and foreground of the TextBox at red. Also the TextBox applies a Template designed for validation error, using which a red '*' symbol is shown at the right of the Textbox. The effect of validation is shown in Figure 21.

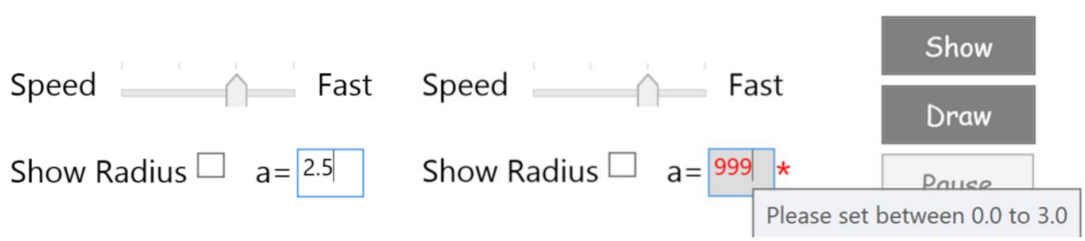


Figure 21 Validation effect

The ‘aInfoWithValidation’ above is just an example. Indeed because requirements of parameters are different between different curves, further some curves require more than one parameter and allowed ranges of their value are also different, so multiple properties need be defined inside IDataErrorInfo instances to bind to all TextBoxes. In this application many objects of IDataErrorInfo is defined to fit every curve UserControl, and they are named from ‘aInfoWithValidation1’. It is also feasible to define all properties in one IDataErrorInfo instance, while UserControls bind their TextBoxes to different properties. But this method may be confusing as scores of properties are put inside an IDataErrorInfo instance, thus it not applied.

Moreover, the click event handling functions of both Show button and Click button check if parameters are valid before enabling timer. So not only message will show at Textbox, but no curve will be drawn if the user input an invalid value into a Textbox.

6. Evaluation

This project succeeds in developing graphical user interface to introduce some classic curves. Introduction for each curve includes presenting its shape, stating literal description, giving equation, and relating to associated curves. In addition, this application realises animation so the user can trace the curves’ generation, meanwhile some shapes and coordinates can be seen during this process to help comprehension. To further enhance interactivity, validation mechanism is applied for textboxes, and the speed of drawing a curve can be adjusted through a slider. Thus, overall this application can be a useful tool for curve learners.

Also some work is done to improve the appearance of the application, such as customizing window, designing style template that contains triggers for buttons and scroll bars. These attempts might be meaningless when functionality and performance of the application are concerned only. However, making nice interface is a common task for developing applications with graphical user interfaces, while in this application only some common skills are applied, so the interface is relatively fundamental.

Though the application runs well, it still has some flaws. The biggest problem is that the process of drawing curve is often slower than it is set to be (which is t). For some curves, it takes time to wait till the show curve process is finished, and that is unacceptable. Unfortunately, the actual reason for that has still not been found, and it requires further studying to figure it out. Here lists three guesses to explain that. The first guess is maybe it takes more time for the AnimatedDraw function to draw a new line than the timer ticks once. Thus every time AnimatedDraw is called by Tick event, it finishes its work after another Tick event is created. So there is delay in drawing each line, as a result, the drawing process lasts longer than expected. But if this guess is correct, then if timer's interval is shorter than the duration of AnimatedDraw, the speed of drawing a curve will be the same regardless of how short timer's interval is. However if interval is set as 0 the drawing process is still quicker than when interval is set as some small positive numbers. Another guess is that maybe the timer and AnimatedDraw are run in the same thread by the system rather than handled asynchronously or in different threads, thus duration between neighbouring ticks prolongs the drawing process. But this is also unlikely correct because a DispatcherTimer is really run standalone in a background thread according to official introduction. The last guess is that on the contrary too many backgurand threads are created that increases workload of CPU and reduced its efficiency. Lacking more professional knowledge, it is hard to believe since only one DispatcherTimer is enabled during the drawing process.

Furthermore, some functions that have been planned are not actually undertaken due to short of time, and the application can be extended in the

following ways. First of all, the application lacks derivation process of equations for some curves and this can be added into introduction to enrich its content. Also, the application does not include right click menu as planned, with a right click menu the functionality of the application will surely be improved, and the application will be more convenient to use. Another way to improve its functionality is creating more global buttons. For example, navigation buttons like back button and forward button make switching between curves even easier. And if more time is allowed, the application may include more curves. Now it only introduces some plane curves, while it may also introduce some space curves. So the application can still to a great extent be improved.

References

Algebraic Curves. (n.d.). Retrieved June 3, 2015, from http://mathworld.wolfram.com/topics/AlgebraicCurves.html

Blog.csdn.net,. (2015). [WPF,XAML,ScrollBar,ScrollViewer,Style] Custom scroll bar - - blog channel - CSDN.NET. Retrieved 3 June 2015, from http://blog.csdn.net/qqamoon/article/details/7317891

Famous Curves Index. (2015, April 1). Retrieved June 3, 2015, from http://www-history.mcs.st-and.ac.uk/Curves/Curves.html

Msdn.microsoft.com. (2015). ControlTemplate Class (System.Windows.Controls). Retrieved 3 June 2015, from https://msdn.microsoft.com/en-us/library/system.windows.controls.controltemplate(v=vs.110).aspx

Msdn.microsoft.com. (2015). IDataErrorInfo Interface (System.ComponentModel). Retrieved 3 June 2015, from https://msdn.microsoft.com/en-us/library/system.componentmodel.idataerrorinfo(v=vs.110).aspx

Msdn.microsoft.com. (2015). Introduction to WPF. Retrieved 3 June 2015, from https://msdn.microsoft.com/en-us/library/aa970268(v=vs.110).aspx

Msdn.microsoft.com. (2015). ScrollViewer Overview. Retrieved 3 June 2015, from https://msdn.microsoft.com/en-us/library/ms750665%28v=vs.110%29.aspx

Sharp, John. Microsoft Visual C♯ 2010 Step By Step. Redmond, Wash.: Microsoft Press, 2010. Print.

Wassenaar, J. (2013, September 21). Mathematical curves. Retrieved June 3, 2015, from http://www.2dcurves.com/

Wetzel, T, & Bez, H. (n.d.). Index of Special Curves with Rational Parametrisations.