

## Project 2 Report – Pythagorean Tree Fractal

D11315807

Ardiawan Bagus Harisa

Department of CSIE

In this project, we are asked to create a Pythagorean fractal using three rectangles, with these specifications:

1. Different size: 3:4:5, 5:12:13, free ratio → choose using bottom right dropdown button.  
In case of 3:4:5 and 5:12:13, the alpha angle will be determined automatically.
2. Size of rectangles (default 100x100) → choose using size up-down button.
3. Color gradient → choose using color dropdown button.

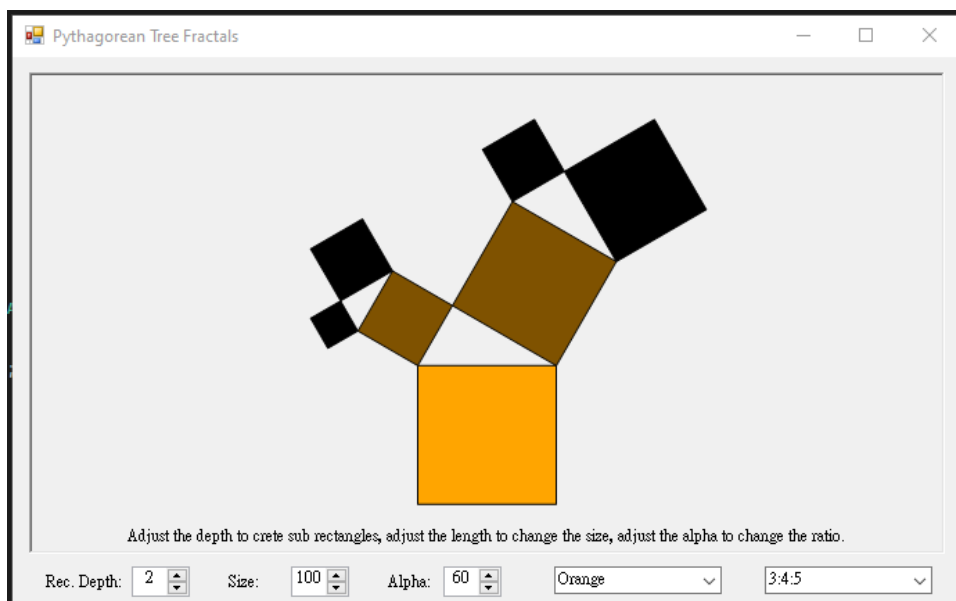
This time, I implement the homework using .NET C# in windows form app. This [tutorial](#) by Rod Stephens really gives the great step through the fractal generation process.

### Setup

To be able to run the project, you just need to use Visual Studio (2022), .NET C#, and windows form SDK installed. Then you can just run the debug or release version of the .exe file.

### How to Use

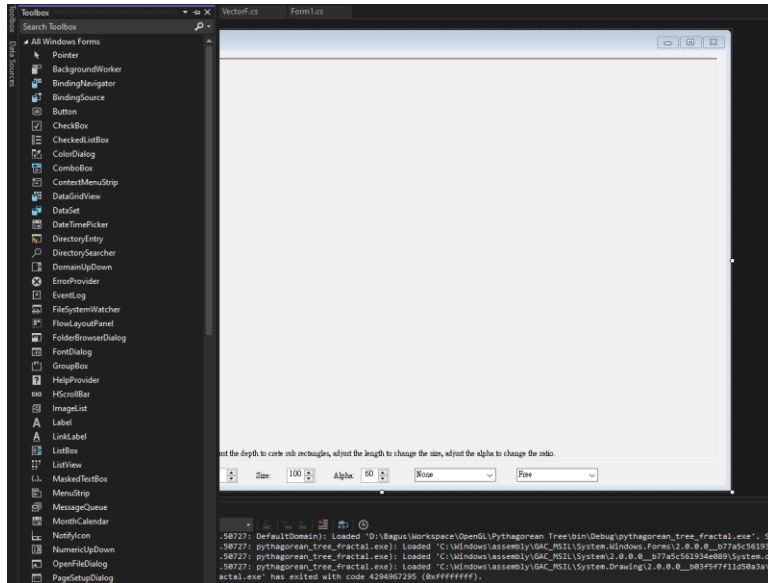
Here is the visual look of the Pythagorean fractal. In here, you can set the recursive depth of the tree, the size of the rectangle, the alpha (angle), and the color.



## Step-by-Step

### Form1.cs

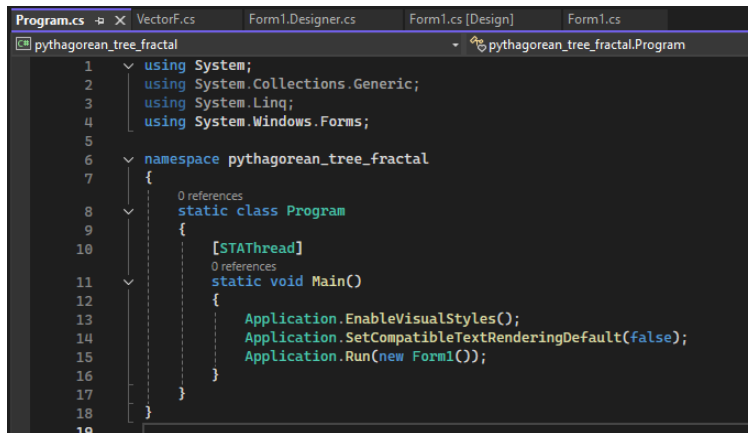
First, I create the form window and attach some UI elements from the toolbox such as text label, combo box dropdown, numeric up and down arrow, and a picture box. The picture box is the main element to perform the drawing of the fractal.



And here is the code view of that form. Basically, in windows form app, there are two files (views): code and designer views. The code is basically generated version of the visual designer form. In the following code, we can see the list of the UI components.

```
cs [Design] Form1.Designer.cs VectorF.cs Form1.cs
pythagorean_tree_fractal
1 namespace pythagorean_tree_fractal
2 {
3     3 references
4     partial class Form1
5     {
6         /// Required designer variable.
7         private System.ComponentModel.IContainer components = null;
8
9         /// Clean up any resources being used. ...
10        0 references
11        protected override void Dispose(bool disposing) ...
12
13        #region Windows Form Designer generated code
14
15        /// Required method for Designer support - do not modify
16        /// the contents of this method with the code editor.
17
18        1 reference
19        private void InitializeComponent() ...
20
21        #endregion
22
23        private System.Windows.Forms.NumericUpDown nudAlpha;
24        private System.Windows.Forms.Label label4;
25        private System.Windows.Forms.NumericUpDown nudLength;
26        private System.Windows.Forms.Label label2;
27        private System.Windows.Forms.NumericUpDown nudDepth;
28        private System.Windows.Forms.Label label1;
29        private System.Windows.Forms.PictureBox picCanvas;
30        private System.Windows.Forms.ComboBox comboBox1;
31        private System.Windows.Forms.Label label3;
32        private System.Windows.Forms.ComboBox comboBox2;
33    }
34}
```

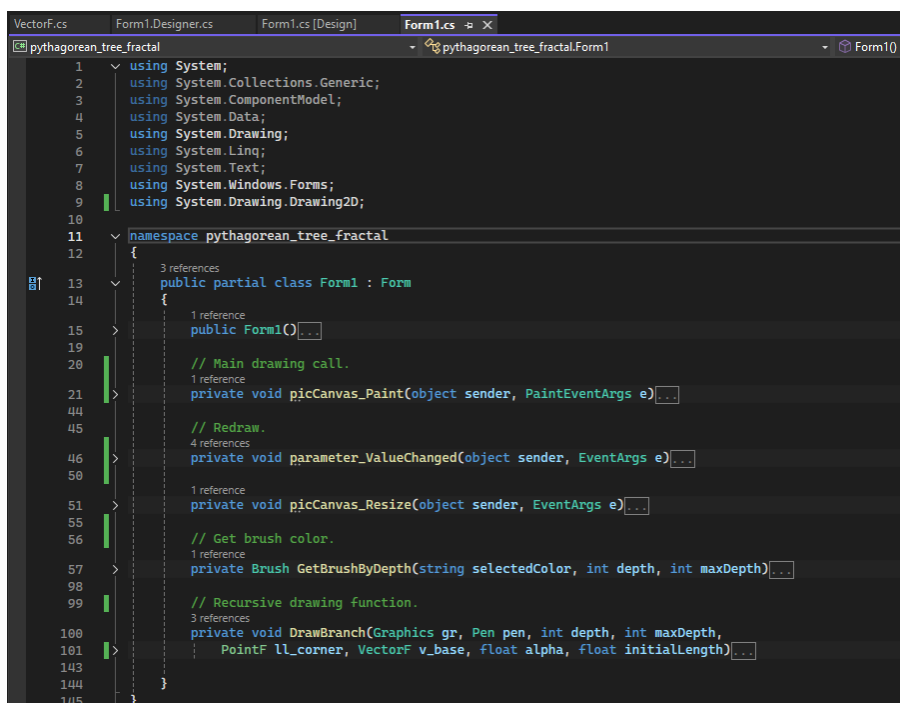
Generally, in Windows Form App, we can create as many forms as we want, then just call them onto the main program as follows. In here, I call the Form1 that I just created in the main function as entry point of the application. First, I enable the visual style of the form to follow the current modern style by calling the `EnableVisualStyles()`. Then, I set the application to use the newer version of text rendering (using GDI+ from .NET). Finally, I call the form that I created where I implemented the logic of the fractal.



```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Windows.Forms;
5
6  namespace pythagorean_tree_fractal
7  {
8      0 references
9      static class Program
10     {
11         0 references
12         static void Main()
13         {
14             Application.EnableVisualStyles();
15             Application.SetCompatibleTextRenderingDefault(false);
16             Application.Run(new Form1());
17         }
18     }
19 }
```

So, here is the steps to create the fractal:

1. Initialize all of the UI element.
2. Setup all of the logic variables including drawing function call.
  - a. Here is where the recursive function to draw the tree is called.
3. I redraw if there is any change on the UI elements.



```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Data;
5  using System.Drawing;
6  using System.Linq;
7  using System.Text;
8  using System.Windows.Forms;
9  using System.Drawing.Drawing2D;
10
11 namespace pythagorean_tree_fractal
12 {
13     3 references
14     public partial class Form1 : Form
15     {
16         1 reference
17         public Form1()...
18     }
19
20     // Main drawing call.
21     1 reference
22     private void picCanvas_Paint(object sender, PaintEventArgs e)...
```

## 1. Initialize UI element

Basically, if we create the windows form using visual editor, the UI elements will all be automatically generated by the system. So, I don't need to do much. Probably just check and rename.

```
21 // Required method for Designer support - do not modify
22 // the contents of this method with the code editor.
23 private void InitializeComponent()
24 {
25     this.nudAlpha = new System.Windows.Forms.NumericUpDown();
26     this.label4 = new System.Windows.Forms.Label();
27     this.nudLength = new System.Windows.Forms.NumericUpDown();
28     this.label2 = new System.Windows.Forms.Label();
29     this.nudDepth = new System.Windows.Forms.NumericUpDown();
30     this.label1 = new System.Windows.Forms.Label();
31     this.picCanvas = new System.Windows.Forms.PictureBox();
32     this.comboBox1 = new System.Windows.Forms.ComboBox();
33     this.label3 = new System.Windows.Forms.Label();
34     ((System.ComponentModel.ISupportInitialize)(this.nudAlpha)).BeginInit();
35     ((System.ComponentModel.ISupportInitialize)(this.nudLength)).BeginInit();
36     ((System.ComponentModel.ISupportInitialize)(this.nudDepth)).BeginInit();
37     ((System.ComponentModel.ISupportInitialize)(this.picCanvas)).BeginInit();
38     this.SuspendLayout();
39     //
40     // nudAlpha
41     //
42     this.nudAlpha.Anchor = System.Windows.Forms.AnchorStyles.Bottom;
43     this.nudAlpha.Location = new System.Drawing.Point(606, 621);
44     this.nudAlpha.Maximum = new decimal(new int[] {
45         360,
46         0,
47         0,
48         0});
```

## 2. Setup logical variables and call the drawing function

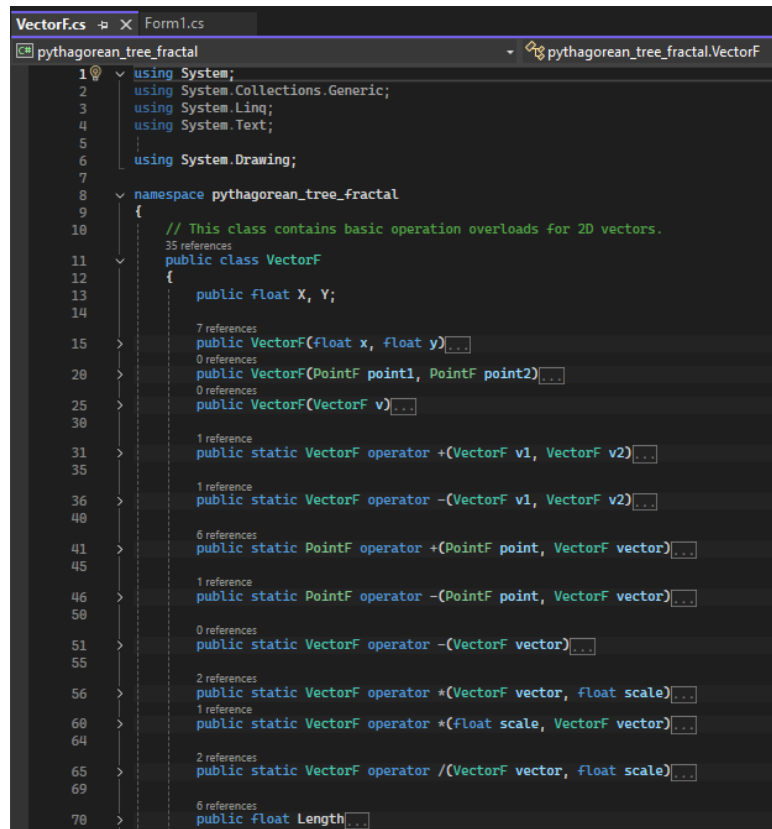
In this `picCanvas_Paint()`, I intend to call the recursive function to draw the fractal tree. First, I make sure to clean up the background color of the canvas that later will be used as the main medium to draw. Second, I assign the parameters of recursive depth, the size of the triangle, the angle of the triangle, the position of the tree, and finally just call the recursive drawing function.

```
19 // Main drawing call.
20 private void picCanvas_Paint(object sender, PaintEventArgs e)
21 {
22     e.Graphics.Clear(picCanvas.BackColor);
23     e.Graphics.SmoothingMode = SmoothingMode.AntiAlias;
24
25     try
26     {
27         // Free aspect ratio default
28         float oriAlpha = (float)nudAlpha.Value;
29
30         if (comboBox2.Text == "3:4:5")
31         {
32             oriAlpha = 60;
33             nudAlpha.Value = 60;
34         }
35         else if (comboBox2.Text == "5:12:13")
36         {
37             oriAlpha = 67.4f;
38             nudAlpha.Value = 67;
39         }
40
41         int recDepth = (int)nudDepth.Value;
42         int length = (int)nudLength.Value;
43         float alpha = (float)((double)oriAlpha * Math.PI / 180.0); // {(float)((double)nud
44         float root_x = picCanvas.ClientSize.Width / 2;
45         float root_y = picCanvas.ClientSize.Height * 0.9f;
46         VectorF v_base = new VectorF(length, 0);
47         PointF ll_corner = new PointF(root_x, root_y) - v_base / 2;
48
49         DrawBranch(e.Graphics, Pens.Black, recDepth, recDepth,
50             ll_corner, v_base, alpha, v_base.Length);
51     }
52     catch
53     {
54     }
55 }
56
57
```

There you can see where I set the ratio of the triangle according to the user preference.

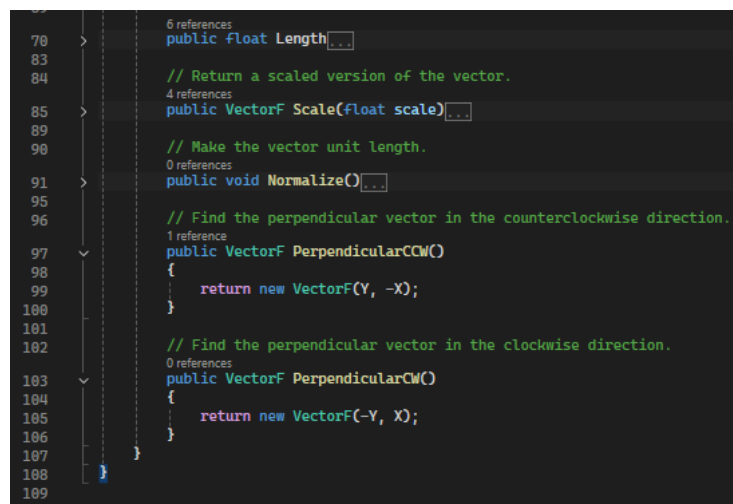
### 3. Draw the tree

To be able to draw the tree with modularity, I have a class to handle the vector operations, call VectorF. This class literally just handle the overload operations of vector and point.



```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  using System.Drawing;
7
8  namespace pythagorean_tree_fractal
9  {
10     // This class contains basic operation overloads for 2D vectors.
11     public class VectorF
12     {
13         public float X, Y;
14
15         public VectorF(float x, float y)
16         {
17             X = x; Y = y;
18         }
19
20         public VectorF(PointF point1, PointF point2)
21         {
22             X = point2.X - point1.X;
23             Y = point2.Y - point1.Y;
24         }
25
26         public VectorF(VectorF v)
27         {
28             X = v.X; Y = v.Y;
29         }
30
31         public static VectorF operator +(VectorF v1, VectorF v2)
32         {
33             return new VectorF(v1.X + v2.X, v1.Y + v2.Y);
34         }
35
36         public static VectorF operator -(VectorF v1, VectorF v2)
37         {
38             return new VectorF(v1.X - v2.X, v1.Y - v2.Y);
39         }
40
41         public static PointF operator +(PointF point, VectorF vector)
42         {
43             return new PointF(point.X + vector.X, point.Y + vector.Y);
44         }
45
46         public static PointF operator -(PointF point, VectorF vector)
47         {
48             return new PointF(point.X - vector.X, point.Y - vector.Y);
49         }
50
51         public static VectorF operator -(VectorF vector)
52         {
53             return new VectorF(-vector.X, -vector.Y);
54         }
55
56         public static VectorF operator *(VectorF vector, float scale)
57         {
58             return new VectorF(vector.X * scale, vector.Y * scale);
59         }
60
61         public static VectorF operator *(float scale, VectorF vector)
62         {
63             return new VectorF(vector.X * scale, vector.Y * scale);
64         }
65
66         public static VectorF operator /(VectorF vector, float scale)
67         {
68             return new VectorF(vector.X / scale, vector.Y / scale);
69         }
70
71         public float Length
72         {
73             get
74             {
75                 return Math.Sqrt(X * X + Y * Y);
76             }
77         }
78     }
79 }
```

Here are some functions to handle the calculation of the angle to create the triangle. CW stands for clockwise, and CCW stands for counter clockwise.



```
70     public float Length
71     {
72         get
73         {
74             return Math.Sqrt(X * X + Y * Y);
75         }
76     }
77
78     // Return a scaled version of the vector.
79     public VectorF Scale(float scale)
80     {
81         return new VectorF(X * scale, Y * scale);
82     }
83
84     // Make the vector unit length.
85     public void Normalize()
86     {
87         float length = Length;
88         X /= length; Y /= length;
89     }
90
91     // Find the perpendicular vector in the counterclockwise direction.
92     public VectorF PerpendicularCCW()
93     {
94         return new VectorF(-Y, X);
95     }
96
97     // Find the perpendicular vector in the clockwise direction.
98     public VectorF PerpendicularCW()
99     {
100         return new VectorF(Y, -X);
101     }
102 }
```

The graphics and pen type from the argument are data type in .NET graphic library to render the fractal tree. The use of Graphics gr is to draw shapes like polygons and lines. So, the graphics gr is passed to the main drawing canvas. The object pen will define the style of the lines to draw the outline of the rectangles.

To render the rectangle, we need to calculate its corner points. First, we need to determine the base width and orientation of the rectangle, using vector, in the counter clockwise direction. Then we calculate the all four points in array of points. We use lower left corner as a base to compute the other. Here is the order:

1. Lower left
2. Lower right
3. Upper right
4. Upper left

```
112 // Recursive drawing function.
113
114 private void DrawBranch(Graphics gr, Pen pen, int depth, int maxDepth,
115     PointF ll_corner, VectorF v_base, float alpha, float initialLength)
116 {
117     // Compute corners
118     VectorF v_height = v_base.PerpendicularCCW();
119     PointF[] points =
120     {
121         ll_corner,
122         ll_corner + v_base,
123         ll_corner + v_base + v_height,
124         ll_corner + v_height,
125     };
126 }
```

Then, I draw the rectangle(s). First, I need to prepare the drawing tool: brush and pen. Those are used to define the style of the drawing. For instance, the color of the rectangle is defined by brush, which is defined by the dropdown button and the depth of the tree.

```

55 // Get brush color.
56
57 1 reference
58 private Brush GetBrushByDepth(string selectedColor, int depth, int maxDepth)
59 {
60     double fade = depth / (double)maxDepth;
61     int a = 255;
62     int r = 0, g = 0, b = 0;
63
64     switch (selectedColor)
65     {
66         case "Red":
67             r = (int)(255 * fade);
68             break;
69
70         case "Purple":
71             r = (int)(128 * fade);
72             b = (int)(128 * fade);
73             break;
74
75         case "Blue":
76             b = (int)(255 * fade);
77             break;
78
79         case "Green":
80             g = (int)(255 * fade);
81             break;
82
83         case "Yellow":
84             r = (int)(255 * fade);
85             g = (int)(255 * fade);
86             break;
87
88         case "Orange":
89             r = (int)(255 * fade);
90             g = (int)(165 * fade);
91             break;
92
93         default:
94             return null;
95     }
96
97     return new SolidBrush(Color.FromArgb(a, r, g, b));
98
99 // Drawing drawing function

```

```

// Draw rectangle
Brush brush = GetBrushByDepth(comboBox1.SelectedItem?.ToString(), depth, maxDepth);
if (brush != null || comboBox1.SelectedItem?.ToString() != "None")
{
    gr.FillPolygon(brush, points);
}
gr.DrawPolygon(pen, points);

if (depth > 0)
{
    // LEFT BRANCH
    double w1 = v_base.Length * Math.Cos(alpha);
    float wb1 = (float)(w1 * Math.Cos(alpha));
    float wh1 = (float)(w1 * Math.Sin(alpha));
    VectorF v_base1 = v_base.Scale(wb1) + v_height.Scale(wh1);
    PointF ll_corner1 = ll_corner + v_height;

    DrawBranch(gr, pen, depth - 1, maxDepth, ll_corner1, v_base1, alpha, initialLength);

    // RIGHT BRANCH
    double beta = Math.PI / 2.0 - alpha;
    double w2 = v_base.Length * Math.Sin(alpha);
    float wb2 = (float)(w2 * Math.Cos(beta));
    float wh2 = (float)(w2 * Math.Sin(beta));
    VectorF v_base2 = v_base.Scale(wb2) - v_height.Scale(wh2);
    PointF ll_corner2 = ll_corner1 + v_base1;

    DrawBranch(gr, pen, depth - 1, maxDepth, ll_corner2, v_base2, alpha, initialLength);
}
}

```

Then using the following recursive concept, I draw the rectangles.

1. Draw 1 rectangle
2. If depth > 0, draw two more rectangle
3. Recursive call

So, first, draw a rectangle as the base, as I mentioned before. Then, if the depth is more than zero, meaning I should draw two branches. Then, draw the left branch of the tree by calculating the base size and its relative position and orientation by calling its recursive function. Then, perform the similar call to the other (right) branch. The process will continue until there is no more depth to perform (depth = 0).

And these two functions will be called to redraw in case any update from the form (i.e., user edits the value).

```
45 // Redraw.
46 4 references
47 private void parameter_ValueChanged(object sender, EventArgs e)
48 {
49     picCanvas.Refresh();
50 }
51 1 reference
52 private void picCanvas_Resize(object sender, EventArgs e)
53 {
54     picCanvas.Refresh();
55 }
```

## Simulations

