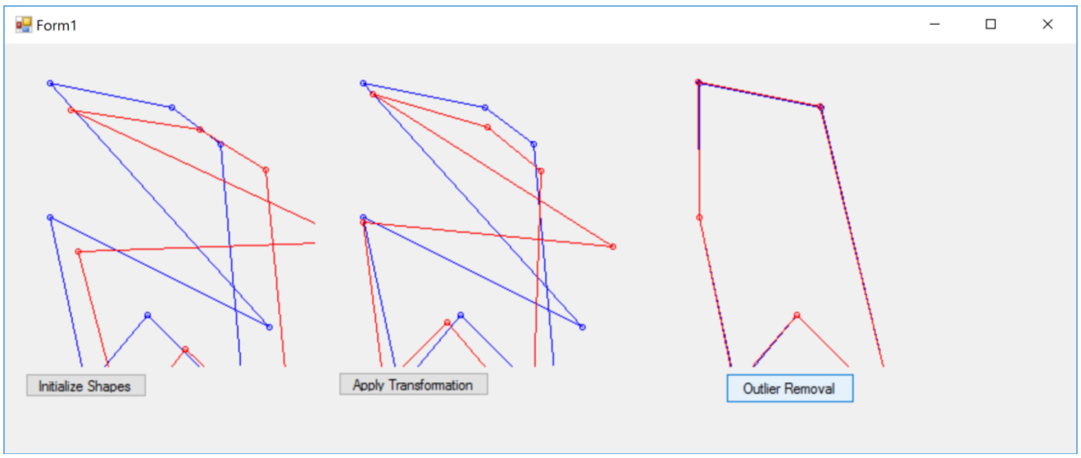
CPEG 585 – Assignment #5

a) Outlier Removal using Exhaustive Evaluation:

Suppose you are given a set of points in two shapes. Some of these points are outliers. We can remove these via exhaustive evaluation or via random sampling through a technique called RANSAC.
The following partial shows how you will test for adding and removing outliers. You need to write the code for the outlier removal button handler, so that the two shapes get aligned as shown in the right most part of the screen shot below.




```
public partial class Form1 : Form
    {
        List<Point> Shape1 = new List<Point>();
        List<Point> Shape2 = new List<Point>();
        List<Point> Shape2Transformed = new List<Point>();

        public Form1()
        {
            InitializeComponent();
        }

        private void btnInitializeShapes_Click(object sender, EventArgs e)
        {
            Shape1.Clear();
            Shape2.Clear();
            Point p1a = new Point(20, 30);
            Point p2a = new Point(120, 50);
            Point p3a = new Point(160, 80);
            Point p4a = new Point(180, 300);
            Point p5a = new Point(100, 220);
            Point p6a = new Point(50, 280);
            Point p7a = new Point(20, 140);

            Shape1.Add(p1a);
            Shape1.Add(p2a);
            Shape1.Add(p3a);
            Shape1.Add(p4a);
            Shape1.Add(p5a);
            Shape1.Add(p6a);
            Shape1.Add(p7a);
```

```
            Transformation T2 = new Transformation();
            T2.A = 1.05; T2.B = 0.05; T2.T1 = 15; T2.T2 = 22;
            Shape2 = ApplyTransformation(T2, Shape1);
            Shape2[2] = new Point(Shape2[2].X + 10, Shape2[2].Y + 3);// change one point
            // add outliers to both shapes
            Point ptOutlier1 = new Point(200, 230);
            Shape1.Add(ptOutlier1);
            Point ptOutLier2 = new Point(270, 160);
            Shape2.Add(ptOutLier2);


            Pen pBlue = new Pen(Brushes.Blue, 1);
            Pen pRed = new Pen(Brushes.Red, 1);
            Graphics g = panShape1.CreateGraphics();
            DisplayShape(Shape1, pBlue, g);
            DisplayShape(Shape2, pRed, g);
        }

        private void btnApplyTransformation_Click(object sender, EventArgs e)
        {
            Transformation T = ICPTransformation.ComputeTransformation(Shape1, Shape2);
            MessageBox.Show("Cost = " + ICPTransformation.ComputeCost(Shape1, Shape2,
T).ToString());
            List<Point> Shape2T = ApplyTransformation(T, Shape2);
            Pen pBlue = new Pen(Brushes.Blue, 1);
            Pen pRed = new Pen(Brushes.Red, 1);
            Graphics g = panShape2.CreateGraphics();
            DisplayShape(Shape1, pBlue, g);
            DisplayShape(Shape2T, pRed, g);

        }

        void DisplayShape(List<Point> Shp, Pen pen, Graphics g)
        {
            Point? prevPoint = null;  // nullable
            foreach (Point pt in Shp)
            {
                g.DrawEllipse(pen, new Rectangle(pt.X - 2, pt.Y - 2, 4, 4));
                if (prevPoint != null)
                    g.DrawLine(pen, (Point)prevPoint, pt);
                prevPoint = pt;
            }
            g.DrawLine(pen, Shp[0], Shp[Shp.Count - 1]);
        }

        List<Point> ApplyTransformation(Transformation T, List<Point> shpList)
        {
            List<Point> TList = new List<Point>();
            foreach (Point pt in shpList)
            {
                double xprime = T.A * pt.X + T.B * pt.Y + T.T1;
                double yprime = T.B * pt.X * -1 + T.A * pt.Y + T.T2;
                Point pTrans = new Point((int)xprime, (int)yprime);
                TList.Add(pTrans);
            }
            return TList;
        }
```

```
class ICPTransformation
    {

        public static Transformation ComputeTransformation(List<Point> Shp1, List<Point>
Shp2)
        {
            Matrix A = new Matrix(4, 4);
            Matrix B = new Matrix(4, 1);
            for (int i = 0; i < Shp1.Count; i++)
            {
                A[0, 0] += 2 * Shp2[i].X * Shp2[i].X + 2 * Shp2[i].Y * Shp2[i].Y;
                // provide the code for remaining A matrix entries …
                A[3, 3] += -2;

                B[0, 0] += 2 * Shp1[i].X * Shp2[i].X + 2 * Shp1[i].Y * Shp2[i].Y;
                // provide the code for remaining B matrix entries …

                B[3, 0] += -2 * Shp1[i].Y;

            }

            Matrix Ainv = A.Inverse;
            Matrix Res = Ainv * B;
            Transformation T = new Transformation();
            T.A = Res[0, 0];
            T.B = Res[1, 0];
            T.T1 = Res[2, 0];
            T.T2 = Res[3, 0];
            return T;
        }


        public static double ComputeCost(List<Point> P1List, List<Point> P2List,
Transformation T)
        {
            double cost = 0;
            for (int i = 0; i < P1List.Count; i++)
            {
                double xprime = T.A * P2List[i].X + T.B * P2List[i].Y + T.T1;
                double yprime = -1 * T.B * P2List[i].X + T.A * P2List[i].Y + T.T2;
                cost += (P1List[i].X - xprime) * (P1List[i].X - xprime) +
                    (P1List[i].Y - yprime) * (P1List[i].Y - yprime);
            }
            return cost;
        }
    }


public class Transformation
    {
        public double A { get; set; }
        public double B { get; set; }
        public double T1 { get; set; }
        public double T2 { get; set; }
    }
```

b) **Outlier Removal Using RANSAC**:

Repeat part a) using the RANSAC algorithm.

The pseudocode for the RANSAC algorithm is shown below (Ref: http://www.thefullwiki.org/RANSAC)

```
input:
    data - a set of observations
    model - a model that can be fitted to data
    n - the minimum number of data required to fit the model
    k - the number of iterations performed by the algorithm
    t - a threshold value for determining when a datum fits a model
    d - the number of close data values required to assert that a model fits
well to data
output:
    best_model - model parameters which best fit the data (or nil if no good
model is found)
    best_consensus_set - data point from which this model has been estimated
    best_error - the error of this model relative to the data

iterations := 0
best_model := nil
best_consensus_set := nil
best_error := infinity
while iterations < k
    maybe_inliers := n randomly selected values from data
    maybe_model := model parameters fitted to maybe_inliers
    consensus_set := maybe_inliers

    for every point in data not in maybe_inliers
        if point fits maybe_model with an error smaller than t
            add point to consensus_set

    if the number of elements in consensus_set is > d
        (this implies that we may have found a good model,
        now test how good it is)
        better_model := model parameters fitted to all points in
consensus_set
        this_error := a measure of how well better_model fits these points
        if this_error < best_error
            (we have found a model which is better than any of the previous
ones,
            keep it until a better one is found)
            best_model := better_model
            best_consensus_set := consensus_set
            best_error := this_error

    increment iterations

return best_model, best_consensus_set, best_error
```