|  |  |
| --- | --- |
|  |  |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего профессионального образования  **«Московский технологический университет»**  **МИРЭА** | | |

Институт Информационных Технологий

Кафедра Корпоративных Информационных Систем

**ОТЧЕТ**

по Лабораторной Работе №1

на тему

«Перегрузка опреций»

по дисциплине

«ООП»

Выполнил студент группы ИСБО-11-16 Шайхуллин С.В.

Принял Cтарший преподаватель Мирзоян Д.И.

Выполнено «\_\_\_» \_\_\_\_\_\_\_\_\_2017 г.

Зачтено «\_\_\_» \_\_\_\_\_\_\_\_\_ 2017г.

Москва, 2017

**Теоретическая часть**

В C#, подобно любому языку программирования, имеется готовый набор лексем, используемых для выполнения базовых операций над встроенными типами. Например, известно, что операция + может применяться к двум целым, чтобы дать их сумму. По сути, функциональность операции + уникальным образом базируются на представленных типах данных (строках или целых в данном случае). Когда операция + применяется к числовым типам, мы получаем арифметическую сумму операндов. Однако когда та же операция применяется к строковым типам, получается конкатенация строк.

Язык C# предоставляет возможность строить специальные классы и структуры, которые также уникально реагируют на один и тот же набор базовых лексем (вроде операции +). Имейте в виду, что абсолютно каждую встроенную операцию C# перегружать нельзя. Перегрузка операторов тесно связана с перегрузкой методов. Для перегрузки оператора служит ключевое слово operator, определяющее операторный метод, который, в свою очередь, определяет действие оператора относительно своего класса. Существуют две формы операторных методов (operator): одна - для унарных операторов, другая - для бинарных. Унарные операторы перегружаются таким же образом, как и бинарные. Главное отличие заключается, конечно, в том, что у них имеется лишь один операнд. Для любого заданного класса и оператора имеется также возможность перегрузить сам операторный метод. Это, в частности, требуется для того, чтобы разрешить операции с типом класса и другими типами данных, в том числе и встроенными.

**Задание**

Составить приложение, использующее классы/структуры с перегруженными стандартными операциями.

Вариант 1:

1. Шестиугольник
2. Двадцатиконечная звезда
3. Греческий крест

**Алгоритм решения задачи**

1. Определить типы вектора и матрицы, операции с ними
2. Создать форму-представление с необходимыми на ней компонентами
3. Описать бизнес-логику приложения
4. Протестировать результат

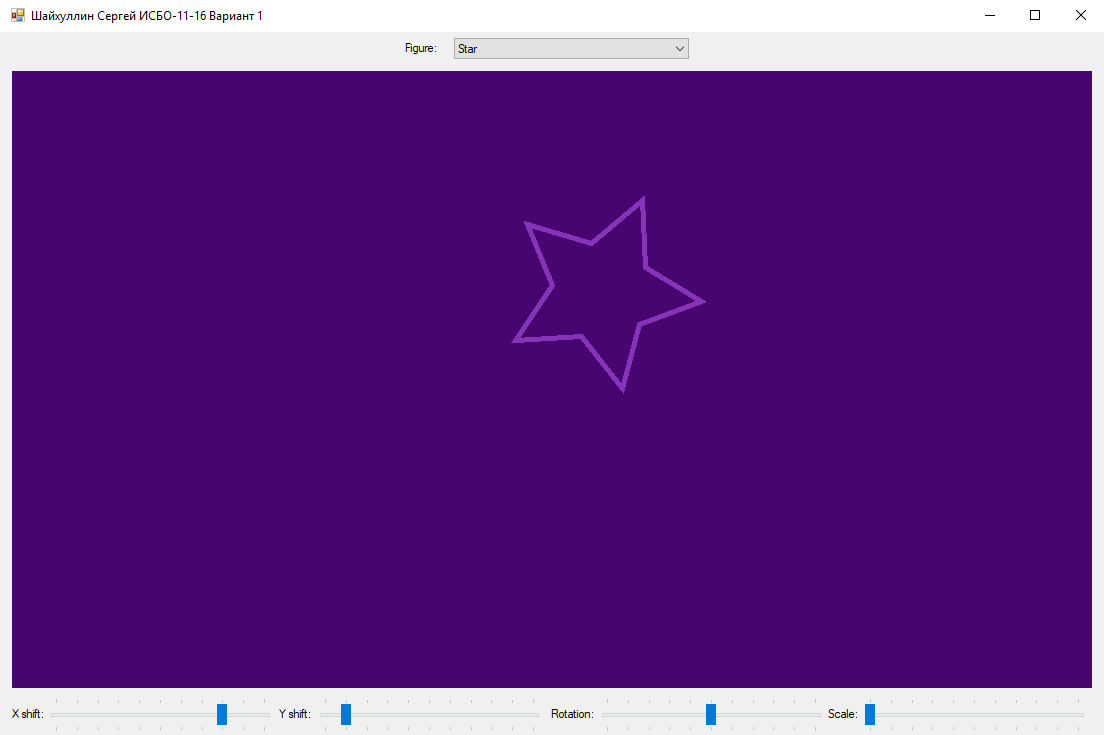
**Тестирование**

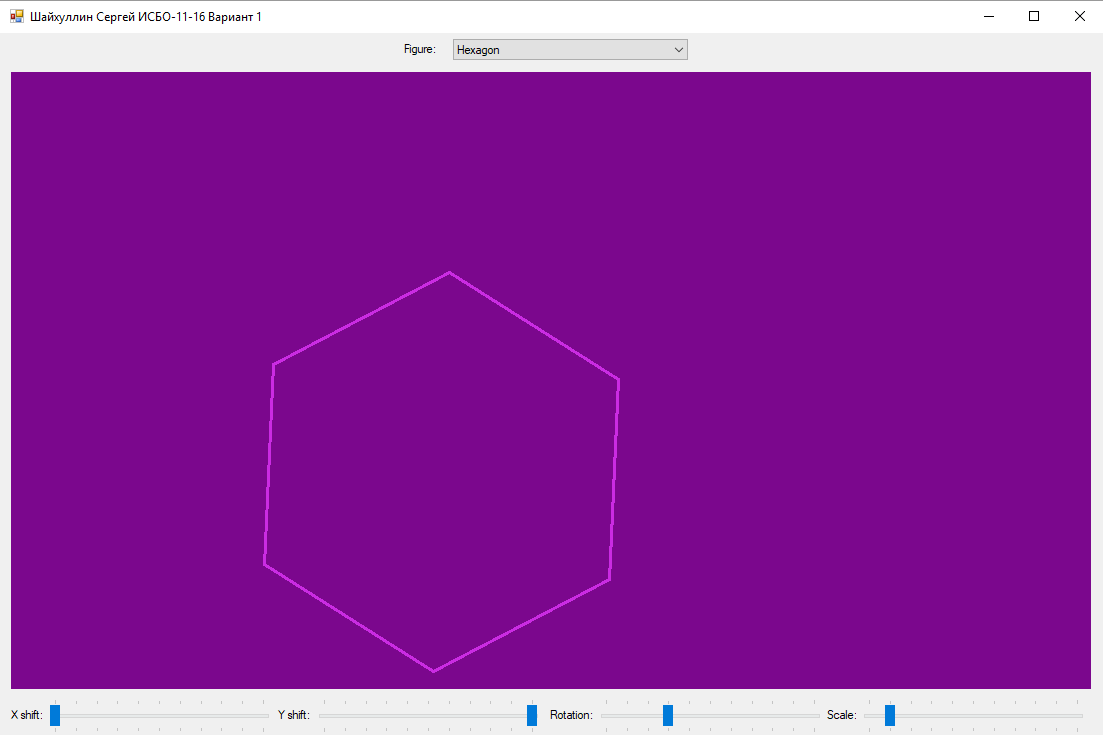
1. Протестировать изменение фигур
2. Протестировать смещение фигур по оси Х
3. Протестировать смещение фигур по оси Y
4. Протестировать поворот фигур
5. Протестировать увеличение фигур
6. Протестировать всевозможные вариации из пунктов выше

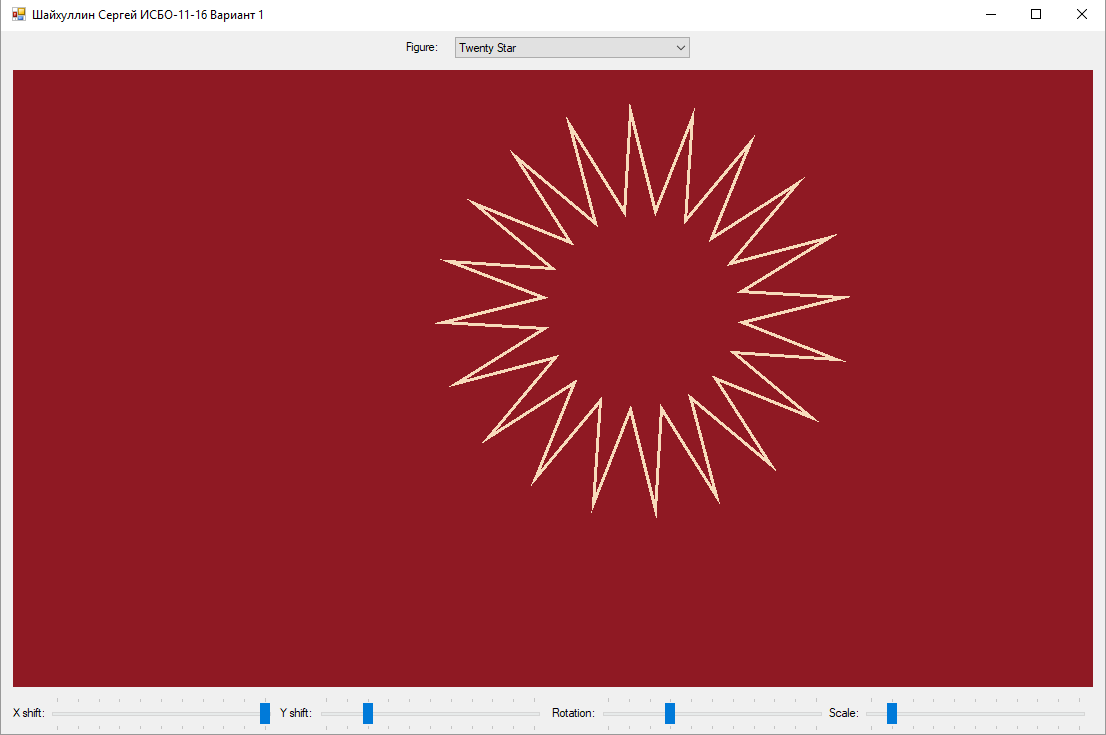
**Заключение**

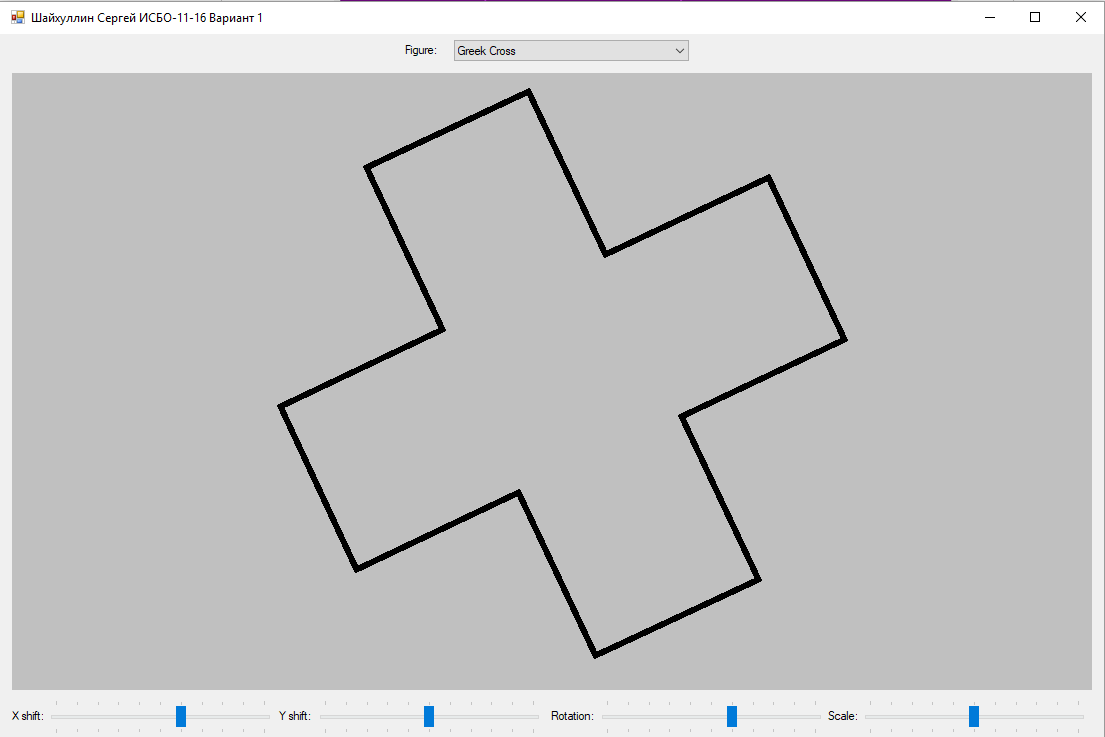
В данной лабораторной работе мы научились переопределять операторы, описывать взаимодействия между сущностями, создавать удобные для пользования интерфейсы программы, а также познакомились с некоторыми аффинными преобразованиями и их реализацией на языке C#.

**Пример работы**









**Исходный код**

using System;

using System.Windows.Forms;

using Shaykhullin.Lab1.Views;

using Shaykhullin.DependencyInjection.App;

using System.Collections.Generic;

using Shaykhullin.Shared.Lab1.Drawables;

using Shaykhullin.Shared.Lab1.ProcessingStrategies;

using System.Drawing;

using Shaykhullin.Shared.Lab1;

namespace Shaykhullin.Lab1

{

static class Program

{

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

var service = new AppServiceBuilder()

.Register<MainView>()

.AsSingleton()

.Register<Dictionary<string, Type>>()

.Returns(new Dictionary<string, Type>

{

["Star"] = typeof(StarDrawable),

["Hexagon"] = typeof(HexagonDrawable),

["Twenty Star"] = typeof(TwentyStarDrawable),

["Greek Cross"] = typeof(GreekCrossDrawable)

})

.As<IReadOnlyDictionary<string, Type>>()

.AsSingleton()

.Register<IStateProcessingStrategy<Matrix3x3, int>[]>()

.Returns(new IStateProcessingStrategy<Matrix3x3, int>[]

{

new RotationProcessingStrategy(),

new ScaleProcessingStrategy(),

new XShiftProcessingStrategy(),

new YShiftProcessingStrategy()

})

.AsSingleton()

.Register<LoopProcessingStrategy>()

.As<IStateValueProcessingStrategy<Bitmap, DrawableBase, (int, int)>>()

.AsSingleton()

.Register<Bitmap>()

.Returns(new Bitmap(1080, 617))

.AsSingleton()

.Service;

Application.Run(service.Resolve<MainView>());

}

}

}

using System;

using System.Drawing;

using System.Windows.Forms;

using System.Collections.Generic;

using Shaykhullin.Extensions;

using Shaykhullin.Shared.Lab1.Drawables;

using Shaykhullin.Shared.Lab1.ProcessingStrategies;

using Shaykhullin.DependencyInjection;

using Shaykhullin.DependencyInjection.Abstraction;

namespace Shaykhullin.Lab1.Views

{

public partial class MainView : Form

{

[Inject]

private IService service;

[Inject]

private IReadOnlyDictionary<string, Type> drawables;

[Inject]

private IStateValueProcessingStrategy<Bitmap, DrawableBase, (int, int)> strategy;

[Inject]

private Bitmap bitmap;

public MainView()

{

InitializeComponent();

}

private void OnLoad(object sender, EventArgs e)

{

image.Image = bitmap;

}

private void OnSelected(object sender, EventArgs e)

{

strategy.State = service.Create<DrawableBase>(drawables[(sender as ComboBox).Text]);

image.Image = strategy.Process();

}

private void OnValueChanged(object sender, EventArgs e)

{

if (strategy.State == null)

{

MessageBox.Show("Select figure!", "Figure not selected", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);

return;

}

strategy.Value = (sender.ParseTag(), (sender as TrackBar).Value);

image.Image = strategy.Process();

}

}

}

using System;

using System.Collections;

using System.Collections.Generic;

namespace Shaykhullin.Shared.Lab1

{

public struct Quaternion

{

public static QuaternionFactory Factory { get; } = new QuaternionFactory();

public double X { get; }

public double Y { get; }

public double Degree { get; }

private Quaternion(double x, double y) : this(x, y, 1.0)

{

}

private Quaternion(double x, double y, double degree)

{

X = x;

Y = y;

Degree = degree;

}

public static Quaternion operator +(Quaternion a, Quaternion b) =>

a.Degree == b.Degree

? Factory.WithDegree(a.X + b.X, a.Y + b.Y, a.Degree)

: Factory.With(a.X / a.Degree + b.X / a.Degree, a.Y / b.Degree + b.Y / b.Degree);

public static Quaternion operator -(Quaternion a, Quaternion b) =>

a.Degree == b.Degree

? Factory.WithDegree(a.X - b.X, a.Y - b.Y, a.Degree)

: Factory.With(a.X / a.Degree - b.X / a.Degree, a.Y / b.Degree - b.Y / b.Degree);

public static Quaternion operator \*(Quaternion a, Quaternion b) =>

a.Degree == b.Degree

? Factory.WithDegree(a.X \* b.X, a.Y \* b.Y, a.Degree)

: Factory.With(a.X / a.Degree \* b.X / a.Degree, a.Y / b.Degree \* b.Y / b.Degree);

public class QuaternionFactory

{

public QuaternionFlientFactory AsFluent(Func<double, double> selector = null) =>

new QuaternionFlientFactory(selector ?? (p => p));

public Quaternion With(double x, double y) => new Quaternion(x, y);

public Quaternion WithDegree(double x, double y, double degree) => new Quaternion(x, y, degree);

public class QuaternionFlientFactory : IEnumerable<Quaternion>

{

private Func<double, double> selector;

private List<Quaternion> quaternionList = new List<Quaternion>();

public QuaternionFlientFactory(Func<double, double> selector)

{

this.selector = selector;

}

public QuaternionFlientFactory With(double x, double y)

{

quaternionList.Add(new Quaternion(selector(x), selector(y)));

return this;

}

public QuaternionFlientFactory WithDegree(double x, double y, double degree)

{

quaternionList.Add(new Quaternion(selector(x), selector(y), degree));

return this;

}

public IEnumerator<Quaternion> GetEnumerator()

{

foreach (var quaternion in quaternionList)

{

yield return quaternion;

}

}

IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using Shaykhullin.Extensions;

namespace Shaykhullin.Shared.Lab1

{

public class Matrix3x3

{

public const int Size = 3;

public static Matrix3x3Factory Factory { get; } = new Matrix3x3Factory();

private double[,] matrix = new double[Size, Size];

public double this[int x, int y]

{

get

{

if (x.IsInRange(0, 2).And(y, 0, 2))

{

return matrix[x, y];

}

throw new InvalidOperationException($"X and Y must be in range 0-{Size}");

}

set

{

if (x.IsInRange(0, 2).And(y, 0, 2))

{

matrix[x, y] = value;

}

else

{

throw new InvalidOperationException($"X and Y must be in range 0-{Size}");

}

}

}

public static Matrix3x3 operator +(Matrix3x3 a, Matrix3x3 b) =>

Factory.Zero

.Loop((matrix, x, y) => matrix[x, y] = a[x, y] + b[x, y]);

public static Matrix3x3 operator -(Matrix3x3 a, Matrix3x3 b) =>

Factory.Zero

.Loop((matrix, x, y) => matrix[x, y] = a[x, y] - b[x, y]);

public static Matrix3x3 operator \*(Matrix3x3 a, Matrix3x3 b) =>

Factory.Zero.AsFluent()

.In(0, 0).Has(a[0, 0] \* b[0, 0] + a[0, 1] \* b[1, 0] + a[0, 2] \* b[2, 0])

.In(0, 1).Has(a[0, 0] \* b[0, 1] + a[0, 1] \* b[1, 1] + a[0, 2] \* b[2, 1])

.In(0, 2).Has(a[0, 0] \* b[0, 2] + a[0, 1] \* b[1, 2] + a[0, 2] \* b[2, 2])

.In(1, 0).Has(a[1, 0] \* b[0, 0] + a[1, 1] \* b[1, 0] + a[1, 2] \* b[2, 0])

.In(1, 1).Has(a[1, 0] \* b[0, 1] + a[1, 1] \* b[1, 1] + a[1, 2] \* b[2, 1])

.In(1, 2).Has(a[1, 0] \* b[0, 2] + a[1, 1] \* b[1, 2] + a[1, 2] \* b[2, 2])

.In(2, 0).Has(a[2, 0] \* b[0, 0] + a[2, 1] \* b[1, 0] + a[2, 2] \* b[2, 0])

.In(2, 1).Has(a[2, 0] \* b[0, 1] + a[2, 1] \* b[1, 1] + a[2, 2] \* b[2, 1])

.In(2, 2).Has(a[2, 0] \* b[0, 2] + a[2, 1] \* b[1, 2] + a[2, 2] \* b[2, 2])

.Matrix;

public static Quaternion operator \*(Matrix3x3 matrix, Quaternion quaternion) =>

Quaternion.Factory.AsSeparate()

.WithX(matrix[0, 0] \* quaternion.X + matrix[0, 1] \* quaternion.Y + matrix[0, 2] \* quaternion.Degree)

.WithY(matrix[1, 0] \* quaternion.X + matrix[1, 1] \* quaternion.Y + matrix[1, 2] \* quaternion.Degree)

.WithDegree(matrix[2, 0] \* quaternion.X + matrix[2, 1] \* quaternion.Y + matrix[2, 2] \* quaternion.Degree)

.Quaternion;

public static Quaternion operator \*(Quaternion quaternion, Matrix3x3 matrix) =>

Quaternion.Factory.AsSeparate()

.WithX(quaternion.X \* matrix[0, 0] + quaternion.Y \* matrix[1, 0] + quaternion.Degree \* matrix[2, 0])

.WithY(quaternion.X \* matrix[0, 1] + quaternion.Y \* matrix[1, 1] + quaternion.Degree \* matrix[2, 1])

.WithDegree(quaternion.X \* matrix[0, 2] + quaternion.Y \* matrix[1, 2] + quaternion.Degree \* matrix[2, 2])

.Quaternion;

public static IEnumerable<Quaternion> operator \*(Matrix3x3 matrix, IEnumerable<Quaternion> quaternions) =>

quaternions.Select(quaternion => matrix \* quaternion);

public static IEnumerable<Quaternion> operator \*(IEnumerable<Quaternion> quaternions, Matrix3x3 matrix) =>

quaternions.Select(quaternion => quaternion \* matrix);

public static Matrix3x3 operator !(Matrix3x3 other) =>

Factory.Zero.Loop((matrix, x, y) => matrix[x, y] = other[y, x]);

public class Matrix3x3Factory

{

public Matrix3x3 Zero => new Matrix3x3();

public Matrix3x3 Identity =>

Factory.Zero.AsFluent()

.In(0, 0).Has(1)

.In(1, 1).Has(1)

.In(2, 2).Has(1)

.Matrix;

public Matrix3x3 ToTranslational(double? shiftX = null, double? shiftY = null) =>

Factory.Identity.AsFluent()

.In(2, 0).Has(shiftX) // 2 0

.In(2, 1).Has(shiftY) // 2 1

.Matrix;

public Matrix3x3 ToRotational(double angle) =>

Factory.Identity.AsFluent()

.In(0, 0).Has(Math.Cos(angle))

.In(1, 1).Has(Math.Cos(angle))

.In(0, 1).Has(-Math.Sin(angle))

.In(1, 0).Has(Math.Sin(angle))

.Matrix;

public Matrix3x3 ToScalable(double scaleX, double scaleY) =>

Factory.Identity.AsFluent()

.In(0, 0).Has(scaleX)

.In(1, 1).Has(scaleY)

.Matrix;

}

}

}

namespace Shaykhullin.Shared.Lab1.ProcessingStrategies

{

public class RotationProcessingStrategy : IStateProcessingStrategy<Matrix3x3, int>

{

public int State { get; set; }

public Matrix3x3 Process()

{

return Matrix3x3.Factory.ToRotational(State \* 4);

}

}

}

namespace Shaykhullin.Shared.Lab1.ProcessingStrategies

{

public class ScaleProcessingStrategy : IStateProcessingStrategy<Matrix3x3, int>

{

public int State { get; set; }

public Matrix3x3 Process()

{

return Matrix3x3.Factory.ToScalable(State + 1, State + 1);

}

}

}

namespace Shaykhullin.Shared.Lab1.ProcessingStrategies

{

public class XShiftProcessingStrategy : IStateProcessingStrategy<Matrix3x3, int>

{

public int State { get; set; } = 5;

public Matrix3x3 Process()

{

return Matrix3x3.Factory.ToTranslational(shiftX: State \* 20);

}

}

}

namespace Shaykhullin.Shared.Lab1.ProcessingStrategies

{

public class YShiftProcessingStrategy : IStateProcessingStrategy<Matrix3x3, int>

{

public int State { get; set; } = 5;

public Matrix3x3 Process()

{

return Matrix3x3.Factory.ToTranslational(shiftY: State \* 20);

}

}

}

using System.Linq;

using System.Drawing;

using Shaykhullin.DependencyInjection;

namespace Shaykhullin.Shared.Lab1.Drawables

{

public abstract class DrawableBase : IDrawable<Bitmap, Matrix3x3>

{

[Inject]

public Bitmap Bitmap { get; set; }

protected Quaternion[] Points { get; set; }

protected abstract Pen Pen { get; }

protected abstract Brush Brush { get; }

public DrawableBase()

{

}

public DrawableBase(int pointsCount)

{

this.Points = new Quaternion[pointsCount];

}

public Bitmap Draw(Matrix3x3 matrix)

{

var graphics = Graphics.FromImage(Bitmap);

graphics.FillRectangle(Brush, 0, 0, Bitmap.Width, Bitmap.Height);

graphics.DrawPolygon(Pen, (Points \* matrix)

.Select(point => new Point((int)point.X + 430, (int)point.Y + 200))

.ToArray());

return Bitmap;

}

}

}

using System.Drawing;

using System.Linq;

namespace Shaykhullin.Shared.Lab1.Drawables

{

public class GreekCrossDrawable : DrawableBase

{

protected override Pen Pen { get; } = new Pen(Color.Black, 6.0f);

protected override Brush Brush { get; } = new SolidBrush(Color.Silver);

private int size = 30;

public GreekCrossDrawable() =>

Points = Quaternion.Factory.AsFluent(prop => prop - size \* 1.5)

.With(size, 0)

.With(size, size)

.With(0, size)

.With(0, 2 \* size)

.With(size, 2 \* size)

.With(size, 3 \* size)

.With(2 \* size, 3 \* size)

.With(2 \* size, 2 \* size)

.With(3 \* size, 2 \* size)

.With(3 \* size, 1 \* size)

.With(2 \* size, 1 \* size)

.With(2 \* size, 0)

.ToArray();

}

}

using System;

using System.Drawing;

namespace Shaykhullin.Shared.Lab1.Drawables

{

public class HexagonDrawable : DrawableBase

{

protected override Pen Pen { get; } = new Pen(Color.FromArgb(255, 202, 44, 226), 3.0f);

protected override Brush Brush { get; } = new SolidBrush(Color.FromArgb(255, 123, 7, 141));

public HexagonDrawable() : base(6)

{

for (int i = 0; i < Points.Length; i++)

{

Points[i] = Quaternion.Factory.With(

Math.Cos(i \* 60 \* Math.PI / 180) \* 100,

Math.Sin(i \* 60 \* Math.PI / 180) \* 100);

}

}

}

}

namespace Shaykhullin.Shared.Lab1.Drawables

{

public interface IDrawable<TInput, TResult>

{

TInput Draw(TResult draw);

}

}

using System;

using System.Drawing;

namespace Shaykhullin.Shared.Lab1.Drawables

{

public class StarDrawable : DrawableBase

{

protected override Pen Pen { get; } = new Pen(Color.FromArgb(255, 133, 52, 183), 5.0f);

protected override Brush Brush { get; } = new SolidBrush(Color.FromArgb(255, 71, 5, 112));

public StarDrawable() : base(pointsCount: 10)

{

for (int i = 0; i < Points.Length / 2; i++)

{

Points[i \* 2] = Quaternion.Factory

.With(Math.Cos(i \* 72 \* Math.PI / 180) \* 100,

Math.Sin(i \* 72 \* Math.PI / 180) \* 100);

Points[i \* 2 + 1] = Quaternion.Factory

.With(Math.Cos((i \* 72 + 36) \* Math.PI / 180) \* 50,

Math.Sin((i \* 72 + 36) \* Math.PI / 180) \* 50);

}

}

}

}

using System;

using System.Drawing;

namespace Shaykhullin.Shared.Lab1.Drawables

{

public class TwentyStarDrawable : DrawableBase

{

protected override Pen Pen { get; } = new Pen(Color.FromArgb(255, 247, 219, 187), 3.0f);

protected override Brush Brush { get; } = new SolidBrush(Color.FromArgb(255, 143, 25, 35));

public TwentyStarDrawable() : base(40)

{

for (int i = 0; i < Points.Length / 2; i++)

{

Points[i \* 2] = Quaternion.Factory

.With(Math.Cos(i \* 18 \* Math.PI / 180) \* 100,

Math.Sin(i \* 18 \* Math.PI / 180) \* 100);

Points[i \* 2 + 1] = Quaternion.Factory

.With(Math.Cos((i \* 18 + 11) \* Math.PI / 180) \* 50,

Math.Sin((i \* 18 + 11) \* Math.PI / 180) \* 50);

}

}

}

}