# Computer Graphics Programming

MOD006127

## Evaluation

The program meets the specifications defined in the brief with most functions added. The program can draw from a selection of 4 different shapes which all inherit features from a main shape class. The shapes are stored within a list which was selected due to it deleting the shape from memory when it is removed from a list. Transformations were implemented for multiple functions those being rotation, scaling and translation (moving shapes), both scaling and rotation utilise matrix multiplication as specified. All functions work as planned, deletion removes a shape from the screen while also removing it from the list and memory. All shapes are drawn from basic principles as opposed to using GDI functions, although the square was built mainly from code provided and the circle utilised code provided in week 5’s materials. Rotation works as expected and scaling can both upscale and downscale shapes. The movement was implemented as planned to allow both user input from mouse and keyboard.

While overall the implementation was strong some improvements could be made to the program. The addition of further shapes would give the program more variety. Further transformations could have been implemented such as reflection adding to options given to the user. Overall, the code layout was clean with operations broken down into separate methods to give the code a modular structure allowing for easy refactoring of methods. The main code is contained within the Grafpack class with over 400 lines I feel this could have been broken down with potential use of other classes to give it a better overall view and making it easier to view.

The key strengths lie in the easy addition of new shapes adhering the same structure as the implemented 4 shapes. The user input form can easily be tailored to each situation take different inputs for input type and a max and min input value allowing easy application for further added code. The multiplication methods implemented allow easy application for any shape class accepting the input and transformation of any point from any shape allowing for easy implementation to further added shapes.

## 2.0 Appendix 1: Instructions

Run the executable file Grafpack, options for creating new shapes, selecting shapes, transformations, movement and deletions will all be displayed at the top of the window.

## Appendix 2: Pseudocode

## Grafpack.cs

using Grafpack;

using System;

using System.Drawing;

using System.Windows.Forms;

using System.Collections.Generic;

namespace GrafPack

{

public partial class GrafPack : Form

{

private List<Shape> listOfShapes = new List<Shape>(); //Definition of List used to store all shapes

//Definition of various bools to define which action is being performed

private bool selectCircleStatus = false;

private bool selectSquareStatus = false;

private bool selectTriangleStatus = false;

private bool selectRectStatus = false;

private bool selectShapeStatus = false;

private bool selectShapeUsingMouse = false;

private bool moveUsingMouseID = false;

private bool isShapeSelected = false;

//Definition of variables used for storage in various methods

private int clicknumber = 0;

private int selectedShape = 0;

private Point one, two, three, clickedPoint;

private int lowestDistance;

public GrafPack()

{

this.SetStyle(ControlStyles.ResizeRedraw, true);

this.WindowState = FormWindowState.Maximized;

this.BackColor = Color.White;

this.KeyPreview = true;

// The following approach uses menu items coupled with mouse clicks

//All menu items added for various functions implemented

MainMenu mainMenu = new MainMenu();

MenuItem createShape = new MenuItem();

MenuItem selectShape = new MenuItem();

MenuItem usingKeyboard = new MenuItem();

MenuItem usingMouse = new MenuItem();

MenuItem moveShape = new MenuItem();

MenuItem moveUsingKey = new MenuItem();

MenuItem moveUsingMouse = new MenuItem();

MenuItem transformShape = new MenuItem();

MenuItem deleteShape = new MenuItem();

MenuItem exit = new MenuItem();

MenuItem squareShape = new MenuItem();

MenuItem triangleShape = new MenuItem();

MenuItem circleShape = new MenuItem();

MenuItem rectShape = new MenuItem();

MenuItem scale = new MenuItem();

MenuItem rotate = new MenuItem();

//Definition of text displayed to user for all menu options

createShape.Text = "&Create";

squareShape.Text = "&Square";

triangleShape.Text = "&Triangle";

circleShape.Text = "&Circle";

rectShape.Text = "&Rectangle";

selectShape.Text = "&Select";

moveShape.Text = "&Move Shape";

moveUsingKey.Text = "&Move using keyboard input";

moveUsingMouse.Text = "&Move using Mouse input";

usingKeyboard.Text = "&Select using Keyboard";

usingMouse.Text = "&Select using Mouse";

transformShape.Text = "&Transform";

scale.Text = "&Scale";

rotate.Text = "&Rotate";

deleteShape.Text = "&Delete";

exit.Text = "&exit";

//Addition of various menu items to form

mainMenu.MenuItems.Add(createShape);

createShape.MenuItems.Add(squareShape);

createShape.MenuItems.Add(triangleShape);

createShape.MenuItems.Add(circleShape);

createShape.MenuItems.Add(rectShape);

mainMenu.MenuItems.Add(selectShape);

selectShape.MenuItems.Add(usingKeyboard);

selectShape.MenuItems.Add(usingMouse);

mainMenu.MenuItems.Add(moveShape);

moveShape.MenuItems.Add(moveUsingKey);

moveShape.MenuItems.Add(moveUsingMouse);

mainMenu.MenuItems.Add(transformShape);

transformShape.MenuItems.Add(scale);

transformShape.MenuItems.Add(rotate);

mainMenu.MenuItems.Add(deleteShape);

mainMenu.MenuItems.Add(exit);

//Click events added to active methods upon menu items being clicked

exit.Click += new System.EventHandler(this.SelectExit);

deleteShape.Click += new System.EventHandler(this.DeleteShape);

squareShape.Click += new System.EventHandler(this.SelectSquare);

triangleShape.Click += new System.EventHandler(this.SelectTriangle);

rectShape.Click += new System.EventHandler(this.SelectRect);

circleShape.Click += new System.EventHandler(this.SelectCircle);

moveUsingKey.Click += new System.EventHandler(this.MoveUsingKeyboard);

moveUsingMouse.Click += new System.EventHandler(this.MoveUsingMouse);

scale.Click += new System.EventHandler(this.ScaleShapes);

rotate.Click += new System.EventHandler(this.RotateShape);

usingKeyboard.Click += new System.EventHandler(this.SelectShapeKey);

usingMouse.Click += new System.EventHandler(this.SelectShapeMouse);

this.Menu = mainMenu;

this.MouseClick += MousePointClick; //Mouse click events stored

this.KeyDown += KeyboardPressed; // Key events stored

}

//Method for exitig the program

private void SelectExit(object sender, EventArgs e)

{

MessageBox.Show("Closing Application now"); // Displays to user

Application.Exit();// Terminates the program

}

// Generally, all methods of the form are usually private

private void SelectSquare(object sender, EventArgs e)

{

selectSquareStatus = true; //Bool set to true for activation of another method

MessageBox.Show("Click OK and then click once each at two locations to create a square");

}

//Method for triangle selected in menu

private void SelectTriangle(object sender, EventArgs e)

{

selectTriangleStatus = true; //Bool set to true for activation of another method

MessageBox.Show("Click OK and then click once each at three locations to draw a triangle");

}

//Method for Circle selected in menu

private void SelectCircle(object sender, EventArgs e)

{

selectCircleStatus = true; //Bool set to true for activation of another method

MessageBox.Show("Click OK and then click two locations to set the diameter of the circle");

}

private void SelectRect(object sender, EventArgs e)

{

selectRectStatus = true;

MessageBox.Show("Click OK and then click two locations to set the 2 corners of the rectangle");

}

//Method for select shape using keyboard being clicked

private void SelectShapeKey(object sender, EventArgs e)

{

selectShapeStatus = true;

if (listOfShapes.Count == 0) // Checks if any shapes are currently stored in the List

{

MessageBox.Show("No shapes currently stored.");

}

else

{

MessageBox.Show("You selected the Select option selected shapes will be highlighted with a red outline"); //Displays to user

MessageBox.Show("Use the arrow keys to navigate shapes and enter when desired shape is selected");

}

}

//Method for selecting a shape using mouse

private void SelectShapeMouse(object sender, EventArgs e)

{

selectShapeStatus = true;

if (listOfShapes.Count == 0) // Check if the list currently contains any shapes

{

MessageBox.Show("No shapes currently stored.");

}

else

{

MessageBox.Show("Please click on the middle of the shape you wish to select.");

selectShapeUsingMouse = true; //Bool set to true for activtion 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 MousePointClick(object sender, MouseEventArgs e)

{

if (e.Button == MouseButtons.Left)

{

int xDistance;

int yDistance;

// '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.

if (selectShapeUsingMouse == true)

{

lowestDistance = 9999999; //set to a high value so we can obtain the closest object

clickedPoint = new Point(e.X, e.Y); //Obtain the point clicked by user

for (int i = 0; i < listOfShapes.Count; i++) //Loop through all stored shapes

{

Point currentShapeMid = listOfShapes[i].GetMid(); //Returns midpoint of current shape being examined

xDistance = Math.Abs(clickedPoint.X - currentShapeMid.X); // calculation of difference in X distance

yDistance = Math.Abs(clickedPoint.Y - currentShapeMid.Y);//Calculation of difference in Y distance

if (xDistance < lowestDistance || yDistance < lowestDistance) // Check if either distance is lower than currently stored lowest

{

lowestDistance = xDistance;

if(yDistance < xDistance)

{

lowestDistance = yDistance;

}

selectedShape = i; // If distance is lowest we know this was the shape the user wanted to select

}

}

selectShapeUsingMouse = false; //Reset bool to false so as to not retrigger the method

isShapeSelected = true; // set to true to show that the user has selected a shape

RedrawShapes(); // Redraw shapes will show selected in red outline

}

else if (moveUsingMouseID == true)

{

clickedPoint = new Point(e.X, e.Y); //Stores the point the user clicked

Point currentShapeMid = listOfShapes[selectedShape].GetMid();//Returns mid point of currently selected shape

xDistance = clickedPoint.X - currentShapeMid.X;

yDistance = clickedPoint.Y - currentShapeMid.Y;

listOfShapes[selectedShape].Move(xDistance, yDistance); //Activates the method stored to move a shape by the distance from current mid point to the clicked point

this.Refresh();//Refreshes the form

RedrawShapes();//Redraws all shapes so the selected shape is moved to its new position

moveUsingMouseID = false; //Set to false so code does not reactivate

}

else if (selectSquareStatus == true) // Activates if user is drawing a new square

{

if (clicknumber == 0) //click number used for storing points of square

{

one = new Point(e.X, e.Y);

clicknumber = 1; //Increase click to activate else statement

}

else

{

two = new Point(e.X, e.Y);

NewSquare(one, two); // Creation of new square with user input points

}

}

else if (selectTriangleStatus == true) // creation of new triangle

{

if (clicknumber == 0)

{

one = new Point(e.X, e.Y);

clicknumber++;

}

else if (clicknumber == 1)

{

two = new Point(e.X, e.Y);

clicknumber++;

}

else

{

three = new Point(e.X, e.Y);

NewTriangle(one, two, three); //Creates new triangle with 3 user defined points

}

}

else if (selectCircleStatus == true)//Creation of new circle

{

if (clicknumber == 0)

{

one = new Point(e.X, e.Y);

clicknumber = 1;

}

else

{

two = new Point(e.X, e.Y);

NewCircle(one, two); //Create the new circle with user defined points

}

}

else if (selectRectStatus == true)//Creation of new Rectangle

{

if (clicknumber == 0)

{

one = new Point(e.X, e.Y);

clicknumber = 1;

}

else

{

two = new Point(e.X, e.Y);

NewRectangle(one, two); //Create the new Rectangle with user defined points

}

}

}

}

//Method for new square

private void NewSquare(Point pt1, Point pt2)

{

clicknumber = 0; //Reset click numbers for later use

selectSquareStatus = false; // reset the bool

Graphics g = this.CreateGraphics(); //creation of graphics object

Pen blackpen = new Pen(Color.Black); // Creation of pen for drawing

Square aShape = new Square(pt1, pt2); //Creation of square object

aShape.Draw(g, blackpen);//Draw the shape using draw method in the shapes class

listOfShapes.Add(aShape);//Addition of Square to the shapes list

}

//Method for new triangle utilises similar code to square

private void NewTriangle(Point pt1, Point pt2, Point pt3)

{

clicknumber = 0;

selectTriangleStatus = false;

Graphics g = this.CreateGraphics();

Pen blackpen = new Pen(Color.Black);

Triangle aShape = new Triangle(pt1, pt2, pt3);

aShape.Draw(g, blackpen);

listOfShapes.Add(aShape);

}

//Method for new circle Utilises similar code to circle

private void NewCircle(Point pt1, Point pt2)

{

clicknumber = 0;

selectCircleStatus = false;

Graphics g = this.CreateGraphics();

Pen blackPen = new Pen(Color.Black);

Circle aShape = new Circle(pt1, pt2);

aShape.Draw(g, blackPen);

listOfShapes.Add(aShape);

}

//Method for a new rectangle

private void NewRectangle(Point pt1, Point pt2)

{

clicknumber = 0;

selectRectStatus = false;

Graphics g = this.CreateGraphics();

Pen blackPen = new Pen(Color.Black);

Grafpack.Rectangle aShape = new Grafpack.Rectangle(pt1, pt2);

aShape.Draw(g, blackPen);

listOfShapes.Add(aShape);

}

//Method for shape movement using keyboard

private void MoveUsingKeyboard(object sender, EventArgs e)

{

float check = 999999999;

if (isShapeSelected == false) //Check a shape has been selected

{

MessageBox.Show("No shape currently selected");

}

else if (listOfShapes.Count == 0) //CHeck if list of shapes is empty

{

MessageBox.Show("No shapes currently stored.");

}

else

{

MessageBox.Show("You are now going to move the currently selected shape using keyboard input movement values.");

float movementx = GetUserInput("movement value for X between ", -500, 500);//Takes a user input for X movement

float movementy = GetUserInput("movement value for Y between ", -500, 500);//Takes a user input for Y movement

if (movementx == check || movementy == check)

{

}

else

{

listOfShapes[selectedShape].Move(movementx, movementy); // Activates move method uniquely in each shapes class

this.Refresh();

RedrawShapes();

}

}

}

//Method for movement using mouse

private void MoveUsingMouse(object sender, EventArgs e)

{

if (isShapeSelected == false) //Check a shape has been selected

{

MessageBox.Show("No shape currently selected");

}

else if (listOfShapes.Count == 0) //CHeck if list of shapes is empty

{

MessageBox.Show("No shapes currently stored.");

}

else

{

MessageBox.Show("You are now going to move the currently selected shape using Mouse input.");

MessageBox.Show("Please click where you would like to move the currently selected shape to");

moveUsingMouseID = true;

}

}

//Method to detect keypresses

private void KeyboardPressed(object sender, KeyEventArgs e)

{

if (selectShapeStatus == true)

{

if (e.KeyCode == Keys.Left)//Activates if key pressed is left

{

selectedShape -= 1; //Moves through the list of shapes selectedShape ultimately stores a selected shapes location in the list

if (selectedShape <= -1) //Checks if we reached the start of the list

{

selectedShape = listOfShapes.Count - 1;

}

}

else if (e.KeyCode == Keys.Right)

{

selectedShape += 1;

if (selectedShape == listOfShapes.Count)//Check if we reached end of list

{

selectedShape = 0;

}

}

else if (e.KeyCode == Keys.Enter) //Finishes the method when user hits enter key

{

selectShapeStatus = false; //reset bool

isShapeSelected = true;//Set the bool to identify if a shape has been selected

}

}

RedrawShapes(); //Redraw so currently selected shape is shown to user in red

}

//Method to redraw all currently stored shapes

public void RedrawShapes()

{

Graphics g = this.CreateGraphics();

Pen blackPen = new Pen(Color.Black);

Pen redPen = new Pen(Color.Red); //Red pen used to identify selected shape

for (int i = 0; i < listOfShapes.Count; i ++) // Loop through all shapes

{

if(i == selectedShape) //Check if the current shape is the selected shape

{

listOfShapes[i].Draw(g, redPen); //Activate draw method with red pen to highlight selected shape

}

else

{

listOfShapes[i].Draw(g, blackPen); //Activate draw method

}

}

}

//Method for scaling of shapes

private void ScaleShapes(object sender, EventArgs e)

{

if(isShapeSelected == false) //Check a shape has been selected

{

MessageBox.Show("No shape currently selected");

}

else

{

float scalerInput = GetUserInput("Scaler value between ", 0, 10); //Takes user input using user input form with defined variables on creation

if (scalerInput == 999999999) // used when a user cancels a scaling

{

return;

}

else

{

listOfShapes[selectedShape].Scale(scalerInput);

this.Refresh();

RedrawShapes();

}

}

}

private void DeleteShape(object sender, EventArgs e)

{

if (isShapeSelected == false) //Check a shape has been selected

{

MessageBox.Show("No shape currently selected");

}

//Displays a message to user to confirm shape deletion

else if (MessageBox.Show("You are about to delete the currently selected shape", "Please Confirm", MessageBoxButtons.YesNo, MessageBoxIcon.Question) == DialogResult.Yes)

{

listOfShapes.Remove(listOfShapes[selectedShape]); //Deletion of the currently selected shape

this.Refresh();//Refresh the screen to remove deleted shape

}

else

{

return;

}

RedrawShapes(); //Redraw shape method called

}

private void InitializeComponent()

{

this.SuspendLayout();

//

// GrafPack

//

this.ClientSize = new System.Drawing.Size(278, 244);

this.Name = "GrafPack";

this.Load += new System.EventHandler(this.GrafPack\_Load);

this.ResumeLayout(false);

}

private void GrafPack\_Load(object sender, EventArgs e)

{

}

//Method for getting user input takes input for form definitions

public float GetUserInput(string inputType, int lower, int upper)

{

float input;

UserInput userInput = new UserInput(inputType, lower, upper); //Creation of a user input form

userInput.ShowDialog(); //Brings user input form to the front and stops actions on main form

input = userInput.GetInput(); // Take the users input from user input form

userInput.Close(); // Close deletes the form and removes all resources used by it

return input; //Returns the users input from the form

}

//Method for the rotation of a shape

private void RotateShape(object sender, EventArgs e)

{

if (isShapeSelected == false) //Check a shape has been selected

{

MessageBox.Show("No shape currently selected");

}

else if (listOfShapes[selectedShape].GetType() == "Circle") //check if shape is a circle as no point rotating a circle

{

MessageBox.Show("Cannot rotate a circle");

}

else

{

float rotateInput = GetUserInput("Rotation angle between ", 0, 360); //Definitions for user input form for rotation

if (rotateInput == 999999999) // Used in the event of cancel clicked on form

{

return;

}

else

{

listOfShapes[selectedShape].Rotate(rotateInput); //Calls the rotation method on the selected shape

this.Refresh(); //Refresh screen and redraw all shapes to display the rotation

RedrawShapes();

}

}

}

public static void Main()

{

Application.Run(new GrafPack()); //Runs the application

}

}

}

## Circle.cs

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Grafpack

{

class Circle : Shape

{

private double diameter, radius; //variables defined for storage

Point[] points = new Point[2];

Point plotPt, centre;

private string type;

public new static Point midPoint;

public Circle(Point pt1, Point pt2)

{

points[0] = pt1;

points[1] = pt2;

centre.X = (points[0].X + points[1].X) / 2;

centre.Y = (points[0].Y + points[1].Y) / 2;

type = "Circle";

midPoint = pt1;

diameter = Math.Sqrt(Math.Pow(Convert.ToDouble((points[1].X - points[0].X)), 2) + Math.Pow(Convert.ToDouble((points[1].Y - points[0].Y)), 2));

radius = diameter / 2;

}

public override string GetType()

{

return type;

}

public override Point GetMid()

{

return midPoint;

}

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);

}

}

//Draw method Utilises the BresCircle code from Week 5 representation of geometrical shapes

public override void Draw(Graphics g, Pen pen)

{

int x = 0;

int y = Convert.ToInt32(radius);

int d = 3 - 2 \* Convert.ToInt32(radius); // initial value

while (x <= y)

{

// plot pixel in each octant

plotPt.X = x + centre.X;

plotPt.Y = y + centre.Y;

putPixel(g, plotPt, pen);

plotPt.X = y + centre.X;

plotPt.Y = x + centre.Y;

putPixel(g, plotPt, pen);

plotPt.X = y + centre.X;

plotPt.Y = -x + centre.Y;

putPixel(g, plotPt, pen);

plotPt.X = x + centre.X;

plotPt.Y = -y + centre.Y;

putPixel(g, plotPt, pen);

plotPt.X = -x + centre.X;

plotPt.Y = -y + centre.Y;

putPixel(g, plotPt, pen);

plotPt.X = -y + centre.X;

plotPt.Y = -x + centre.Y;

putPixel(g, plotPt, pen);

plotPt.X = -y + centre.X;

plotPt.Y = x + centre.Y;

putPixel(g, plotPt, pen);

plotPt.X = -x + centre.X;

plotPt.Y = y + centre.Y;

putPixel(g, plotPt, pen);

// update d value

if (d <= 0)

{

d = d + 4 \* x + 6;

}

else

{

d = d + 4 \* (x - y) + 10;

y--;

}

x++;

}

}

public override void Scale(float scaler)

{

//First we set the objects midpoint to the origin (0,0)

this.MoveToOrigin();

//we can then multiply these points by the scaler to scale the object

//Definition of scaler matrix

float[,] scalerMatrix = { {scaler, 0},

{0, scaler }};

//Multiply both points of the circle by the scaler matrix

float[,] firstPoint = { { points[0].X, points[0].Y } };

float[,] secondPoint = { { points[1].X, points[1].Y } };

firstPoint = MatrixMultiplication.ScalerMultiplier(firstPoint, scalerMatrix);

points[0].X = Convert.ToInt32(firstPoint[0, 0]);

points[0].Y = Convert.ToInt32(firstPoint[0, 1]);

secondPoint = MatrixMultiplication.ScalerMultiplier(secondPoint, scalerMatrix);

points[1].X = Convert.ToInt32(secondPoint[0, 0]);

points[1].Y = Convert.ToInt32(secondPoint[0, 1]);

//we then move the object back to its original origin;

this.MoveToOriginalPlacement();

//calculate the new diameter radius for the new points

diameter = Math.Sqrt(Math.Pow(Convert.ToDouble((points[1].X - points[0].X)), 2) + Math.Pow(Convert.ToDouble((points[1].Y - points[0].Y)), 2));

radius = diameter / 2;

}

//Method to move object midpoint to 0,0 for transformations

public void MoveToOrigin()

{

for (int i = 0; i < 2; i++)

{

points[i].X -= centre.X;

points[i].Y -= centre.Y;

}

}

//Method to move the object back to its original place

public void MoveToOriginalPlacement()

{

for (int i = 0; i < 2; i++)

{

points[i].X += centre.X;

points[i].Y += centre.Y;

}

}

public override void Move(float x, float y) //Method for moving a shape used for both mouse click and user input

{

int newX = Convert.ToInt32(x);

int newY = Convert.ToInt32(y);

for (int i = 0; i < 2; i++) //set new points for all points stored in array

{

points[i].X += newX;

points[i].Y += newY;

}

centre.X = (points[0].X + points[1].X) / 2; //calculate new centre

centre.Y = (points[0].Y + points[1].Y) / 2;

midPoint = points[0]; //calculate new midpoint

diameter = Math.Sqrt(Math.Pow(Convert.ToDouble((points[1].X - points[0].X)), 2) + Math.Pow(Convert.ToDouble((points[1].Y - points[0].Y)), 2));

radius = diameter / 2;

}

}

}

## MatrixMultiplication.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Grafpack

{

class MatrixMultiplication

{

public static float[,] m3 = { { 0, 0 } }; // used to store new point returned

public static int incrementer;

//Method containing matrix multiplication for scaling of a given point

public static float[,] ScalerMultiplier(float[,] m1, float[,] m2)

{

//Loops used for matrix multiplication

for (int i = 0; i < 1; i++)

{

for (int j = 0; j < 2; j++)

{

m3[i, j] = 0;

for (incrementer = 0; incrementer < 2; incrementer++) // Incrementer is used to access X and Y locations

{

m3[i, j] += m1[i, j] \* m2[incrementer, j];

}

}

}

return m3;

}

//Method containing calculations for rotation of a given point of a shape

public static float[,] RotationMultiplier(float[,] m1, float[,] m2)

{

m3[0, 0] = (m1[0, 0] \* m2[0, 0]) - (m1[0, 1] \* m2[0, 1]);

m3[0, 1] = (m1[0, 0] \* m2[0, 1]) + (m1[0, 1] \* m2[0, 0]);

return m3;

}

}

}

## Rectangle.cs

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Grafpack

{

class Rectangle : Shape

{

private Point[] points = new Point[4]; //Points array of the 4 points of rectangle

public new static Point midPoint;

private string type;

public Rectangle(Point pt1, Point pt2)

{

points[0] = pt1;

points[1] = pt2;

midPoint.X = (pt1.X + pt2.X) / 2;

midPoint.Y = (pt1.Y + pt2.Y) / 2;

type = "Rectangle";

//Creation of other 2 points

points[2].X = points[0].X;

points[2].Y = points[1].Y;

points[3].X = points[1].X;

points[3].Y = points[0].Y;

}

public override string GetType()

{

return type;

}

public override Point GetMid()

{

return midPoint;

}

public override void Draw(Graphics g, Pen blackPen) //Draw method for rectangle

{

// draw Rectangle

g.DrawLine(blackPen, points[0].X, points[0].Y, points[2].X, points[2].Y);

g.DrawLine(blackPen, points[2].X, points[2].Y, points[1].X, points[1].Y);

g.DrawLine(blackPen, points[1].X, points[1].Y, points[3].X, points[3].Y);

g.DrawLine(blackPen, points[3].X, points[3].Y, points[0].X, points[0].Y);

}

//Scaling method

public override void Scale(float scaler)

{

//First we set the objects midpoint to the origin (0,0)

this.MoveToOrigin();

//Definition of scaler matrix

float[,] scalerMatrix = { {scaler, 0},

{0, scaler }};

//Multiply all points of the rectangle by the scaler matrix

float[,] firstPoint = { { points[0].X, points[0].Y } };

float[,] secondPoint = { { points[1].X, points[1].Y } };

float[,] thirdPoint = { { points[2].X, points[2].Y } };

float[,] fourthPoint = { { points[3].X, points[3].Y } };

firstPoint = MatrixMultiplication.ScalerMultiplier(firstPoint, scalerMatrix);

points[0].X = Convert.ToInt32(firstPoint[0, 0]);

points[0].Y = Convert.ToInt32(firstPoint[0, 1]);

secondPoint = MatrixMultiplication.ScalerMultiplier(secondPoint, scalerMatrix);

points[1].X = Convert.ToInt32(secondPoint[0, 0]);

points[1].Y = Convert.ToInt32(secondPoint[0, 1]);

thirdPoint = MatrixMultiplication.ScalerMultiplier(thirdPoint, scalerMatrix);

points[2].X = Convert.ToInt32(thirdPoint[0, 0]);

points[2].Y = Convert.ToInt32(thirdPoint[0, 1]);

fourthPoint = MatrixMultiplication.ScalerMultiplier(fourthPoint, scalerMatrix);

points[3].X = Convert.ToInt32(fourthPoint[0, 0]);

points[3].Y = Convert.ToInt32(fourthPoint[0, 1]);

//we then move the object back to its original origin;

this.MoveToOriginalPlacement();

}

public override void Rotate(float angle)

{

//First we set the objects midpoint to the origin (0,0)

this.MoveToOrigin();

//Next Definition of the scaler matrix

float cosa = (float)Math.Cos(angle \* Math.PI / 180.0);

float sina = (float)Math.Sin(angle \* Math.PI / 180.0);

float negativeSina = sina \* -1;

//Definition of rotation matrix

float[,] rotationMatrix = { {cosa, sina},

{negativeSina, cosa } };

float[,] firstPoint = { { points[0].X, points[0].Y } };

float[,] secondPoint = { { points[1].X, points[1].Y } };

float[,] thirdPoint = { { points[2].X, points[2].Y } };

float[,] fourthPoint = { { points[3].X, points[3].Y } };

firstPoint = MatrixMultiplication.RotationMultiplier(firstPoint, rotationMatrix);

points[0].X = Convert.ToInt32(firstPoint[0, 0]);

points[0].Y = Convert.ToInt32(firstPoint[0, 1]);

secondPoint = MatrixMultiplication.RotationMultiplier(secondPoint, rotationMatrix);

points[1].X = Convert.ToInt32(secondPoint[0, 0]);

points[1].Y = Convert.ToInt32(secondPoint[0, 1]);

thirdPoint = MatrixMultiplication.RotationMultiplier(thirdPoint, rotationMatrix);

points[2].X = Convert.ToInt32(thirdPoint[0, 0]);

points[2].Y = Convert.ToInt32(thirdPoint[0, 1]);

fourthPoint = MatrixMultiplication.RotationMultiplier(fourthPoint, rotationMatrix);

points[3].X = Convert.ToInt32(fourthPoint[0, 0]);

points[3].Y = Convert.ToInt32(fourthPoint[0, 1]);

//we then move the object back to its original origin;

this.MoveToOriginalPlacement();

}

//Method to move object midpoint to 0,0 for transformations

public void MoveToOrigin()

{

for (int i = 0; i < 4; i++)

{

points[i].X -= midPoint.X;

points[i].Y -= midPoint.Y;

}

}

//Method to move the object back to its original place

public void MoveToOriginalPlacement()

{

for (int i = 0; i < 4; i++)

{

points[i].X += midPoint.X;

points[i].Y += midPoint.Y;

}

}

public override void Move(float x, float y) //Method to move a shape

{

int newX = Convert.ToInt32(x);

int newY = Convert.ToInt32(y);

for (int i = 0; i < 4; i++)

{

points[i].X += newX;

points[i].Y += newY;

}

midPoint.X = (points[0].X + points[1].X) / 2;

midPoint.Y = (points[0].Y + points[1].Y) / 2;

}

}

}

## Shape.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Drawing;

namespace Grafpack

{

abstract class Shape

{

private string type;

public Point midPoint;

// 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.

public Shape() // constructor

{

}

//Main draw Method overriden in each shape instance

public virtual void Draw(Graphics g, Pen blackPen)

{

}

//Getters

public new virtual string GetType()

{

return type;

}

public virtual Point GetMid()

{

return midPoint;

}

//Methods overriden in each shape class containing relevent code for each shape

public virtual void Scale(float scaler)

{

}

public virtual void Rotate(float Angle)

{

}

public virtual void Move(float x, float y)

{ }

}

}

## Square.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Drawing;

using System.Windows.Forms;

using Grafpack;

class Square : Shape

{

//This class contains the specific details for a square defined in terms of opposite corners

Point[] points = new Point[2];

private string type;

public new static Point midPoint;

public Square( Point pt1, Point pt2) // constructor

{

points[0] = pt1;

points[1] = pt2;

midPoint.X = (pt1.X + pt2.X) / 2;

midPoint.Y = (pt1.Y + pt2.Y) / 2;

type = "Square";

}

public override string GetType()

{

return type;

}

public override Point GetMid()

{

return midPoint;

}

// 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 utilse user-developed code.

public override void Draw(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 points of x & y

// calculate ranges and mid points

xDiff = points[1].X - points[0].X;

yDiff = points[1].Y - points[0].Y;

xMid = (points[1].X + points[0].X) / 2;

yMid = (points[1].Y + points[0].Y) / 2;

// draw square

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

g.DrawLine(blackPen, (int)(xMid + yDiff / 2), (int)(yMid - xDiff / 2), (int)points[1].X, (int)points[1].Y);

g.DrawLine(blackPen, (int)points[1].X, (int)points[1].Y, (int)(xMid - yDiff / 2), (int)(yMid + xDiff / 2));

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

}

public override void Scale(float scaler)

{

//First we set the objects midpoint to the origin (0,0)

this.MoveToOrigin();

//Definition of scaler matrix

float[,] scalerMatrix = { {scaler, 0},

{0, scaler }};

//Multiply both points of the square by the scaler matrix

float[,] firstPoint = { { points[0].X, points[0].Y } };

float[,] secondPoint = { { points[1].X, points[1].Y } };

firstPoint = MatrixMultiplication.ScalerMultiplier(firstPoint, scalerMatrix);

points[0].X = Convert.ToInt32(firstPoint[0,0]);

points[0].Y = Convert.ToInt32(firstPoint[0, 1]);

secondPoint = MatrixMultiplication.ScalerMultiplier(secondPoint, scalerMatrix);

points[1].X = Convert.ToInt32(secondPoint[0, 0]);

points[1].Y = Convert.ToInt32(secondPoint[0, 1]);

//we then move the object back to its original origin;

this.MoveToOriginalPlacement();

}

public override void Rotate(float angle)

{

//First we set the objects midpoint to the origin (0,0)

this.MoveToOrigin();

//Next Definition of the scaler matrix

float cosa = (float)Math.Cos(angle \* Math.PI / 180.0);

float sina = (float)Math.Sin(angle \* Math.PI / 180.0);

float negativeSina = sina \* -1;

float[,] rotationMatrix = { {cosa, sina},

{negativeSina, cosa } };

float[,] firstPoint = { { points[0].X, points[0].Y } };

float[,] secondPoint = { { points[1].X, points[1].Y } };

firstPoint = MatrixMultiplication.RotationMultiplier(firstPoint, rotationMatrix);

points[0].X = Convert.ToInt32(firstPoint[0, 0]);

points[0].Y = Convert.ToInt32(firstPoint[0, 1]);

secondPoint = MatrixMultiplication.RotationMultiplier(secondPoint, rotationMatrix);

points[1].X = Convert.ToInt32(secondPoint[0, 0]);

points[1].Y = Convert.ToInt32(secondPoint[0, 1]);

//we then move the object back to its original origin;

this.MoveToOriginalPlacement();

}

//Method to move object midpoint to 0,0 for transformations

public void MoveToOrigin()

{

points[0].X -= midPoint.X;

points[0].Y -= midPoint.Y;

points[1].X -= midPoint.X;

points[1].Y -= midPoint.Y;

}

//Method to move the object back to its original place

public void MoveToOriginalPlacement()

{

points[0].X += midPoint.X;

points[0].Y += midPoint.Y;

points[1].X += midPoint.X;

points[1].Y += midPoint.Y;

}

public override void Move(float x, float y) //method to move a square

{

int newX = Convert.ToInt32(x);

int newY = Convert.ToInt32(y);

points[0].X += newX;

points[0].Y += newY;

points[1].X += newX;

points[1].Y += newY;

midPoint.X = (points[0].X + points[1].X) / 2;

midPoint.Y = (points[0].Y + points[1].Y) / 2;

}

}

## Triangle.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Drawing;

namespace Grafpack

{

class Triangle : Shape

{

private Point[] points = new Point[3]; // Points array to store points of a triangle

public new static Point midPoint; // Point to store shapes midpoint

string type;

public Triangle( Point pt1, Point pt2, Point pt3) // Traingle constructer

{

points[0] = pt1; // points stored in array

points[1] = pt2;

points[2] = pt3;

midPoint.X = (points[0].X + points[1].X + points[2].X) / 3; // MidPoint calculated

midPoint.Y = (points[0].Y + points[1].Y + points[2].Y) / 3;

type = "Triangle";

}

public override Point GetMid()

{

return midPoint; //Method to return midpoint of triangle

}

public override string GetType()

{

return type;

}

public override void Draw(Graphics g, Pen pen) //Method draws the shape

{

g.DrawLine(pen, (int)points[0].X, (int)points[0].Y, (int)points[1].X, points[1].Y); // Each line of triangle drawn

g.DrawLine(pen, (int)points[1].X, (int)points[1].Y, (int)points[2].X, points[2].Y);

g.DrawLine(pen, (int)points[2].X, (int)points[2].Y, (int)points[0].X, points[0].Y);

}

public override void Scale(float scaler)

{

//First we set the objects midpoint to the origin (0,0)

this.MoveToOrigin();

//Definition of scaler matrix

float[,] scalerMatrix = { {scaler, 0},

{0, scaler }};

//we can then multiply the 3 points by the scaler matrix to scale the object

float[,] firstPoint = { { points[0].X, points[0].Y } };

float[,] secondPoint = { { points[1].X, points[1].Y } };

float[,] thirdPoint = { { points[2].X, points[2].Y } };

firstPoint = MatrixMultiplication.ScalerMultiplier(firstPoint, scalerMatrix);

points[0].X = Convert.ToInt32(firstPoint[0, 0]);

points[0].Y = Convert.ToInt32(firstPoint[0, 1]);

secondPoint = MatrixMultiplication.ScalerMultiplier(secondPoint, scalerMatrix);

points[1].X = Convert.ToInt32(secondPoint[0, 0]);

points[1].Y = Convert.ToInt32(secondPoint[0, 1]);

thirdPoint = MatrixMultiplication.ScalerMultiplier(thirdPoint, scalerMatrix);

points[2].X = Convert.ToInt32(thirdPoint[0, 0]);

points[2].Y = Convert.ToInt32(thirdPoint[0, 1]);

//we then move the object back to its original origin;

this.MoveToOriginalPlacement();

}

public override void Rotate(float angle) //Rotation method

{

//First we set the objects midpoint to the origin (0,0)

this.MoveToOrigin();

//Definition of the rotation matrix

float cosa = (float)Math.Cos(angle \* Math.PI / 180.0);

float sina = (float)Math.Sin(angle \* Math.PI / 180.0);

float negativeSina = sina \* -1;

float[,] rotationMatrix = { {cosa, sina},

{negativeSina, cosa } };

//Calculation of new points when multiplied by rotation matrix

float[,] firstPoint = { { points[0].X, points[0].Y } };

float[,] secondPoint = { { points[1].X, points[1].Y } };

float[,] thirdPoint = { { points[2].X, points[2].Y } };

firstPoint = MatrixMultiplication.RotationMultiplier(firstPoint, rotationMatrix);

points[0].X = Convert.ToInt32(firstPoint[0, 0]);

points[0].Y = Convert.ToInt32(firstPoint[0, 1]);

secondPoint = MatrixMultiplication.RotationMultiplier(secondPoint, rotationMatrix);

points[1].X = Convert.ToInt32(secondPoint[0, 0]);

points[1].Y = Convert.ToInt32(secondPoint[0, 1]);

thirdPoint = MatrixMultiplication.RotationMultiplier(thirdPoint, rotationMatrix);

points[2].X = Convert.ToInt32(thirdPoint[0, 0]);

points[2].Y = Convert.ToInt32(thirdPoint[0, 1]);

//we then move the object back to its original origin;

this.MoveToOriginalPlacement();

}

//Method to move object midpoint to 0,0 for transformations

public void MoveToOrigin()

{

for (int i = 0; i < 3; i++) //Loops through and moves all points to centre on midpoint (0,0)

{

points[i].X -= midPoint.X;

points[i].Y -= midPoint.Y;

}

}

//Method to move the object back to its original place

public void MoveToOriginalPlacement()

{

for (int i = 0; i < 3; i++)

{

points[i].X += midPoint.X;

points[i].Y += midPoint.Y;

}

}

public override void Move(float x, float y) //Method to move a shape to a new midpoint

{

int newX = Convert.ToInt32(x);

int newY = Convert.ToInt32(y);

for (int i = 0; i < 3; i++) //Loops through all points

{

points[i].X += newX;

points[i].Y += newY;

}

midPoint.X = (points[0].X + points[1].X + points[2].X) / 3; //Calculation of new midpoit

midPoint.Y = (points[0].Y + points[1].Y + points[2].Y) / 3;

}

}

}

## UserInput.cs

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Grafpack

{

public partial class UserInput : Form

{

private float inputValue, upperBound, lowerBound;

private string inputType;

public UserInput(string type, float lower, float upper)

{

InitializeComponent(); //Initialise the form

inputType = type; // setting of variables

upperBound = upper;

lowerBound = lower;

label1.Text = "";

label1.Text = Convert.ToString("Enter a " + inputType + lowerBound + " and " + upperBound); //Displays to user with customised text dependant on method

}

private void UserInput\_Load(object sender, EventArgs e)

{

}

private void Button2\_Click(object sender, EventArgs e) //Method if user clicks cancel

{

inputValue = 999999999; //Number used to identify if user clicks cancel

this.Close();

}

private void Label1\_Click(object sender, EventArgs e)

{

}

public float GetInput()

{

return inputValue;

}

private void Button1\_Click(object sender, EventArgs e)

{

inputValue = float.Parse(textBox1.Text); //set the textbox text based on tranform type

if (inputValue >= lowerBound && inputValue <= upperBound) //Check the input is within upper and lower bounds

{

this.Close();

}

else

{

MessageBox.Show("Please input a number between" + lowerBound + " and " + upperBound); //Displays if userinputs incorrectly

this.Close();

UserInput userInput = new UserInput(inputType, lowerBound,upperBound); //Creates a new form for new input

userInput.TopMost = true;

userInput.Show();

}

}

}

}