



MOD006127

**COMPUTER GRAPHICS  
PROGRAMMING**

**MAIN EXERCISES**

*Prepared for: Ashim Chakraborty*

*Prepared by: **SID\_19-03-999***

Due Date: 10<sup>th</sup> of June 2022

Word Count: 750 words

**Table of Contents**

**INSTRUCTIONS ..... 2**

**THE APPLICATION..... 2**

**PRO’S ..... 3**

**CON’S..... 3**

**APPENDIX GRAFPACK: ..... 4**

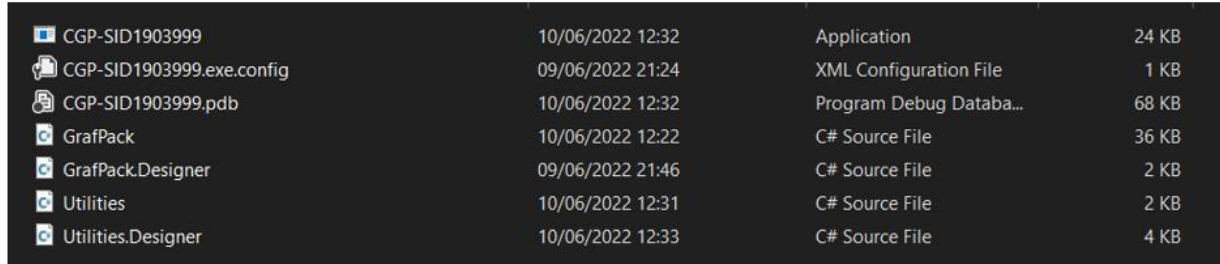
**APPENDIX GRAFPACK.DESIGNER:..... 25**

**APPENDIX UTILITIES:..... 26**

**APPENDIX UTILITIES.DESIGNER: ..... 28**

## Instructions

Open folder ( SID1903999/CGPMain/MainApp/ ) where you will find all of the .cs files along with the .exe file to run the app as demonstrated in with Fig1, below.



CGP-SID1903999	10/06/2022 12:32	Application	24 KB
CGP-SID1903999.exe.config	09/06/2022 21:24	XML Configuration File	1 KB
CGP-SID1903999.pdb	10/06/2022 12:32	Program Debug Databa...	68 KB
GrafPack	10/06/2022 12:22	C# Source File	36 KB
GrafPack.Designer	09/06/2022 21:46	C# Source File	2 KB
Utilities	10/06/2022 12:31	C# Source File	2 KB
Utilities.Designer	10/06/2022 12:33	C# Source File	4 KB

Fig1: Main Assignment Content

## The Application

The Program was modified to draw 3x shapes i.e., a square, a circle, and a triangle. The circle and square are created with the help of 2pts whereas the triangle is with the help of 3pts. As requested the program shapes are all 3x defined as separate classes that inherit from the class shape.

A list to store all of the shapes was created and which plays the role of adding new shapes through creating or deleting (toRemove) the shapes from the screen and memory.

In order to create the square, I developed from the existing GrafPack code from canvas and borrowed some of the code from brescircles.cs file on canvas to draw the circle. The transform section contains 3 items of choice which is to resize, move and rotate the shape. The circle cannot be rotated and a trigger message follows if the user tries to do so.

The rotation works on the principle of degrees i.e., the user is asked to input a value between 0 and 360 degrees which will rotate the shape accordingly. The resize option works up to 5 scales otherwise the shape is exceeding the window size.

As for the delete and exit option the user is prompted with a warning message each time, thus so they can confirm the action they wish to go ahead with i.e. delete the shape or exit the application.

The program works around using separate methods so the code can be red easily. Lines that were self-explanatory were left without comments whereas where needed comments have been added.

## Pro's

The program meets all of the assignment requirements but it does not exceed them. The program can create 3x shapes and upon each creation stores the shapes in a list which is accessed in the background to find the shape required which can then later be deleted (completely removed), or selected in order to be modified.

The program will highlight the selected shape with red surrounding, which gives a clear visual representation of which shape is being selected.

The program can modify the already created shapes. In Transform there are 3x selectable options, resize, move and rotate all calculated with the help of linear algebra. A shape has to be firstly selected, once selected it will be highlighted with a red surrounding color, the if move option is selected the shape can be moved with the help of drag and drop operation. As for the resizing and rotation, similar processes are followed, which once pressed will show a message box asking the user for input. The input is then applied to the shape as per user requirements.

The program can delete the shapes. As mentioned earlier a list of shapes was created in order to keep track of all of the created shapes, in order to delete a shape, the user is asked to first select the desired shape, and then once the delete option is pressed a trigger message box will appear on the screen asking the user if they wish to go ahead or cancel the deletion.

The program can be exited cleanly. Once the exit button is pressed, the program will show a trigger message to ask the user to double confirm the wish to exit. At this point, the user has two options to cancel the operation or to confirm it which in essence is self-explanatory.

## Con's

The program does not exceed the assignment requirements, in my opinion further development can make the program generate more than just the 3 required shapes, which is something that needs more work on if the expectations are to build a paint app clone. The following cons also apply to the transformation section where more options could've been added.

Shape creation is made through discrete mouse clicks, while this works accordingly and stores all of the newly created shapes in a list, the rubber-banding method will perhaps be more intuitive to use as well as easier to visualize.

GUI can be worked on to look more user-friendly, perhaps on screen paint the shapes created option can be added, the option to have multiple shape pen drawing color i.e., graphics can be implemented as currently the program only works on 2 pen colors which black and red.

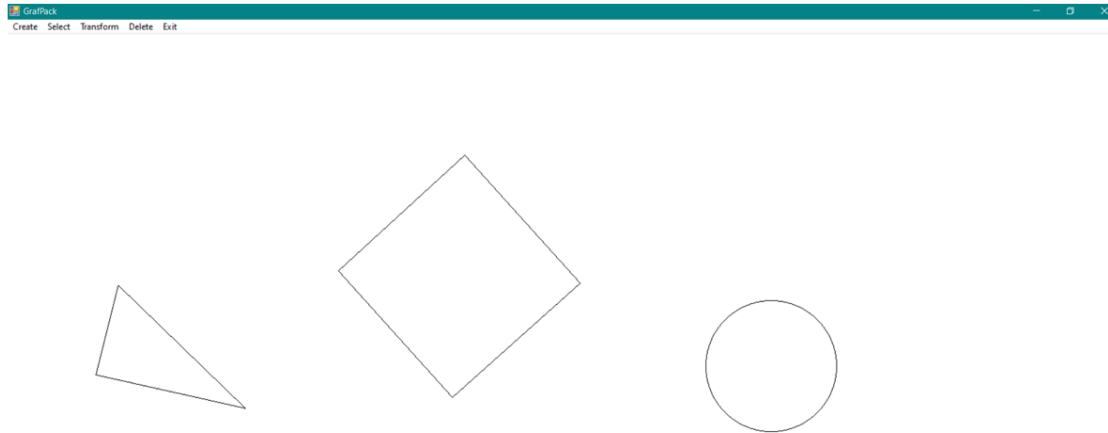


Fig2: Application working and drawing 3 shapes

## Appendix GrafPack:

```
using System;
using System.Drawing;
using System.Collections.Generic;
using System.Windows.Forms;

namespace Grafpack
{
    public partial class GrafPack : Form
    {
        // MainMenu;
        private MainMenu mainMenu;

        // Declare graphics var
        Graphics g;
        Pen blackPen;
        Pen redPen;

        // Bool var to select the shapes to be drawn
        private bool selectSquareStatus = false;
        private bool selectCircleStatus = false;
        private bool selectTriangleStatus = false;

        private bool currentShapeStatus = false;
        private bool selectedShapes = false;
        private bool moveDragAnDrop = false;
```

```

// false - no current shape selected
private bool aShapeHasBeenSelected = false;

// Count var
private int selectionCount = 0;
private int markedShape = 0;

// Point var to draw the shapes
private Point one;
private Point two;
private Point three;

/* Declare variables
 * markTheClickedPoint by User
 * lengthVal - lowest length value
 */
private Point markTheClickedPoint;
private int lengthVal;

// Declare a list to store all of the Shapes
private List<Shape> shapeList;

public GrafPack()
{
    loadProgram();
    loadInterfacelItems();
}

// Set window size and background color
private void loadProgram()
{
    InitializeComponent();
    this.SetStyle(ControlStyles.ResizeRedraw, true);
    this.WindowState = FormWindowState.Maximized;
    this.BackColor = Color.White;
}

// Load menu objects and declare menu items
private void loadInterfacelItems()
{
    mainMenu = new MainMenu();
    // Selectables
    MenuItem createlItem = new MenuItem();
    MenuItem selectItem = new MenuItem();
    // Shapes
    MenuItem squareItem = new MenuItem();
    MenuItem circleItem = new MenuItem();

```

```

MenuItem triangleItem = new MenuItem();
// Transform
MenuItem transformItem = new MenuItem();
//MenuItem moveShape = new MenuItem();
MenuItem move = new MenuItem();
MenuItem resize = new MenuItem();
MenuItem rotate = new MenuItem();
// Delete
MenuItem deleteItem = new MenuItem();
// Exit
MenuItem exitApplication = new MenuItem();

// Give Text Value to menu items
// Selectables
createItem.Text = "&Create";
selectItem.Text = "&Select";
// Shapes
squareItem.Text = "&Square";
circleItem.Text = "&Circle";
triangleItem.Text = "&Triangle";
// Transform
transformItem.Text = "&Transform";
move.Text = "&Move Shape";
resize.Text = "&Resize Shape";
rotate.Text = "&Rotate Shape";
// Delete
deleteItem.Text = "&Delete";
// Exit
exitApplication.Text = "&Exit";

// Assign menu items to main menu object
// Selectables
mainMenu.MenuItems.Add(createItem);
mainMenu.MenuItems.Add(selectItem);
// Shapes
createItem.MenuItems.Add(squareItem);
createItem.MenuItems.Add(circleItem);
createItem.MenuItems.Add(triangleItem);
// Transform
mainMenu.MenuItems.Add(transformItem);
transformItem.MenuItems.Add(resize);
transformItem.MenuItems.Add(move);
transformItem.MenuItems.Add(rotate);
// Delete
mainMenu.MenuItems.Add(deleteItem);
// Exit

```

```

mainMenu.MenuItems.Add(exitApplication);

//Handling Mouse Events to the menu items
selectItem.Click += new System.EventHandler(this.selectShape);
// Shapes
squareItem.Click += new System.EventHandler(this.SelectSquare);
circleItem.Click += new System.EventHandler(this.SelectCircle);
triangleItem.Click += new System.EventHandler(this.SelectTriangle);

// Transform
move.Click += new System.EventHandler(this.transG_MoveShape);
resize.Click += new System.EventHandler(this.transG_resizeShape);
rotate.Click += new System.EventHandler(this.transG_RotateShape);
// Delete
deleteItem.Click += new System.EventHandler(this.deleteShape);
// Exit
exitApplication.Click += new System.EventHandler(this.exitSelected);

// Assign menu object to form menu
this.Menu = mainMenu;
// initialise the list of shapes
shapeList = new List<Shape>();

// Mouse Listener
this.MouseClick += mouseClick;
}
// Exit: Terminate the program cleanly.
private void exitSelected(object sender, EventArgs e)
{
    string title = "Confirm Exit!";
    string message = "Are you sure, you want to exit?";
    MessageBoxButtons buttons = MessageBoxButtons.YesNo;
    DialogResult dr = MessageBox.Show(message, title, buttons);
    if (dr == DialogResult.Yes)
    {
        this.Close();
        Application.Exit();
    }
    if (dr == DialogResult.No)
    {
        selectionCount = 0;
    }
}

// Generally, all methods of the form are usually private
private void SelectSquare(object sender, EventArgs e)

```



```

    {
        selectSquareStatus = true;
        MessageBox.Show("Click OK and then click once each at two locations to
create a square");
    }
    private void SelectCircle(object sender, EventArgs e)
    {
        selectCircleStatus = true;
        MessageBox.Show("Click OK and then click once each at two locations to
create a circle");
    }
    private void SelectTriangle(object sender, EventArgs e)
    {
        selectTriangleStatus = true;
        MessageBox.Show("Click OK and then click once each at three locations to
create a triangle");
    }
    private void selectShape(object sender, EventArgs e)
    {
        currentShapeStatus = true;

        if (shapeList.Count == 0)
        {
            MessageBox.Show("No shapes available to select");
        }
        else
        {
            MessageBox.Show("Please click using the mouse the shape, you'd like to
select");
            selectedShapes = true; //Bool set to true for activation of other code
        }
    }

    // This method is quite important and detects all mouse clicks - other methods may
need
    // to be implemented to detect other kinds of event handling eg keyboard presses.
    private void mouseClicked(object sender, MouseEventArgs e)
    {
        if (e.Button == MouseButtons.Left)
        {

            // 'if' statements can distinguish different selected menu operations to
implement.
            // There may be other (better, more efficient) approaches to event handling,
            // but this approach works.
            int xLength, yLength;

```

```

if (selectedShapes == true)
{
    int shapeX;
    lengthVal = 2147483647; // biggest int value to identify the closest item
    markTheClickedPoint = new Point(e.X, e.Y);

    // for loop to loop through all of the current shapes in the list
    for (shapeX = 0; shapeX < shapeList.Count; shapeX++)
    {
        // calc mid points of selected shape
        Point calcMidPoint = shapeList[shapeX].calcMidPoint();
        xLength = Math.Abs(markTheClickedPoint.X - calcMidPoint.X);
        yLength = Math.Abs(markTheClickedPoint.Y - calcMidPoint.Y);

        if (lengthVal > xLength || lengthVal > yLength)
        {
            lengthVal = xLength;

            if (xLength > yLength)
            {
                lengthVal = yLength;
            }
            markedShape = shapeX; // lowestVal = desired shape identified
        }
    }

    aShapeHasBeenSelected = true;
    // Reset
    selectedShapes = false;
    // HighlightTheShape
    highlightTheShape();
}
if (moveDragAnDrop == true)
{
    // store the location
    markTheClickedPoint = new Point(e.X, e.Y);
    // calc mid points of selected shape
    Point calcMidPoint = shapeList[markedShape].calcMidPoint();
    xLength = markTheClickedPoint.X - calcMidPoint.X;
    yLength = markTheClickedPoint.Y - calcMidPoint.Y;
    // Call the method to move the shape by drag&drop
    shapeList[markedShape].transformMove(xLength, yLength);
    this.Refresh(); // invalidate, and then update
    highlightTheShape();
    moveDragAnDrop = false;
}

```

```

}
if (selectSquareStatus == true)
{
    if (selectionCount == 0)
    {
        one = new Point(e.X, e.Y);
        selectionCount = 1;
    }
    else
    {
        two = new Point(e.X, e.Y);
        drawASqaure(one, two);
    }
}
if (selectCircleStatus == true)
{
    if (selectionCount == 0)
    {
        one = new Point(e.X, e.Y);
        selectionCount = 1;
    }
    else
    {
        two = new Point(e.X, e.Y);
        drawACircle(one, two);
    }
}

if (selectTriangleStatus == true)
{
    if (selectionCount == 0)
    {
        one = new Point(e.X, e.Y);
        selectionCount++;
    }
    else if (selectionCount == 1)
    {
        two = new Point(e.X, e.Y);
        selectionCount++;
    }
    else
    {
        three = new Point(e.X, e.Y);
        drawATriangle(one, two, three);
    }
}

```

```

    }
}
//Shapes - Start
// drawASqaure() method
private void drawASqaure(Point pts1, Point pts2)
{
    selectionCount = 0; //Reset click numbers for later use
    selectSquareStatus = false; // reset the bool
    // Initialise Graphics
    g = this.CreateGraphics();
    blackPen = new Pen(Color.Black);

    Square aShape = new Square(pts1, pts2); //Creation of square object
    aShape.drawM(g, blackPen); //Draw the shape using draw method in the shapes
class
    shapeList.Add(aShape); //Addition of Square to the shapes list
}
// drawACircle() method
private void drawACircle(Point pts1, Point pts2)
{
    selectionCount = 0;
    selectCircleStatus = false;
    // Initialise Graphics
    g = this.CreateGraphics();
    blackPen = new Pen(Color.Black);

    Circle aShape = new Circle(pts1, pts2);
    aShape.drawM(g, blackPen);
    shapeList.Add(aShape);
}
// drawATriangle() method
private void drawATriangle(Point pts1, Point pts2, Point pts3)
{
    selectionCount = 0;
    selectTriangleStatus = false;
    // Initialise Graphics
    g = this.CreateGraphics();
    blackPen = new Pen(Color.Black);

    Triangle aShape = new Triangle(pts1, pts2, pts3);
    aShape.drawM(g, blackPen);
    shapeList.Add(aShape);
}
//Shapes - End

```

```

// Transform - Start (transformGroup)
// Method to Resize
private void transG_resizeShape(object sender, EventArgs e)
{
    if (aShapeHasBeenSelected == false)
    {
        MessageBox.Show("No shape has been selected");
    }
    else
    {
        float resizeFactor = retrieveGetSet("Resize value between ", 0, 5);

        if (resizeFactor == 2147483647) // cancelled op
        {
            return;
        }
        else
        {
            shapeList[markedShape].transformResize(resizeFactor);
            this.Refresh(); // invalidate, and then update
            highlightTheShape();
        }
    }
}

// Method to Move
private void transG_MoveShape(object sender, EventArgs e)
{
    if (aShapeHasBeenSelected == false || shapeList.Count == 0)
    {
        MessageBox.Show("No shape has been selected");
    }
    else
    {
        MessageBox.Show("Please Drag and Drop the shape with the help of mouse click and release");
        moveDragAnDrop = true;
    }
}

//Method to Rotate
private void transG_RotateShape(object sender, EventArgs e)
{
    if (shapeList[markedShape].GetItemToDraw() == "Circle") // can't rotate a circle
    {
        MessageBox.Show("Can't rotate what's already rounded");
    }
}

```

```

else if (aShapeHasBeenSelected == false)
{
    MessageBox.Show("No shape has been selected");
}
else
{
    float rotateShape = retrieveGetSet("Enter a value in-between ", 0, 360);

    if (rotateShape == 2147483647) // Used in the event of cancel clicked on form
    {
        return;
    }
    else
    {
        shapeList[markedShape].transformRotate(rotateShape); //Calls the rotation
method on the selected shape

        this.Refresh(); // invalidate, and then update
        highlightTheShape();
    }
}
}
// Transform - End (transformGroup)

//Method to highlight the stored Shape
public void highlightTheShape()
{
    int shapeX;
    // Initialise Graphics object
    g = this.CreateGraphics();
    blackPen = new Pen(Color.Black);
    redPen = new Pen(Color.Red);

    // for loop to loop through all of the current shapes in the list
    for (shapeX = 0; shapeX < shapeList.Count; shapeX++)
    {
        if (shapeX == markedShape)
        {
            shapeList[shapeX].drawM(g, redPen);
        }
        else
        {
            shapeList[shapeX].drawM(g, blackPen);
        }
    }
}
}

```

```

// Delete: If a shape has been selected, this will delete it from the screen and from
memory
private void deleteShape(object sender, EventArgs e)
{
    if (aShapeHasBeenSelected == false) // no shape selected
    {
        MessageBox.Show("No shape has been selected");
    }

    else if (MessageBox.Show("Delete the Shape?", "Confirm Delete!",
        MessageBoxButtons.YesNo, MessageBoxIcon.Question) == DialogResult.Yes)
    {
        shapeList.Remove(shapeList[markedShape]); // delete it from the screen and
        from memory.
        this.Refresh();// invalidate, and then update
    }
    else
    {
        return;
    }
    highlightTheShape(); //Redraw shape method called
}

//Method to retrieve from public partial class Utilities : Form
public float retrieveGetSet(string GetItemToDraw, float highest, float lowest)
{
    float userInputVal;
    // form to write
    Utilities userInputBox = new Utilities(GetItemToDraw, lowest, highest);

    userInputBox.ShowDialog(); //Brings user input form to the front and stops
    actions on main form
    userInputVal = userInputBox.inputVal(); // Take the users input from user input
    form
    userInputBox.Close();
    return userInputVal;
}
}
abstract class Shape
{
    // This is the base class for Shapes in the application. It should allow an array or LL
    // to be created containing different kinds of shapes.
    private string ItemToDraw;
    public Point midPt;
}

```

```

public Shape() { } // constructor
public virtual void drawM(Graphics g, Pen blackPen) { }
public virtual void transformResize(float scalarVal) { }
public virtual void transformMove(float x, float y) { }
public virtual void transformRotate(float Angle) { }
public new virtual string GetItemToDraw() { return ItemToDraw; }
public virtual Point calcMidPoint() { return midPt; }
}
class calc
{
    public static float[,] ptsReturned = { { 0, 0 } };
    public static int adOn;
    public static int shapeX;
    public static int shapeY;

    // Method to calc ScalarVal
    public static float[,] scalarVal(float[,] ptsReturned1, float[,] ptsReturned2)
    {
        for (shapeX = 0; shapeX < 1; shapeX++)
        {
            for (int shapeY = 0; shapeY < 2; shapeY++)
            {
                ptsReturned[shapeX, shapeY] = 0;

                // loop through shapeX and shapeY
                for (adOn = 0; adOn < 2; adOn++)
                {
                    ptsReturned[shapeX, shapeY] += ptsReturned1[shapeX, shapeY] *
ptsReturned2[adOn, shapeY];
                }
            }
        }
        return ptsReturned;
    }

    // Method to calc RotationVal
    public static float[,] rotationVal(float[,] ptsReturned1, float[,] ptsReturned2)
    {
        ptsReturned[0, 0] = (ptsReturned1[0, 0] * ptsReturned2[0, 0]) - (ptsReturned1[0,
1] * ptsReturned2[0, 1]);
        ptsReturned[0, 1] = (ptsReturned1[0, 0] * ptsReturned2[0, 1]) + (ptsReturned1[0,
1] * ptsReturned2[0, 0]);
        return ptsReturned;
    }
}
class Square : Shape

```



```

{
    //This class contains the specific details for a square defined in terms of opposite
    corners
    Point[] pts = new Point[2]; // these points identify opposite corners of the square
    Point midPt = new Point();
    string ItemToDraw;

    public Square(Point pts1, Point pts2) // constructor
    {
        pts[0] = pts1;
        pts[1] = pts2;
        // calc midPt
        midPt.X = (pts1.X + pts2.X) / 2;
        midPt.Y = (pts1.Y + pts2.Y) / 2;
        ItemToDraw = "Square";
    }
    // Abstract
    public override string GetItemToDraw() { return ItemToDraw; }
    public override Point calcMidPoint() { return midPt; }

    // You will need a different draw method for each kind of shape. Note the square is
    drawn
    // from first principles. All other shapes should similarly be drawn from first
    principles.
    // Ideally no C# standard library class or method should be used to create, draw or
    transform a shape
    // and instead should utilise user-developed code.
    public override void drawM(Graphics g, Pen blackPen)
    {
        // This method draws the square by calculating the positions of the other 2
        corners
        double xDiff, yDiff, xMid, yMid; // range and mid pts of x & y

        // calculate ranges and mid pts
        xDiff = pts[1].X - pts[0].X;
        yDiff = pts[1].Y - pts[0].Y;
        xMid = (pts[1].X + pts[0].X) / 2;
        yMid = (pts[1].Y + pts[0].Y) / 2;

        // draw square
        g.DrawLine(blackPen, (int)pts[0].X, (int)pts[0].Y, (int)(xMid + yDiff / 2), (int)(yMid
        - xDiff / 2));
        g.DrawLine(blackPen, (int)(xMid + yDiff / 2), (int)(yMid - xDiff / 2), (int)pts[1].X,
        (int)pts[1].Y);
        g.DrawLine(blackPen, (int)pts[1].X, (int)pts[1].Y, (int)(xMid - yDiff / 2), (int)(yMid
        + xDiff / 2));
    }
}

```

```

        g.DrawLine(blackPen, (int)(xMid - yDiff / 2), (int)(yMid + xDiff / 2), (int)pts[0].X,
(int)pts[0].Y);
    }

    //Method to move the square to Origin
    public void Origin()
    {
        pts[0].X -= midPt.X;
        pts[0].Y -= midPt.Y;
        pts[1].X -= midPt.X;
        pts[1].Y -= midPt.Y;
    }

    //Method to move the square to initialCoordinates
    public void initialCoordinates()
    {
        pts[0].X += midPt.X;
        pts[0].Y += midPt.Y;
        pts[1].X += midPt.X;
        pts[1].Y += midPt.Y;
    }

    // Method to resize the square
    public override void transformResize(float scalarVal)
    {
        // move to origin
        this.Origin();

        // Scalar Matrix
        float[,] scalarMatrixVal = { { scalarVal, 0 }, { 0, scalarVal } };

        // square pts * scalarMatrixVal
        float[,] firstPts = { { pts[0].X, pts[0].Y } };
        firstPts = calc.scalarVal(firstPts, scalarMatrixVal);
        pts[0].X = Convert.ToInt32(firstPts[0, 0]);
        pts[0].Y = Convert.ToInt32(firstPts[0, 1]);

        float[,] secondPts = { { pts[1].X, pts[1].Y } };
        secondPts = calc.scalarVal(secondPts, scalarMatrixVal);
        pts[1].X = Convert.ToInt32(secondPts[0, 0]);
        pts[1].Y = Convert.ToInt32(secondPts[0, 1]);

        // move to initial origin
        this.initialCoordinates();
    }

    // Method to move the square
    public override void transformMove(float x, float y)
    {

```

```

int oppPtX = Convert.ToInt32(x);
int oppPtY = Convert.ToInt32(y);
pts[0].X += oppPtX;
pts[0].Y += oppPtY;
pts[1].X += oppPtX;
pts[1].Y += oppPtY;

// calc new midPt
midPt.X = (pts[0].X + pts[1].X) / 2;
midPt.Y = (pts[0].Y + pts[1].Y) / 2;
}
// Method to rotate the square
public override void transformRotate(float angle)
{
    // move to origin
    this.Origin();
    // Rotation Matrix
    float cosa = (float)Math.Cos(angle * Math.PI / 180.0);
    float sina = (float)Math.Sin(angle * Math.PI / 180.0);
    float sinaN = sina * -1;
    float[,] rotationMatrixVal = { { cosa, sina }, { sinaN, cosa } };
    // square pts * scalarMatrixVal
    float[,] firstPts = { { pts[0].X, pts[0].Y } };
    firstPts = calc.rotationVal(firstPts, rotationMatrixVal);
    pts[0].X = Convert.ToInt32(firstPts[0, 0]);
    pts[0].Y = Convert.ToInt32(firstPts[0, 1]);
    float[,] secondPts = { { pts[1].X, pts[1].Y } };
    secondPts = calc.rotationVal(secondPts, rotationMatrixVal);
    pts[1].X = Convert.ToInt32(secondPts[0, 0]);
    pts[1].Y = Convert.ToInt32(secondPts[0, 1]);

    // move to initial origin
    this.initialCoordinates();
}
}
class Circle : Shape
{
    double distance; // distance between the 2x given pts
    double radius;
    int shapeX;

    //This class contains the specific details for a square defined in terms of opposite
    corners
    Point[] pts = new Point[2]; // these points identify opposite corners of the circle
    Point midPt = new Point();
    string ItemToDraw;

```

```

Point pixel = new Point();
Point center = new Point();

public Circle(Point pts1, Point pts2) // constructor
{
    pts[0] = pts1;
    pts[1] = pts2;
    // calc midPt
    center.X = (pts[0].X + pts[1].X) / 2;
    center.Y = (pts[0].Y + pts[1].Y) / 2;
    ItemToDraw = "Circle";
    midPt = pts1;

    // calc distance radius
    distance = Math.Sqrt(Math.Pow(Convert.ToDouble((pts[1].X - pts[0].X)), 2) +
Math.Pow(Convert.ToDouble((pts[1].Y - pts[0].Y)), 2));
    radius = distance / 2;
}

private void putPixel(Graphics g, Point pixel, Pen pen)
{
    if (pen.Color == Color.Red)
    {
        Brush aBrush = (Brush)Brushes.Red;
        // FillRectangle call fills at location x y and is 1 pixel high by 1 pixel wide
        g.FillRectangle(aBrush, pixel.X, pixel.Y, 1, 1);
    }
    else
    {
        Brush aBrush = (Brush)Brushes.Black;
        // FillRectangle call fills at location x y and is 1 pixel high by 1 pixel wide
        g.FillRectangle(aBrush, pixel.X, pixel.Y, 1, 1);
    }
}

// Abstract
public override string GetItemToDraw() { return ItemToDraw; }
public override Point calcMidPoint() { return midPt; }

// You will need a different draw method for each kind of shape. Note the square is
drawn
// from first principles. All other shapes should similarly be drawn from first
principles.
// Ideally no C# standard library class or method should be used to create, draw or
transform a shape
// and instead should utilise user-developed code.

```

```

public override void drawM(Graphics g, Pen blackPen)
{
    int x = 0;
    int y = Convert.ToInt32(radius);
    int d = 3 - 2 * Convert.ToInt32(radius); // initial value

    while (x <= y)
    {
        // put pixel in each octant
        pixel.X = x + center.X;
        pixel.Y = y + center.Y;
        putPixel(g, pixel, blackPen);

        pixel.X = y + center.X;
        pixel.Y = x + center.Y;
        putPixel(g, pixel, blackPen);

        pixel.X = y + center.X;
        pixel.Y = -x + center.Y;
        putPixel(g, pixel, blackPen);

        pixel.X = x + center.X;
        pixel.Y = -y + center.Y;
        putPixel(g, pixel, blackPen);

        pixel.X = -x + center.X;
        pixel.Y = -y + center.Y;
        putPixel(g, pixel, blackPen);

        pixel.X = -y + center.X;
        pixel.Y = -x + center.Y;
        putPixel(g, pixel, blackPen);

        pixel.X = -y + center.X;
        pixel.Y = x + center.Y;
        putPixel(g, pixel, blackPen);

        pixel.X = -x + center.X;
        pixel.Y = y + center.Y;
        putPixel(g, pixel, blackPen);

        // update d value
        if (d <= 0)
        {
            d = d + 4 * x + 6;
        }
    }
}

```

```

        else
        {
            d = d + 4 * (x - y) + 10;
            y--;
        }
        x++;
    }
}

//Method to move the circle to Origin
public void Origin()
{
    for (shapeX = 0; shapeX < 2; shapeX++)
    {
        pts[shapeX].X -= center.X;
        pts[shapeX].Y -= center.Y;
    }
}

//Method to move the circle to initialCoordinates
public void initialCoordinates()
{
    for (shapeX = 0; shapeX < 2; shapeX++)
    {
        pts[shapeX].X += center.X;
        pts[shapeX].Y += center.Y;
    }
}

// Method to resize the circle
public override void transformResize(float scalarVal)
{
    // move to origin
    this.Origin();

    // Scalar Matrix
    float[,] scalarMatrixVal = { {scalarVal, 0}, {0, scalarVal} };

    // circle pts * scalarMatrixVal
    float[,] firstPts = { { pts[0].X, pts[0].Y } };
    firstPts = calc.scalarVal(firstPts, scalarMatrixVal);
    pts[0].X = Convert.ToInt32(firstPts[0, 0]);
    pts[0].Y = Convert.ToInt32(firstPts[0, 1]);

    float[,] secondPts = { { pts[1].X, pts[1].Y } };
    secondPts = calc.scalarVal(secondPts, scalarMatrixVal);
    pts[1].X = Convert.ToInt32(secondPts[0, 0]);
    pts[1].Y = Convert.ToInt32(secondPts[0, 1]);
}

```

```

        // move to initial origin
        this.initialCoordinates();

        // re-calc distance radius
        distance = Math.Sqrt(Math.Pow(Convert.ToDouble((pts[1].X - pts[0].X)), 2) +
Math.Pow(Convert.ToDouble((pts[1].Y - pts[0].Y)), 2));
        radius = distance / 2;

    }
    // Method to move the circle
    public override void transformMove(float x, float y)
    {
        int oppPtX = Convert.ToInt32(x);
        int oppPtY = Convert.ToInt32(y);

        // reset all pts
        for (shapeX = 0; shapeX < 2; shapeX++)
        {
            pts[shapeX].X += oppPtX;
            pts[shapeX].Y += oppPtY;
        }
        // calc new center
        center.X = (pts[0].X + pts[1].X) / 2;
        center.Y = (pts[0].Y + pts[1].Y) / 2;
        // calc new midPt
        midPt = pts[0];

        // re-calc distance radius
        distance = Math.Sqrt(Math.Pow(Convert.ToDouble((pts[1].X - pts[0].X)), 2) +
Math.Pow(Convert.ToDouble((pts[1].Y - pts[0].Y)), 2));
        radius = distance / 2;
    }
}
class Triangle : Shape
{
    //This class contains the specific details for a square defined in terms of opposite
    corners
    Point[] pts = new Point[3]; // these points identify opposite corners of the triangle
    Point midPt = new Point();
    string ItemToDraw;
    int shapeX;
    public Triangle(Point pts1, Point pts2, Point pts3) // constructor
    {
        // three pts used to draw
        pts[0] = pts1;
        pts[1] = pts2;
    }
}

```

```

    pts[2] = pts3;
    // calc midPt
    midPt.X = (pts[0].X + pts[1].X + pts[2].X) / 3;
    midPt.Y = (pts[0].Y + pts[1].Y + pts[2].Y) / 3;
    ItemToDraw = "Triangle";
}
// Abstract
public override Point calcMidPoint() { return midPt; }
public override string GetItemToDraw() { return ItemToDraw; }

// You will need a different draw method for each kind of shape. Note the square is
drawn
// from first principles. All other shapes should similarly be drawn from first
principles.
// Ideally no C# standard library class or method should be used to create, draw or
transform a shape
// and instead should utilise user-developed code.
public override void drawM(Graphics g, Pen blackPen)
{
    g.DrawLine(blackPen, (int)pts[0].X, (int)pts[0].Y, (int)pts[1].X, pts[1].Y);
    g.DrawLine(blackPen, (int)pts[1].X, (int)pts[1].Y, (int)pts[2].X, pts[2].Y);
    g.DrawLine(blackPen, (int)pts[2].X, (int)pts[2].Y, (int)pts[0].X, pts[0].Y);
}

//Method to move the triangle to Origin
public void Origin()
{
    for (shapeX = 0; shapeX < 3; shapeX++)
    {
        pts[shapeX].X -= midPt.X;
        pts[shapeX].Y -= midPt.Y;
    }
}

//Method to move the triangle to initialCoordinates
public void initialCoordinates()
{
    for (int shapeX = 0; shapeX < 3; shapeX++)
    {
        pts[shapeX].X += midPt.X;
        pts[shapeX].Y += midPt.Y;
    }
}

// Method to resize the triangle
public override void transformResize(float scalarVal)
{

```



```

// move to origin
this.Origin();

// Scalar Matrix
float[,] scalarMatrixVal = { { scalarVal, 0 }, { 0, scalarVal } };

// triangle pts * scalarMatrixVal
float[,] firstPts = { { pts[0].X, pts[0].Y } };
firstPts = calc.scalarVal(firstPts, scalarMatrixVal);
pts[0].X = Convert.ToInt32(firstPts[0, 0]);
pts[0].Y = Convert.ToInt32(firstPts[0, 1]);

float[,] secondPts = { { pts[1].X, pts[1].Y } };
secondPts = calc.scalarVal(secondPts, scalarMatrixVal);
pts[1].X = Convert.ToInt32(secondPts[0, 0]);
pts[1].Y = Convert.ToInt32(secondPts[0, 1]);

float[,] thirdPoint = { { pts[2].X, pts[2].Y } };
thirdPoint = calc.scalarVal(thirdPoint, scalarMatrixVal);
pts[2].X = Convert.ToInt32(thirdPoint[0, 0]);
pts[2].Y = Convert.ToInt32(thirdPoint[0, 1]);

// move to initial origin
this.initialCoordinates();
}

// Method to move the square
public override void transformMove(float x, float y)
{
    int oppPtX = Convert.ToInt32(x);
    int oppPtY = Convert.ToInt32(y);

    // reset all pts
    for (shapeX = 0; shapeX < 3; shapeX++)
    {
        pts[shapeX].X += oppPtX;
        pts[shapeX].Y += oppPtY;
    }

    // calc new midPt
    midPt.X = (pts[0].X + pts[1].X + pts[2].X) / 3;
    midPt.Y = (pts[0].Y + pts[1].Y + pts[2].Y) / 3;
}

// Method to rotate the triangle
public override void transformRotate(float angle)
{
    // move to origin

```

```

this.Origin();

// Rotation Matrix
float cosa = (float)Math.Cos(angle * Math.PI / 180.0);
float sina = (float)Math.Sin(angle * Math.PI / 180.0);
float sinaN = sina * -1;

float[,] rotationMatrixVal = { { cosa, sina }, { sinaN, cosa } };

// square pts * scalarMatrixVal
float[,] firstPts = { { pts[0].X, pts[0].Y } };
firstPts = calc.rotationVal(firstPts, rotationMatrixVal);
pts[0].X = Convert.ToInt32(firstPts[0, 0]);
pts[0].Y = Convert.ToInt32(firstPts[0, 1]);

float[,] secondPts = { { pts[1].X, pts[1].Y } };
secondPts = calc.rotationVal(secondPts, rotationMatrixVal);
pts[1].X = Convert.ToInt32(secondPts[0, 0]);
pts[1].Y = Convert.ToInt32(secondPts[0, 1]);

float[,] thirdPoint = { { pts[2].X, pts[2].Y } };
thirdPoint = calc.rotationVal(thirdPoint, rotationMatrixVal);
pts[2].X = Convert.ToInt32(thirdPoint[0, 0]);
pts[2].Y = Convert.ToInt32(thirdPoint[0, 1]);

// move to initial origin
this.initialCoordinates();
}
}
// Program.cs - Method Main()
public class GrafPackDemo
{
    public static void Main()
    {
        Application.Run(new GrafPack());
    }
}
}

```

## Appendix GrafPack.Designer:

```

namespace Grafpack
{
    partial class GrafPack
    {
        /// <summary>
        /// Required designer variable.

```

```

    /// </summary>
    private System.ComponentModel.IContainer components = null;

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    /// <param name="disposing">true if managed resources should be disposed;
    otherwise, false.</param>
    protected override void Dispose(bool disposing)
    {
        if (disposing && (components != null))
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }

    #region Windows Form Designer generated code

    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        this.components = new System.ComponentModel.Container();
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.Text = "GrafPack";
    }

    #endregion
}

```

## Appendix Utilities:

```

using CGP_SID1903999;
using System;
using System.Windows.Forms;

namespace Grafpack
{
    public partial class Utilities : Form
    {
        // Declare Var
        private string selectedShapeltem;
        private float userInputVal;
    }
}

```

```

private float lowestVal;
private float highestVal;

public Utilities(string itemToDraw, float highest, float lowest)
{
    loadUtilities();

    selectedShapelItem = itemToDraw;
    highestVal = highest;
    lowestVal = lowest;
    label1.Text = "";
    label1.Text = (String.Format("{0}, {1} and {2}", selectedShapelItem, lowestVal,
highestVal));
}
private void loadUtilities() { InitializeComponent(); }
private void Utilities_Load(object sender, EventArgs e) { }
private void Label1_Click(object sender, EventArgs e) { }
private void Cancel_Click(object sender, EventArgs e)
{
    userInputVal = 2147483647; // cancelled op
    this.Close();
}
public float inputVal() { return userInputVal; }
private void Enter_Button(object sender, EventArgs e)
{
    int nothing = 0;
    userInputVal = float.Parse(textBox1.Text); // Parse userInputVal to text

    if (userInputVal == nothing)
    {
        MessageBox.Show("Cannot Scale by 0");
        this.Close();
    }
    else if (userInputVal >= lowestVal) { this.Close(); }
    else if (userInputVal <= highestVal) { this.Close(); }
    else
    {
        // if scale value is beyond reach
        MessageBox.Show(String.Format("Please input a number between {0} and
{1}", lowestVal, highestVal));
        this.Close();
        // new form
        Utilities getSet = new Utilities(selectedShapelItem, lowestVal, highestVal);
    }
}
}

```

```
}  
}
```

## Appendix Utilities.Designer:

```
using CGP_SID1903999;  
  
namespace Grafpack  
{  
    partial class Utilities  
    {  
        /// <summary>  
        /// Required designer variable.  
        /// </summary>  
        private System.ComponentModel.IContainer components = null;  
  
        /// <summary>  
        /// Clean up any resources being used.  
        /// </summary>  
        /// <param name="disposing">true if managed resources should be disposed;  
        otherwise, false.</param>  
        protected override void Dispose(bool disposing)  
        {  
            if (disposing && (components != null))  
            {  
                components.Dispose();  
            }  
            base.Dispose(disposing);  
        }  
  
        #region Windows Form Designer generated code  
  
        /// <summary>  
        /// Required method for Designer support - do not modify  
        /// the contents of this method with the code editor.  
        /// </summary>  
        private void InitializeComponent()  
        {  
            this.button1 = new System.Windows.Forms.Button();  
            this.textBox1 = new System.Windows.Forms.TextBox();  
            this.label1 = new System.Windows.Forms.Label();  
            this.button2 = new System.Windows.Forms.Button();  
            this.SuspendLayout();  
  
            // Enter Button  
            this.button1.Location = new System.Drawing.Point(40, 130);  
            this.button1.Name = "button1";  
        }  
    }  
}
```

```

this.button1.Size = new System.Drawing.Size(75, 25);
this.button1.TabIndex = 0;
this.button1.Text = "Enter";
this.button1.UseVisualStyleBackColor = true;
this.button1.Click += new System.EventHandler(this.Enter_Button);
// textbox1
this.textBox1.Location = new System.Drawing.Point(140, 90);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(100, 20);
this.textBox1.TabIndex = 1;
// label1
this.label1.AutoSize = true;
this.label1.Font = new System.Drawing.Font("Times New Roman", 12.0F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.label1.Location = new System.Drawing.Point(95, 30);
this.label1.Name = "label1";
this.label1.Size = new System.Drawing.Size(55, 20);
this.label1.TabIndex = 2;
this.label1.Text = "label1";
this.label1.Click += new System.EventHandler(this.Label1_Click);
// Cancel Button
this.button2.Location = new System.Drawing.Point(250, 130);
this.button2.Name = "button2";
this.button2.Size = new System.Drawing.Size(75, 25);
this.button2.TabIndex = 3;
this.button2.Text = "Cancel";
this.button2.UseVisualStyleBackColor = true;
this.button2.Click += new System.EventHandler(this.Cancel_Click);
// Utilities
this.AccessibleName = "Utilities";
this.AutoScaleDimensions = new System.Drawing.SizeF(6.0F, 13.0F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleModeMode.Font;
this.ClientSize = new System.Drawing.Size(400, 165);
this.Controls.Add(this.button2);
this.Controls.Add(this.label1);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Transform Mode ON";
this.Text = "Transform Mode ON";
this.Load += new System.EventHandler(this.Utilities_Load);
this.ResumeLayout(false);
this.PerformLayout();

}
#endregion

```

```
private System.Windows.Forms.Button button1;  
private System.Windows.Forms.TextBox textBox1;  
private System.Windows.Forms.Button button2;  
public System.Windows.Forms.Label label1;  
}  
}
```