using System.ComponentModel;

using System.Linq;

namespace SmearsMaker.Common.BaseTypes

{

public abstract class BaseShape

{

public PointCollection Points { get; }

private System.Windows.Point? \_centerPoint;

private Point \_center;

protected BaseShape()

{

Points = new PointCollection();

Points.PropertyChanged += PropertyChanged;

}

public System.Windows.Point GetCenter()

{

if (\_centerPoint == null)

{

\_centerPoint = ComputeCenterWinPoint();

}

return \_centerPoint.Value;

}

public Point GetCenterPoint()

{

return \_center ?? (\_center = ComputeCenterPoint());

}

public Pixel GetCenter(string layer)

{

return GetCenterPoint().Pixels[layer];

}

private System.Windows.Point ComputeCenterWinPoint()

{

var x = 0d;

var y = 0d;

foreach (var point in Points)

{

x += point.Position.X;

y += point.Position.Y;

}

x /= Points.Count;

y /= Points.Count;

return new System.Windows.Point(x, y);

}

private Pixel ComputeCenter(string layer)

{

var dataArr = new float[Points.First().Pixels[layer].Length];

foreach (var point in Points)

{

var currData = point.Pixels[layer].Data;

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

{

dataArr[i] += currData[i];

}

}

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

{

dataArr[i] /= Points.Count;

}

return new Pixel(dataArr);

}

private Point ComputeCenterPoint()

{

var point = new Point(GetCenter());

foreach (var layer in Points.Layers)

{

var pixel = ComputeCenter(layer);

point.Pixels.AddPixel(layer, pixel);

}

return point;

}

private void PropertyChanged(object obj, PropertyChangedEventArgs args)

{

ClearCache();

}

private void ClearCache()

{

\_center = null;

\_centerPoint = null;

}

}

}

using System;

using System.Linq;

namespace SmearsMaker.Common.BaseTypes

{

public class Pixel

{

public static Pixel CreateInstance(float[] data)

{

if (data == null)

{

throw new ArgumentNullException(nameof(data));

}

return new Pixel(data);

}

public float Sum => Data.Sum();

public float Average => Data.Average();

public float[] Data { get; set; }

public int Length => Data.Length;

public float GrayScale

{

get

{

{

//var grayscale = (byte)(0.2126 \* r + 0.7152 \* g + 0.0722 \* b);

var grayscale = (float)(0.299 \* Data[0] + 0.587 \* Data[1] + 0.114 \* Data[2]);

return grayscale;

};

}

}

internal Pixel()

{

Data = new float[3];

}

internal Pixel(float[] data)

{

Data = (float[])data?.Clone() ?? throw new ArgumentNullException(nameof(data));

}

internal Pixel(Pixel pixel)

{

if (pixel == null)

{

throw new ArgumentNullException(nameof(pixel));

}

var data = pixel.Data;

Data = (float[])data?.Clone();

}

public double Distance(float[] data)

{

var dist = 0d;

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

{

dist += Distance(Data[i], data[i]);

}

return dist;

}

private static float Distance(float first, float second)

{

var distance = first - second;

if (distance < 0)

{

return -distance;

}

return distance;

}

}

}

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace SmearsMaker.Common.BaseTypes

{

public class PixelCollection : IEnumerable<Pixel>

{

private readonly Dictionary<string, Pixel> \_pixels;

internal List<string> Layers => \_pixels.Keys.ToList();

internal PixelCollection()

{

\_pixels = new Dictionary<string, Pixel>();

}

internal PixelCollection(PixelCollection collection)

{

\_pixels = new Dictionary<string, Pixel>();

foreach (var p in collection.\_pixels)

{

var newPixel = new Pixel(p.Value.Data);

\_pixels.Add(p.Key, newPixel);

}

}

internal void Addlayer(string layer)

{

\_pixels.Add(layer, new Pixel());

}

internal void AddPixel(string layer, Pixel value)

{

\_pixels.Add(layer, value);

}

public Pixel this[string index]

{

get => \_pixels[index];

set => \_pixels[index] = new Pixel(value);

}

public IEnumerator<Pixel> GetEnumerator() => \_pixels.Values.GetEnumerator();

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

}

}

namespace SmearsMaker.Common.BaseTypes

{

public class Point

{

public System.Windows.Point Position { get; }

public PixelCollection Pixels { get; }

internal Point(System.Windows.Point point)

{

Pixels = new PixelCollection();

Position = point;

}

internal Point(double posX, double posY) : this(new System.Windows.Point(posX, posY))

{

}

internal Point(Point point)

{

Pixels = new PixelCollection(point.Pixels);

Position = point.Position;

}

internal void AddLayer(string layer)

{

if (!Pixels.Layers.Contains(layer))

{

Pixels.Addlayer(layer);

}

}

public Point Clone()

{

return new Point(this);

}

}

}

using System;

using System.Collections;

using System.Collections.Generic;

using System.Collections.ObjectModel;

using System.Collections.Specialized;

using System.ComponentModel;

using System.Linq;

using SmearsMaker.Common.Helpers;

namespace SmearsMaker.Common.BaseTypes

{

public class PointCollection : IEnumerable<Point>

{

internal event PropertyChangedEventHandler PropertyChanged;

private readonly ObservableCollection<Point> \_points;

internal readonly List<string> Layers;

public PointCollection()

{

\_points = new ObservableCollection<Point>();

\_points.CollectionChanged += CollectionChanged;

Layers = new List<string>();

}

internal PointCollection(PointCollection collection) : this()

{

Layers.AddRange(collection.Layers);

foreach (var p in collection.\_points)

{

\_points.Add(new Point(p));

}

}

public PointCollection Clone()

{

return new PointCollection(this);

}

public void AddLayers(IEnumerable<string> layers)

{

if (layers == null)

{

throw new NullReferenceException(nameof(layers));

}

foreach (var layer in layers)

{

Addlayer(layer);

}

}

public void Addlayer(string layer)

{

if (Layers.Contains(layer))

throw new ArgumentException(nameof(layer));

Layers.Add(layer);

foreach (var point in \_points)

{

point.AddLayer(layer);

}

}

public void AddRange(PointCollection points)

{

AddRange(points.ToList());

}

public void AddRange(List<Point> points)

{

foreach (var point in points)

{

Add(point);

}

}

public void Add(Point point)

{

\_points.Add(point);

}

public int Count => \_points.Count;

public void Clear()

{

\_points.Clear();

Layers.Clear();

}

public Point this[int index]

{

get => \_points[index];

set => \_points[index] = value;

}

public void ForEach(Action<Point> action)

{

\_points.ForEach(action);

}

public IEnumerator<Point> GetEnumerator() => \_points.GetEnumerator();

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

private void CollectionChanged(object obj, NotifyCollectionChangedEventArgs args)

{

PropertyChanged?.Invoke(obj, new PropertyChangedEventArgs(nameof(\_points)));

}

}

}

using System.Windows.Media.Imaging;

using SmearsMaker.Common.BaseTypes;

namespace SmearsMaker.Common.Helpers

{

internal class ImageHelper

{

private const int DataFormatSize = 4;

internal static PointCollection ConvertToPixels(BitmapSource source)

{

var inputData = new PointCollection();

var stride = source.PixelWidth \* DataFormatSize;

var size = source.PixelHeight \* stride;

var data = new byte[size];

source.CopyPixels(data, stride, 0);

for (int x = 0; x < source.PixelWidth; x++)

{

for (int y = 0; y < source.PixelHeight; y++)

{

var idx = y \* stride + DataFormatSize \* x;

var rgbArray = new float[] { data[idx], data[idx + 1], data[idx + 2], data[idx + 3] };

var point = new Point(x, y);

point.Pixels.AddPixel(Layers.Original, new Pixel(rgbArray));

inputData.Add(point);

}

}

return inputData;

}

internal static BitmapSource ConvertRgbToBitmap(BitmapSource source, PointCollection points, string layer)

{

var stride = source.PixelWidth \* DataFormatSize;

var size = source.PixelHeight \* stride;

var data = new byte[size];

foreach (var pixel in points)

{

var indexPixel = (int)(pixel.Position.Y \* stride + DataFormatSize \* pixel.Position.X);

var dataArray = pixel.Pixels[layer].Data;

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

{

data[indexPixel + i] = (byte)dataArray[i];

}

}

var image = BitmapSource.Create(source.PixelWidth, source.PixelHeight, source.DpiX, source.DpiY, source.Format, source.Palette, data, stride);

return image;

}

}

}

using System;

using System.Collections.Generic;

namespace SmearsMaker.Common.Helpers

{

public static class ObservableCollectionExtention

{

public static void ForEach<T>(this IEnumerable<T> enumerable, Action<T> action)

{

foreach (var cur in enumerable)

{

action(cur);

}

}

}

}

using System;

using System.Windows.Media.Imaging;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.Common.Helpers;

namespace SmearsMaker.Common.Image

{

public class ImageModel : IDisposable

{

public int Width => Image.PixelWidth;

public int Height => Image.PixelHeight;

public PointCollection Points => \_points ?? (\_points = ImageHelper.ConvertToPixels(Image));

public BitmapSource Image { get; }

private PointCollection \_points;

public ImageModel(BitmapSource image)

{

Image = image ?? throw new NullReferenceException("image");

}

public BitmapSource GetDifference(BitmapSource image, string layer)

{

if (image == null)

{

throw new NullReferenceException(nameof(image));

}

var diffCollection = new PointCollection();

var secondModel = new ImageModel(image);

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

{

var firstArray = Points[i].Pixels[layer].Sum;

var secondArray = secondModel.Points[i].Pixels[Layers.Original].Sum;

var diffArray = new float[Points[i].Pixels[layer].Length];

for (int j = 0; j < diffArray.Length; j++)

{

diffArray[j] = 255 - firstArray - secondArray;

}

var pixel = new Pixel(diffArray);

var diffPoint = new Point(Points[i].Position);

diffPoint.Pixels.AddPixel(Layers.Original, pixel);

diffCollection.Add(diffPoint);

}

return ImageHelper.ConvertRgbToBitmap(Image, diffCollection, Layers.Original);

}

public BitmapSource GetDifference(string firstLayer, string secondLayer)

{

if (!Points.Layers.Contains(firstLayer))

{

throw new ArgumentOutOfRangeException(nameof(firstLayer));

}

if (!Points.Layers.Contains(secondLayer))

{

throw new ArgumentOutOfRangeException(nameof(secondLayer));

}

var diffCollection = new PointCollection();

diffCollection.Addlayer(firstLayer);

foreach (var point in Points)

{

var firstArray = point.Pixels[firstLayer].Data;

var secondArray = point.Pixels[secondLayer].Data;

var diffArray = new float[firstArray.Length];

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

{

diffArray[i] = 255 - firstArray[i] - secondArray[i];

}

var pixel = new Pixel(diffArray);

var diffPoint = new Point(point.Position);

diffPoint.Pixels.AddPixel(firstLayer, pixel);

diffCollection.Add(diffPoint);

}

return ImageHelper.ConvertRgbToBitmap(Image, diffCollection, firstLayer);

}

public BitmapSource ConvertToBitmapSource(PointCollection points, string layer)

{

return ImageHelper.ConvertRgbToBitmap(Image, points, layer);

}

public void Dispose()

{

\_points?.Clear();

}

}

}

namespace SmearsMaker.Common.Image

{

public class ImageSetting

{

public string Name { get; set; }

public double Value { get; set; }

}

}

using System.Windows.Media.Imaging;

namespace SmearsMaker.Common.Image

{

public class ImageView

{

public BitmapSource Source;

public string Name;

public ImageView(BitmapSource source, string name)

{

Source = source;

Name = name;

}

}

}

using System;

namespace SmearsMaker.Common

{

public interface IProgress

{

event EventHandler<ProgressBarEventArgs> UpdateProgress;

void NewProgress(string msg, int minimum, int maximum);

void NewProgress(string msg);

void Update(int step);

}

public class ProgressBarEventArgs : EventArgs

{

public string Msg { get; }

public int Percentage { get; }

public ProgressBarEventArgs(string msg)

{

Msg = msg;

}

public ProgressBarEventArgs(string msg, int percentage)

{

Msg = msg;

Percentage = percentage;

}

}

}

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using SmearsMaker.Common.Image;

using SmearsMaker.HPGL;

namespace SmearsMaker.Common

{

public interface ITracer : IPltCreator, IDisposable

{

IProgress Progress { get; }

List<ImageSetting> Settings { get;}

List<ImageView> Views { get; }

Task Execute();

}

}

namespace SmearsMaker.Common

{

public struct Layers

{

public static string Original = "Original";

public static string Filtered = "Filtered";

public static string SuperPixels = "SuperPixels";

public static string Gradient = "Gradient";

public static string Curves = "Curves";

public static string BrushStrokes = "BrushStrokes";

}

}

using System;

namespace SmearsMaker.Common

{

internal class ProgressImpl : IProgress

{

public event EventHandler<ProgressBarEventArgs> UpdateProgress;

private string \_message;

private int \_position;

private int \_minimum;

private int \_maximum;

private int CurrPercent => \_minimum - \_maximum != 0 ? (\_minimum + \_position) \* 100 / (\_maximum - \_minimum) : 0;

public void NewProgress(string msg, int minimum, int maximum)

{

\_message = msg;

\_minimum = minimum;

\_maximum = maximum;

\_position = 0;

UpdateProgress?.Invoke(this, new ProgressBarEventArgs(msg, CurrPercent));

}

public void NewProgress(string msg)

{

\_message = msg;

UpdateProgress?.Invoke(this, new ProgressBarEventArgs(msg, CurrPercent));

}

public void Update(int step)

{

\_position += step;

UpdateProgress?.Invoke(this, new ProgressBarEventArgs(\_message, CurrPercent));

}

}

}

using System.Collections.Generic;

using System.Threading.Tasks;

using System.Windows.Media.Imaging;

using NLog;

using SmearsMaker.Common.Image;

namespace SmearsMaker.Common

{

public abstract class TracerBase : ITracer

{

public IProgress Progress { get; }

public abstract List<ImageSetting> Settings { get; }

public abstract List<ImageView> Views { get; }

protected ImageModel Model { get; }

protected static readonly ILogger Log = LogManager.GetCurrentClassLogger();

protected TracerBase(BitmapSource image)

{

Model = new ImageModel(image);

Progress = new ProgressImpl();

}

public abstract Task Execute();

public abstract string CreatePlt();

public virtual void Dispose()

{

Model?.Dispose();

}

}

}

namespace SmearsMaker.HPGL

{

public interface IPltCreator

{

string CreatePlt();

}

}

using System.Windows.Media.Imaging;

namespace SmearsMaker.HPGL

{

public interface IPltReader

{

BitmapSource ReadPlt(string path);

}

}

using System;

using System.Collections.Generic;

using System.Windows;

using System.Windows.Media;

namespace SmearsMaker.HPGL

{

internal class Painter

{

private const int Radius = 1;

private SolidColorBrush \_brush;

private Pen \_circlePen;

private Pen \_linePen;

private readonly DrawingGroup \_geometries;

public DrawingImage Image => new DrawingImage(\_geometries);

internal Painter()

{

\_geometries = new DrawingGroup();

}

internal void SetPens(byte[] data, int width)

{

\_brush = new SolidColorBrush(Color.FromRgb(data[0], data[1], data[2]));

\_circlePen = new Pen(\_brush, width - 1);

\_linePen = new Pen(\_brush, width);

}

internal void AddPoints(List<Point> points)

{

if (points == null)

throw new ArgumentNullException(nameof(points));

if (points.Count <= 1)

return;

for (int i = 0; i < points.Count - 1; i++)

{

var line = new LineGeometry(points[i], points[i + 1]);

var head = new EllipseGeometry(points[i], Radius, Radius);

var tail = new EllipseGeometry(points[i + 1], Radius, Radius);

\_geometries.Children.Add(new GeometryDrawing(\_brush, \_circlePen, head));

\_geometries.Children.Add(new GeometryDrawing(\_brush, \_circlePen, tail));

\_geometries.Children.Add(new GeometryDrawing(\_brush, \_linePen, line));

}

}

}

}

using System.Windows;

using System.Windows.Media;

using System.Windows.Media.Imaging;

namespace SmearsMaker.HPGL

{

internal static class PltHelper

{

internal static BitmapSource ToBitmapSource(DrawingImage source, int heigth, int width)

{

var drawingVisual = new DrawingVisual();

var drawingContext = drawingVisual.RenderOpen();

var bmp = new RenderTargetBitmap(width, heigth, 96, 96, PixelFormats.Pbgra32);

drawingContext.DrawImage(source, new Rect(new Point(0, 0), new Size(width, heigth)));

drawingContext.Close();

bmp.Render(drawingVisual);

return bmp;

}

}

}

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text.RegularExpressions;

using System.Windows;

using System.Windows.Media.Imaging;

namespace SmearsMaker.HPGL

{

public class PltReaderImpl : IPltReader

{

private readonly Painter \_painter;

private static readonly string \_pattern = @"PW(?<Width>\d\*),\d\*;(PC\d\*,(?<Color>(\d\*,\*)\*);(PU[\d,]\*;(PD[\d,]\*);\*)\*)\*";

private static readonly string \_smearPattern = @"PC\d\*,(?<Color>(\d\*,\*)\*);PU(?<Start>[\d,]\*);PD(?<Points>[\d,]\*);";

private static readonly string \_positionsPattern = @"(PD|PU)((?<Position>\d\*,\d\*)(,|;))\*";

public PltReaderImpl()

{

\_painter = new Painter();

}

public BitmapSource ReadPlt(string path)

{

if (string.IsNullOrEmpty(path))

{

throw new NullReferenceException(nameof(path));

}

var file = File.ReadAllText(path);

var matches = Regex.Matches(file, \_pattern);

var (width, height) = GetDimensions(file);

var widthOfSmears = new List<int>();

foreach (Match match in matches)

{

try

{

var smearwidth = Convert.ToInt32(match.Groups["Width"].Value);

widthOfSmears.Add(smearwidth);

var pens = Regex.Matches(match.Value, \_smearPattern);

foreach (Match smearPen in pens)

{

var colorArr = smearPen.Groups["Color"].Value.Split(',');

var colorData = colorArr.Select(p => Convert.ToByte(p)).ToArray();

\_painter.SetPens(colorData, smearwidth);

var points = new List<Point>();

var startArr = smearPen.Groups["Start"].Value.Split(',');

var startPointArr = startArr.Select(Convert.ToSingle).ToArray();

var startPoint = new Point(startPointArr[0], height - startPointArr[1]);

points.Add(startPoint);

var pointsGroup = smearPen.Groups["Points"];

foreach (Capture capture in pointsGroup.Captures)

{

var pointArr = capture.Value.Split(',').Select(Convert.ToSingle).ToArray();

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

{

var point = new Point(pointArr[i], height - pointArr[i + 1]);

points.Add(point);

}

}

\_painter.AddPoints(points);

}

}

catch (Exception ex)

{

throw new Exception($"Ошибка анализа plt файла!\n{ex.Message}", ex);

}

}

int averageWidth = (int)widthOfSmears.Average();

return PltHelper.ToBitmapSource(\_painter.Image, height / averageWidth \* 10, width / averageWidth \* 10);

}

private static (int width, int height) GetDimensions(string file)

{

int maxWidth = 0;

int maxHeight = 0;

int minWidth = int.MaxValue;

int minHeight = int.MaxValue;

foreach (Match position in Regex.Matches(file, \_positionsPattern))

{

foreach (Capture capture in position.Groups["Position"].Captures)

{

var posArr = capture.Value.Split(',').Select(p => Convert.ToInt32(p)).ToArray();

if (posArr[0] > maxWidth)

{

maxWidth = posArr[0];

}

if (posArr[1] > maxHeight)

{

maxHeight = posArr[1];

}

if (posArr[0] < minWidth)

{

minWidth = posArr[0];

}

if (posArr[1] < minHeight)

{

minHeight = posArr[1];

}

}

}

return (maxWidth - minWidth, maxHeight - minHeight);

}

}

}

using SmearsMaker.Common.BaseTypes;

namespace SmearsMaker.ImageProcessing.Clustering

{

public class Cluster : BaseShape

{

public float[] Centroid;

public float[] LastCentroid;

public double DistanceBeetweenCentroids => GetDistance();

private double GetDistance()

{

var dist = 0d;

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

{

dist += Distance(Centroid[i], LastCentroid[i]);

}

return dist;

}

private static float Distance(float first, float second)

{

var distance = first - second;

if (distance < 0)

{

return -distance;

}

return distance;

}

}

}

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using SmearsMaker.Common;

using SmearsMaker.Common.BaseTypes;

namespace SmearsMaker.ImageProcessing.Clustering

{

public abstract class Kmeans : IClusterizer

{

protected readonly List<Point> Points;

protected readonly List<Cluster> Clusters;

private readonly double \_precision;

private readonly int \_maxIteration;

protected Kmeans(int clustersCount, double precision, int maxIteration)

{

Points = new List<Point>();

Clusters = new List<Cluster>();

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

{

Clusters.Add(new Cluster());

}

\_precision = precision;

\_maxIteration = maxIteration;

}

public List<Cluster> Clustering(PointCollection points)

{

Points.AddRange(points.Select(p => p.Clone()));

FillCentroidsWithInitialValues();

double delta;

var counter = 0;

do

{

UpdateMeans();

UpdateCentroids();

delta = Clusters.Sum(cluster => cluster.DistanceBeetweenCentroids);

counter++;

}

while (delta > \_precision && counter < \_maxIteration);

MergingSmallClusters();

return Clusters;

}

protected virtual void MergingSmallClusters()

{

var smallData = new List<Point>();

for (int i = 0; i < Clusters.Count; i++)

{

if (Clusters[i].Points.Count != 0) continue;

smallData.AddRange(Clusters[i].Points);

Clusters.Remove(Clusters[i]);

}

Parallel.ForEach(smallData, d =>

{

var index = NearestCentroid(d.Pixels[Layers.Filtered]);

lock (smallData)

{

Clusters[index].Points.Add(d);

}

});

}

private void UpdateMeans()

{

foreach (var cluster in Clusters)

{

cluster.Points.Clear();

}

Parallel.ForEach(Points, d =>

{

var index = NearestCentroid(d.Pixels[Layers.Original]);

lock (Points)

{

Clusters[index].Points.Add(d);

}

});

Clusters.RemoveAll(c => c.Points.Count == 0);

}

private void UpdateCentroids()

{

Parallel.ForEach(Clusters, UpdateCentroid);

}

protected abstract void FillCentroidsWithInitialValues();

protected abstract void UpdateCentroid(Cluster cluster);

protected abstract int NearestCentroid(Pixel pixel);

protected abstract double Distance(IReadOnlyList<double> left, IReadOnlyList<double> right);

}

}

using System.Collections.Generic;

using System.Linq;

using SmearsMaker.Common;

using SmearsMaker.Common.BaseTypes;

namespace SmearsMaker.ImageProcessing.Clustering

{

public sealed class KmeansClassic : Kmeans

{

public KmeansClassic(int clustersCount, double precision, int maxIteration) : base(clustersCount, precision, maxIteration)

{

}

protected override double Distance(IReadOnlyList<double> left, IReadOnlyList<double> right)

{

double dictance = 0;

for (int i = 0; i < left.Count; i++)

{

var d = left[i] - right[i];

if (d < 0)

{

dictance -= d;

}

else

{

dictance += d;

}

}

return dictance;

}

protected override void FillCentroidsWithInitialValues()

{

var sortedArray = Points.OrderBy(p => p.Pixels[Layers.Filtered].Sum).ToArray();

var step = Points.Count / Clusters.Count;

for (int i = 0; i < Clusters.Count; i++)

{

var point = sortedArray[i \* step / 2];

Clusters[i].Centroid = point.Pixels[Layers.Filtered].Data;

}

}

protected override void UpdateCentroid(Cluster cluster)

{

var newCentroid = new float[cluster.Centroid.Length];

foreach (var data in cluster.Points)

{

var dataArray = data.Pixels[Layers.Filtered].Data;

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

{

newCentroid[i] += dataArray[i];

}

}

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

{

newCentroid[i] /= cluster.Points.Count;

}

cluster.LastCentroid = cluster.Centroid;

cluster.Centroid = newCentroid;

}

protected override int NearestCentroid(Pixel pixel)

{

var index = 0;

var min = pixel.Distance(Clusters[0].Centroid);

for (int i = 0; i < Clusters.Count; i++)

{

var distance = pixel.Distance(Clusters[i].Centroid);

if (min > distance)

{

min = distance;

index = i;

}

}

return index;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using SmearsMaker.Common;

using SmearsMaker.Common.BaseTypes;

namespace SmearsMaker.ImageProcessing.FeatureDetection

{

public class Sobel : IDetector

{

private readonly int \_width;

private readonly int \_height;

public Sobel(int width, int height)

{

\_width = width;

\_height = height;

}

public PointCollection Compute(PointCollection points)

{

var result = points.Clone();

result.Addlayer(Layers.Gradient);

result.Addlayer(Layers.Curves);

var arrLength = result.First().Pixels[Layers.Original].Length;

Parallel.For(0, \_width, (coordX) =>

{

for (int coordY = 0; coordY < \_height; coordY++)

{

var mask = GetMask(result, coordX, coordY);

var pos = coordX \* \_height + coordY;

var gradient = new float[arrLength];

var curve = new float[arrLength];

var gx = (int)((mask[6].GrayScale + mask[7].GrayScale \* 2 + mask[8].GrayScale) - (mask[0].GrayScale + mask[1].GrayScale \* 2 + mask[2].GrayScale));

var gy = (mask[2].GrayScale + mask[5].GrayScale \* 2 + mask[8].GrayScale) - (mask[0].GrayScale + mask[3].GrayScale \* 2 + mask[6].GrayScale);

var norm = (float)Math.Sqrt(gx \* gx + gy \* gy);

var tetta = (float)(Math.Atan(gy / gx) \* 180 / Math.PI);

if (gx < 0)

{

tetta += 180;

}

else if (gx == 0)

{

tetta = 0;

}

for (int i = 0; i < gradient.Length - 1; i++)

{

gradient[i] = tetta;

curve[i] = norm;

}

lock (result)

{

result[pos].Pixels[Layers.Gradient].Data = gradient;

result[pos].Pixels[Layers.Curves].Data = curve;

}

}

});

return result;

}

private List<Pixel> GetMask(PointCollection units, int x, int y)

{

var mask = new List<Pixel>();

var size = units.First().Pixels[Layers.Original].Length;

for (int coordMaskX = x - 1; coordMaskX <= x + 1; coordMaskX++)

{

for (int coordMaskY = y - 1; coordMaskY <= y + 1; coordMaskY++)

{

var idx = coordMaskX \* \_height + coordMaskY;

if (idx < \_height \* \_width && coordMaskX >= 0 && coordMaskY >= 0)

{

mask.Add(units[idx].Pixels[Layers.Filtered]);

}

else

{

mask.Add(Pixel.CreateInstance(new float[size]));

}

}

}

return mask;

}

}

}

using System.Collections.Generic;

using System.Linq;

using SmearsMaker.Common;

using SmearsMaker.Common.BaseTypes;

namespace SmearsMaker.ImageProcessing.Filtering

{

public class MedianFilter : IFilter

{

private readonly int \_rank;

private readonly int \_width;

private readonly int \_height;

public MedianFilter(int rank, int width, int height)

{

\_rank = rank;

\_width = width;

\_height = height;

}

public PointCollection Filtering(PointCollection points)

{

var filteredPoints = new PointCollection();

for (int coordX = 0; coordX < \_width; coordX++)

{

for (int coordY = 0; coordY < \_height; coordY++)

{

var mask = GetMask(points, coordX, coordY);

var pos = coordX \* \_height + coordY;

var median = mask.OrderByDescending(v => v.Sum).ToArray()[mask.Count / 2].Data;

var clonePoint = points[pos].Clone();

clonePoint.Pixels[Layers.Filtered] = Pixel.CreateInstance(median);

filteredPoints.Add(clonePoint);

}

}

return filteredPoints;

}

private List<Pixel> GetMask(PointCollection units, int x, int y)

{

var mask = new List<Pixel>();

for (int coordMaskX = x - \_rank; coordMaskX <= x + \_rank; coordMaskX++)

{

for (int coordMaskY = y - \_rank; coordMaskY <= y + \_rank; coordMaskY++)

{

var idx = coordMaskX \* \_height + coordMaskY;

if (idx < \_height \* \_width && coordMaskX >= 0 && coordMaskY >= 0)

{

mask.Add(units[idx].Pixels[Layers.Original]);

}

}

}

return mask;

}

}

}

using SmearsMaker.Common.BaseTypes;

namespace SmearsMaker.ImageProcessing.Segmenting

{

public class Segment : BaseShape

{

public Segment()

{

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using SmearsMaker.Common.BaseTypes;

namespace SmearsMaker.ImageProcessing.StrokesFormation

{

public abstract class BrushStroke

{

public Guid Id { get; }

public List<BaseShape> Objects { get; }

public virtual System.Windows.Point Head => Objects.First().GetCenter();

public virtual System.Windows.Point Tail => Objects.Last().GetCenter();

public abstract Pixel AverageData { get; }

public abstract int Width { get; }

public abstract int Length { get; }

public abstract double GetDistance(BrushStroke stroke);

protected BrushStroke()

{

Objects = new List<BaseShape>();

Id = Guid.NewGuid();

}

protected BrushStroke(List<BaseShape> baseObjects)

{

Objects = baseObjects;

Id = Guid.NewGuid();

}

}

}

using System.Collections.Generic;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.ImageProcessing.Clustering;

namespace SmearsMaker.ImageProcessing

{

public interface IClusterizer

{

List<Cluster> Clustering(PointCollection points);

}

}

using SmearsMaker.Common.BaseTypes;

namespace SmearsMaker.ImageProcessing

{

public interface IDetector

{

PointCollection Compute(PointCollection points);

}

}

using SmearsMaker.Common.BaseTypes;

namespace SmearsMaker.ImageProcessing

{

public interface IFilter

{

PointCollection Filtering(PointCollection points);

}

}

using System.Collections.Generic;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.ImageProcessing.Segmenting;

namespace SmearsMaker.ImageProcessing

{

public interface ISplitter

{

List<Segment> Splitting(BaseShape segment);

}

}

using System.Collections.Generic;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.ImageProcessing.StrokesFormation;

namespace SmearsMaker.ImageProcessing

{

public interface IStrokesBuilder

{

List<BrushStroke> Execute(IEnumerable<BaseShape> objs);

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using SmearsMaker.ImageProcessing.StrokesFormation;

using SmearsMaker.Tracers.Helpers;

namespace SmearsMaker.Tracers.Extentions

{

public static class BrushStrokeExtension

{

public static bool IsSameColor(this BrushStroke firstSegment, BrushStroke secondSegment, double tolerance)

{

var distance = Math.Abs(firstSegment.AverageData.Average - secondSegment.AverageData.Average);

return distance < tolerance;

}

public static BrushStroke FindFirstNearestByLayer(this BrushStroke obj, IEnumerable<BrushStroke> objs, string layer)

{

var gradHead = obj.Objects.First().GetCenter(layer).Data[0];

var gradTail = obj.Objects.Last().GetCenter(layer).Data[0];

var grads = new List<float>();

return objs.OrderBy(p =>

{

grads.Add(Math.Abs(gradHead - obj.Objects.First().GetCenter(layer).Data[0]));

grads.Add(Math.Abs(gradTail - obj.Objects.Last().GetCenter(layer).Data[0]));

grads.Add(Math.Abs(gradHead - obj.Objects.Last().GetCenter(layer).Data[0]));

grads.Add(Math.Abs(gradTail - obj.Objects.First().GetCenter(layer).Data[0]));

return grads.Min();

}).First();

}

public static List<BrushStroke> FindAllNearest(this BrushStroke seq, IList<BrushStroke> sequences)

{

var result = new List<BrushStroke>();

var head = seq.Head;

var tail = seq.Tail;

double minDistance;

if (sequences.First().Id != seq.Id)

{

minDistance = Utils.SqrtDistance(sequences[0].Head, head);

result.Add(sequences.First());

}

else if (sequences.Count > 1)

{

minDistance = Utils.SqrtDistance(sequences[1].Head, head);

result.Add(sequences[1]);

}

else

{

return new List<BrushStroke>();

}

foreach (var sequence in sequences.Where(s => s.Id != seq.Id))

{

new List<double>{

Utils.SqrtDistance(sequence.Head, head),

Utils.SqrtDistance(sequence.Head, tail),

Utils.SqrtDistance(sequence.Tail, tail),

Utils.SqrtDistance(sequence.Tail, head)

}.ForEach(dist =>

{

if (minDistance > dist)

{

minDistance = dist;

result = new List<BrushStroke> { sequence };

}

else if (minDistance == dist)

{

result.Add(sequence);

}

});

}

return result;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using SmearsMaker.Common.BaseTypes;

namespace SmearsMaker.Tracers.Extentions

{

public static class BaseShapeExtentions

{

public static bool IsSameColor(this BaseShape firstSegment, BaseShape secondSegment, double tolerance, string layer)

{

var distance = Math.Abs(firstSegment.GetCenter(layer).Average - secondSegment.GetCenter(layer).Average);

return distance < tolerance;

}

public static BaseShape FindNearest(this BaseShape obj, IEnumerable<BaseShape> objs, string layer)

{

var data = obj.GetCenter(layer).Data[0];

var result = objs.OrderBy(p => Math.Abs(data - p.GetCenter(layer).Data[0]));

return result.First();

}

public static float[] GetAverageData(this BaseShape seg, string layer)

{

var length = seg.Points.First().Pixels[layer].Length;

var averData = new float[length];

foreach (var point in seg.Points)

{

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

{

averData[i] += point.Pixels[layer].Data[i];

}

}

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

{

averData[i] /= seg.Points.Count;

}

return averData;

}

public static (double minx, double miny, double maxx, double maxy) GetExtremums(this BaseShape segment)

{

var points = segment.Points;

//coordinates for compute vector

double minX = points[0].Position.X;

double minY = points[0].Position.Y;

double maxX = 0;

double maxY = 0;

foreach (var data in points)

{

//find min and max coordinates in segment

if (data.Position.X < minX)

{

minX = data.Position.X;

}

if (data.Position.Y < minY)

{

minY = data.Position.Y;

}

if (data.Position.X > maxX)

{

maxX = data.Position.X;

}

if (data.Position.Y > maxY)

{

maxY = data.Position.Y;

}

}

return (minX, minY, maxX, maxY);

}

}

}

using System;

using System.Collections.Generic;

using SmearsMaker.Common.Image;

namespace SmearsMaker.Tracers.GradientTracers

{

public class GtImageSettings

{

#region properties

public List<ImageSetting> Settings { get; }

public ImageSetting FilterRank { get; }

public ImageSetting HeightPlt { get; }

public ImageSetting WidthPlt { get; }

public ImageSetting WidthSmearUI { get; }

public ImageSetting WidthSmear { get; }

public ImageSetting Tolerance { get; }

public ImageSetting Tolerance2 { get; }

#endregion

public GtImageSettings(int width, int height)

{

FilterRank = new ImageSetting

{

Value = (int)Math.Sqrt((double)(width + height) / 80 / 2),

Name = "Ранг фильтра"

};

HeightPlt = new ImageSetting

{

Value = 7600,

Name = "Ширина plt"

};

WidthPlt = new ImageSetting

{

Value = 5200,

Name = "Высота plt"

};

WidthSmear = new ImageSetting

{

Value = width \* height / 5000 + 1,

Name = "Ширина мазка в plt"

};

WidthSmearUI = new ImageSetting

{

Value = width \* height / 10000 + 1,

Name = "Ширина мазка"

};

Tolerance = new ImageSetting

{

Value = 10,

Name = "Погрешность в rgb"

};

Tolerance2 = new ImageSetting

{

Value = 13,

Name = "Погрешность в rgb для коротких мазков"

};

Settings = new List<ImageSetting>

{

FilterRank,

HeightPlt,

WidthPlt,

WidthSmearUI,

WidthSmear,

Tolerance,

Tolerance2

};

}

}

}

using System.Linq;

using System.Diagnostics;

using SmearsMaker.Common;

using System.Threading.Tasks;

using SmearsMaker.Common.Image;

using System.Collections.Generic;

using SmearsMaker.Tracers.Helpers;

using System.Windows.Media.Imaging;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.ImageProcessing.Segmenting;

using SmearsMaker.ImageProcessing.StrokesFormation;

namespace SmearsMaker.Tracers.GradientTracers

{

public abstract class GTracer : TracerBase

{

public override List<ImageSetting> Settings => GtSettings.Settings;

public override List<ImageView> Views => CreateViews();

private PointCollection \_filteredPoints;

private PointCollection \_detectorPoints;

private List<Segment> \_superPixels;

private List<BrushStroke> \_strokes;

internal readonly GtImageSettings GtSettings;

internal IServicesFactory Factory;

protected GTracer(BitmapSource image) : base(image)

{

GtSettings = new GtImageSettings(Model.Width, Model.Height);

}

public override Task Execute()

{

var detector = Factory.CreateDetector();

var splitter = Factory.CreateSplitter();

var filter = Factory.CreateFilter();

var bsm = Factory.CreateStrokesBuilder();

var segment = new Segment();

var points = Model.Points;

return Task.Run(() =>

{

Log.Trace("Начало обработки изображения");

Progress.NewProgress("Фильтрация");

var sw = Stopwatch.StartNew();

\_filteredPoints = filter.Filtering(points);

Log.Trace($"Фильтрация заняла {sw.Elapsed.Seconds} с.");

sw.Restart();

Progress.NewProgress("Вычисление градиентов");

Log.Trace("Вычисление градиентов");

\_detectorPoints = detector.Compute(\_filteredPoints);

Log.Trace($"Операция заняла {sw.Elapsed.Seconds} с.");

sw.Restart();

segment.Points.AddRange(\_detectorPoints);

\_superPixels = splitter.Splitting(segment);

Log.Trace($"Операция заняла {sw.Elapsed.Seconds} с.");

sw.Restart();

Log.Trace("Создание мазков");

\_strokes = bsm.Execute(\_superPixels);

Log.Trace($"Операция заняла {sw.Elapsed.Seconds} с.");

Log.Trace($"Сформировано {\_superPixels.Count} суперпикселей, {\_strokes.Count} мазков");

Log.Trace("Обработка изображения завершена");

Progress.NewProgress("Готово");

});

}

private List<ImageView> CreateViews()

{

Progress.NewProgress("Вычисление суперпикселей");

var spixels = ImageHelper.CreateRandomImage(\_superPixels, Layers.SuperPixels, Model);

Progress.NewProgress("Вычисление градиентов суперпикселей");

var spixelsGrad = ImageHelper.CreateImage(\_superPixels, Layers.Gradient, Model);

Progress.NewProgress("Вычисление мазков");

var smears = ImageHelper.CreateImageFromStrokes(\_strokes, Layers.Filtered, Model);

Progress.NewProgress("Вычисление мазков (линии)");

var smearsMap = ImageHelper.PaintStrokes(Model.Image, \_strokes, (int)GtSettings.WidthSmearUI.Value);

Progress.NewProgress("Вычисление размытия");

var blurredImage = Model.ConvertToBitmapSource(\_filteredPoints, Layers.Filtered);

Progress.NewProgress("Вычисление градиентов");

var sobelGradients = Model.ConvertToBitmapSource(\_detectorPoints, Layers.Gradient);

Progress.NewProgress("Вычисление границ");

var sobelCurves = Model.ConvertToBitmapSource(\_detectorPoints, Layers.Curves);

Progress.NewProgress("Вычисление центров");

var newCollection = new PointCollection();

newCollection.AddRange(\_superPixels.Select(s => s.GetCenterPoint()).ToList());

var centres = Model.ConvertToBitmapSource(newCollection, Layers.Original);

Progress.NewProgress("Вычисление погрешности");

var diffImage = Model.GetDifference(smearsMap, Layers.Original);

return new List<ImageView>

{

new ImageView(Model.Image, "Оригинал"),

new ImageView(blurredImage, "Размытое изображение"),

new ImageView(sobelCurves, "Границы"),

new ImageView(sobelGradients, "Поле градиентов"),

new ImageView(spixelsGrad, "Суперпиксели-градиенты"),

new ImageView(centres, "Центры суперпикселей"),

new ImageView(spixels, "Суперпиксели"),

new ImageView(smearsMap, "Мазки (Линии)"),

new ImageView(smears, "Мазки"),

new ImageView(diffImage, "Погрешность")

};

}

public override string CreatePlt()

{

return PltHelper.GetPlt(\_strokes, GtSettings.HeightPlt.Value, GtSettings.WidthPlt.Value, GtSettings.WidthSmear.Value, Model.Height, Model.Width);

}

public override void Dispose()

{

\_filteredPoints?.Clear();

\_detectorPoints?.Clear();

\_superPixels?.Clear();

\_strokes?.Clear();

base.Dispose();

}

}

}

using System.Windows.Media.Imaging;

using SmearsMaker.Common;

namespace SmearsMaker.Tracers.GradientTracers

{

public sealed class HexagonGTracer : GTracer

{

public HexagonGTracer(BitmapSource image) : base(image)

{

Factory = new HexagonTracerFactory(GtSettings, Model, Progress);

}

}

}

using System;

using SmearsMaker.Common;

using SmearsMaker.Common.Image;

using SmearsMaker.ImageProcessing;

using SmearsMaker.ImageProcessing.FeatureDetection;

using SmearsMaker.ImageProcessing.Filtering;

using SmearsMaker.Tracers.Logic;

namespace SmearsMaker.Tracers.GradientTracers

{

public class HexagonTracerFactory : IServicesFactory

{

private readonly ImageModel \_model;

private readonly IProgress \_progress;

private readonly GtImageSettings \_settings;

public HexagonTracerFactory(GtImageSettings settings, ImageModel model, IProgress progress)

{

\_model = model;

\_progress = progress;

\_settings = settings;

}

public IStrokesBuilder CreateStrokesBuilder()

{

return new GradientStrokesBuilder(\_progress, \_settings.WidthSmear.Value \* Math.Sqrt(3) / 2 + 3, (float)\_settings.Tolerance.Value, (float)\_settings.Tolerance2.Value);

}

public ISplitter CreateSplitter()

{

return new HexagonSplitter(\_progress, (int)\_settings.WidthSmear.Value);

}

public IClusterizer CreateClusterizer()

{

throw new NotImplementedException();

}

public IFilter CreateFilter()

{

return new MedianFilter((int)\_settings.FilterRank.Value, \_model.Width, \_model.Height);

}

public IDetector CreateDetector()

{

return new Sobel(\_model.Width, \_model.Height);

}

}

}

using System.Windows.Media.Imaging;

using SmearsMaker.Common;

namespace SmearsMaker.Tracers.GradientTracers

{

public sealed class SquadGTracer : GTracer

{

public SquadGTracer(BitmapSource image) : base(image)

{

Factory = new SquadTracerFactory(GtSettings, Model, Progress);

}

}

}

using System;

using SmearsMaker.Common;

using SmearsMaker.Common.Image;

using SmearsMaker.ImageProcessing;

using SmearsMaker.ImageProcessing.FeatureDetection;

using SmearsMaker.ImageProcessing.Filtering;

using SmearsMaker.Tracers.Logic;

namespace SmearsMaker.Tracers.GradientTracers

{

public class SquadTracerFactory : IServicesFactory

{

private readonly ImageModel \_model;

private readonly IProgress \_progress;

private readonly GtImageSettings \_settings;

public SquadTracerFactory(GtImageSettings settings, ImageModel model, IProgress progress)

{

\_model = model;

\_progress = progress;

\_settings = settings;

}

public IStrokesBuilder CreateStrokesBuilder()

{

return new GradientStrokesBuilder(\_progress, \_settings.WidthSmear.Value \* 2 - 2, (float)\_settings.Tolerance.Value, (float)\_settings.Tolerance2.Value);

}

public ISplitter CreateSplitter()

{

return new SuperpixelSplitter(\_progress, (int)\_settings.WidthSmear.Value);

}

public IClusterizer CreateClusterizer()

{

throw new NotImplementedException();

}

public IFilter CreateFilter()

{

return new MedianFilter((int)\_settings.FilterRank.Value, \_model.Width, \_model.Height);

}

public IDetector CreateDetector()

{

return new Sobel(\_model.Width, \_model.Height);

}

}

}

using System.Collections.Generic;

using System.Linq;

using System.Windows;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using SmearsMaker.Common;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.Common.Image;

using SmearsMaker.ImageProcessing.StrokesFormation;

using SmearsMaker.Tracers.Extentions;

using PointCollection = SmearsMaker.Common.BaseTypes.PointCollection;

namespace SmearsMaker.Tracers.Helpers

{

internal static class ImageHelper

{

internal static BitmapSource CreateRandomImage(IEnumerable<BaseShape> objects, string layer, ImageModel model)

{

var data = new PointCollection();

foreach (var obj in objects)

{

var points = obj.Points.Clone();

var rand = Utils.GetGandomData(3).ToArray();

points.ForEach(d =>

{

d.Pixels[layer].Data = rand;

});

data.AddRange(points);

}

return model.ConvertToBitmapSource(data, layer);

}

internal static BitmapSource CreateImage(IEnumerable<BaseShape> objects, string layer, ImageModel model)

{

var data = new PointCollection();

foreach (var obj in objects)

{

var averData = obj.GetAverageData(layer);

var points = obj.Points.Clone();

foreach (var point in points)

{

for (int i = 0; i < point.Pixels[layer].Length; i++)

{

point.Pixels[layer].Data[i] = averData[i];

}

}

data.AddRange(points);

}

return model.ConvertToBitmapSource(data, layer);

}

internal static BitmapSource CreateImageFromStrokes(IEnumerable<BrushStroke> strokes, string layer, ImageModel model)

{

var data = new PointCollection();

foreach (var stroke in strokes)

{

var averageData = stroke.AverageData;

foreach (var obj in stroke.Objects)

{

var points = obj.Points.Clone();

foreach (var point in points)

{

point.Pixels[layer] = averageData;

}

data.AddRange(points);

}

}

return model.ConvertToBitmapSource(data, layer);

}

public static BitmapSource ToBitmapSource(DrawingImage source, int width, int heigth)

{

var drawingVisual = new DrawingVisual();

var drawingContext = drawingVisual.RenderOpen();

var bmp = new RenderTargetBitmap(width, heigth, 96, 96, PixelFormats.Pbgra32);

drawingContext.DrawImage(source, new Rect(new System.Windows.Point(0, 0), new Size(width, heigth)));

drawingContext.Close();

bmp.Render(drawingVisual);

return bmp;

}

internal static BitmapSource PaintStrokes(BitmapSource source, IEnumerable<BrushStroke> strokes, float width)

{

var geometries = new DrawingGroup();

var radius = 1;

foreach (var stroke in strokes.OrderByDescending(s => s.Objects.Count))

{

var center = Utils.GetAverageData(stroke.Objects, Layers.Filtered);

var brush = new SolidColorBrush(GetColorFromArgb(center));

var linePen = new Pen(brush, width);

var circlePen = new Pen(brush, width - 1);

var points = stroke.Objects.Select(point => new System.Windows.Point((int)point.GetCenter().X, (int)point.GetCenter().Y)).ToArray();

var lastPoint = new EllipseGeometry(points.Last(), radius, radius);

geometries.Children.Add(new GeometryDrawing(brush, circlePen, lastPoint));

if (points.Length > 1)

{

for (int i = 0; i < points.Length - 1; i++)

{

var line = new LineGeometry(points[i], points[i + 1]);

var head = new EllipseGeometry(points[i], radius, radius);

var tail = new EllipseGeometry(points[i + 1], radius, radius);

geometries.Children.Add(new GeometryDrawing(brush, circlePen, head));

geometries.Children.Add(new GeometryDrawing(brush, circlePen, tail));

geometries.Children.Add(new GeometryDrawing(brush, linePen, line));

}

}

}

return ToBitmapSource(new DrawingImage(geometries), source.PixelWidth, source.PixelHeight);

}

private static Color GetColorFromArgb(IReadOnlyList<float> data)

{

return Color.FromArgb((byte)data[3], (byte)data[2], (byte)data[1], (byte)data[0]);

}

}

}

using System.Collections.Generic;

using System.Linq;

using System.Text;

using SmearsMaker.ImageProcessing.StrokesFormation;

namespace SmearsMaker.Tracers.Helpers

{

public static class PltHelper

{

internal static string GetPlt(List<BrushStroke> strokes, double pltHeight, double pltWidth, double widthStroke, int imageHeight, int imageWidth)

{

var delta = (float)pltHeight / imageHeight;

var widthImage = imageWidth \* delta;

if (widthImage > pltWidth)

{

delta \*= (float)pltWidth / widthImage;

}

var smearWidth = (int)(widthStroke \* delta);

const int index = 1;

//start plt and add pen

var plt = new StringBuilder().Append($"IN;SP{index};");

//add pen pltWidth

plt.Append($"PW{smearWidth},{index};");

foreach (var stroke in strokes.OrderBy(c => c.AverageData.GrayScale).ThenBy(b => b.Objects.Count))

{

var average = stroke.AverageData.Data;

//add pen color

plt.Append($"PC{index},{(uint)average[2]},{(uint)average[1]},{(uint)average[0]};");

//add strokes

plt.Append($"PU{(uint)(stroke.Head.X \* delta)},{(uint)(pltHeight - stroke.Head.Y \* delta)};");

plt.Append("PD");

for (int i = 1; i < stroke.Objects.Count - 1; i++)

{

var point = stroke.Objects[i].GetCenter();

plt.Append($"{(uint)(point.X \* delta)},{(uint)(pltHeight - point.Y \* delta)},");

}

if (stroke.Objects.Count == 1)

{

plt.Append($"{(uint)(stroke.Tail.X \* delta)},{(uint)(pltHeight - stroke.Tail.Y \* delta)},");

//Hack: paint shortest lines

plt.Append($"{(uint)(stroke.Tail.X \* delta) + 1},{(uint)(pltHeight - stroke.Tail.Y \* delta) + 1};");

}

else

{

plt.Append($"{(uint)(stroke.Tail.X \* delta)},{(uint)(pltHeight - stroke.Tail.Y \* delta)};");

}

}

return plt.ToString();

}

}

}

using System.Collections.Generic;

using System.Linq;

using System.Security.Cryptography;

using SmearsMaker.Common.BaseTypes;

namespace SmearsMaker.Tracers.Helpers

{

public static partial class Utils

{

internal static float[] GetAverageData(IReadOnlyCollection<BaseShape> segments, string layer)

{

var averData = new float[4];

foreach (var segment in segments)

{

segment.Points.ForEach(d =>

{

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

{

averData[i] += d.Pixels[layer].Data[i];

}

});

}

var count = segments.Sum(s => s.Points.Count);

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

{

averData[i] /= count;

}

return averData;

}

internal static List<float> GetGandomData(uint length)

{

var c = new RNGCryptoServiceProvider();

var randomNumber = new byte[length];

c.GetBytes(randomNumber);

return randomNumber.Select(b => (float)b).ToList();

}

}

}

using System;

namespace SmearsMaker.Tracers.Helpers

{

public partial class Utils

{

internal static double ManhattanDistance(System.Windows.Point first, System.Windows.Point second)

{

double dictance = 0;

var d = first.X - second.X;

if (d < 0)

{

dictance -= d;

}

else

{

dictance += d;

}

d = first.Y - second.Y;

if (d < 0)

{

dictance -= d;

}

else

{

dictance += d;

}

return dictance;

}

internal static double SqrtDistance(System.Windows.Point first, System.Windows.Point second)

{

var sum = Math.Pow(first.X - second.X, 2);

sum += Math.Pow(first.Y - second.Y, 2);

return Math.Sqrt(sum);

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using SmearsMaker.Common;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.ImageProcessing;

using SmearsMaker.ImageProcessing.StrokesFormation;

using SmearsMaker.Tracers.Extentions;

using SmearsMaker.Tracers.Helpers;

using SmearsMaker.Tracers.Model;

namespace SmearsMaker.Tracers.Logic

{

public class GradientStrokesBuilder : IStrokesBuilder

{

private double \_tolerance;

private readonly Random \_rnd;

private readonly double \_maxDistance;

private readonly double \_toleranceSecond;

private readonly IProgress \_progress;

public GradientStrokesBuilder(IProgress progress, double width, float toleranceFirst, float toleranceSecond)

{

\_toleranceSecond = toleranceSecond;

\_tolerance = toleranceFirst;

\_maxDistance = width;

\_progress = progress;

\_rnd = new Random();

}

public List<BrushStroke> Execute(IEnumerable<BaseShape> objs)

{

var strokesFromGroups = new List<BrushStroke>();

var groups = SplitSegmentsByColor(objs);

\_progress.NewProgress("Создание мазков", 0, groups.Count);

Parallel.ForEach(groups, group =>

{

var pairs = Pairing(group);

var brushStrokes = pairs.Count > 1 ? Combining(pairs) : pairs;

lock (strokesFromGroups)

{

\_progress.Update(1);

strokesFromGroups.AddRange(brushStrokes);

}

});

\_tolerance = \_toleranceSecond;

return ConcatStrokes(strokesFromGroups);

}

private List<BrushStroke> ConcatStrokes(IEnumerable<BrushStroke> strokesFromGroups)

{

var groupsWithAloneStrokes = SplitSegmentsByColor(strokesFromGroups);

var result = new List<BrushStroke>();

\_progress.NewProgress("Группировка одиночных суперпикселей", 0, groupsWithAloneStrokes.Count);

foreach (var group in groupsWithAloneStrokes)

{

var sameColorStrokes = group.ToList();

var aloneStrokes = sameColorStrokes.Where(stroke => stroke.Objects.Count == 1).ToList();

//sameColorStrokes.RemoveAll(stroke => stroke.Objects.Count == 1);

foreach (var aloneStroke in aloneStrokes)

{

if (sameColorStrokes.Any())

{

var strokes = aloneStroke.FindAllNearest(sameColorStrokes);

if (strokes.Count > 0)

{

if (aloneStroke.GetDistance(strokes.First()) < \_maxDistance)

{

var segmentToBeDeleted = aloneStroke.FindFirstNearestByLayer(strokes, Layers.Gradient);

var combinedSequence = Combine(segmentToBeDeleted, aloneStroke);

sameColorStrokes.Remove(segmentToBeDeleted);

result.Add(combinedSequence);

}

else

{

result.Add(aloneStroke);

}

}

else

{

result.Add(aloneStroke);

}

}

else

{

result.Add(aloneStroke);

}

}

result.AddRange(sameColorStrokes);

\_progress.Update(1);

}

return result;

}

private List<List<BaseShape>> SplitSegmentsByColor(IEnumerable<BaseShape> objs)

{

var groups = new List<List<BaseShape>>();

var segments = new List<BaseShape>();

segments.AddRange(objs);

while (segments.Count > 0)

{

var segment = segments[\_rnd.Next(0, segments.Count)];

var searchableSegments = segments.FindAll(s => s.IsSameColor(segment, \_tolerance, Layers.Gradient));

var group = new List<BaseShape>();

group.AddRange(searchableSegments);

segments.RemoveAll(p => searchableSegments.Contains(p));

segments.Remove(segment);

groups.Add(group);

}

return groups;

}

private List<List<BrushStroke>> SplitSegmentsByColor(IEnumerable<BrushStroke> strokes)

{

var groups = new List<List<BrushStroke>>();

var segments = new List<BrushStroke>();

segments.AddRange(strokes);

while (segments.Count > 0)

{

var segment = segments[\_rnd.Next(0, segments.Count)];

var searchableSegments = segments.FindAll(s => s.IsSameColor(segment, \_tolerance));

var group = new List<BrushStroke>();

group.AddRange(searchableSegments);

segments.RemoveAll(p => searchableSegments.Contains(p));

segments.Remove(segment);

groups.Add(group);

}

return groups;

}

private IList<BrushStroke> Pairing(IEnumerable<BaseShape> objs)

{

var brushStrokes = new List<BrushStroke>();

var points = new List<BaseShape>();

points.AddRange(objs);

while (points.Count > 0)

{

var list = new List<BaseShape>();

var main = points[\_rnd.Next(0, points.Count)];

list.Add(main);

points.Remove(main);

var nearestPoints = points.FindAll(p => Utils.SqrtDistance(p.GetCenter(), main.GetCenter()) < \_maxDistance);

if (nearestPoints.Any())

{

var next = main.FindNearest(nearestPoints, Layers.Gradient);

list.Add(next);

points.Remove(next);

}

brushStrokes.Add(new BrushStrokeImpl(list));

}

return brushStrokes;

}

private IEnumerable<BrushStroke> Combining(IList<BrushStroke> sequences)

{

bool segmentsAreMerged;

do

{

segmentsAreMerged = false;

for (int i = 0; i < sequences.Count; i++)

{

var strokes = sequences[i].FindAllNearest(sequences);

var distance = sequences[i].GetDistance(strokes.First());

if (distance < \_maxDistance)

{

var segmentToBeDeleted = sequences[i].FindFirstNearestByLayer(strokes, Layers.Gradient);

var combinedSequence = Combine(segmentToBeDeleted, sequences[i]);

sequences.RemoveAt(i);

sequences.Remove(segmentToBeDeleted);

sequences.Add(combinedSequence);

segmentsAreMerged = true;

}

}

} while (segmentsAreMerged && sequences.Count > 1);

return sequences;

}

private static BrushStroke Combine(BrushStroke first, BrushStroke second)

{

var newSequence = new BrushStrokeImpl();

var a = Utils.SqrtDistance(first.Head, second.Head);

var b = Utils.SqrtDistance(first.Tail, second.Tail);

var c = Utils.SqrtDistance(first.Head, second.Tail);

var d = Utils.SqrtDistance(first.Tail, second.Head);

if (a < b)

{

if (a < c)

{

if (a < d)

{

first.Objects.Reverse();

newSequence.Objects.AddRange(first.Objects);

newSequence.Objects.AddRange(second.Objects);

}

else

{

newSequence.Objects.AddRange(first.Objects);

newSequence.Objects.AddRange(second.Objects);

}

}

else

{

if (c < d)

{

newSequence.Objects.AddRange(second.Objects);

newSequence.Objects.AddRange(first.Objects);

}

else

{

newSequence.Objects.AddRange(first.Objects);

newSequence.Objects.AddRange(second.Objects);

}

}

}

else

{

if (b < c)

{

if (b < d)

{

second.Objects.Reverse();

newSequence.Objects.AddRange(first.Objects);

newSequence.Objects.AddRange(second.Objects);

}

else

{

newSequence.Objects.AddRange(first.Objects);

newSequence.Objects.AddRange(second.Objects);

}

}

else

{

if (c < d)

{

newSequence.Objects.AddRange(second.Objects);

newSequence.Objects.AddRange(first.Objects);

}

else

{

newSequence.Objects.AddRange(first.Objects);

newSequence.Objects.AddRange(second.Objects);

}

}

}

return newSequence;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using SmearsMaker.Common;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.ImageProcessing;

using SmearsMaker.ImageProcessing.Segmenting;

using SmearsMaker.Tracers.Extentions;

using SmearsMaker.Tracers.Helpers;

namespace SmearsMaker.Tracers.Logic

{

public class HexagonSplitter : ISplitter

{

private double \_inscribedRadius;

private readonly double \_circumscribedRadius;

private readonly IProgress \_progress;

public HexagonSplitter(IProgress progress, int length)

{

\_circumscribedRadius = length;

\_progress = progress;

}

public List<Segment> Splitting(BaseShape segment)

{

segment.Points.Addlayer(Layers.SuperPixels);

\_inscribedRadius = Math.Sqrt(3) / 2 \* \_circumscribedRadius;

//spliting complex segment into superPixels

var data = segment.Points;

var samples = PlacementCenters(\_inscribedRadius, segment);

//Search for winners and distribution of data

\_progress.NewProgress("Создание суперпикселей", 0, data.Count);

Parallel.ForEach(data, unit =>

{

var winner = NearestCentroid(unit, samples);

lock (samples)

{

\_progress.Update(1);

winner.Points.Add(unit);

}

});

var result = new List<Segment>();

foreach (var row in samples)

{

result.AddRange(row.Where(superPixel => superPixel.Item2.Points.Count > 0).Select(s => s.Item2).ToList());

}

result.ForEach(s =>

{

s.Points.Addlayer(Layers.Original);

s.Points.Addlayer(Layers.Gradient);

s.Points.Addlayer(Layers.SuperPixels);

s.Points.Addlayer(Layers.Curves);

s.Points.Addlayer(Layers.Filtered);

}

);

return result;

}

protected Segment NearestCentroid(Point pixel, List<List<(System.Windows.Point, Segment)>> superPixels)

{

var nearest = superPixels[0].First().Item2;

var min = Utils.SqrtDistance(superPixels[0].First().Item1, pixel.Position);

var minIndex = 0;

foreach (var row in superPixels)

{

var minY = Math.Abs(row.First().Item1.Y - pixel.Position.Y);

if (minIndex < 0)

{

minIndex = 0;

}

for (var i = minIndex; i < row.Count; i++)

{

var center = row[i].Item1;

var dy = Math.Abs(center.Y - pixel.Position.Y);

if (dy <= minY)

{

minY = dy;

minIndex = i - 1;

}

else

{

break;

}

var distance = Utils.SqrtDistance(center, pixel.Position);

if (min > distance)

{

min = distance;

nearest = row[i].Item2;

}

}

}

return nearest;

}

protected List<List<(System.Windows.Point, Segment)>> PlacementCenters(double diameter, BaseShape segment)

{

var (minx, miny, maxx, maxy) = segment.GetExtremums();

var samplesData = new List<List<(System.Windows.Point, Segment)>>();

var count = 0;

for (double i = minx + diameter / 2; i < maxx; i += diameter)

{

var offset = count % 2 == 0 ? diameter : diameter / 2;

var row = new List<(System.Windows.Point, Segment)>();

for (double j = miny + offset; j < maxy; j += diameter)

{

var segments = (p: new System.Windows.Point(i, j), s: new Segment());

row.Add(segments);

}

samplesData.Add(row);

count++;

}

return samplesData;

}

}

}

using System.Collections.Generic;

using System.Linq;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.ImageProcessing;

using SmearsMaker.ImageProcessing.StrokesFormation;

using SmearsMaker.Tracers.Helpers;

using SmearsMaker.Tracers.Model;

namespace SmearsMaker.Tracers.Logic

{

public class PairStrokesBuilder : IStrokesBuilder

{

private readonly double \_maxDistance;

public PairStrokesBuilder(double width)

{

\_maxDistance = width;

}

public List<BrushStroke> Execute(IEnumerable<BaseShape> objs)

{

var objsList = objs.ToList();

var pairs = Pairing(objsList);

var brushStrokes = Combining(pairs);

return brushStrokes;

}

private List<BrushStroke> Combining(List<BrushStroke> sequences)

{

bool distanceCheck;

do

{

distanceCheck = false;

if (sequences.Count > 1)

{

for (int i = 0; i < sequences.Count && sequences.Count > 1; i++)

{

var index = NearestPart(sequences, sequences[i]);

if (sequences[i].GetDistance(sequences[index]) < \_maxDistance)

{

distanceCheck = true;

var combinedSequence = Combine(sequences[i], sequences[index]);

var segmentToBeDeleted = sequences[index];

sequences.RemoveAt(i);

sequences.Remove(segmentToBeDeleted);

sequences.Add(combinedSequence);

}

}

}

} while (distanceCheck);

return sequences;

}

private static BrushStroke Combine(BrushStroke first, BrushStroke second)

{

var newSequence = new BrushStrokeImpl();

var a = Utils.SqrtDistance(first.Objects.First().GetCenter(), second.Objects.First().GetCenter());

var b = Utils.SqrtDistance(first.Objects.Last().GetCenter(), second.Objects.Last().GetCenter());

var c = Utils.SqrtDistance(first.Objects.First().GetCenter(), second.Objects.Last().GetCenter());

var d = Utils.SqrtDistance(first.Objects.Last().GetCenter(), second.Objects.First().GetCenter());

if (a < b)

{

if (a < c)

{

if (a < d)

{

first.Objects.Reverse();

newSequence.Objects.AddRange(first.Objects);

newSequence.Objects.AddRange(second.Objects);

}

else

{

newSequence.Objects.AddRange(first.Objects);

newSequence.Objects.AddRange(second.Objects);

}

}

else

{

if (c < d)

{

newSequence.Objects.AddRange(second.Objects);

newSequence.Objects.AddRange(first.Objects);

}

else

{

newSequence.Objects.AddRange(first.Objects);

newSequence.Objects.AddRange(second.Objects);

}

}

}

else

{

if (b < c)

{

if (b < d)

{

second.Objects.Reverse();

newSequence.Objects.AddRange(first.Objects);

newSequence.Objects.AddRange(second.Objects);

}

else

{

newSequence.Objects.AddRange(first.Objects);

newSequence.Objects.AddRange(second.Objects);

}

}

else

{

if (c < d)

{

newSequence.Objects.AddRange(second.Objects);

newSequence.Objects.AddRange(first.Objects);

}

else

{

newSequence.Objects.AddRange(first.Objects);

newSequence.Objects.AddRange(second.Objects);

}

}

}

return newSequence;

}

private static int NearestPart(IList<BrushStroke> sequences, BrushStroke seq)

{

var headPosition = seq.Objects.First().GetCenter();

var tailPosition = seq.Objects.Last().GetCenter();

double minDistance;

int index;

if (sequences.First() != seq)

{

minDistance = Utils.SqrtDistance(sequences[0].Objects.First().GetCenter(), headPosition);

index = 0;

}

else

{

minDistance = Utils.SqrtDistance(sequences[1].Objects.First().GetCenter(), headPosition);

index = 1;

}

foreach (var sequence in sequences.Where(s => s != seq))

{

var hh = Utils.SqrtDistance(sequence.Objects.First().GetCenter(), headPosition);

var ht = Utils.SqrtDistance(sequence.Objects.First().GetCenter(), tailPosition);

var tt = Utils.SqrtDistance(sequence.Objects.Last().GetCenter(), tailPosition);

var th = Utils.SqrtDistance(sequence.Objects.Last().GetCenter(), headPosition);

if (hh > 0 && minDistance > hh)

{

index = sequences.IndexOf(sequence);

minDistance = hh;

}

if (ht > 0 && minDistance > ht)

{

index = sequences.IndexOf(sequence);

minDistance = ht;

}

if (tt > 0 && minDistance > tt)

{

index = sequences.IndexOf(sequence);

minDistance = tt;

}

if (th > 0 && minDistance > th)

{

index = sequences.IndexOf(sequence);

minDistance = th;

}

}

return index;

}

private static int FindNearestSequence(IReadOnlyList<BrushStroke> sequences, Point point)

{

var index = 0;

var minDistance = Utils.SqrtDistance(sequences.First().Objects.First().GetCenter(), point.Position);

for (int i = 0; i < sequences.Count; i++)

{

var distanceHead = Utils.SqrtDistance(sequences[i].Objects.First().GetCenter(), point.Position);

var distanceTail = Utils.SqrtDistance(sequences[i].Objects.Last().GetCenter(), point.Position);

if (minDistance > distanceHead)

{

minDistance = distanceHead;

index = i;

}

if (minDistance > distanceTail)

{

minDistance = distanceTail;

index = i;

}

}

return index;

}

private static BrushStroke Combine(BrushStroke first, BaseShape second)

{

var newSequence = new BrushStrokeImpl();

var distanceHead = Utils.SqrtDistance(first.Objects.First().GetCenter(), second.GetCenter());

var distanceTail = Utils.SqrtDistance(first.Objects.Last().GetCenter(), second.GetCenter());

if (distanceHead > distanceTail)

{

newSequence.Objects.AddRange(first.Objects);

newSequence.Objects.Add(second);

}

else

{

newSequence.Objects.Add(second);

newSequence.Objects.AddRange(first.Objects);

}

return newSequence;

}

private static System.Windows.Point GetCenter(IReadOnlyCollection<BaseShape> objs)

{

var x = 0d;

var y = 0d;

foreach (var obj in objs)

{

x += obj.GetCenter().X;

y += obj.GetCenter().Y;

}

return new System.Windows.Point(x / objs.Count, y / objs.Count);

}

private List<BrushStroke> Pairing(IReadOnlyCollection<BaseShape> objs)

{

var brushStrokes = new List<BrushStroke>();

var points = new List<BaseShape>();

points.AddRange(objs);

if (objs.Count > 1)

{

var center = GetCenter(objs);

var startPoint = objs.OrderBy(p => Utils.SqrtDistance(center, p.GetCenter())).First();

while (points.Count > 0)

{

if (points.Count > 1)

{

var list = new List<BaseShape>();

var main = points.OrderBy(p => Utils.SqrtDistance(startPoint.GetCenter(), p.GetCenter())).Last();

list.Add(main);

points.Remove(main);

var next = points.OrderBy(p => Utils.SqrtDistance(main.GetCenter(), p.GetCenter())).First();

if (Utils.SqrtDistance(next.GetCenter(), main.GetCenter()) < \_maxDistance)

{

list.Add(next);

points.Remove(next);

brushStrokes.Add(new BrushStrokeImpl(list));

}

else

{

var index = FindNearestSequence(brushStrokes, list.First().GetCenterPoint());

var newSequence = Combine(brushStrokes[index], list.First());

brushStrokes.RemoveAt(index);

brushStrokes.Add(newSequence);

}

}

else

{

var index = FindNearestSequence(brushStrokes, points.First().GetCenterPoint());

var newSequence = Combine(brushStrokes[index], points.First());

brushStrokes.RemoveAt(index);

brushStrokes.Add(newSequence);

points = new List<BaseShape>();

}

}

}

else

{

var brushStroke = new BrushStrokeImpl();

brushStroke.Objects.Add(objs.First());

brushStrokes.Add(brushStroke);

}

return brushStrokes;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.ImageProcessing;

using SmearsMaker.ImageProcessing.StrokesFormation;

using SmearsMaker.Tracers.Helpers;

using SmearsMaker.Tracers.Model;

using Point = System.Windows.Point;

namespace SmearsMaker.Tracers.Logic

{

public class SimpleStrokesBuilder : IStrokesBuilder

{

private readonly double \_maxLemgth;

private Point \_finishPoint;

public SimpleStrokesBuilder(double width)

{

\_maxLemgth = width;

}

public List<BrushStroke> Execute(IEnumerable<BaseShape> objs)

{

if (objs == null)

throw new NullReferenceException(nameof(objs));

var objsList = objs.ToList();

FindPoints(objsList);

if (objsList.Count > 0)

{

var size = Math.Sqrt(objs.First().Points.Count);

var brushStrokes = new List<BrushStroke>();

var points = new List<BaseShape>();

points.AddRange(objs);

double length = 0;

var list = new List<BaseShape>();

while (points.Count > 0)

{

var main = points.OrderBy(p => Utils.SqrtDistance(\_finishPoint, p.GetCenter())).First();

list.Add(main);

points.Remove(main);

if (points.Count > 0)

{

do

{

var next = points.OrderBy(p => Utils.SqrtDistance(list.Last().GetCenter(), p.GetCenter())).First();

if (Utils.SqrtDistance(list.Last().GetCenter(), next.GetCenter()) / 2 < size)

{

length += Utils.SqrtDistance(list.Last().GetCenter(), next.GetCenter());

if (length <= \_maxLemgth)

{

\_finishPoint = next.GetCenter();

list.Add(next);

points.Remove(next);

}

}

else

{

break;

}

} while (length <= \_maxLemgth && points.Count > 0);

brushStrokes.Add(new BrushStrokeImpl(list));

if (length <= \_maxLemgth)

{

length = 0;

list = new List<BaseShape>();

}

else

{

length = 0;

list = new List<BaseShape>

{

brushStrokes.Last().Objects.Last()

};

}

}

else

{

brushStrokes.Add(new BrushStrokeImpl(list));

}

}

return brushStrokes;

}

return new List<BrushStroke>();

}

private void FindPoints(IEnumerable<BaseShape> objs)

{

var finish = objs.First().Points.First().Position;

double maxDistance = 0;

foreach (var objOne in objs)

{

foreach (var objTwo in objs)

{

if (Utils.SqrtDistance(objOne.GetCenter(), objTwo.GetCenter()) > maxDistance)

{

maxDistance = Utils.SqrtDistance(objOne.GetCenter(), objTwo.GetCenter());

finish = objTwo.GetCenter();

}

}

}

\_finishPoint = finish;

}

}

}

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using SmearsMaker.Common;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.ImageProcessing;

using SmearsMaker.ImageProcessing.Segmenting;

using SmearsMaker.Tracers.Extentions;

using SmearsMaker.Tracers.Helpers;

namespace SmearsMaker.Tracers.Logic

{

public class SuperpixelSplitter : ISplitter

{

private readonly int \_length;

private readonly IProgress \_progress;

public SuperpixelSplitter(IProgress progress, int length)

{

\_length = length;

\_progress = progress;

}

public virtual List<Segment> Splitting(BaseShape segment)

{

segment.Points.Addlayer(Layers.SuperPixels);

//spliting complex segment into superPixels

var data = segment.Points;

var samples = PlacementCenters(\_length, segment);

//Search for winners and distribution of data

\_progress.NewProgress("Создание суперпикселей", 0, data.Count);

Parallel.ForEach(data, unit =>

{

var winner = NearestCentroid(unit, samples);

\_progress.Update(1);

lock (samples)

{

winner.Points.Add(unit);

}

});

//Deleting empty cells and cells with small data count

var segments = samples.Values.Where(superPixel => superPixel.Points.Count > 0).ToList();

segments.ForEach(s =>

{

s.Points.Addlayer(Layers.Original);

s.Points.Addlayer(Layers.Gradient);

s.Points.Addlayer(Layers.SuperPixels);

s.Points.Addlayer(Layers.Curves);

s.Points.Addlayer(Layers.Filtered);

}

);

return segments;

}

protected Dictionary<System.Windows.Point, Segment> PlacementCenters(double diameter, BaseShape segment)

{

var samplesData = new Dictionary<System.Windows.Point, Segment>();

var (minx, miny, maxx, maxy) = segment.GetExtremums();

var firstPoint = new System.Windows.Point(minx, miny);

var secondPoint = new System.Windows.Point(maxx, maxy);

var widthCount = (int)(maxx - minx);

var heightCount = (int)(maxy - miny);

if (widthCount > diameter && heightCount > diameter)

{

for (double i = firstPoint.X; i < secondPoint.X; i += diameter)

{

for (double j = firstPoint.Y; j < secondPoint.Y; j += diameter)

{

samplesData.Add(new System.Windows.Point(i + diameter / 2, j + diameter / 2), new Segment());

}

}

}

else

{

samplesData.Add(new System.Windows.Point((firstPoint.X + secondPoint.X) / 2, (firstPoint.Y + secondPoint.Y) / 2), new Segment());

}

return samplesData;

}

protected Segment NearestCentroid(Point pixel, Dictionary<System.Windows.Point,Segment> superPixels)

{

var nearest = superPixels.First().Value;

var min = Utils.ManhattanDistance(superPixels.First().Key, pixel.Position);

foreach (var superPixel in superPixels)

{

var distance = Utils.ManhattanDistance(superPixel.Key, pixel.Position);

if (min > distance)

{

min = distance;

nearest = superPixel.Value;

}

}

return nearest;

}

}

}

using System.Collections.Generic;

using System.Linq;

using SmearsMaker.Common;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.ImageProcessing.StrokesFormation;

using SmearsMaker.Tracers.Helpers;

namespace SmearsMaker.Tracers.Model

{

public class BrushStrokeImpl : BrushStroke

{

public BrushStrokeImpl()

{

}

public BrushStrokeImpl(List<BaseShape> baseObjects) : base(baseObjects)

{

}

public override int Width { get; }

public override int Length

{

get

{

var length = 0d;

for (int i = 1; i < Objects.Count; i++)

{

length += Utils.SqrtDistance(Objects[i - 1].GetCenter(), Objects[i].GetCenter());

}

return (int)length;

}

}

public override double GetDistance(BrushStroke stroke)

{

return new List<double>

{

Utils.SqrtDistance(Head, stroke.Head),

Utils.SqrtDistance(Head, stroke.Tail),

Utils.SqrtDistance(Tail, stroke.Tail),

Utils.SqrtDistance(Tail, stroke.Head)

}.Min();

}

public override Pixel AverageData => ComputeAverageData();

private Pixel ComputeAverageData()

{

var centers = Objects.Select(o => o.GetCenter(Layers.Original)).ToList();

var length = centers.First().Data.Length;

var averageData = new float[length];

foreach (var center in centers)

{

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

{

averageData[i] += center.Data[i];

}

}

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

{

averageData[i] /= centers.Count;

}

return Pixel.CreateInstance(averageData);

}

}

}

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.Tracers.Helpers;

namespace SmearsMaker.Tracers.SmearTracer

{

public static class Merger

{

internal static void MergePointsWithSmears(List<Smear> smears, List<Point> points)

{

foreach (var point in points)

{

var minDist = Utils.ManhattanDistance(smears.First().BrushStroke.Objects.First().GetCenter(), point.Position);

var nearestObj = smears.First().BrushStroke.Objects.First();

Parallel.ForEach(smears, smear =>

{

foreach (var obj in smear.BrushStroke.Objects)

{

var dist = Utils.ManhattanDistance(obj.GetCenter(), point.Position);

if (dist < minDist)

{

lock (nearestObj)

{

nearestObj = obj;

}

minDist = dist;

}

}

});

nearestObj.Points.Add(point);

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.ImageProcessing.Segmenting;

using SmearsMaker.Tracers.Helpers;

namespace SmearsMaker.Tracers.SmearTracer

{

public static class SegmentSplitter

{

public static List<Segment> Split(BaseShape shape)

{

var segments = new List<Segment>();

var data = shape.Points.OrderBy(p => p.Position.X).ToList();

while (data.Count > 0)

{

int countPrevious, countNext;

var segment = new Segment();

segment.Points.Add(data[0]);

data.RemoveAt(0);

do

{

var segmentData = new List<Point>();

countPrevious = data.Count;

foreach (var pixel in data.OrderBy(p => Utils.ManhattanDistance(p.Position, segment.Points.Last().Position)))

{

if (Contains(pixel, segment.Points))

{

segment.Points.Add(pixel);

}

else

{

segmentData.Add(pixel);

}

}

data = segmentData;

countNext = segmentData.Count;

} while (countPrevious != countNext);

segments.Add(segment);

}

return segments;

}

private static bool Contains(Point pixel, IEnumerable<Point> units)

{

return units.Any(unit => Math.Abs(pixel.Position.X - unit.Position.X) < 2 && Math.Abs(pixel.Position.Y - unit.Position.Y) < 2);

}

}

}

using SmearsMaker.ImageProcessing.Clustering;

using SmearsMaker.ImageProcessing.Segmenting;

using SmearsMaker.ImageProcessing.StrokesFormation;

namespace SmearsMaker.Tracers.SmearTracer

{

public class Smear

{

public BrushStroke BrushStroke { get; }

public Segment Segment { get; set; }

public Cluster Cluster { get; set; }

public Smear(BrushStroke brushStroke)

{

BrushStroke = brushStroke;

}

}

}

using SmearsMaker.Common;

using SmearsMaker.Common.Image;

using SmearsMaker.ImageProcessing;

using SmearsMaker.ImageProcessing.Clustering;

using SmearsMaker.ImageProcessing.FeatureDetection;

using SmearsMaker.ImageProcessing.Filtering;

using SmearsMaker.Tracers.Logic;

namespace SmearsMaker.Tracers.SmearTracer

{

public class SmearFactory : IServicesFactory

{

private readonly ImageModel \_model;

private readonly IProgress \_progress;

private readonly StImageSettings \_settings;

public SmearFactory(StImageSettings settings, ImageModel model, IProgress progress)

{

\_model = model;

\_progress = progress;

\_settings = settings;

}

public IStrokesBuilder CreateStrokesBuilder()

{

return new PairStrokesBuilder(\_settings.MaxSmearDistance.Value);

}

public ISplitter CreateSplitter()

{

return new SuperpixelSplitter(\_progress, (int)\_settings.WidthSmear.Value);

}

public IClusterizer CreateClusterizer()

{

return new KmeansClassic((int)\_settings.ClustersCount.Value, \_settings.ClustersPrecision.Value, (int)\_settings.ClusterMaxIteration.Value);

}

public IFilter CreateFilter()

{

return new MedianFilter((int)\_settings.FilterRank.Value, \_model.Width, \_model.Height);

}

public IDetector CreateDetector()

{

return new Sobel(\_model.Width, \_model.Height);

}

}

}

using System;

using System.Collections.Generic;

using SmearsMaker.Common.Image;

namespace SmearsMaker.Tracers.SmearTracer

{

public class StImageSettings

{

#region properties

public List<ImageSetting> Settings { get; }

public ImageSetting FilterRank { get; }

public ImageSetting ClustersCount { get; }

public ImageSetting MaxSmearDistance { get; }

public ImageSetting MinSizeSuperpixel { get; }

public ImageSetting MaxSizeSuperpixel { get; }

public ImageSetting ClustersPrecision { get; }

public ImageSetting ClusterMaxIteration { get; }

public ImageSetting WidthSmear { get; }

public ImageSetting HeightPlt { get; }

public ImageSetting WidthPlt { get; }

#endregion

public StImageSettings(int width, int height)

{

ClustersCount = new ImageSetting

{

Value = (int)(Math.Sqrt(width + height) + Math.Sqrt(width + height) / 2) + 3,

Name = "Кол-во кластеров"

};

ClusterMaxIteration = new ImageSetting

{

Value = 100,

Name = "Макс. число итераций класт."

};

MinSizeSuperpixel = new ImageSetting

{

Value = (int)(width \* height / 1000),

Name = "Мин. размер суперпикселя"

};

MaxSizeSuperpixel = new ImageSetting

{

Value = (int)(width \* height / 500),

Name = "Мaкс. размер суперпикселя"

};

MaxSmearDistance = new ImageSetting

{

Value = width \* height,

Name = "Максимальная длина мазка"

};

FilterRank = new ImageSetting

{

Value = (int)Math.Sqrt((double)(width + height) / 80 / 2),

Name = "Ранг фильтра"

};

ClustersPrecision = new ImageSetting

{

Value = 0.001f,

Name = "Точность поиска кластеров"

};

WidthSmear = new ImageSetting

{

Value = width \* height / 5000 + 1,

Name = "Ширина мазка в plt"

};

WidthSmear = new ImageSetting

{

Value = width \* height / 5000 + 1,

Name = "Ширина мазка в plt"

};

HeightPlt = new ImageSetting

{

Value = 7600,

Name = "Ширина plt"

};

WidthPlt = new ImageSetting

{

Value = 5200,

Name = "Высота plt"

};

Settings = new List<ImageSetting>

{

ClustersCount,

ClusterMaxIteration,

MinSizeSuperpixel,

MaxSizeSuperpixel,

MaxSmearDistance,

FilterRank,

ClustersPrecision,

WidthSmear,

HeightPlt,

WidthPlt

};

}

}

}

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Media.Imaging;

using SmearsMaker.Common;

using SmearsMaker.Common.BaseTypes;

using SmearsMaker.Common.Image;

using SmearsMaker.Tracers.Helpers;

using Point = SmearsMaker.Common.BaseTypes.Point;

namespace SmearsMaker.Tracers.SmearTracer

{

public class STracer : TracerBase

{

public override List<ImageSetting> Settings => \_settings.Settings;

public override List<ImageView> Views => CreateViews();

private readonly StImageSettings \_settings;

private readonly IServicesFactory \_factory;

private List<Smear> \_smears;

public STracer(BitmapSource image) : base(image)

{

\_settings = new StImageSettings(Model.Width, Model.Height);

\_factory = new SmearFactory(\_settings, Model, Progress);

}

public override Task Execute()

{

var filter = \_factory.CreateFilter();

var kmeans = \_factory.CreateClusterizer();

var supPixSplitter = \_factory.CreateSplitter();

var bsm = \_factory.CreateStrokesBuilder();

var points = Model.Points;

var segmentsCount = 0;

var smearsCount = 0;

return Task.Run(() =>

{

\_smears = new List<Smear>();

Log.Trace("Начало обработки изображения");

Progress.NewProgress("Фильтрация");

var sw = Stopwatch.StartNew();

var filteredPoints = filter.Filtering(points);

Log.Trace($"Фильтрация заняла {sw.Elapsed.Seconds} с.");

sw.Restart();

Progress.NewProgress("Кластеризация");

var clusters = kmeans.Clustering(filteredPoints);

Log.Trace($"Кластеризация заняла {sw.Elapsed.Seconds} с.");

var defectedPixels = new List<Point>();

Progress.NewProgress("Обработка", 0, clusters.Sum(c => c.Points.Count));

Parallel.ForEach(clusters, (cluster) =>

{

var swClusters = Stopwatch.StartNew();

var segments = SegmentSplitter.Split(cluster);

segmentsCount += segments.Count;

Log.Trace($"Сегментация кластера размером {cluster.Points.Count} пикселей заняла {swClusters.Elapsed.Seconds} с.");

swClusters.Reset();

Parallel.ForEach(segments, segment =>

{

var superPixels = supPixSplitter.Splitting(segment);

Parallel.ForEach(superPixels, (supPix) =>

{

if (supPix.Points.Count < \_settings.MinSizeSuperpixel.Value)

{

lock (defectedPixels)

{

defectedPixels.AddRange(supPix.Points);

}

}

});

if (superPixels.Count > 0)

{

var smears = bsm.Execute(superPixels);

smearsCount += smears.Count;

foreach (var smear in smears)

{

var newSmear = new Smear(smear)

{

Cluster = cluster,

Segment = segment

};

lock (\_smears)

{

\_smears.Add(newSmear);

}

Progress.Update(newSmear.BrushStroke.Objects.Sum(o => o.Points.Count));

}

}

});

Log.Trace($"Разбиение на мазки кластера размером {cluster.Points.Count} пикселей заняло {swClusters.Elapsed.Seconds} с.");

});

if (defectedPixels.Any(point => point == null))

{

throw new NullReferenceException("point");

}

Progress.NewProgress("Распределение удаленных точек");

Merger.MergePointsWithSmears(\_smears, defectedPixels);

Log.Trace("Обработка изображения завершена");

Progress.NewProgress("Готово");

Log.Trace($"Сформировано {clusters.Count} кластеров, {segmentsCount} сегментов, {smearsCount} мазков");

});

}

private List<ImageView> CreateViews()

{

Progress.NewProgress("Вычисление погрешности");

var diffImage = Model.GetDifference(BrushStrokesPlt(), Layers.Original);

return new List<ImageView>

{

new ImageView(Model.Image, "Оригинал"),

new ImageView(SuperPixels(), "Суперпиксели"),

new ImageView(Clusters(), "Кластеры"),

new ImageView(Segments(), "Сегменты"),

new ImageView(BrushStrokes(), "Мазки"),

new ImageView(RandomBrushStrokes(), "Мазки(рандом)"),

new ImageView(BrushStrokesLines(), "Мазки(линии)"),

new ImageView(BrushStrokesPlt(), "Мазки"),

new ImageView(diffImage, "Погрешность")

};

}

private BitmapSource SuperPixels()

{

Progress.NewProgress("Вычисление суперпикселей");

var data = new PointCollection();

foreach (var smear in \_smears)

{

foreach (var obj in smear.BrushStroke.Objects)

{

var color = Utils.GetGandomData(3);

obj.Points.ForEach(d => d.Pixels[Layers.Filtered] = Pixel.CreateInstance(color.ToArray()));

//obj.Data.ForEach(d => d.Pixels[Layers.Filtered] = new Pixel(obj.Centroid.Pixels[Layers.Original].Data));

data.AddRange(obj.Points);

}

}

return Model.ConvertToBitmapSource(data, Layers.Filtered);

}

private BitmapSource Clusters()

{

Progress.NewProgress("Вычисление кластеров");

var data = new PointCollection();

foreach (var smear in \_smears)

{

foreach (var obj in smear.BrushStroke.Objects)

{

obj.Points.ForEach(d => d.Pixels[Layers.Filtered] = Pixel.CreateInstance(smear.Cluster.Centroid));

data.AddRange(obj.Points);

}

}

return Model.ConvertToBitmapSource(data, Layers.Filtered);

}

private BitmapSource Segments()

{

Progress.NewProgress("Вычисление сегментов");

var data = new PointCollection();

foreach (var smear in \_smears)

{

foreach (var obj in smear.BrushStroke.Objects)

{

obj.Points.ForEach(d => d.Pixels[Layers.Filtered] = Pixel.CreateInstance(obj.GetCenter(Layers.Original).Data));

data.AddRange(obj.Points);

}

}

return Model.ConvertToBitmapSource(data, Layers.Filtered);

}

private BitmapSource BrushStrokes()

{

Progress.NewProgress("Вычисление мазков");

var data = new PointCollection();

foreach (var smear in \_smears)

{

var objs = smear.BrushStroke.Objects;

var center = objs.OrderBy(p => p.GetCenter(Layers.Original).Sum).ToList()[objs.Count / 2].GetCenter(Layers.Original).Data;

foreach (var obj in objs)

{

var pointsClone = obj.Points.Clone();

pointsClone.ForEach(d => d.Pixels[Layers.Filtered] = Pixel.CreateInstance(center));

data.AddRange(pointsClone);

}

}

return Model.ConvertToBitmapSource(data, Layers.Filtered);

}

private BitmapSource RandomBrushStrokes()

{

Progress.NewProgress("Вычисление мазков(рандом)");

var data = new PointCollection();

foreach (var smear in \_smears)

{

var objs = smear.BrushStroke.Objects;

var color = Utils.GetGandomData(3);

color.Add(255);

foreach (var obj in objs)

{

var pointsClone = obj.Points.Clone();

pointsClone.ForEach(d => d.Pixels[Layers.Filtered] = Pixel.CreateInstance(color.ToArray()));

data.AddRange(pointsClone);

}

}

return Model.ConvertToBitmapSource(data, Layers.Filtered);

}

private BitmapSource BrushStrokesLines()

{

Progress.NewProgress("Вычисление мазков (линии)");

return ImageHelper.PaintStrokes(Model.Image, \_smears.Select(s => s.BrushStroke), ((int)\_settings.MinSizeSuperpixel.Value) / 20 + 1);

}

private BitmapSource BrushStrokesPlt()

{

Progress.NewProgress("Вычисление мазков");

return ImageHelper.PaintStrokes(Model.Image, \_smears.Select(s => s.BrushStroke), (float)\_settings.WidthSmear.Value);

}

public override string CreatePlt()

{

return PltHelper.GetPlt(\_smears.Select(s=>s.BrushStroke).ToList(), \_settings.HeightPlt.Value, \_settings.WidthPlt.Value, \_settings.WidthSmear.Value, Model.Height, Model.Width);

}

public override void Dispose()

{

\_smears?.Clear();

base.Dispose();

}

}

}

using SmearsMaker.ImageProcessing;

namespace SmearsMaker.Tracers

{

public interface IServicesFactory

{

IStrokesBuilder CreateStrokesBuilder();

ISplitter CreateSplitter();

IClusterizer CreateClusterizer();

IFilter CreateFilter();

IDetector CreateDetector();

}

}

using System;

using System.Collections.Generic;

using System.Configuration;

using System.Data;

using System.Linq;

using System.Threading.Tasks;

using System.Windows;

namespace SmearsMaker.Wpf

{

/// <summary>

/// Interaction logic for App.xaml

/// </summary>

public partial class App : Application

{

}

}

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Linq;

using System.Runtime.CompilerServices;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using NLog;

using SmearsMaker.Common;

using SmearsMaker.Common.Image;

using SmearsMaker.HPGL;

namespace SmearsMaker.Wpf

{

public class ApplicationViewModel : INotifyPropertyChanged

{

public List<Type> Tracers { get; }

public event PropertyChangedEventHandler PropertyChanged;

public List<ImageSetting> Settings { get; set; }

public ICommand OpenImage { get; }

public ICommand SavePlt { get; }

public ICommand SaveImages { get; }

public string Label

{

get => \_label;

set

{

\_label = value;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nameof(Label)));

}

}

public ImageSource CurrentImage

{

get => \_currentImage;

set

{

\_currentImage = value;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nameof(CurrentImage)));

}

}

private string \_label;

private ImageSource \_currentImage;

private readonly List<ImageView> \_images;

private static readonly ILogger Log = LogManager.GetCurrentClassLogger();

private int \_currentImageIndex;

private ITracer \_tracer;

private BitmapImage \_image;

private readonly PltReaderImpl \_readerImpl;

public ApplicationViewModel()

{

OpenImage = new Command(OpenFile);

SavePlt = new Command(SavePtl);

SaveImages = new Command(SaveImagesInFolder);

\_images = new List<ImageView>();

\_currentImageIndex = 0;

Label = "Выберите изображение";

Tracers = FileHelper.LoadLibraries();

\_readerImpl = new PltReaderImpl();

}

public void SetAlgorithm(Type tracer)

{

if (\_image == null || tracer == null) return;

\_tracer?.Dispose();

\_tracer = Activator.CreateInstance(tracer, (BitmapSource)\_image) as ITracer;

\_tracer.Progress.UpdateProgress += UpdateProgress;

Settings = \_tracer.Settings;

}

public async Task Run()

{

try

{

if (\_image == null) return;

if (\_tracer == null && Tracers.Any())

{

SetAlgorithm(Tracers.First());

}

\_images.Clear();

await \_tracer.Execute();

\_images.AddRange(\_tracer.Views);

if (\_images.Count < \_currentImageIndex)

{

\_currentImageIndex = \_images.Count - 1;

}

Label = \_images[\_currentImageIndex].Name;

CurrentImage = \_images[\_currentImageIndex].Source;

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка! {ex.Message}");

Log.Error(ex);

}

}

private void SavePtl()

{

if (\_tracer == null) return;

try

{

var plt = \_tracer.CreatePlt();

FileHelper.SavePlt(plt);

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка! {ex.Message}");

}

}

public void ChangeImage(KeyEventArgs e)

{

if (\_images.Count == 0) return;

switch (e.Key)

{

case Key.Right:

\_currentImageIndex = \_currentImageIndex < \_images.Count - 1 ? \_currentImageIndex + 1 : 0;

break;

case Key.Left:

\_currentImageIndex = \_currentImageIndex > 0 ? \_currentImageIndex - 1 : \_images.Count - 1;

break;

default:

return;

}

CurrentImage = \_images[\_currentImageIndex].Source;

Label = \_images[\_currentImageIndex].Name;

}

public void OpenPlt()

{

var name = FileHelper.OpenPlt();

if (name != null)

{

CurrentImage = \_readerImpl.ReadPlt(name);

}

}

private void UpdateProgress(object sender, ProgressBarEventArgs args)

{

Label = args.Percentage != 0 ? $"{args.Msg} {args.Percentage}%" : args.Msg;

}

private void OpenFile()

{

var imageUri = FileHelper.OpenImage();

if (imageUri == null) return;

\_image = new BitmapImage(imageUri);

CurrentImage = \_image;

Label = "Нажмите кнопку старт";

if (\_tracer != null)

{

SetAlgorithm(\_tracer.GetType());

}

}

private void SaveImagesInFolder()

{

var path = FileHelper.GetFolder();

foreach (var image in \_images)

{

FileHelper.SaveBitmapSource(path, image.Name, image.Source);

}

}

protected virtual void OnPropertyChanged([CallerMemberName] string prop = null)

{

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(prop));

}

}

}

using System;

using System.Windows.Input;

namespace SmearsMaker.Wpf

{

public class Command : ICommand

{

protected Action Action;

private readonly bool \_canExecute;

public Command(Action action, bool canExecute = true)

{

// Set the action.

this.Action = action;

this.\_canExecute = canExecute;

}

bool ICommand.CanExecute(object parameter)

{

return \_canExecute;

}

public void Execute(object parameter)

{

Action?.Invoke();

}

public event EventHandler CanExecuteChanged;

}

}

using System;

using System.Collections.Generic;

using System.Configuration;

using System.IO;

using System.Linq;

using System.Reflection;

using System.Text;

using System.Windows.Media.Imaging;

using Microsoft.Win32;

using SmearsMaker.Common;

namespace SmearsMaker.Wpf

{

public static class FileHelper

{

internal static List<Type> LoadLibraries()

{

var libs = new List<Type>();

var tracerLibs = ConfigurationManager.AppSettings;

foreach (var tracerLib in tracerLibs.AllKeys)

{

var value = tracerLibs.Get(tracerLib);

var assemblyFolder = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);

if (assemblyFolder == null) continue;

var assemblyPath = Path.Combine(assemblyFolder, value);

if (!File.Exists(assemblyPath)) continue;

var assembly = Assembly.LoadFrom(assemblyPath);

var tracers = assembly.GetTypes().Where(t => typeof(ITracer).IsAssignableFrom(t) && !t.IsAbstract).ToList();

libs.AddRange(tracers);

}

return libs;

}

internal static string GetFolder()

{

var sf = new SaveFileDialog

{

FileName = "select folder"

};

return sf.ShowDialog() != true ? null : Path.GetDirectoryName(sf.FileName);

}

internal static void SaveBitmapSource(string path, string name, BitmapSource source)

{

if (path == null) return;

var encoder = new PngBitmapEncoder();

encoder.Frames.Add(BitmapFrame.Create(source));

using (var filestream = new FileStream(Path.Combine(path, $"{name}.bmp"), FileMode.Create))

{

encoder.Save(filestream);

}

}

internal static Uri OpenImage()

{

//считывание с файла

var fileDialog = new OpenFileDialog

{

Filter =

"JPG Files (\*.jpg)|\*.jpg|bmp files (\*.bmp)|\*.bmp|JPEG Files (\*.jpeg)|\*.jpeg|PNG Files (\*.png)" +

"|\*.png|GIF Files (\*.gif)|\*.gif|All files (\*.\*)|\*.\*",

RestoreDirectory = true

};

return fileDialog.ShowDialog() == true ? new Uri(fileDialog.FileName) : null;

}

internal static string OpenPlt()

{

//считывание с файла

var fileDialog = new OpenFileDialog

{

Filter =

"Plt Files (\*.plt)|\*.plt|All files (\*.\*)|\*.\*",

RestoreDirectory = true

};

return fileDialog.ShowDialog() != true ? null : fileDialog.FileName;

}

internal static void SavePlt(string plt)

{

var fileDialog = new SaveFileDialog

{

FileName = "pltFile",

DefaultExt = ".plt",

Filter = "Plt Files (\*.plt)|\*.plt|All files (\*.\*)|\*.\*",

RestoreDirectory = true

};

if (fileDialog.ShowDialog() == true)

{

File.WriteAllText(fileDialog.FileName, plt, Encoding.ASCII);

}

}

}

}

<Window x:Class="SmearsMaker.Wpf.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:SmearsMaker.Wpf"

mc:Ignorable="d"

Title="SmearTracer" Height="650" Width="600" WindowStartupLocation="CenterScreen" KeyUp="Window\_KeyUp">

<Grid x:Name="MainGrid">

<Grid.RowDefinitions>

<RowDefinition Height="20"></RowDefinition>

<RowDefinition Height="\*"></RowDefinition>

<RowDefinition Height="30"></RowDefinition>

</Grid.RowDefinitions>

<Menu Grid.Row="0">

<MenuItem Header="File">

<MenuItem Header="Open" Command="{Binding OpenImage}"></MenuItem>

<MenuItem Header="Open plt" Click="MenuItem\_Click" ></MenuItem>

<MenuItem Header="Save images" Command="{Binding SaveImages}"></MenuItem>

<MenuItem Header="Save plt" Command="{Binding SavePlt}"></MenuItem>

<Separator></Separator>

<MenuItem x:Name="ExitMenuItem" Header="Exit" Click="ExitMenuItem\_Click"></MenuItem>

</MenuItem>

<MenuItem Header="Edit">

<MenuItem x:Name="ClipboardCopyMenuItem" Header="Copy" Click="ClipboardCopyMenuItem\_Click"></MenuItem>

<Separator></Separator>

<MenuItem x:Name="SettingsMenuItem" Header="Settings" Click="SettingsMenuItem\_Click"></MenuItem>

</MenuItem>

<MenuItem x:Name="Algorithms" Header="Algorithms" Click="TracerMenuItem\_Click">

<MenuItem.ItemContainerStyle>

<Style TargetType="{x:Type MenuItem}">

<Setter Property="Header" Value="{Binding Name}"/>

</Style>

</MenuItem.ItemContainerStyle>

</MenuItem>

</Menu>

<Image Grid.Row="1" HorizontalAlignment="Center" Margin="5,5,5,30" VerticalAlignment="Top" x:Name="Image" Stretch="Uniform" Source="{Binding CurrentImage}"/>

<Button Grid.Row="2" x:Name="ButtonRun" Content="Старт" Margin="0,0,0,5" HorizontalAlignment="Center" VerticalAlignment="Bottom" Width="76" Click="ButtonRun\_Click" />

<Label Grid.Row="2" Name="LabelStatus" HorizontalAlignment="Right" VerticalAlignment="Bottom" Content="{Binding Label}"/>

</Grid>

</Window>

using System;

using System.Collections.Generic;

using System.IO;

using System.Text;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

using Microsoft.Win32;

namespace SmearsMaker.Wpf

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

private readonly ApplicationViewModel \_model;

public MainWindow()

{

InitializeComponent();

\_model = new ApplicationViewModel();

DataContext = \_model;

Algorithms.ItemsSource = \_model.Tracers;

}

private async void ButtonRun\_Click(object sender, RoutedEventArgs e)

{

try

{

await \_model.Run();

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

private void Window\_KeyUp(object sender, KeyEventArgs e)

{

\_model.ChangeImage(e);

}

private void ExitMenuItem\_Click(object sender, RoutedEventArgs e)

{

Close();

}

private void SettingsMenuItem\_Click(object sender, RoutedEventArgs e)

{

var settingsForm = new SettingsWindow(\_model.Settings);

settingsForm.Show();

}

private void TracerMenuItem\_Click(object sender, RoutedEventArgs e)

{

var tracer = (e.OriginalSource as MenuItem)?.DataContext as Type;

\_model.SetAlgorithm(tracer);

}

private void ClipboardCopyMenuItem\_Click(object sender, RoutedEventArgs e)

{

if (Image.Source != null)

{

Clipboard.SetImage((BitmapSource)Image.Source);

}

}

private void MenuItem\_Click(object sender, RoutedEventArgs e)

{

try

{

\_model.OpenPlt();

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

}

}

<Window x:Class="SmearsMaker.Wpf.SettingsWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:SmearsMaker.Wpf"

mc:Ignorable="d"

Title="SettingsWindow" Height="400" Width="400">

<Grid Name="SettingsGrid">

<Grid.RowDefinitions>

<RowDefinition Height="10\*"></RowDefinition>

<RowDefinition Height="\*"></RowDefinition>

</Grid.RowDefinitions>

<ListBox Grid.Row="0" Name="lstSettings" Margin="5" SelectionMode="Single">

<ListBox.ItemContainerStyle>

<Style TargetType="{x:Type ListBoxItem}">

<Setter Property="Margin" Value="3"/>

<Setter Property="Padding" Value="3"/>

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="{x:Type ListBoxItem}">

<StackPanel Orientation="Horizontal">

<Label Content="{Binding Name}" Width="170"/>

<TextBox Text="{Binding Value}" Width="190"/>

</StackPanel>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

</ListBox.ItemContainerStyle>

</ListBox>

<Button Grid.Row="1" Width="76" Height="23" Margin="0,0,0,5" VerticalAlignment="Bottom" Content="Ок" Click="Button\_Click" ></Button>

</Grid>

</Window>

using System.Collections.Generic;

using System.Windows;

using SmearsMaker.Common.Image;

namespace SmearsMaker.Wpf

{

/// <summary>

/// Interaction logic for SettingsWindow.xaml

/// </summary>

public partial class SettingsWindow : Window

{

public SettingsWindow(IEnumerable<ImageSetting> settings)

{

InitializeComponent();

lstSettings.ItemsSource = settings;

}

private void Button\_Click(object sender, RoutedEventArgs e)

{

Close();

}

}

}