**«Санкт-Петербургский государственный электротехнический университет**

**«ЛЭТИ» им. В.И.Ульянова (Ленина)»**

**(СПбГЭТУ “ЛЭТИ”)**

|  |  |  |
| --- | --- | --- |
| **Направление** | 09.03.01 - Информатика и вычислительная техника | |
| **Профиль** | Системы автоматизированного проектирования | |
| **Факультет** | КТИ | |
| **Кафедра** | САПР | |
| *К защите допустить* |  | |
| Зав. кафедрой |  | Рыжов Н.Г. |

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**БАКАЛАВРА**

**Тема: Программа разбиения изображения на колористические примитивы для машинной живописи**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент |  |  |  | Мезенцев Р.С. |
|  |  | *подпись* |  |  |
| Руководитель | ассистент |  |  | Каримов А.И. |
|  | *(Уч. степень, уч. звание)* | *подпись* |  |  |
| Консультанты | к.т.н., доцент |  |  | Иванов А.Н. |
|  | *(Уч. степень, уч. звание)* | *подпись* |  |  |

Санкт-Петербург

2016

**ЗАДАНИЕ**

**на выпускную квалификационную работу**

|  |  |
| --- | --- |
|  | Утверждаю |
|  | Зав. кафедрой САПР |
|  | Рыжов Н.Г. |
|  | «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Студент | МезенцевР.С. | | | |  | Группа | 2302 | |
| Тема работы: Программа разбиения изображения на колористические примитивы для машинной живописи | | | | | | | | |
| Место выполнения ВКР: кафедра САПР, СПбГЭТУ | | | | | | | | |
| Исходные данные (технические требования):  Спецификация робота, техническое задание на программный модуль, текстовое описание технологии живописи на холсте. | | | | | | | | |
| Перечень отчетных материалов: пояснительная записка, иллюстративный материал | | | | | | | | |
| Дополнительные разделы: Аспекты эргономики программного обеспечения | | | | | | | | |
|  | | | | | | | | |
| Дата выдачи задания | | | Дата представления ВКР к защите | | | | | |
| «01»\_февраля 2016 г. | | | «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. | | | | | |
|  | | |  | | | | | |
| Студент | |  | | Мезенцев Р.С. | | | |
| Руководитель | |  | | Каримов А.И. | | | |
|  | |  | |  | | | |

**календарный планвыполнения**

**выпускной квалификационной работы**

|  |  |
| --- | --- |
|  | Утверждаю |
|  | Зав. кафедрой САПР |
|  | Рыжов Н.Г. |
|  | «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | Мезенцев Р.С. |  | Группа | 2302 |
| Тема работы: Программа разбиения изображения на колористические примитивы для машинной живописи | | | | |

|  |  |  |
| --- | --- | --- |
| № п/п | Наименование работ | Срок выполнения |
| 1 | Обзор литературы по теме работы | 01.02 – 01.03 |
| 2 | Выбор технологий реализации | 01.03 – 07.03 |
| 3 | Определение архитектуры приложения | 07.03 – 14.03 |
| 4 | Программная реализация | 14.03 – 01.05 |
| 5 | Оформление пояснительной записки | 01.05 – 20.05 |
| 6 | Оформление иллюстративного материала | 20.05 – 30.05 |

|  |  |  |
| --- | --- | --- |
| Студент |  | Мезенцев Р.С. |
| Руководитель |  | КаримовА.И. |

СОДЕРЖАНИЕ

[ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ 5](#_Toc452764859)

[ВВЕДЕНИЕ 6](#_Toc452764860)

[1. МАТЕМАТИЧЕСКОЕ ОПИСАНИЕ АЛГОРИТМОВ ОБРАБОТКИ ИЗОБРАЖЕНИЙ 8](#_Toc452764861)

[1.1. Алгоритмы фильтрации изображений 8](#_Toc452764862)

[1.2. Алгоритмы кластеризации изображений 11](#_Toc452764863)

[1.3. Алгоритмы сегментации изображений 14](#_Toc452764864)

[1.4. Алгоритмы преобразования изображения в карту мазков 20](#_Toc452764865)

[1.5. Алгоритм преобразования изображения в карту мазков на основе кластеризации 23](#_Toc452764866)

[2. ОПИСАНИЕ ПРОГРАММНЫХ СРЕДСТВ И ТЕХНОЛОГИЙ 25](#_Toc452764867)

[2.1. Язык разработки программного модуля 25](#_Toc452764868)

[2.2. Среда разработки программного модуля 28](#_Toc452764869)

[3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ 31](#_Toc452764870)

[3.1. Общая структура программы 31](#_Toc452764871)

[3.2. Описание объектов 32](#_Toc452764872)

[3.3. Статические классы 36](#_Toc452764873)

[3.4. Разработка пользовательского интерфейса 45](#_Toc452764874)

[3.5. Тестирование разработанной подсистемы 49](#_Toc452764875)

[4. СПЕЦИАЛЬНЫЕ ВОПРОСЫ ОБЕСПЕЧЕНИЯ БЕЗОПАСНОСТИ 54](#_Toc452764876)

[ЗАКЛЮЧЕНИЕ 57](#_Toc452764877)

[СПИСОК ЛИТЕРАТУРЫ 59](#_Toc452764878)

[ПРИЛОЖЕНИЕ А 60](#_Toc452764879)

# ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ПО – Программное обеспечение.

EM (Expectation-Maximization) – метод поиска оценок функции правдоподобия в моделях со скрытыми переменными.

GUI (GraphicalUserInterface) – графический интерфейс пользователя.

VS (VisualStudio) – это интегрированная среда разработки, основанная на использовании компонентов и других технологий, служит для разработки и совместного проектирования мощных приложений.

# ВВЕДЕНИЕ

Робототехника является одним из важнейших направлений научно-технического прогресса. В настоящее время происходит повсеместное внедрение роботов, многие процессы автоматизируются. Современные роботы используются во всех отраслях – промышленности, медицине, в быту и даже в искусстве.

**Целью** выпускной работы является разработка и написание алгоритма преобразования изображения в карту мазков (колористических примитивов), связанных с движением кистью робота-живописца. Для достижения цели ставятся **задачи**:

1) разработка алгоритма уменьшения глубины цвета без ощутимой для человека-зрителя потери качества;

2) разработка алгоритма определения границ зон одного цвета;

3) разработка алгоритма преобразования одноцветных областей в карту движений кистью;

4) программная реализация разработанных алгоритмов.

На сегодняшний день существует огромное количество алгоритмов, связанных с обработкой изображений. Алгоритмы фильтрации, кластеризации, сегментации тесно связаны между собой, но при этом каждый из них имеет свою определенную область применения и зависит от входных данных, поэтому необходимо найти такие алгоритмы, которые будут решать поставленные задачи лучше остальных. Таким образом, необходимо провести анализ алгоритмов и на их основе разработать программный компонент для разбиения изображения на колористические примитивы для машинной живописи.

В первой главе данной работы рассматривается математическое описание алгоритмов, участвующих в работе модуля, а также сравнение их с другими алгоритмами.

Вовторой главе описываются программные средства и технологии, которые необходимы для реализации программного модуля.

В третей главе приводится описание программной реализации, описывается архитектура приложения, внутренние объекты и их взаимодействие между собой, а также производится тестирование пользовательского приложения.

В четвертой главе рассматриваются аспекты эргономики проектирования и разработки графического интерфейса пользователя.

# МАТЕМАТИЧЕСКОЕ ОПИСАНИЕ АЛГОРИТМОВ ОБРАБОТКИ ИЗОБРАЖЕНИЙ

# Алгоритмы фильтрации изображений

Первостепенной задачей системы фильтрации является улучшение качества изображения. Проблема шумоподавления является очень актуальной и распространенной проблемой в области обработки изображений. Самыми распространенными видами шумов являются Гауссов и импульсный шумы, а также их комбинация (см. рис. 1.1).

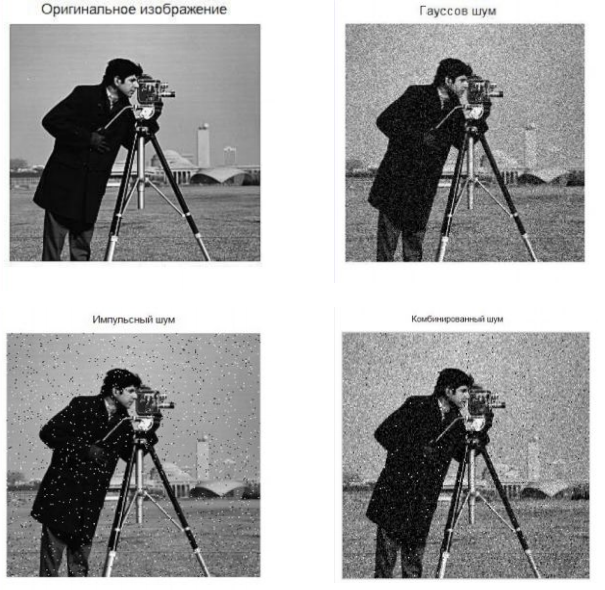


Рисунок 1.1 – Виды шума

Алгоритмы шумоподавления обычно используются для снижения какого-то конкретного вида шума. Пока что не существует универсальных фильтров, которые подавляют и детектируют все виды шумов, но многие шумы можно приблизить моделью белого Гауссова шума, именно поэтому большинство алгоритмов ориентировано на подавление этого вида шума. Самые распространенные методы удаления шумов:

1. сглаживающие фильтры;
2. фильтры Винера;
3. медианные фильтры;
4. ранжирующие фильтры.

Для подавления Гауссова шума используются как линейные, так и нелинейные фильтры. Линейный фильтр определяется ядром фильтра, заданным на растровом изображении. Сама фильтрация производится при помощи операции дискретной свертки (взвешенного суммирования).

Потенциально лучшие результаты обработки изображения в результатах фильтрации достигаются при использовании фильтра Винера [4]. Его использование связано с предположением о том, что изображение стационарно. Так как наличие краев изображения служит нарушением его стационарности, то эта фильтрация не является строго оптимальной. Но при размерах кадра, превышающих промежуток корреляции изображения, влияние границ является небольшим. Технически виннеровский фильтр реализуется с помощью использования дискретного преобразования Фурье в частотной области.

Но применениелинейных методов фильтрации не дает возможность получить приемлемое решение в широком спектре задач. Необходимо принимать во внимание нелинейный характер процессов кодирования, передачи и восприятия информации, например, канала связи, датчиков информации, зрительной системы человека и т.п.

В отличие от теории линейных методов фильтрации, определение единой теории нелинейной фильтрации вряд ли возможно. Любой из перечисленных классов имеет свои преимущества и свою область применения. Так, например, известно, что самые лучшие результаты для сохранения перепадов (резких переходов) оттенков, локальных пиков яркости, и различных границ на искаженных импульсным шумом изображениях может дать использование медианной фильтрации [2].

Медианный фильтр в отличие от обычного сглаживающего фильтра использует нелинейную процедуру подавления шумов. Медианный фильтр представляет из себя окно *w* (апертура фильтра), охватывающее нечетное число отсчетов. Центральный отсчет заменяется медианным значением всех элементов изображения, которые попали в окно. Медианным значением дискретной последовательности является средний по порядку член ряда, образующийся при упорядочении исходной последовательности. Медианный фильтр, как и сглаживающий, используется для подавления импульсного и аддитивного шумов на изображениях. Характерной особенностью медианного фильтра можно назвать сохранение перепадов яркости (см. рис.1.2).

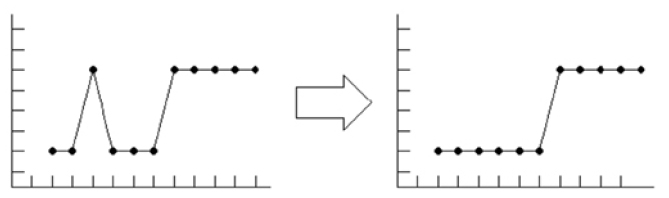


Рисунок 1.2 – Сохранение перепадов

Если перепады яркости больше дисперсии аддитивного белого шума, то медианная фильтрация дает гораздо лучшие результаты, чем оптимальный линейный фильтр. Наиболее эффективным медианный фильтр является в случае подавления импульсного шума.

Ранжирующий фильтр, как и сглаживающий, тоже использует для преобразования изображения маску. Маска может включать или не включать центральный пиксель. Значения элементов, попадающих в маску, можно расположить в упорядоченный ряд и упорядочить по убыванию или возрастанию, и вычислить некоторые моменты этого ряда, например, среднее значение дисперсии и интенсивности. Выходным значением фильтра, которым заменяется центральный отсчет, является взвешенная сумма интенсивности центрального пикселя и медианы получившегося ряда. Коэффициенты обычно связаны определенной зависимостью со статистикой пикселей в окне фильтра.

Из рассмотренных алгоритмов фильтрации наиболее подходящим для обработки изображения оказался медианный фильтр, так как он сохраняет перепады контуров.

# Алгоритмы кластеризации изображений

Кластеризация – деление множества входных векторов на кластеры (группы) по “степени схожести” друг на друга.

Чтобы сравнивать два любых объекта, необходимо иметь какой-то критерий, на основании которого можно произвести сравнение этих объектов. Таким критерием, как правило, является расстояние между этими объектами. Существует множество мер расстояния, рассмотрим некоторые из них:

1. Евклидово расстояние  – самое распространенное расстояние. Оно представляет собой геометрическое расстояние в многомерном пространстве;
2. Квадрат евклидова расстояния. Можно возвести в квадрат евклидово расстояние, чтобы придать большие веса для отдаленных друг от друга объектов;
3. Расстояние Чебышева. Это расстояние является полезным, когда необходимо определить два объекта как «различные», при условии, что они различаются по какой-либо координате (или каким-либо одним измерением);
4. Расстояние городских кварталов (манхэттенское расстояние). Данное расстояние выглядит как просто среднее разностей по всем координатам. Как правило, эта мера расстояния приводит к схожим результатам, как и для расстояния Евклида. Однако стоит отметить, что для нее влияние отдельных больших разностей уменьшается (так как онипросто не возводятся в квадрат);
5. Степенное расстояние. Иногда можно прогрессивно уменьшить или увеличить вес, который относится к размерности, для которой имеющиеся объекты сильно отличаются друг от друга. Это может быть реализованос помощью степенного расстояния.

Выбор критерия схожести (расстояния) лежит полностью на самом исследователе. При различных мерах результаты кластеризации будут, как правило, существенно отличаться.

k-means (метод k-средних) – самый популярный и очень простой метод кластеризации. Его особенностью является минимизация суммарного квадратичного отклонения точек кластеров от центров этих кластеров:

, (1)



где - число кластеров, - полученные кластеры, и - центры масс векторов.

Алгоритм представляет из себя версию EM-алгоритма, возможностью которого является применение его для разделения смеси гауссиан. Он разбивает множество элементов векторного пространства на заранее известное число кластеров k.

Главная идея этого алгоритма заключается в том, что на каждой итерации выполнения пересчитывается центр масс каждого кластера, который был получен на предыдущем шаге, а после векторы снова разбиваются на кластеры в соответствии с тем, какой из новых центров масс оказался ближе к точке по указанной метрике.

Алгоритм заканчивает свою работу, если на какой-то итерации не изменяется центр масс кластеров. Это происходит за определенное число итераций, так как количество всех возможных разбиений конечного множества не бесконечно, а на каждом шаге суммарное квадратичное отклонение не увеличивается, и поэтому зацикливание просто невозможно.

Проблемы алгоритма:

1. этот алгоритм очень чувствительный к шуму, который в свою очередь может существенно исказить результаты кластеризации;
2. нет гарантии достижения глобального минимума суммарного квадратичного отклонения V, а гарантируется достижение только локального минимума;
3. результат сильно зависит от выбора исходных центров масс кластеров, однако их оптимальный выбор является неизвестным;
4. количество кластеров нужно знать заранее;

k-means++ – это более совершенная версия алгоритма кластеризации к-средней. Его идея заключается в поискенаиболее “хороших” начальных значений центров масс кластеров. Обычный k-means не учитывает выполнение этого этапа, и поэтому он является нестабильным.

Шаги алгоритма:

1. Выбрать первый центр массслучайно (среди всех точек).
2. Для всех точек найти квадрат расстояния до ближайшего центра масс (только из тех, которые уже выбраны) dx².
3. Выбрать из этих точек следующий центр масс такой, чтобы вероятность выбора точки была пропорциональной вычисленному для неё квадрату расстояния. При выборе каждого следующего центра масс необходимо следить за тем, чтобы он не совпадал с одной из уже выбранных в качестве центров масс точек, нет необходимости, так как вероятность повторного выбора некоторой точки практически равна 0.
4. Выполнять шаги 2 и 3 до тех пор, пока не будут найдены все центры масс.

Затем выполняется обычный алгоритм K-Means.

Еще одним примером решения проблемы нестабильности k-means является алгоритм с-means. Вместо однозначного определения принадлежности к кластеру объекта, он определяет вероятность принадлежности объекта к тому или иному кластеру. Другими словами, утверждение“объект А принадлежит к кластеру 1 с вероятностью 60%, к кластеру 2-40%” верно и гораздо более удобно. Однако остальные проблемы у с-means точно такие же, как и у k-means, но стоит отметить, что все они нивелируются благодаря нечеткости разбиения.

Из рассмотренных алгоритмов кластеризации для решения поставленной задачи наиболее подходящим оказался алгоритм K-means, ввиду простоты реализации и высокой эффективности.

# Алгоритмы сегментации изображений

Сегментация изображения – это разделение изображения на определенные области (сегменты), однородные по некоторому критерию. Цель сегментации упростить представление изображения, чтобы его было легче и проще анализировать. Существуют различные варианты разбиения:

1. сегментация на неперекрывающиеся области;
2. сегментация на перекрывающиеся области;
3. иерархическое разбиение (все фрагменты одного уровня разбиваются на несколько сегментов).

Учитывая то, что разные области одного и того же объекта существенно отличаются друг от друга по признакам, и единственное общее у них – это семантика, можно сказать, что задача сегментации очень похода на задачу распознавания. Выделение границ с помощью различных алгоритмов не дают достаточно точную информацию о границах и областях, так как на сегментацию влияет много различных факторов. Сегментация является как правило частью некоторой системы, а значит качество работы необходимо оценивать исходя из работы системы целиком, так как один и тот же алгоритм сегментации может решать разные задачи с разной степенью эффективности.

Часто алгоритмы классифицируют следующим образом:

1. пороговые алгоритмы;
2. алгоритмы наращивания областей;
3. граничные алгоритмы;
4. алгоритмы сегментации на основе кластеризации.

Если нужно распознать изображение, области внутри которого сильно различаются по яркости пикселей, необходимо применять пороговые алгоритмы, например, для сегментации пороговым алгоритмом нужно получить бинарное изображение из полутонового. Для этого необходимо установить пороговое ограничение. После этого функция изображения отображает элементы изображения с уровнем яркости, который выше порогового, в значение 1, в противном случае в значение 0. В этих случаях оптимальное значение порога определяется просто при помощи анализа гистограмм. Следующим шагом является мультипороговая сегментация.

В случае отсутствия шума и других возможных возмущений, изменения яркости происходит непосредственно на границах выделяемой области. Эта модель может быть упрощена до единой размерности, и тогда распределение яркости будет однозначно совпадать с формой контура границы. По сути, распределение яркости является первой производной от контура границ.

Данные алгоритмы могут сегментировать простые изображения, но они не дают хорошего результата на изображении с наличием шумов, помех, теней, что является большим минусом. Также если существуют большие интервалы между экстремумами, то определение значимого минимума будет затруднительным. Для уменьшения влияния этих недостатков разработаны целые группы алгоритмов, которые выполняют анализ взвешенных значений экстремумов.

В общем случае, для корректного применения алгоритмов данной группы, необходимо избегать смещения при определении порогового значения с помощью ужесточения контроля одинаковости распределения в светлых и темных областях гистограммы яркости, а также нужно делить изображение на малые части, чтобы гистограмма яркостей имела ярко выраженные экстремумы, эти части должны быть также достаточно большими, чтобы была возможность оценить местоположения экстремумов и описать окрестность.

Если на изображении присутствует устойчивая связность внутри различных сегментов, то стоит использовать алгоритмы наращивания областей. С их помощью происходит объединение граничных элементов с одинаковыми или очень близкими уровнями яркости в однородные области. Самыми распространенными алгоритмами нахождения сегментов включают в себя алгоритмы расщепления и слияния областей, алгоритмы водоразделов и алгоритмы центроидного связывания.

Самыми примитивными способами наращивания областей является использование совокупности свойств разных групп точек для определения простых областей. Более сложные варианты алгоритмов строятся на объединении простых областей.

При использовании центроидного связывания при помощи информации об объекте выбираются начальные точки, которым присваиваются разные метки, точки с одинаковыми метками образуют отдельные наборы. Этот алгоритм пригоден для сегментации примитивных изображений. Для сложных изображений необходимо точки выбирать по итерациям, на каждой итерации рассматривается набор точек на принадлежность их соседей данному множеству. Точки, которые включены в множество на предыдущих итерациях, не рассматриваются. Таким образом проводится анализ всех множеств. Точки, которые были добавлены к множеству на этой текущей итерации, называются фронтом, а объединение фронтов называется волной, именно поэтому такие алгоритмы называются волновыми.

Алгоритм слияния и расщепления делит точки путем деления некоторым образом изображение на квадратные области, которые в дальнейшем анализируются для их проверки на однородную яркость. Если квадратная область не удовлетворяет этим условиям, то она заменяется четырьмя меньшими областями, а подходящие по условию однородности 4 квадратные области могут быть заменены одной областью. Из недостатков можно выделить то, что эти алгоритмы выделяют общие фрагменты, при этом часто не показывая, информацию об изменениях яркостей внутри области и возможных границах, но несмотря на это они зарекомендовали себя как более эффективные применительно к зашумленным изображениям.

Еще одним способом сегментации являются граничные методы. Их особенностью является поиск точек, лежащих на границах областей. Основной характеристикой для определения границ является яркость, но также часто используются другие характеристики, так как градиент и текстура.

Границей называются сильные изменения яркости на изображении, границей можно назвать резкое изменение первой производной яркости изображения. Такие переходы могут быть как пошаговыми, в случае резких изменений значений вплоть до противоположных значений, так и линейными, в этом случае значение яркости тоже резко меняется, но через очень короткий промежуток возвращается на место. Последние ситуации встречаются редко вследствие сглаживания в изображениях, получаемых при использовании любых устройств работы с изображениями. Поэтому при рассмотрении изменений в пределах конечного отрезка пошаговые изменения фактически идентифицируются в качестве линейно возрастающих, а линейные представляются треугольными. Иногда встречаются случаи, когда удобно определять границу по обоим признакам сразу.

Фрагмент границы соответствует координатам границы и направленности границы, которая может быть градиентным углом. Контур состоит из набора границ или математических кривых, моделирующих набор границ. Существуют алгоритмы объединения границ, создающие упорядоченные наборы границ, они используют локальную информацию для определения принадлежности пикселя какой-либо границе, а также алгоритмы отслеживания границ, которые в свою очередь могут использовать глобальную информацию.

Эти алгоритмы состоят из фильтрации, усиления, выделения и определения местоположения и направления. Разработано множество таких алгоритмов, например, операторы Робертса, Собела, Превитта, Канни.

Существующие алгоритмы делятся на алгоритмы сравнения с эталоном и градиентно-дифференциальные алгоритмы. Каждый из них определяет, в каком случае колебание градиента яркости g становится чересчур большим, чтобы заявить, что здесь существует граница. Существенными отличиями этих алгоритмов являются способ оценки градиента и определение локального направления границы, что является важным аспектов в области распознавания объектов.

Для определения градиентов обе группы алгоритмов используют матрицы свертки. В случае сравнения с эталоном применяется до 12 масок свертки, которые оценивают локальные компоненты градиента.

При сравнении с эталоном изменение локального градиента границы аппроксимируется с помощью выбора наибольшего значения среди всех откликов компонентов сверток (2):

 (2)



где обычно *n* = 8 или *n* = 12.

При дифференциально-градиентном подходе локальное значение границ можно рассчитать с помощью векторов, используя нелинейное преобразование (3)

 . (3)

При сравнении с эталоном направление границ может оцениваться как маска, дающая самое большое число градиента в уравнении. В дифференциально-градиентном подходе уравнением (4)

 . (4)

Последние два способа имеют высокую вычислительную сложность, но при этом являются очень точными. Однако в некоторых ситуациях алгоритм сравнения с эталонами считается более предпочтительным, например, когда информация о направленности не требуется. В целом, выделение границ позволяет хорошо интерпретировать изображения. Информацию о границах можно получать в разных масштабах, это дает возможность получать коррелированные результаты. Такие алгоритмы требуют меньше ресурсов, и информация о результатах занимает меньше места для хранения. Данные алгоритмы стоит использовать тогда, когда границы четкие и непрерывистые.

Из недостатков можно заявить, что большинство схем имеют высокую вычислительную сложность, и это не позволяет применять алгоритмы в реальном времени. Также стоит учитывать тот факт, что для точной оценки значения и направления границ нужно использовать различные маски, так как они не всегда будут давать достоверный результат.

Отдельно стоит упомянуть об алгоритмах сегментации, основанных на кластеризации. Примеры таких алгоритмов приведены в предыдущей главе. Недостатками этих алгоритмов является возможность сбоя в случае некорректного выбора параметров модели по отношению к классифицируемым данным, или если модель не содержит в должном количестве характеристики сегментов. Также во многих алгоритмах существуют погрешности, когда данные состоят из сегментов разных форм, плотностей и размеров. Существует алгоритм, который измеряет сходство двух кластеров посредством использования динамической модели. В процессе кластеризации объединение двух кластеров происходит только при условии внутренней связности и схожести классов между собой. Кроме этого, существуют алгоритмы, в которых используется жесткая кластеризация, а также кластеризация, в основе которой лежит нечеткая логика, статистика появления признаков, гистограмм, иерархической кластеризации, и другие.

Каждый из алгоритмов сегментации имеет определенную область применения, которая в первую очередь зависит от входных данных, требования для распознавания, вычислительных затрат и времени на принятие решений. Следовательно, для конкретной задачи нужно подбирать алгоритм, который соответствует выдвигаемым требованиям. Но опять же, рассмотренных алгоритмов сегментации недостаточно, и требуется анализ полученных областей, для выявления связей между отдельными фрагментами.

# Алгоритмы преобразования изображения в карту мазков

Интересный алгоритм был предложен Аараном Хертманом[6]. Он заключается в рисовании изображения с помощью сплайнов (см. рис. 1.3). Мазки выбираются таким образом, чтобы максимально соответствовать цветам в исходном изображении. Картина строится с помощью серии слоев, начиная с самого грубого эскиза, проведенной большой кистью. Эскиз закрашивается с постепенно уменьшающимися мазками, но только в тех местах, где эскиз отличается от исходного изображения. Таким образом, демонстрируется техника рисования длинными, изогнутыми мазками, выровненными по нормалям градиентов изображения, в результате получаются выразительные и сложные мазки.

Вместо того, чтобы обрабатывать изображение в одном стиле, в данной работе представлена основа для описания широкого спектра визуальных стилей. Стиль представляет собой интуитивный набор параметров для алгоритма закраски, что позволяет дизайнеру вирировать стиль живописи по своему усмотрению. В качестве демонстрации работы алгоритма приведена визуализация нескольких шагов (см. рис. 1.3). Мазок начинается в контрольной точке и продолжается в направлении, перпендикулярном относительно направления градиента.

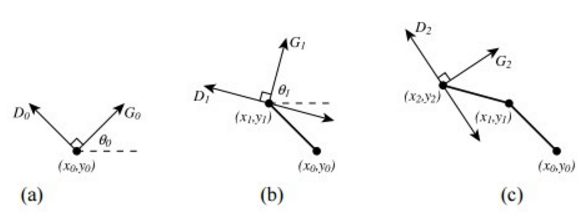


Рисунок 1.3–Создание мазка (по Хертману)

а – начало мазка; b– движение мазка; c – выбор направления.

Из второй точки мазка существует два перпендикулярных направления, выбираем направление, так как оно создает менее искривленную форму мазка. Процедура повторяется до тех пор, пока мазок не нарисован. Затем он будет отрисован с помощью B-сплайна с контрольных точек. Расстояние между контрольными точками равно радиусу кисти. Пиксели изображения представлены в виде точек с плавающими координатами. Контрольная точкадобавляется к сплайну, а цвет исходного изображения в этой точке используется как цвет сплайна. На следующем шаге происходит вычисление следующей точки вдоль заданной кривой. Направление градиента в точке вычисляется с помощью применения оператора Собеля к исходному изображению. Для примера рассмотрим первые три итерации алгоритма на изображении (см. рис. 1.4).



Рисунок 1.4 – Исходное изображение

Первая итерация алгоритма с использованием трех кистей с радиусом 8 (см. рис 1.5) сформировала первый слой будущей картины.



Рисунок 1.5 – Первый слой

На рис.1.6 продемонстрирован результат работы алгоритма с кистями радиуса 4, а финальный слой (см. рис. 1.7) был нарисован при использовании кистей с радиусом 2. Стоит отметить, что в алгоритме реализована возможность увеличивать и уменьшать кривизну мазка кисти путем фильтрации направления движения мазка.



Рисунок 1.6 – Второй слой

Также стоит обратить внимание, что мазки предыдущих слоев видны на финальной картине.



Рисунок 1.7– Третий слой

Из рассмотренного примера данного алгоритма можно сделать вывод, что алгоритм обладает хорошей точностью и высокой художественной выразительностью. Недостатки алгоритма: высокая вычислительная сложность, склонность порождать неоправданно длинные и изогнутые мазки.

# Алгоритм преобразования изображения в карту мазков на основе кластеризации

Выбранный в нашей программе алгоритм для составления карты мазков работает следующим образом.

1. Выбирается небольшое число кластеров, например, 4, и изображение разбивается на эти кластеры (см. рис. 1.8).

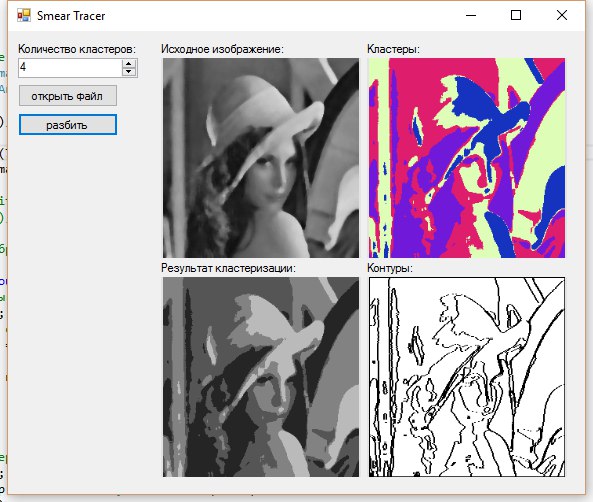


Рисунок 1.8 – Пример кластеризации

2. Если площадь кластера не превышает некоторый порог, связанный с шириной выбранной кисти, то перейти к п. 3, иначе к п.

3. Для кластеров с, которые могут быть отрисованы кистью заданной ширины, выбирается длина мазка и его направление. Мазок добавляется в карту нанесенных мазков. В текущей версии мазок представляет собой прямоугольник однородного цвета с наклоном, определяемым по границам кластера.

4. Полученное изображение с частично нанесенными мазками вычитается из исходного изображения (см. рис. 1.9). Те области, которые имеют ошибку по цвету меньше заданного порога, помечаются как законченные и в дальнейшем не обрабатываются.

5. Для всех оставшихся кластеров производится повторная кластеризация с тем же количеством кластеров, далее переход к п.2.



Рисунок 1.9 – Результат вычитания полученного и исходного изображений

Алгоритм завершается, когда погрешность ни в одной точке картины не превышает.

# ОПИСАНИЕ ПРОГРАММНЫХ СРЕДСТВ И ТЕХНОЛОГИЙ

# Язык разработки программного модуля

При планировании программного проекта имеется большой выбор языков программирования, а значит, при выборе языка необходимо учитывать множество факторов.

Python – очень простой язык программирования. Он многофункционален, его используют для web-разработки приложений, пользовательских интерфейсов, однако основная версия языка не позволяет писать многопоточные приложения. Основным его преимуществом является огромные возможности работы с большими наборами данных. Он поддерживает несколько парадигм программирования, в том числе структурное, объектно-ориентированное, императивное, аспектно-ориентированное и функциональное.

Преимущества языка Python:

1. Скорость выполнения написанных на нем программ высока. Это достигается за счет того, что его основные библиотеки написаны на C++ и выполнение задачи занимает меньше времени, чем на других языках высокого уровня.
2. Не надо заранее объявлять тип переменной (динамическая типизация).
3. К нему легко можно написать свой модуль и использовать в других программах.
4. Поддержка объектно-ориентированного программирования.
5. Автоматическая сборка мусора, отсутствие утечек памяти.
6. Кроссплатформенность.

Java владеет большинством достоинств Python и при этом избавлена от части его недостатков. Это объектно-ориентированный язык программирования, разработанный в SunMicrosystems. Java является одним из наиболее часто используемых языков программирования, он считается идеальным языком для web-разработчиков и программистов, он используется для создания мобильных и desktop приложений, а также в управленческих решениях для корпораций. Одним из важных преимуществ является его универсальность. Будучи компилируемым, язык может работать несколькими способами: либо в браузере, либо при запуске виртуальной машины, и такая гибкость играет очень важную роль, когда необходимо повторно использовать код или обновлять программное обеспечение.

Преимущества языка Java:

1. переносимость программы;
2. безопасность;
3. надежность;
4. сборщик мусора;
5. стандартные библиотеки;
6. автоматическая генерация документации;
7. многообразие типов приложений.

С++ является расширением языка С. Он представляет собой гибкий и мощный язык программирования, поддерживает такие парадигмы программирования, как объектно-ориентированное программирование, процедурное программирование, абстракцию данных, обобщенное программирование, обеспечивает модульность раздельную компиляцию, обработку исключений, виртуальные функции, и много другое. Он широко используется для разработки программного обеспечения, является одним из самых популярных языков программирования. Областью его применения являются создание операционных систем, множества прикладных программ, драйверов, приложений для встраиваемых систем, высокопроизводительных серверов, а также игр. Таким образом, его можно использовать для разработки приложений практически для любых целей и любой платформе. Однако его нельзя назвать современных средством программирования, он прошел длинный путь развития и на сегодняшний день существует множество его различных реализаций.

Его преимуществами являются:

1. поддержка различных стилей и технологий программирования;
2. предсказуемое выполнение программ;
3. автоматический вызов деструкторов объектов при их уничтожении;
4. использование шаблонов;
5. кроссплатформенность;
6. эффективность;
7. сочетание высокоуровневых и низкоуровневых средств;
8. статическая типизация.

С# – это современный объектно-ориентированный язык программирования, предназначенный для разработки разнообразных безопасный и мощных приложений. С его помощью можно создавать приложения Windows, XML веб-службы, распределенные компоненты, приложения "клиент-сервер", приложения баз данных и другие. Синтаксис C# очень выразителен, но прост в изучении. Все, кто знаком рассмотренными ранее языками, с легкостью узнают синтаксис, характерный для этого языка. Разработчики, знающие каждый из этих языков, как правило, могут добиться эффективной и качественной работы с языком C# за короткое время. Синтаксис этого языка делает проще то, что было сложно в C++, и обеспечивает мощные возможности, такие как перечисления, делегаты, лямбда-выражения и прямой доступ к памяти, чего нет в Java. C# поддерживает универсальные методы и типы, обеспечивая более высокий уровень безопасности и производительности, а также итераторы, позволяющие при реализации коллекций классов определять собственное поведение итерации, которое может легко использоваться в клиентском коде. Выражения LINQ делают строго типизированный запрос очень удобной языковой конструкцией.

C# – объектно-ориентированный язык, и поэтому поддерживает понятия инкапсуляции, наследования и полиморфизма. Все переменные и методы, включая метод Main – точку входа приложения – инкапсулируются в определения классов. Класс может наследовать непосредственно из одного родительного класса, но может реализовывать любое число интерфейсов. Для методов, которые переопределяют виртуальные методы в родительском классе, необходимо ключевое слово override, чтобы исключить случайное повторное определение. В языке C# структура похожа на облегченный класс: это тип, распределяемый в стеке, реализующий интерфейсы, но не поддерживающий наследование.

Преимуществами языка C# являются:

1. скорость разработки;
2. кроссплатформенность;
3. производительность кода и требовательность к ресурсам;
4. расширяемость системы;
5. степень открытости исходного кода;
6. защищенность и контроль версий;
7. удобство разработки;
8. подлинная объектно-ориентированность.

Из рассмотренных языков разработки был выбран язык C#, так как он обладает наиболее подходящими широким набором технологий, благодаря чемувыбранный язык позволяет легко и просто выполнять работу над растровыми изображениями.

# Среда разработки программного модуля

SharpDevelop – свободная среда разработки, обычно используется как альтернатива VisualStudio.NET. Она представляет собой интегрированный отладчик, использующий свои библиотеки и взаимодействующий с исполняющей средой .NET.

Основные преимущества SharpDevelop:

1. поддержка компиляторов C# от Microsoft и Mono;
2. возможности IntelliSense и расширения программного кода;
3. наличие диалогового окна Add Reference;
4. наличие инструментов визуального проектирования Windows Forms;
5. различные окна для обзора структуры проекта и его составляющих;
6. интегрированная утилита браузера;
7. утилиты для работы с базами данных;
8. утилита конвертирования программного кода C# в VB .NET (и наоборот).

Интегрированная среда разработки Visual C# представляет собой набор средств разработки, предоставляемых через единый пользовательский интерфейс. Некоторые средства используются совместно с другими языками Visual Studio, в то время как другие, например, компилятор C#, свойственны только Visual C#.Эта среда позволяет разрабатывать как консольные приложения, так и приложения с графическим интерфейсом, в том числе с поддержкой технологий Windows Forms, а также дает возможность разрабатывать веб-сайты, веб-приложения, веб-службы.

VS содержит в себе редактор исходного кода с поддержкой технологии IntelliSense, а также обладает возможностью простейшего рефакторинга кода. Встроенный отладчик дает возможность работать как отладчик уровня исходного кода, так и отладчик машинного уровня. Другие встраиваемые инструменты включают в себя такие компоненты, как редактор форм для упрощения создания графического интерфейса приложения, веб-редактор, дизайнер классов и дизайнер схемы базы данных. Visual Studio позволяет создавать и подключать сторонние дополнения для расширения функционала на каждом уровне, включая возможность добавления поддержки систем контроля версий исходного кода, позволяет добавлять новые наборы инструментов для прочих аспектов процесса разработки программного обеспечения.

Преимущества Visual C#:

1. расширенная модель программирования;
2. расширенная безопасность;
3. пользовательские типы и статистические выражения;
4. общая среда;
5. более высокая производительность;
6. набор операторов языка;
7. повторное использование кода;
8. совершенствование существующих навыков;
9. стабильность и надежность.

При выборе среды разработки выбор пал на VisualStudio, так как у нее понятный интерфейс, есть все необходимое для создания настольных приложений, и кроме этого она является удобным инструментом для написания и отладки кода, связанного с обработкой изображений.

# ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

# Общая структура программы

Для реализации программного модуля необходимо построить диаграмму классов, чтобы определится с будущей архитектурой приложения. Для этого было выполнено проектирование различных вариантов будущего приложения.

Наиболее подходящим решением была реализация шести базовых классов, каждый из которых определяет свои данные и поведения типов (см. рис. 3.1). На рисункеприведена диаграмма классовпрограммного модуля с обозначением функций, методов, свойств, и полей, их спецификаторами доступа и другими параметрами.

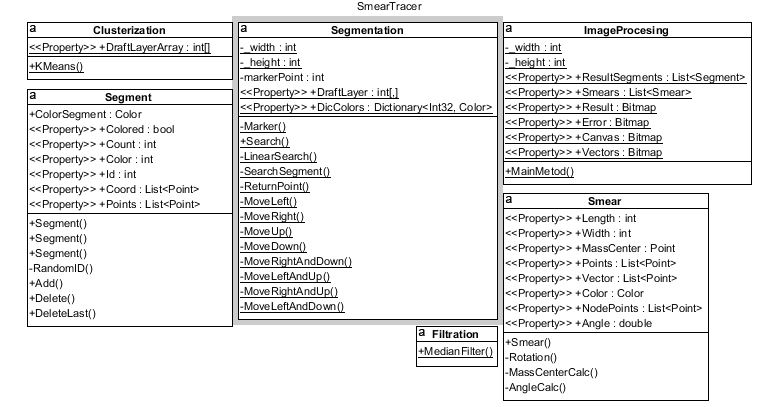


Рисунок 3.1 – Диаграмма Классов

Из рисунка видно, что два класса описывают сущности и их базовую функциональность, остальные же несут статическую функциональность и выступают в роли контейнеров для наборов методов работы с сущностями. Также в приложении используется ещё два класса, но они необходимы для создания графического пользовательского интерфейса, и будут рассмотрены отдельно. На рис.3.2 показана архитектура итогового приложения. Файл Program.cs содержит в себе стандартный метод для автоматического создания экземпляра класса Form, он отвечает за запуск всей программы.



Рисунок 3.2 – Архитектура приложения

MainForm.cs отвечает за графический интерфейс, а остальные файлы будут рассмотрены позднее.

# Описание объектов

Для описания сегмента был создан класс Segment, который содержит следующие поля и свойства:

//состояниесегмента

publicbool Colored { get; set; }

//цвет

publicColor ColorSegment;

//количествоточек

publicint Count { get; set; }

//номерцвета

publicint Color { get; set; }

//id сегмента

publicint Id { get; privateset; }

//списоккоординатконтура

publicList<Point> Coord { get; set; }

//списокточексегмента

publicList<Point> Points { get; set; }

Свойство Colored показывает, был ли закрашен контур, ColorSegment содержит информацию о цвете сегмента, а Color определяет принадлежность сегмента к группе сегментов одного цвета.

Для описания мазка создадим класс Smear, который будет описывать модель будущего мазка. Его свойствами будут являться длина мазка, ширина, центр масс, список точек вектора, цвет, угловые точки, а также угол наклона относительно оси Y.

//длинамазка

publicint Length { get; privateset; }

//ширинамазка

publicint Width { get; privateset; }

//центрмазка

publicPoint MassCenter { get;privateset; }

//списокточекмазка

publicList<Point> Points { get;privateset; }

//виузуализациявектора

publicList<Point> Vector { get; set; }

//цветмазка

publicColor Color { get;privateset; }

//угловыеточки

publicList<Point> NodePoints { get; privateset; }

//угол

publicdoubleAngle { get; privateset; }

Для создания объекта достаточно только одного параметра Color:

public Segment(int color)

{

Count = 0;

Color = color;

Coord = newList<Point>();

Points = newList<Point>();

Id = RandomID();

Colored = false;

}

В роли конструктора выступает метод с одним параметром:

public Smear(Segment segment)

{

//устанавливаемцветмазка

Color = segment.ColorSegment;

//вычисляемуголнаклона

Angle = AngleCalc(segment);

//вычисляемцентрмасс

MassCenter = MassCenterCalc(segment);

//выполняемповорот

Rotation(MassCenter, segment);

}

Для генерации уникального идентификатора служит функция генерации целого числа.

Экземпляр класса RNGCryptoServiceProvider реализует криптографический генератор случайных чисел, который позволяет с высокой долей вероятности создавать уникальный номер для сегмента:

privateintRandomID()

{//генераторслуйчаныйхчисел

RNGCryptoServiceProvider rnd = newRNGCryptoServiceProvider();

byte[] data = newbyte[4];

rnd.GetBytes(data);

returnBitConverter.ToInt32(data, 0);

}

Угол наклона вычисляется как сумма углов всех точек, деленная на количество точек, для этого необходимо взять соседние точки и вычислить арктангенс угла между ними:

currPoint = segment.Coord[i - 1];

nextPoint = segment.Coord[i];

//еслиотрезокимеетнаклон

if (currPoint.X != nextPoint.X)

{

angleSmear = (currPoint.Y - nextPoint.Y) / (currPoint.X - nextPoint.X);

angles.Add(angleSmear);

}//еслиотрезокпарралеленоси Y

elseif (currPoint.Y > nextPoint.Y)

{

angles.Add(3.14 / 2);

}

else

{

angles.Add(-3.14 / 2);

}

i++;

Центр масс вычисляется просто, для этого необходимо сложить соответствующие координаты и поделить получившееся число на их количество:

foreach (Point point in segment.Coord)

{

x += point.X;

y += point.Y;

}

x = Math.Round(x/segment.Coord.Count);

y = Math.Round(y/segment.Coord.Count);

Кроме этого, необходимо описать некоторую базовую функциональность класса. Для этого можно написать несколько простых методов.

Данные методы позволяют выполнять различные операции над объектами:

//добавлениеточки

publicvoid Add(Point point)

{

Count += 1;

Coord.Add(point);

}

//удалениеточки

publicvoid Delete(Point point)

{

this.Count -= 1;

this.Coord.Remove(point);

}

//удаление последней точки

publicvoid DeleteLast()

{

this.Count -= 1;

this.Coord.RemoveAt(this.Count);

}

Важным моментомявляется использование метода Math.Round для округления значения координаты, так как по умолчанию результатом операции деления целых чисел будет число, округленное в большую сторону.

С поворотом мазка всё немного сложнее. Для начала, необходимо найти расстояния от центра масс до каждой точки, и выбрать максимальное:

//все расстояния точек до центров масс

List<double> distance = newList<double>();

foreach (Point point in segment.Coord)

{

distance.Add(Math.Sqrt(Math.Pow(point.X - center.X, 2)

+ Math.Pow(point.Y - center.Y, 2)));

}//сортировкавпорядкеубыванияключа

distance = distance.OrderByDescending(d =>d).ToList();

Затем можно вычислить длину и ширину мазка, длина будет равняться самому большому расстоянию от центра масс, а ширина максимальной разности значений точек на оси Y:

Length = (int) (distance.First()\*2);

Width = segment.Coord.Max(p => p.Y) - segment.Coord.Min(p => p.Y);

Следующим шагом будет поиск угловых точек, которые будут аппроксимировать сегмент прямоугольником. Необходимо рассчитать четыре точки, а затем воспользоваться полученной длиной, шириной, и координатами центра масс:

NodePoints.Add(newPoint(MassCenter.X - Length/2, MassCenter.Y - Width/2));

NodePoints.Add(newPoint(MassCenter.X - Length/2, MassCenter.Y + Width/2));

NodePoints.Add(newPoint(MassCenter.X + Length/2, MassCenter.Y + Width/2));

NodePoints.Add(newPoint(MassCenter.X + Length/2, MassCenter.Y - Width/2));

В результате остается повернуть эти точки относительно центра масс, воспользовавшись следующими формулами:

for (var i = 0; i < NodePoints.Count; i++)

{

int x = center.X + (int)Math.Round((NodePoints[i].X - center.X) \* Math.Cos(Angle) - (NodePoints[i].Y - center.Y) \* Math.Sin(Angle));

int y = center.Y + (int)Math.Round((NodePoints[i].X - center.X) \* Math.Sin(Angle) + (NodePoints[i].Y - center.Y) \* Math.Cos(Angle));

NodePoints[i] = newPoint(y,x);

}

В итоге были написаны два класса, позволяющие создавать объекты и использовать их в качестве входных данных рассматриваемых далее статических классов.

# Статические классы

Для решения задачи фильтрации изображения используется класс Filtration, в котором метод MedianFilter принимает на вход два параметра – ампертуру фильтра и само изображение, и возвращает отфильтрованное изображение в виде объекта Bitmap. В этом методе фильтрация решается путем использования библиотеки AForge.Imaging.Filters, которая содержит в себе класс Median, с его помощью и происходит непосредственно сама фильтрация.

publicstaticBitmap MedianFilter(int rank, Bitmap inputImage)

{

Bitmap image = newBitmap(inputImage);

//медианныйфильтрсмаской rank

Median filter = newMedian(rank);

filter.ApplyInPlace(image);

returnimage;

}

Следующим шагом является применение алгоритма кластеризации с возможностью дальнейшего использования результатов разбиения изображения на кластеры. Для этого был создан класс Clusterization, состоящий из одного метода KMeans и свойства DraftLayerArray. Метод принимает на вход два параметра – colorCount(количество цветов) и inputImage(изображение для кластеризации), и возвращает кластеризованное изображение в виде объекта Bitmap. Свойство DraftLayerArray необходимо для хранения кластеризованного изображения в виде массива значений яркостей, так как в дальнейшем эти данные будут использоваться во многих частях проекта. Кластеризация осуществляется с помощью класса KMeans, который находится в библиотеке Accord.MachineLearning, но его возможности позволяют работать только с изображениями цветовой модели RGB, поэтому входные данные необходимо записать в массив, включающий только три компоненты.

//массив для реализации совместимости разных форматов

double[][] points = newdouble[data.GetLength(0)][];

//убираем прозрачность, если она пристуствует

for (inti = 0; i<data.GetLength(0); i++)

{

points[i] = newdouble[3];

for (int j = data[0].Length - 3; j < data[0].Length; j++)

{

points[i][j - data[0].Length + 3] = data[i][j];

}

}

Кроме этого изображение необходимо конвертировать в массив, элементами которого являются также массивы. Для этого используется библиотека Accord.Imaging.Converters, которая включает в себя конверторы с возможностью нормирования данных.

//сущности для конвертирования картинки в массив и обратно

ImageToArray input = newImageToArray();

ArrayToImage output = newArrayToImage(inputImage.Width, inputImage.Height);

После подготовки входных данных запускается сам алгоритм и происходит конвертирование данных из массива в изображение.

//запускаем алгоритм кластеризации

DraftLayerArray = kmeans.Compute(data);

data.ApplyInPlace((x, i) => kmeans.Clusters.Centroids[DraftLayerArray[i]]);

//выводим результат кластеризации

output.Convert(data, outimage);

Разбив изображение на одноцветные области, необходимо провести сегментацию изображения. Для этого используется класс Segmentation, состоящий из набора методов, свойств и полей для поиска замкнутых контуров и сегментов.

//двумерныймассивсцветамикластеров

publicstaticint[,] DraftLayer { get; privateset; }

//словарьсреальнымицветами

publicstaticDictionary<int, Color> DicColors { get;privateset; }

privatestaticint \_width, \_height;

//маркердляпоискаконтуров

privateconstintMarkerPoint = -1;

Для того, чтобы однозначно определять соответствие между конкретным цветом и значениями массива кластеризации, удобно использовать словарь.

//словарьсреальнымицветами

publicstaticDictionary<int, Color> DicColors { get;privateset; }

Значениями словаря являются реальные цвета, а ключ представляет собой число, определяющее принадлежность пикселя к определенному кластеру.

//ищем все цвета и добавляем в словарь с ключом

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

{

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

{

//если словарь не содержит этот цвет

if (!DicColors.ContainsValue(canvas.GetPixel(j, i)))

{

DicColors.Add(image[n], canvas.GetPixel(j, i));

}

n++;

}

}

Следующим шагом является создание двумерного массива для возможности обхода соседних пикселей.

//формируем матрицу кластеризованного изображения

for (inti = 0; i<image.Length; i++)

{

DraftLayer[k, p] = image[i];

k++;

if (k % \_width == 0 && i != 0)

{

p++;

k = 0;

}

}

Свойство DraftLayer представляет собой двумерную матрицу, преобразованную из одномерного массива путем разбиения его на строки.

На следующем шаге происходит поиск сегментов и присвоение каждому из них цвета из словаря.

//инициализируемсписоксконтурами

List<Segment> segments = newList<Segment>();

//перебираем картинку по всем строкам

while (h < \_height)

{//функция поиска по строке

LinearSearch(h, 0, \_width, segments);

h += 1;

}

//устанавливаемцветасегментам

foreach (varseginsegments)

{

seg.ColorSegment = DicColors[seg.Color];

}

Метод LinearSearch служит для поиска новых контуров на строке с координатой x на промежутке (y0;y1).

//ищемновыйконтур

Segment segment = SearchSegment(y0, x);

//добавляем контур в список, если он правильный

if (segment.Count > 5 && segment.Color != MarkerPoint)

{

segments.Add(segment);

Marker(segment, segments);

}

В случае если сегмент уже найден, выполняется переход на следующую точку этой сроки, не принадлежащую данному сегменту.

if (segments.Exists(seg => seg.Id == DraftLayer[y0, x]))

{ //ищемэтотконтур

Segment segmentFound = segments.Find(seg =>

seg.Id == DraftLayer[y0, x]);

//создаем список точек на прямой х

List<Point> list = segmentFound.Coord.FindAll(p =>

p.X == x && p.Y >= y0).OrderBy(p => p.Y).ToList();

//ищем точку

y0 = list.Last().Y;

for (int j = 0; j < list.Count; j++)

{

if (list[j].Y < \_width - 1 && DraftLayer[list[j].Y + 1, x]

!= segmentFound.Id && DraftLayer[list[j].Y + 1, x]

!= segmentFound.Color)

{

y0 = list[j].Y;

j = list.Count;

}

}

Метод Marker выполняет обход найденного контура, маркируя пиксели, принадлежащие кластеру, а также находит новые контуры внутри него. Его входными параметрами являются найденный сегмент и список всех найденных сегментов. Для обхода всех точек сегмента текущий сегмент копируется в новый сегмент, а затем координаты его контура сортируются в порядке убывания координаты x. После этого выполняется обход в цикле каждой строки с проверкой условий на принадлежность существующему контуру, текущему, либо новому контуру. В противном случае точка помечается как принадлежащая текущему сегменту. Ниже приведена часть кода, отвечающая за добавление нового сегмента.

//еслинашлиновыйконтур

elseif (DraftLayer[i, x0] != copyInputSegment.Id)

{

//ищемконтур

Segment segment = SearchSegment(i, x0);

//если контур больше 4 точек и он не тупиковый

if (segment.Count > 5 && segment.Color != MarkerPoint)

{//добавляемвспискокконтур

segments.Add(segment);

//ищем точку для перехода к сделующей точке

List<Point> list =

segment.Coord.FindAll(p => p.X == x0 && p.Y >=

i).OderBy(p => p.Y).ToList();

i = list.Last().Y;

for (int j = 0; j < list.Count; j++)

{

if (list[j].Y < \_width - 1 &&

DraftLayer[list[j].Y + 1, x0] == inputSegment.Color)

{

i = list[j].Y;

j = list.Count;

}

}

}

}

Особенностью этого метода является последовательная закраска контура с дальнейшей закраской всех новых контуров с помощью рекурсии:

//помечаем контур как закрашенный

inputSegment.Colored = true;

//закрашиваемновыеконтуры

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

{

if (!segments[i].Colored)

{

Marker(segments[i], segments);

}

}

Метод Search служит для подготовки данных для поиска замкнутых контуров, он принимает на вход массив яркостей кластеризованного изображения и само кластерзованное изображение. Метод SearchSegment выполняет поиск контура с начальными координатами (x0, y0) в массиве DraftLayer. Для выполнения поиска необходимо использовать точки для проверки направления, а также хранить координаты предыдущей точки.

//точки для обхода

int x = x0, y = y0;

//точки для проверки направления

int yPrev = y0, xPrev = x0, dY, dX;

//списоктупиковыхточек

List<Point> delPoints = newList<Point>();

//новыйконтур

Segment segment = newSegment(DraftLayer[y0, x0]);

При обходе контура может возникнуть ситуация, когда продолжить движение нельзя, для этого необходимо сохранять тупиковые точки в списке delPoints.Получив на вход координаты начальной точки, необходимо сделать пробный шаг влево, добавить точку в список точек контура, и продолжить поиск следующих точек по часовой стрелке, с учетом приоритета направления движения наружу:

//делаем пробный шаг

MoveLeft(refy, refx);

//вычисляем начальное направление и запоминаем точку

dY = y - yPrev;

dX = x - xPrev;

yPrev = y;

xPrev = x;

//снимаем маркер с первой точки, чтобы была возожность снова в нее попасть

DraftLayer[y0, x0] = segment.Color;

//добавляем первую точку в контур

segment.Add(newPoint(x, y));

do

{

//определяем направление движения по часовой стрелке

if (dX == -1 && dY == 0)

{

MoveLeft(ref y, ref x);

}

elseif (dX == -1 && dY == 1)

{

MoveLeft(ref y, ref x);

}

…

Стоит отметить, что в дальнейшем этот список вернет начальное значение цвета искомого сегмента.

Далее операции повторяются в цикле до тех пор, пока не вернемся в начальную точку, а после снимаем маркер с тупиковых точек и помечаем точки, принадлежащие контуру:

//вовзращаемцветудаленнымточкам

foreach (Point delPoint in delPoints)

{

DraftLayer[delPoint.Y, delPoint.X] = segment.Color;

}

//помечаемточкиконтура

foreach (Point segPoint in segment.Coord)

{

DraftLayer[segPoint.Y, segPoint.X] = segment.Id;

}

Иногда возникает ситуация, когда все соседние точки непригодны для перехода, для этого необходимо вернуться на предыдущую точку и проверить возможность перехода. Для борьбы с этой проблемой используется метод ReturnPoint, суть которого заключается в удалении тупиковой точки из контура и возврат на предыдущую точку до тех пор, пока не будет возможен переход в новую точку, либо пока не вернемся в исходную. Ниже приведен пример части кода удаления последней точки и перехода на предыдущую:

//удаляем точку и маркируем

DraftLayer[y, x] = MarkerPoint;

segment.DeleteLast();

//добавляем точку в список удаленных

points.Add(newPoint(x, y));

//если сегмент имеет точки

if (segment.Count > 0)

{

//ищемновыйпуть

lastY = segment.Coord[segment.Count - 1].Y;

lastX = segment.Coord[segment.Count - 1].X;

dX = x - lastX;

dY = y - lastY;

x = lastX;

y = lastY;

//снимаем маркет для продолжения поиска

DraftLayer[y, x] = segment.Color;

}

Последними методами класса Segmentation являются восемь методов, каждый из которых отвечает за свое направление перехода в новую точку, и каждый из них имеет свое приоритетное направление поиска. Их задачей является поиск точек контура с приоритетом направления по часовой стрелке, осуществляя переход по каждой из координат на величину, равную нулю, либо одному пикселю.

privatestaticvoid MoveLeft(refint y, refint x)

{//ищем крайнюю точку по часовой стрелке

if (y > 0 && x > 0 && DraftLayer[y - 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

x -= 1;

}

elseif (x > 0 && DraftLayer[y, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

}

elseif (y < \_width - 1 && x > 0 && DraftLayer[y + 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

y += 1;

}

…

Следующий класс, названный ImageProcesing, служит для объединения всех рассмотренных ранее алгоритмов в последовательную процедуру поиска небольших контуров и их преобразования в мазки. Он содержит в себе следующие поля и свойства:

//ширинаивысота

privatestaticint \_width;

privatestaticint \_height;

//списоксегментов

publicstaticList<Segment> ResultSegments { get;privateset; }

//списокмазков

publicstaticList<Smear> Smears { get;privateset; }

//результаткластеризации

publicstaticBitmap Result { get;privateset; }

//ошибка

publicstaticBitmap Error { get;privateset; }

//мазки

publicstaticBitmap Canvas { get;privateset; }

//векторы

publicstaticBitmap Vectors { get;privateset; }

Рассматриваемый класс содержит только один метод MainMetod, входные параметры которого это исходное изображение, ампертура фильтра, максимальный размер мазка, компонент BackgroundWorker, и параметр DoWorkEventArgs. Его выполнение начинается с инициализации параметров, после чего выполняется фильтрация.

//начальное число цветов для кластеризации

int сolorCount = 2;

//списоксегментовдляаппроксимации

ResultSegments = newList<Segment>();

//списокмазков

Smears = newList<Smear>();

//списоквсехнайденныхсегментов

List<Segment> segments = newList<Segment>();

//цвет для отсеивания известных контуров

Color temp = Color.White;

//предыдущее значение найденыых точек

intstepPoints = 0;

//счетчикитераций

inti = 0;

//медианный фильтр

curPict = Filtration.MedianFilter(rankFilter, curPict);

На следующем шаге происходит кластеризация и сегментация, после чего из списка найденных сегментов удаляются те, что имеют белый цвет. После этого сегменты, не превышающие размер мазка, добавляются в список ResultSegments, и холст заполняется значениями исходного отфильтрованного изображения.

//удаляем белые сегменты

segments.RemoveAll(seg => temp == seg.ColorSegment);

//добавляем сегменты в список, если они меньше размеров мазка

ResultSegments.AddRange(segments.FindAll(s =>

s.Coord.Count<sizeSmear));

//удаляем сегменты, если они меньше размера мазка

segments.RemoveAll(s => s.Points.Count < sizeSmear);

//заполняем холст исходными значениями

Canvas = newBitmap(curPict);

Для поиска кластеров необходимо также на каждом шаге изменять число цветов для кластеризации в соответствии с правилом:

//количество цветов для кластеризации

сolorCount = segments.Count + i + 1;

Таким образом, мы избежим зацикливания программы, а также кластеризация изображения будет происходить плавно и скорость выполнения алгоритма увеличится за счет равномерной кластеризации.

Для того чтобы избежать добавления уже существующих сегментов, заполняем на холсте белым цветом те пиксели, которые принадлежат списку ResultSegments:

foreach (Segment resultSegment in ResultSegments)

{

foreach (Point point in resultSegment.Points)

{

Canvas.SetPixel(point.Y, point.X, Color.White);

}

foreach (Point point in resultSegment.Coord)

{

Canvas.SetPixel(point.Y, point.X, Color.White);

}

}

Кроме этого, в данном методе реализована возможность отмены операции с помощьюсвойства CancellationPending:

//отмена выполнения операции

if (sender.CancellationPending)

{

e.Cancel = true;

segments = newList<Segment>();

}

Еще одной интересной деталью является передача информации о степени выполнения задачи с помощью объекта ProgressBar:

k = 99 \* ResultSegments.Sum(s =>

s.Points.Count + s.Coord.Count) / (\_width \* \_height);

if (k< 100)

{

sender.ReportProgress(k, null);

}

else

{

sender.ReportProgress(99, null);

}

В конце работы алгоритма небольшие сегменты переводим в мазки:

//формируеммазки

foreach (Segment resultSegment in ResultSegments)

{

Smears.Add(newSmear(resultSegment));

}

//возвращаемчисломазков

return Smears.Count;

Все рассмотренные классы являются статическими, так как они в большей степени содержат методы, выполняющие математические операции.

# Разработка пользовательского интерфейса

Создание пользовательского графического интерфейсаосуществлялось с помощью технологии WindowsForms, и проходило в три этапа:

1. добавление элементов управления на поверхность разработки;
2. установка свойств элементов управления;
3. добавление обработчиков для различных событий.

Сначала были добавлены необходимые элементы управления на форму припомощи конструктораWindowsForms, при этом конструктор автоматически записывает исходный код в файл проекта Form.designer.cs, там же будет записана информация о свойствах компонентов формы. Прототип пользовательского интерфейса после первичного добавления компонентов на форму (см. рис. 3.3) не обладает функциональными возможностями.

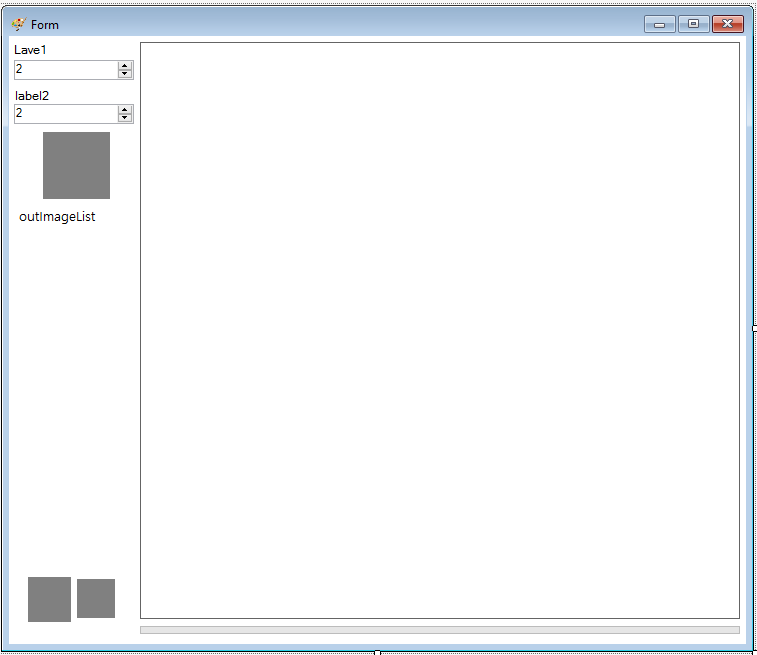


Рисунок 3.3 – Элементы управления

Следующим шагом будет добавление свойств элементов управления. Для этого поменяем фон кнопок и текст по умолчанию на всех элементах управления, а также установим начальные значения входным параметрам и заполним коллекцию элементов для выбора результатов работы алгоритма (см. рис. 3.4).

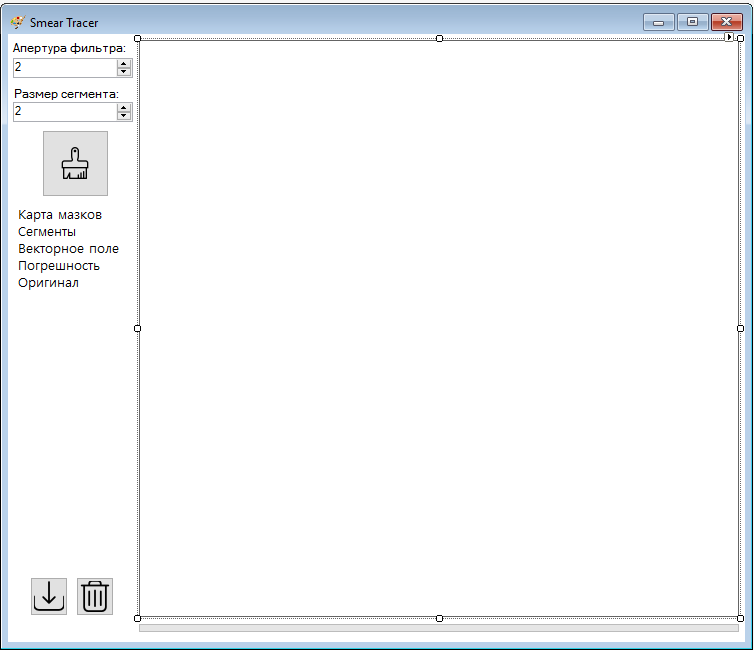


Рисунок 3.4 – Свойства элементов

В окне Свойства можно указать события, добавим события для взаимодействия с пользователем. Событие RunAlg\_Click происходит при нажатии на кнопку запуска алгоритма (кисточка):

privatevoid RunAlg\_Click(object sender, EventArgs e)

{//еслиобъектвыполняетоперацию

if (bWorker.IsBusy)

{//отменяемеё

bWorker.CancelAsync();

}

else

{//запускаемзадачу

bWorker.RunWorkerAsync();

runAlg.BackgroundImage = Properties.Resources.cancel;

HelpMessage.SetToolTip(this.runAlg, "Отмена");

}

}

В событии используется компонент BackgroundWorker, он служит для выполнения фонового процесса в отдельном потоке, что позволяет форме не зависать во время выполнения алгоритма.

Выполнение этого события вызывает событие BWorker\_DoWork, внутри него необходимо выполнить метод MainMetod, который и обеспечивает связь между графическим интерфейсом и бизнес-логикой:

privatevoid BWorker\_DoWork(object sender, DoWorkEventArgs e)

{

e.Result = ImageProcesing.MainMetod(newBitmap(\_currentPicture), (int)rankFilter.Value, (int)smearSize.Value, bWorker, e);

}

При завершении выполнения кода внутри этого события вызывается событие BWorker\_RunWorkerCompleted, внутри него необходимо поместить следующий код:

privatevoid BWorker\_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)

{

progress.Value = 100;

runAlg.BackgroundImage = Properties.Resources.run;

HelpMessage.SetToolTip(this.runAlg, "Разбитьизображениенамазки");

if (e.Error != null)

{//выводим сообщение об ошибке

MessageBox.Show(e.Error.Message);

}

elseif (e.Cancelled)

{//в случае отмены операции выводим сообщение

MessageBox.Show("операция была отменена!");

}

else

{

//выводим информацию

labelNumbSegments.Text = "Всего мазков: " + e.Result;

ImagesPaint();

OutImageList\_SelectedIndexChanged(null, e);

MessageBox.Show("Готово!");

}

}

С помощью компонента ToolTip будем выводить подсказки для пользователя, а компонент ProgressBar показывает ход выполнения операции. Кроме этого, вызывается событие OutImageList\_SelectedIndexChanged для вывода изображения (см. рис. 3.5), которое выбрал пользователь:

privatevoid OutImageList\_SelectedIndexChanged(object sender, EventArgs e)

{

switch (outImageList.SelectedIndex)

{

case 0:

outputImage.Image = ImageProcesing.Canvas;

break;

case 1:

outputImage.Image = ImageProcesing.Result;

break;

case 2:

outputImage.Image = ImageProcesing.Vectors;

break;

case 3:

outputImage.Image = ImageProcesing.Error;

break;

case 4:

outputImage.Image = \_currentPicture;

break;

default:

outputImage.Image = ImageProcesing.Canvas;

break;

}

}

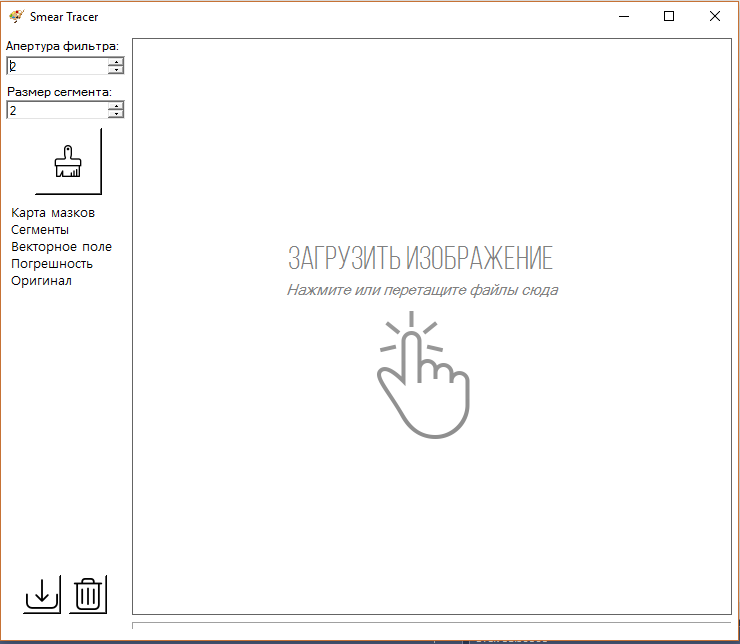


Рисунок 3.5 – Итоговый интерфейс

Еще одним важным моментом является способ загрузки входных данных – изображений. Для этого воспользуемся событием OutputImage\_DragDrop, которое позволяет перетаскивать изображение непосредственно на саму форму:

privatevoid OutputImage\_DragDrop(object sender, DragEventArgs e)

{

int x = this.PointToClient(newPoint(e.X, e.Y)).X;

int y = this.PointToClient(newPoint(e.X, e.Y)).Y;

if (x >= outputImage.Location.X &&

x <= outputImage.Location.X + outputImage.Width

&& y >= outputImage.Location.Y && y <= outputImage.Location.Y

+ outputImage.Height)

{

string[] files = (string[])e.Data.GetData(DataFormats.FileDrop);

\_currentPicture = (Bitmap)Image.FromFile(files[0]);

outputImage.Image = \_currentPicture;

//устанавливаем размер мазка по умолчанию

smearSize.Value = (\_currentPicture.Width \*

\_currentPicture.Height) /

(\_currentPicture.Width +

\_currentPicture.Height) / 2;

}

}

Создание GUI на самом деле является итеративным процессом, и этот результат был получен с помощью тщательной проработки всех его деталей. При разработке интерфейса были учтены показатели качества программного продукта: функциональность, производительность и эстетика.

# Тестирование разработанной подсистемы

В качестве исходного изображения для тестирования выберем картинку разрешением 800 на 600 пикселей (см. рис.3.6). Затем изменим предложенный размер мазка на 150 пикселей, ампертуру фильтра укажем 2, и нажмем кнопку “Разбить”.

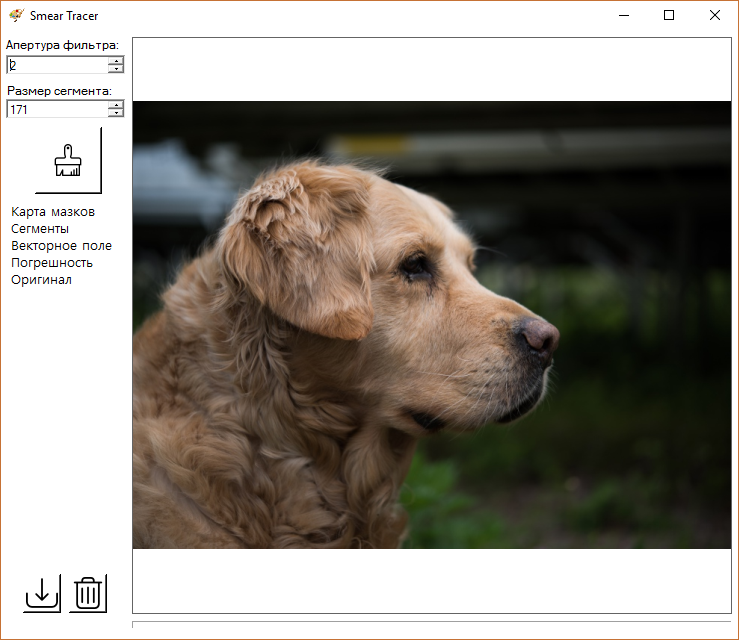


Рисунок 3.6 – Окно с исходным изображением

Дождемся завершения работы алгоритма, для этого необходимо обратить внимание на черную полосу загрузки, расположенную внизу окна. Также по окончании выполнения операции появится соответствующее сообщение (см. рис.3.7).

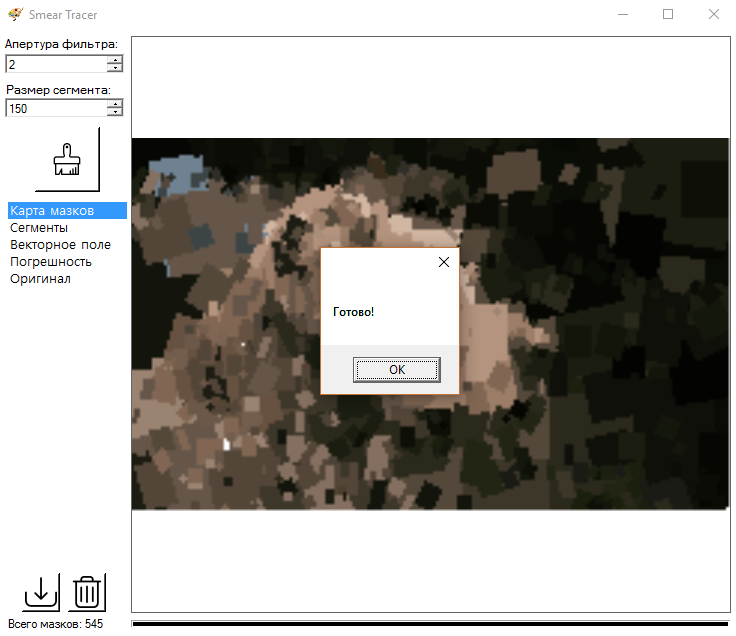


Рисунок 3.7– Успешное завершение операции

Подождав некоторое время, можно посмотреть результат работы программы (см. рис. 3.8) с картой колористических примитивов (мазков).

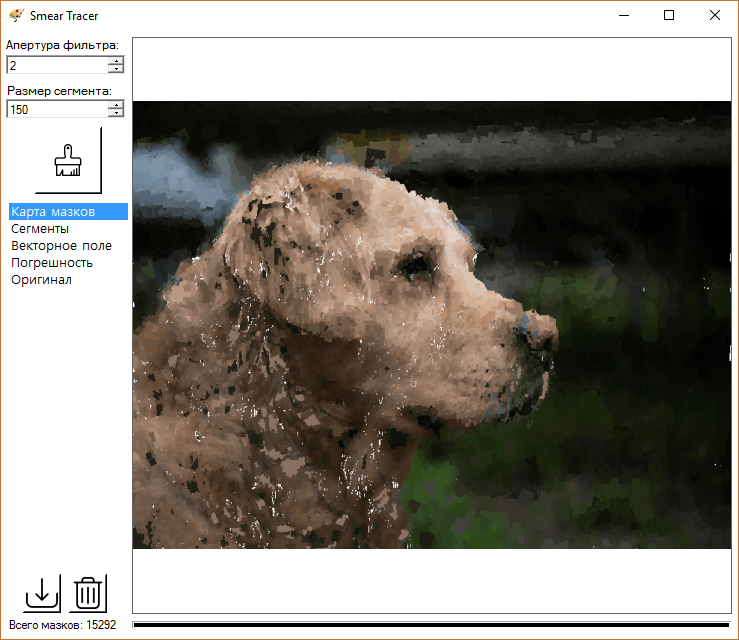


Рисунок 3.8 – Карта мазков

Как видно из рисунка, их количество равно 15292.Число полученных сегментов (см. рис. 3.9) соответствует числу мазков.

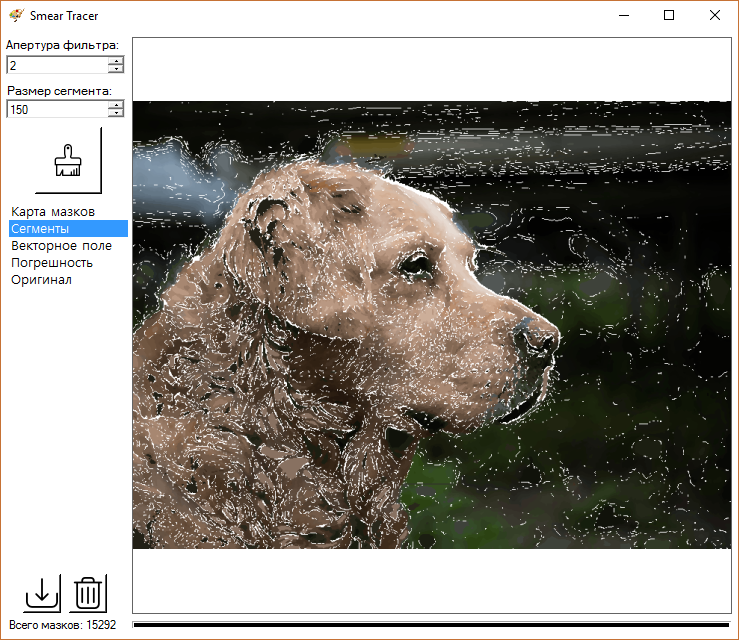


Рисунок 3.9 – Полученные сегменты

Еще одним результатом выполнения программы служит векторное поле (см. рис. 3.10), которое показывает направления движения мазков.

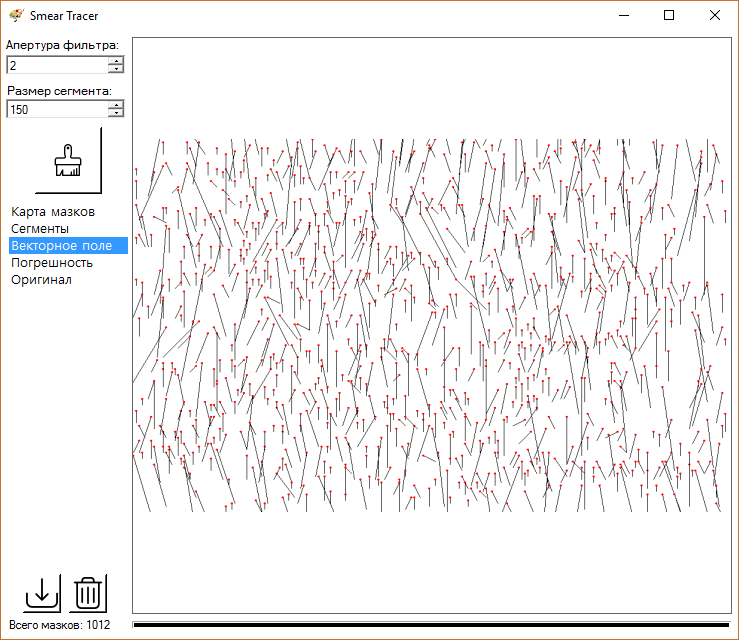


Рисунок 3.10 – Векторное поле

Для демонстрации точности выполнения алгоритма выводится погрешность между исходным изображением и картой мазков (см. рис. 3.11).

Также в качестве тестовых примеров были использованы популярные картинки, такие как “Лена” (см. рис. 3.12) и “Шрек” (см. рис. 3.13)

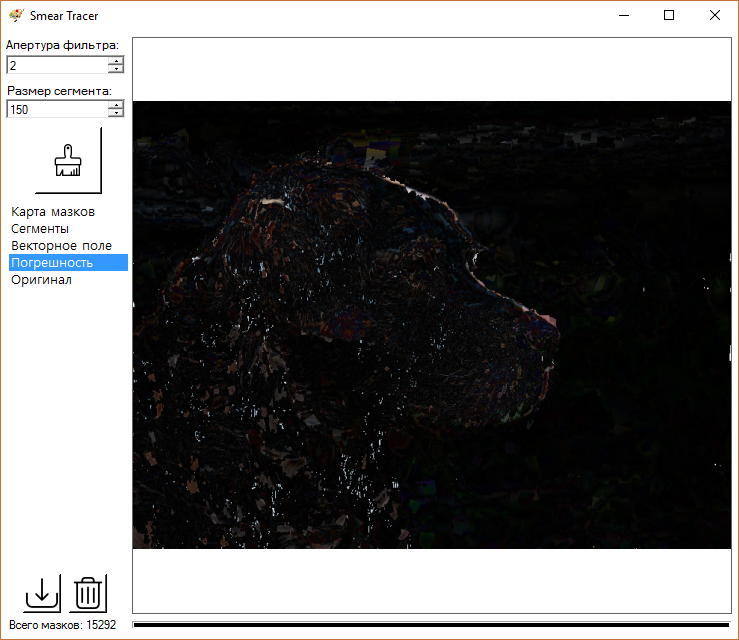


Рисунок 3.11 – Погрешность карты мазков

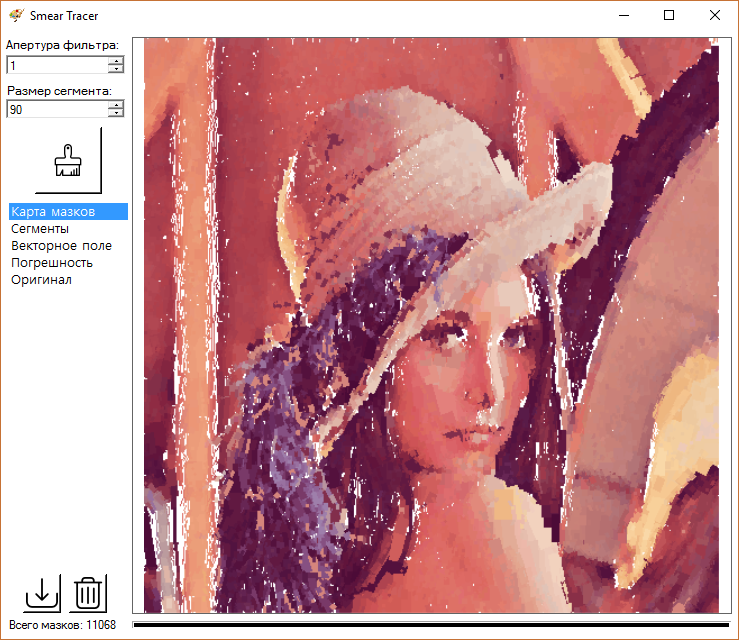


Рисунок 3.12 – Карта мазков изображения “Лена”

Для изображения “Лена” с разрешением 900 на 600 пикселей была построена карта, состоящая из 11068 мазков.

Для изображения “Шрек” с разрешением 400 на 300 пикселей была построена карта колористических примитивов, состоящая из 5321 мазка.

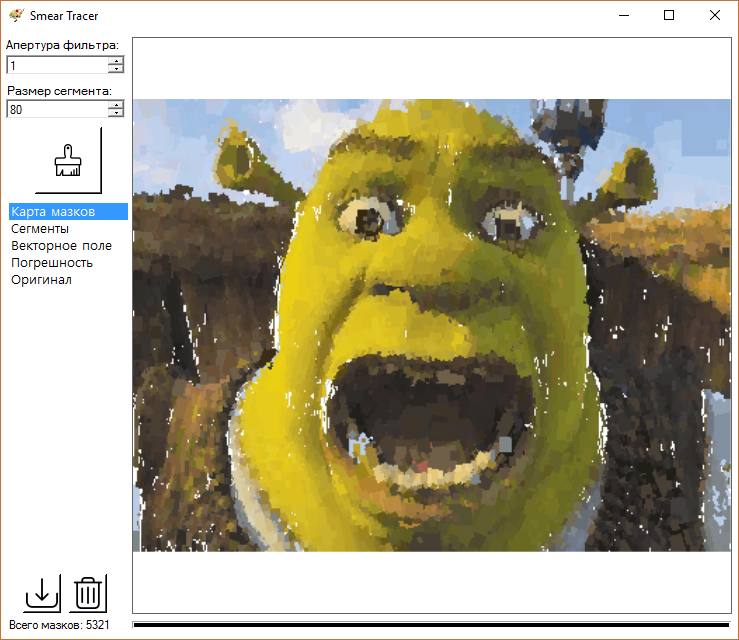


Рисунок 3.13 – Карта мазков изображения “Шрек”

В результате тестирования программного модуля было установлено, что программа хорошо справляется с поставленной задачей.

# СПЕЦИАЛЬНЫЕ ВОПРОСЫ ОБЕСПЕЧЕНИЯБЕЗОПАСНОСТИ

Эргономика – это дисциплина, которая комплексно изучает человека в конкретных условиях, она включается в процессы разработки и тестирования программных продуктов как часть системы качества. Разработка графического пользовательского интерфейса ведется вместе с дизайном всего программного продукта. Рассмотрим основные показатели качества с точки зрения эргономики, которые необходимо достичь в результате проектирования и разработки программного продукта.

Одним из важных показателей качества является эффективность работы. Он показывает, что, используя данный программный продукт, пользователь, будет решать поставленные перед ним задачи с минимальными трудозатратами. Этот показатель включает в себя также точность работы (процент ошибок, которые совершил пользователь), функциональную полноту (степень использования первичных и обработанных данных) и завершенность работы (число пользователей, которые выполнили задание в фиксированные сроки). Для сокращения трудозатрат в разрабатываем интерфейсе максимально упрощен ввод входных данных, он включает в себя только два базовых параметра. Кроме этого, при загрузке входных данных эти параметры устанавливаются рекомендуемыми значениями, что в свою очередь снижает трудозатраты.

Другим показателем качества является производительность работы, этот показатель характеризует объем ресурсов, который необходимо затратить при выполнении задачи. Ресурсы могут быть как вычислительные, так и психологические. Чтобы обеспечить минимизацию усилий пользователя в момент выполнения работы, дизайн программного интерфейса должен приводить к сокращению длительности операций редактирования, чтения, и поиска информации, уменьшению времени выбора команды, повышению общей продуктивности пользователя, увеличению длительности устойчивой работы пользователя. Для этого в разрабатываемом интерфейсе была реализована система подсказок, которая позволяет пользователю получить информацию о необходимых событиях, не прилагая дополнительных усилий.

Еще одним важным показателем является удовлетворенность пользователя от работы. Он тесно связан комфортом взаимодействия пользователя с приложением, и для достижения высокого уровня данного показателя необходимо, чтобы навигация была прозрачной и ясной, тексты и значения иконок понятные, использование стандартных элементов взаимодействия и их традиционное расположение, а также иметь в наличии вспомогательные средства поддержки пользователей. Для достижения этой цели был разработан удобный интерфейс (см. рис. 4.1), разделенный на две области – область отображения результатов работы и рабочую область, состоящую из стандартных элементов управления.

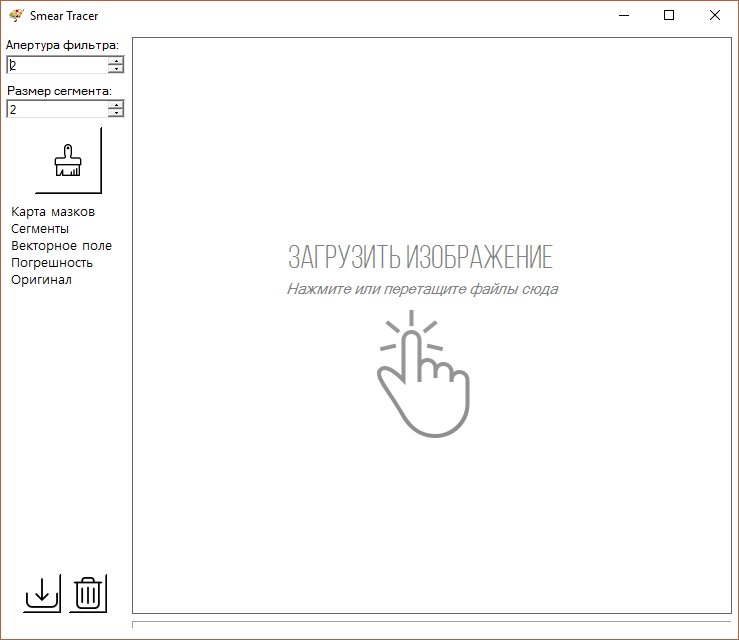


Рисунок 4.1 – Пользовательский интерфейс

Кроме этого, были добавлены возможности сохранения результатов работы в файл, а также возможность очистки текущих результатов.

В разработанном пользовательском интерфейсе также были соблюдены принципы его реализации: стилевая гибкость позволяет использовать различные интерфейсы с одним и тем же приложением, имеется возможность наращивать функциональность без нарушения существующего интерфейса, интерфейс можно легко настраивать и расширять, приложение также допускает ввод и вывод данных с помощью различных способов (клавиш клавиатуры и кнопок мыши), его можно легко перенести на любую другую платформу без потери, как интерфейса, так и самого приложения.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения ВКР был разработан программный модуль, решающий поставленные задачи, с визуализацией результатов работы последовательных этапов алгоритма. В ходе достижения цели были сформулированы требования к разрабатываемому модулю и проведен математический анализ алгоритмов кластеризации, сегментации и фильтрации изображений. Достигнуты следующие результаты:

1. Были рассмотрены алгоритмы фильтрации изображений, проведен их анализ и выбран медианный алгоритм для снижения уровня шума на изображении.
2. Провели анализ популярных методов кластеризации, среди которых выбрали алгоритм k-means в качестве основы этапа кластеризации, определили входные параметры для сокращения вычислительной нагрузки на процессор.
3. Рассмотрели возможные варианты применения алгоритмов сегментации, в результате чего был написан специализированный алгоритм, основанный на приоритетах направлений с поиском контуров на кластеризованном изображении.
4. Разработали алгоритм, способный строить карту мазков путем аппроксимации примитивных сегментов прямоугольниками с вычислением угла наклона и вектора направления.
5. Спроектировали графический интерфейс пользовательского приложения с учетом показателей качества с точки зрения эргономики.

Написали пользовательское приложение на основе разработанного модуля с визуализацией карты мазков, карты векторов, всех примитивных сегментов, а также погрешности карты мазков с исходным изображением.

Таким образом, разработан программный модуль, готовый к применению в составе управляющего программного обеспечения робота-живописца.

Имеются перспективы дальнейшего развития, такие как возможность выбора аппроксимации сегментов из списка предложенных фигур, а также разработка web-приложения на основе имеющегося модуля.

# СПИСОК ЛИТЕРАТУРЫ

1. K-means. SonkaM., HlavacV., BoyleR. Imageprocessing, analysis, and machine vision. – Cengage Learning, 2014, pp. 403-404.
2. Toh K. K. V., Isa N. A. M., Ashidi N. Noise adaptive fuzzy switching median filter for salt-and-pepper noise reduction //IEEE signal processing letters. – 2010. – Т. 17. – №. 3. – С. 281-284.
3. Juan A., Vidal E. Comparison of four initialization techniques for the k-medians clustering algorithm //Advances in Pattern Recognition. – Springer Berlin Heidelberg, 2000. – С. 842-852.
4. Reddy P. R., Amarnadh V., Bhaskar M. Evaluation of stopping criterion in contour tracing algorithms //International Journal of Computer Science and Information Technologies. – 2012. – Т. 3. – №. 3. – С. 3888-3894.
5. Troelsen A. Pro C# 5.0 and the. NET 4.5 Framework. – Apress, 2012.
6. Hertzmann A. A survey of stroke-based rendering //IEEE Computer Graphics and Applications. – 2003. – №. 4. – С. 70-81.

# ПРИЛОЖЕНИЕ А

Исходный код программного модуля

PublicclassSmear

{

//длина мазка

publicintLength { get; privateset; }

//ширинамазка

publicintWidth { get; privateset; }

//центрмазка

publicPointMassCenter { get;privateset; }

//списокточекмазка

publicList<Point> Points { get;privateset; }

//виузуализациявектора

publicList<Point> Vector { get; set; }

//цветмазка

publicColor Color { get;privateset; }

//угловыеточки

publicList<Point> NodePoints { get; privateset; }

//угол

publicdouble Angle { get; privateset; }

//конструктор

public Smear(Segment segment)

{

//устанавливаемцветмазка

Color = segment.ColorSegment;

//вычисляемуголнаклона

Angle = AngleCalc(segment);

//вычисляемцентрмасс

MassCenter = MassCenterCalc(segment);

//выполняемповорот

Rotation(MassCenter, segment);

}

//функцияповорота

privatevoid Rotation(Point center, Segment segment)

{

//всерасстоянияточекдоцентров

List<double> distance = newList<double>();

foreach (Point point in segment.Coord)

{

distance.Add(Math.Sqrt(Math.Pow(point.X - center.X, 2) + Math.Pow(point.Y - center.Y, 2)));

}

distance = distance.OrderByDescending(d => d).ToList();

Length = (int) (distance.First()\*2);

Width = segment.Coord.Max(p => p.Y) - segment.Coord.Min(p => p.Y);

NodePoints = newList<Point>();

Vector = newList<Point>();

NodePoints.Add(newPoint(MassCenter.X - Length/2, MassCenter.Y - Width/2));

NodePoints.Add(newPoint(MassCenter.X - Length/2, MassCenter.Y + Width/2));

NodePoints.Add(newPoint(MassCenter.X + Length/2, MassCenter.Y + Width/2));

NodePoints.Add(newPoint(MassCenter.X + Length/2, MassCenter.Y - Width/2));

Points = newList<Point>();

for (int i = NodePoints[0].X; i <= NodePoints[2].X; i++)

{

for (int j = NodePoints[0].Y; j <= NodePoints[1].Y; j++)

{

int x = center.X + (int) Math.Round((i - center.X)\*Math.Cos(Angle) - (j - center.Y)\*Math.Sin(Angle));

int y = center.Y + (int) Math.Round((i - center.X)\*Math.Sin(Angle) + (j - center.Y)\*Math.Cos(Angle));

Points.Add(newPoint(y, x));

if (j == center.Y)

{

Vector.Add(newPoint(y, x));

}

}

}

for (var i = 0; i < NodePoints.Count; i++)

{

int x = center.X + (int)Math.Round((NodePoints[i].X - center.X) \* Math.Cos(Angle) - (NodePoints[i].Y - center.Y) \* Math.Sin(Angle));

int y = center.Y + (int)Math.Round((NodePoints[i].X - center.X) \* Math.Sin(Angle) + (NodePoints[i].Y - center.Y) \* Math.Cos(Angle));

NodePoints[i] = newPoint(y,x);

}

}

//центрмасс

privatePoint MassCenterCalc(Segment segment)

{

double x = 0, y = 0;

foreach (Point point in segment.Coord)

{

x += point.X;

y += point.Y;

}

x = Math.Round(x / segment.Coord.Count);

y = Math.Round(y / segment.Coord.Count);

returnnewPoint(x.To<int>(), y.To<int>());

}

//угол

privatedouble AngleCalc(Segment segment)

{

//списоквсехуглов

List<double> angles = newList<double>();

//средний угол

double angleSmear;

//две соседние точки

Point currPoint, nextPoint;

int i = 1;

do

{

currPoint = segment.Coord[i - 1];

nextPoint = segment.Coord[i];

//еслиотрезокимеетнаклон

if (currPoint.X != nextPoint.X)

{

angleSmear = (currPoint.Y - nextPoint.Y) / (currPoint.X - nextPoint.X);

angles.Add(angleSmear);

}//еслиотрезокпарралеленоси Y

elseif (currPoint.Y > nextPoint.Y)

{

angles.Add(3.14 / 2);

}

else

{

angles.Add(-3.14 / 2);

}

i++;

} while (i < segment.Coord.Count);

return angles.Average();

}

}

publicclassSegment

{

//состояниесегмента

publicbool Colored { get; set; }

//цвет

publicColor ColorSegment;

//количествоточек

publicint Count { get; set; }

//номерцвета

publicint Color { get; set; }

//id сегмента

publicint Id { get; privateset; }

//списоккоординатконтура

publicList<Point> Coord { get; set; }

//списокточексегмента

publicList<Point> Points { get; set; }

public Segment(int color)

{

Count = 0;

Color = color;

Coord = newList<Point>();

Points = newList<Point>();

Id = RandomID();

Colored = false;

}

public Segment(Segment thisSegment)

{

Count = thisSegment.Count;

Color = thisSegment.Color;

Coord = newList<Point>(thisSegment.Coord);

Points = newList<Point>(thisSegment.Points);

Id = thisSegment.Id;

Colored = thisSegment.Colored;

}

public Segment()

{

}

privateint RandomID()

{//генераторслуйчаныйхчисел

RNGCryptoServiceProvider rnd = newRNGCryptoServiceProvider();

byte[] data = newbyte[4];

rnd.GetBytes(data);

returnBitConverter.ToInt32(data, 0);

}

//добавлениеточки

publicvoid Add(Point point)

{

Count += 1;

Coord.Add(point);

}

//удалениеточки

publicvoid Delete(Point point)

{

this.Count -= 1;

this.Coord.Remove(point);

}

//удаление последней точки

publicvoid DeleteLast()

{

this.Count -= 1;

this.Coord.RemoveAt(this.Count);

}

}

publicstaticclassFiltration

{

publicstaticBitmap MedianFilter(int rank, Bitmap inputImage)

{

Bitmap image = newBitmap(inputImage);

//медианныйфильтрсмаской rank

Median filter = newMedian(rank);

filter.ApplyInPlace(image);

return image;

}

}

publicstaticclassClusterization

{

publicstaticint[] DraftLayerArray { get; privateset; }

publicstaticBitmap KMeans(int colorCount, Bitmap inputImage)

{

Bitmapimage = newBitmap(inputImage);

//сущности для конвертирования картинки в массив и обратно

ImageToArray input = newImageToArray();

ArrayToImage output = newArrayToImage(inputImage.Width, inputImage.Height);

double[][] data;

input.Convert(image, out data);

KMeanskmeans = newKMeans(colorCount);

//массив для реализации совместимости разных форматов

double[][] points = newdouble[data.GetLength(0)][];

//убираем прозрачность, если она пристуствует

for (inti = 0; i<data.GetLength(0); i++)

{

points[i] = newdouble[3];

for (int j = data[0].Length - 3; j < data[0].Length; j++)

{

points[i][j - data[0].Length + 3] = data[i][j];

}

}

//запускаемалгоритмкластеризации

DraftLayerArray = kmeans.Compute(data);

data.ApplyInPlace((x, i) => kmeans.Clusters.Centroids[DraftLayerArray[i]]);

//выводим результат кластеризации

output.Convert(data, out image);

returnimage;

}

}

staticclassSegmentation

{

//двумерныймассивсцветамикластеров

publicstaticint[,] DraftLayer { get; privateset; }

//словарьсреальнымицветами

publicstaticDictionary<int, Color> DicColors { get; privateset; }

//длина и ширина изображения

privatestaticint \_width, \_height;

//маркер для поиска контуров

privateconstint MarkerPoint = -1;

publicstaticList<Segment> Search(int[] image, Bitmap canvas)

{

//задаем ширину, высоту, инициализируем массив пикселей

DraftLayer = newint[canvas.Width, canvas.Height];

\_width = canvas.Width;

\_height = canvas.Height;

//инициализируемсловарь

DicColors = newDictionary<int, Color>();

intn = 0;

//ищем все цвета и добавляем в словарь с ключом

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

{

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

{

//если словарь не содержит этот цвет

if (!DicColors.ContainsValue(canvas.GetPixel(j, i)))

{

DicColors.Add(image[n], canvas.GetPixel(j, i));

}

n++;

}

}

//переменные для перехода по строкам изображения

int p = 0, k = 0;

//формируем матрицу кластеризованного изображения

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

{

DraftLayer[k, p] = image[i];

k++;

if (k % \_width == 0 && i != 0)

{

p++;

k = 0;

}

}

//шагпоиска

inth = 0;

//инициализируемсписоксконтурами

List<Segment> segments = newList<Segment>();

//перебираем картинку по всем строкам

while (h < \_height)

{//функция поиска по строке

LinearSearch(h, 0, \_width, segments);

h += 1;

}

//выводимсегменты

foreach (var seg in segments)

{

try

{

seg.ColorSegment = DicColors[seg.Color];

}

catch (Exception)

{

seg.ColorSegment = Color.Blue;

}

}

return segments;

}

privatestaticvoid LinearSearch(int x, int y0, int y1, List<Segment> segments)

{ //поиcк контуров на отрезке [y0;y1] на строке х

do

{

//если точка есть уже в контуре

if (segments.Exists(seg => seg.Id == DraftLayer[y0, x]))

{ //ищемэтотконтур

Segment segmentFound = segments.Find(seg => seg.Id == DraftLayer[y0, x]);

//создаем список точек на прямой х

List<Point> list = segmentFound.Coord.FindAll(p => p.X == x && p.Y >= y0).OrderBy(p => p.Y).ToList();

//ищем точку, следующая за которой не будет принадлежать найденному контуру

y0 = list.Last().Y;

for (int j = 0; j < list.Count; j++)

{

if (list[j].Y < \_width - 1 && DraftLayer[list[j].Y + 1, x] != segmentFound.Id && DraftLayer[list[j].Y + 1, x] != segmentFound.Color)

{

y0 = list[j].Y;

j = list.Count;

}

}

}

else

{

//ищемновыйконтур

Segment segment = SearchSegment(y0, x);

//добавляем контур в список, если он правильный

if (segment.Count > 5 && segment.Color != MarkerPoint)

{

segments.Add(segment);

Marker(segment, segments);

}

}

y0++;

} while (y0 < y1);

}

privatestaticvoid Marker(Segment inputSegment, List<Segment> segments)

{ //копируемвходнойсегмент

Segment copyInputSegment = newSegment(inputSegment);

//сортируемточкивпорядкевозрастаниякоординатых

copyInputSegment.Coord = copyInputSegment.Coord.OrderBy(p => p.X).ToList();

//текущая строка

int x0;

//обходим все точки контура

for (int k = 0; k < copyInputSegment.Coord.Count && k != -1; k = copyInputSegment.Coord.FindIndex(p => p.X == x0 + 1))

{ //текущаястрока

x0 = copyInputSegment.Coord[k].X;

//список точек на текущей строке, принадлежащих контуру

List<Point> strPoints = inputSegment.Coord.FindAll(p => p.X == x0).OrderBy(p => p.Y).ToList();

//беремкрайниеточки

int y0 = strPoints.First().Y;

int y1 = strPoints.Last().Y;

//ищем между крайними точками

for (inti = y0; i<y1; i++)

{

//если точка цвета сегмента

if (DraftLayer[i, x0] == copyInputSegment.Color)

{ //закрашиваем её и добавляем в список точек сегмента

DraftLayer[i, x0] = copyInputSegment.Id;

inputSegment.Points.Add(newPoint(x0, i));

}

//еслиточкалежитнаконтуре

elseif (DraftLayer[i, x0] == copyInputSegment.Id)

{ // ищем точку, которая будет после себя иметь цвет сегмента

List<Point> list =

copyInputSegment.Coord.FindAll(p => p.X == x0 && p.Y >= i).OrderBy(p => p.Y).ToList();

for (int j = 0; j < list.Count; j++)

{

if (list[j].Y < \_width - 1 && DraftLayer[list[j].Y + 1, x0] == inputSegment.Color)

{

i = list[j].Y;

j = list.Count;

}

}

}

//если точка принадлежит контуру из найденных

elseif (segments.Exists(seg => seg.Id == DraftLayer[i, x0]))

{ //ищем точку, следующая за которой будет цвета сегмента

Segment temp = segments.Find(seg => seg.Id == DraftLayer[i, x0]);

List<Point> list =

temp.Coord.FindAll(p => p.X == x0 && p.Y >= i).OrderBy(p => p.Y).ToList();

i = list.Last().Y;

for (int j = 0; j < list.Count; j++)

{

if (list[j].Y < \_width - 1 && DraftLayer[list[j].Y + 1, x0] == inputSegment.Color)

{

i = list[j].Y;

j = list.Count;

}

}

}//еслинашлиновыйконтур

elseif (DraftLayer[i, x0] != copyInputSegment.Id)

{

//ищемконтур

Segment segment = SearchSegment(i, x0);

//если контур больше 4 точек и он не тупиковый

if (segment.Count > 5 && segment.Color != MarkerPoint)

{//добавляемвспискокконтур

segments.Add(segment);

//ищем точку для перехода к сделующей точке

List<Point> list =

segment.Coord.FindAll(p => p.X == x0 && p.Y >= i).OrderBy(p => p.Y).ToList();

i = list.Last().Y;

for (int j = 0; j < list.Count; j++)

{

if (list[j].Y < \_width - 1 &&

DraftLayer[list[j].Y + 1, x0] == inputSegment.Color)

{

i = list[j].Y;

j = list.Count;

}

}

}

}

}

}//помечаемконтуркакзакрашенный

inputSegment.Colored = true;

//закрашиваемновыеконтуры

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

{

if (!segments[i].Colored)

{

Marker(segments[i], segments);

}

}

}

privatestaticSegment SearchSegment(int y0, int x0)

{

//точки для обхода

int x = x0, y = y0;

//точки для проверки направления

int yPrev = y0, xPrev = x0, dY, dX;

//списоктупиковыхточек

List<Point> delPoints = newList<Point>();

//новыйконтур

Segment segment = newSegment(DraftLayer[y0, x0]);

//добавляем точку в контур

segment.Add(newPoint(x, y));

//делаем пробный шаг

MoveLeft(refy, refx);

//вычисляем начальное направление и запоминаем точку

dY = y - yPrev;

dX = x - xPrev;

yPrev = y;

xPrev = x;

//снимаем маркер с первой точки, чтобы была возожность снова в нее попасть

DraftLayer[y0, x0] = segment.Color;

//добавляем первую точку в контур

segment.Add(newPoint(x, y));

do

{

//определяем направление движения по часовой стрелке

if (dX == -1 && dY == 0)

{

MoveLeft(ref y, ref x);

}

elseif (dX == -1 && dY == 1)

{

MoveLeft(ref y, ref x);

}

elseif (dX == 0 && dY == 1)

{

MoveUp(ref y, ref x);

}

elseif (dX == 1 && dY == 1)

{

MoveUp(ref y, ref x);

}

elseif (dX == 1 && dY == 0)

{

MoveRight(ref y, ref x);

}

elseif (dX == 1 && dY == -1)

{

MoveRight(ref y, ref x);

}

elseif (dX == 0 && dY == -1)

{

MoveDown(ref y, ref x);

}

elseif (dX == -1 && dY == -1)

{

MoveDown(ref y, ref x);

}

//вычисляем направление и запоминаем предыдущую точку

dY = y - yPrev;

dX = x - xPrev;

yPrev = y;

xPrev = x;

//добавляем точку в контур, если направление поменялось

if (dX != 0 || dY != 0)

{

segment.Add(newPoint(x, y));

}

else

{

//ищем новый путь, если остались на месте

delPoints.AddRange(ReturnPoint(ref y, ref x, ref segment, y0, x0));

//если сегмент содержит пиксели, то ищем напаравление и продолжаем поиск

if (segment.Count > 1)

{

dY = y - segment.Coord[segment.Count - 2].Y;

dX = x - segment.Coord[segment.Count - 2].X;

yPrev = y;

xPrev = x;

}

else

{//всегмекнтенетуточек

//чистимтупиковыеточки

foreach (Point delPoint in delPoints)

{

DraftLayer[delPoint.Y, delPoint.X] = segment.Color;

}

returnnewSegment();

}

}

//пока не вернемся в исходную точку

} while ((y != y0 || x != x0));

//вовзращаем цвет удаленным точкам

foreach (Point delPoint in delPoints)

{

DraftLayer[delPoint.Y, delPoint.X] = segment.Color;

}

//помечаемточкиконтура

foreach (Point segPoint in segment.Coord)

{

DraftLayer[segPoint.Y, segPoint.X] = segment.Id;

}

return segment;

}

privatestaticList<Point> ReturnPoint(refint y, refint x, refSegment segment, int y0, int x0)

{//функция отката, если зашли в тупик

//список тупиковых точек на данном участке

List<Point> points = newList<Point>();

//добавляем точку в список удаленных

points.Add(newPoint(x, y));

//маркируем точку, в которой тупик

DraftLayer[y, x] = MarkerPoint;

//удаляем её из контура

segment.DeleteLast();

//беремпредыдущуюточку

int lastY = segment.Coord[segment.Count - 1].Y;

int lastX = segment.Coord[segment.Count - 1].X;

//ищем напаравление движения

int dX = x - lastX;

int dY = y - lastY;

//запоминаем координаты предыдущей точки

x = lastX;

y = lastY;

//возвращаем цвет точки, чтобы прололжить поиск

DraftLayer[y, x] = segment.Color;

do

{

//выбираем направление отката по часовой стрелке

if (dX == -1 && dY == 1)

{

MoveLeftAndUp(ref y, ref x);

}

elseif (dX == 0 && dY == 1)

{

MoveRight(ref y, ref x);

}

elseif (dX == 1 && dY == 1)

{

MoveRightAndUp(ref y, ref x);

}

elseif (dX == 1 && dY == 0)

{

MoveDown(ref y, ref x);

}

elseif (dX == 1 && dY == -1)

{

MoveRightAndDown(ref y, ref x);

}

elseif (dX == 0 && dY == -1)

{

MoveLeft(ref y, ref x);

}

elseif (dX == -1 && dY == -1)

{

MoveLeftAndDown(ref y, ref x);

}

elseif (dX == -1 && dY == 0)

{

MoveUp(ref y, ref x);

}

elseif (dX == 1 && dY == 1)

{

MoveRightAndUp(ref y, ref x);

}

//вычисляем направление и запоминаем точку

dX = x - lastX;

dY = y - lastY;

//запоминаем текущие координаты в качестве последних

lastX = x;

lastY = y;

//если координата поменялась, добавляем точку

if (dX != 0 || dY != 0)

{

segment.Add(newPoint(x, y));

}

else

{//иначеудаляемточкуимаркируем

DraftLayer[y, x] = MarkerPoint;

segment.DeleteLast();

//добавляем точку в список удаленных

points.Add(newPoint(x, y));

//если сегмент имеет точки

if (segment.Count > 0)

{

//ищемновыйпуть

lastY = segment.Coord[segment.Count - 1].Y;

lastX = segment.Coord[segment.Count - 1].X;

dX = x - lastX;

dY = y - lastY;

x = lastX;

y = lastY;

//снимаем маркет для продолжения поиска

DraftLayer[y, x] = segment.Color;

}

}//выполняем вовзрат до тех пор, пока не найдем новый путь. или пока не кончатся точки

} while (segment.Count > 0 && dX == 0 && dY == 0 && (y != y0 || x != x0));

return points;

}

}

privatestaticvoid MoveLeft(refint y, refint x)

{//ищем крайнюю точку по часовой стрелке

if (y > 0 && x > 0 && DraftLayer[y - 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

x -= 1;

}

elseif (x > 0 && DraftLayer[y, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

}

elseif (y < \_width - 1 && x > 0 && DraftLayer[y + 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

y += 1;

}

elseif (y < \_width - 1 && DraftLayer[y + 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

}

elseif (y < \_width - 1 && x < \_height - 1 && DraftLayer[y + 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

x += 1;

}

elseif (x < \_height - 1 && DraftLayer[y, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

}

elseif (x < \_height - 1 && y > 0 && DraftLayer[y - 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

y -= 1;

}

elseif (y > 0 && DraftLayer[y - 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

}

else

{

}

}

privatestaticvoid MoveRight(refint y, refint x)

{

//ищем крайнюю точку по часовой стрелке, приоритет при движении вправо

if (y < \_width - 1 && x < \_height - 1 && DraftLayer[y + 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

x += 1;

}

elseif (x < \_height - 1 && DraftLayer[y, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

}

elseif (x < \_height - 1 && y > 0 && DraftLayer[y - 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

y -= 1;

}

elseif (y > 0 && DraftLayer[y - 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

}

elseif (y > 0 && x > 0 && DraftLayer[y - 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

x -= 1;

}

elseif (x > 0 && DraftLayer[y, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

}

elseif (y < \_width - 1 && x > 0 && DraftLayer[y + 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

y += 1;

}

elseif (y < \_width - 1 && DraftLayer[y + 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

}

else

{

}

}

privatestaticvoid MoveUp(refint y, refint x)

{

//ищем крайнюю точку по часовой стрелке, приоритет согласно направлению движения вверх

if (y < \_width - 1 && x > 0 && DraftLayer[y + 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

y += 1;

}

elseif (y < \_width - 1 && DraftLayer[y + 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

}

elseif (y < \_width - 1 && x < \_height - 1 && DraftLayer[y + 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

x += 1;

}

elseif (x < \_height - 1 && DraftLayer[y, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

}

elseif (x < \_height - 1 && y > 0 && DraftLayer[y - 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

y -= 1;

}

elseif (y > 0 && DraftLayer[y - 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

}

elseif (y > 0 && x > 0 && DraftLayer[y - 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

x -= 1;

}

elseif (x > 0 && DraftLayer[y, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

}

else

{

}

}

privatestaticvoid MoveDown(refint y, refint x)

{

//ищем крайнюю точку по часовой стрелке, приоритет согласно направлению движения вниз

if (x < \_height - 1 && y > 0 && DraftLayer[y - 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

y -= 1;

}

elseif (y > 0 && DraftLayer[y - 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

}

elseif (y > 0 && x > 0 && DraftLayer[y - 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

x -= 1;

}

elseif (x > 0 && DraftLayer[y, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

}

elseif (y < \_width - 1 && x > 0 && DraftLayer[y + 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

y += 1;

}

elseif (y < \_width - 1 && DraftLayer[y + 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

}

elseif (y < \_width - 1 && x < \_height - 1 && DraftLayer[y + 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

x += 1;

}

elseif (x < \_height - 1 && DraftLayer[y, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

}

else

{

}

}

privatestaticvoid MoveRightAndDown(refint y, refint x)

{

//ищем крайнюю точку по часовой стрелке, приоритет согласно направлению движения вверх и влево

if (y > 0 && DraftLayer[y - 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

}

elseif (y > 0 && x > 0 && DraftLayer[y - 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

x -= 1;

}

elseif (x > 0 && DraftLayer[y, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

}

elseif (y < \_width - 1 && x > 0 && DraftLayer[y + 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

y += 1;

}

elseif (y < \_width - 1 && DraftLayer[y + 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

}

elseif (y < \_width - 1 && x < \_height - 1 && DraftLayer[y + 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

x += 1;

}

elseif (x < \_height - 1 && DraftLayer[y, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

}

elseif (x < \_height - 1 && y > 0 && DraftLayer[y - 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

y -= 1;

}

else

{

}

}

privatestaticvoid MoveLeftAndUp(refint y, refint x)

{

//ищем крайнюю точку по часовой стрелке, приоритет согласно направлению движения вверх и вправо

if (y < \_width - 1 && DraftLayer[y + 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

}

elseif (y < \_width - 1 && x < \_height - 1 && DraftLayer[y + 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

x += 1;

}

elseif (x < \_height - 1 && DraftLayer[y, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

}

elseif (x < \_height - 1 && y > 0 && DraftLayer[y - 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

y -= 1;

}

elseif (y > 0 && DraftLayer[y - 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

}

elseif (y > 0 && x > 0 && DraftLayer[y - 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

x -= 1;

}

elseif (x > 0 && DraftLayer[y, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

}

if (y < \_width - 1 && x > 0 && DraftLayer[y + 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

y += 1;

}

else

{

}

}

privatestaticvoid MoveRightAndUp(refint y, refint x)

{

//ищем крайнюю точку по часовой стрелке, приоритет согласно направлению движения вниз и влево

if (x < \_height - 1 && DraftLayer[y, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

}

elseif (x < \_height - 1 && y > 0 && DraftLayer[y - 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

y -= 1;

}

elseif (y > 0 && DraftLayer[y - 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

}

elseif (y > 0 && x > 0 && DraftLayer[y - 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

x -= 1;

}

elseif (x > 0 && DraftLayer[y, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

}

elseif (y < \_width - 1 && x > 0 && DraftLayer[y + 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

y += 1;

}

elseif (y < \_width - 1 && DraftLayer[y + 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

}

elseif (y < \_width - 1 && x < \_height - 1 && DraftLayer[y + 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

x += 1;

}

else

{

}

}

privatestaticvoid MoveLeftAndDown(refint y, refint x)

{

//ищем крайнюю точку по часовой стрелке, приоритет при движении вправо и вниз

if (x > 0 && DraftLayer[y, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

}

elseif (y < \_width - 1 && x > 0 && DraftLayer[y + 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x -= 1;

y += 1;

}

elseif (y < \_width - 1 && DraftLayer[y + 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

}

elseif (y < \_width - 1 && x < \_height - 1 && DraftLayer[y + 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y += 1;

x += 1;

}

elseif (x < \_height - 1 && DraftLayer[y, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

}

elseif (x < \_height - 1 && y > 0 && DraftLayer[y - 1, x + 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

x += 1;

y -= 1;

}

elseif (y > 0 && DraftLayer[y - 1, x] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

}

elseif (y > 0 && x > 0 && DraftLayer[y - 1, x - 1] == DraftLayer[y, x])

{

DraftLayer[y, x] = MarkerPoint;

y -= 1;

x -= 1;

}

else

{

}

}

}

}

publicstaticclassImageProcesing

{

//ширинаивысота

privatestaticint \_width;

privatestaticint \_height;

//списоксегментов

publicstaticList<Segment> ResultSegments { get;privateset; }

//списокмазков

publicstaticList<Smear> Smears { get;privateset; }

//результаткластеризации

publicstaticBitmap Result { get;privateset; }

//ошибка

publicstaticBitmap Error { get;privateset; }

//мазки

publicstaticBitmap Canvas { get;privateset; }

//векторы

publicstaticBitmap Vectors { get;privateset; }

//методформирующийкартупримитивов

publicstaticint MainMetod(Bitmap curPict, int rankFilter, int sizeSmear, BackgroundWorker sender, DoWorkEventArgs e)

{

//инициализируем параметры

//начальное число цветов для кластеризации

int сolorCount = 2;

//список сегментов для аппроксимации

ResultSegments = newList<Segment>();

//списокмазков

Smears = newList<Smear>();

//списоквсехнайденныхсегментов

List<Segment> segments = newList<Segment>();

//длина и ширина изображения

\_width = curPict.Width;

\_height = curPict.Height;

//совйства с результатами работы алгоритма

Vectors = newBitmap(curPict, curPict.Width \* 5, curPict.Height \* 5);

Canvas = newBitmap(curPict);

Error = newBitmap(curPict);

Result = newBitmap(curPict);

//цвет для отсеивания известных контуров

Color temp = Color.White;

//предыдущее значение найденыых точек

intstepPoints = 0;

//счетчикитераций

inti = 0;

//медианныйфильтр

curPict = Filtration.MedianFilter(rankFilter, curPict);

do

{

//алгоритмкластеризации

Canvas = Clusterization.KMeans(сolorCount, Canvas);

//алгоритмпоискасегментов

segments = Segmentation.Search(Clusterization.DraftLayerArray, Canvas);

//находим самый яркий цвет

if (i != 0)

{

temp = Segmentation.DicColors.ToList().Find(

c => c.Value.G == Segmentation.DicColors.Values.ToList().Max(r => r.G)

&& c.Value.R == Segmentation.DicColors.Values.ToList().Max(r => r.R)

&& c.Value.B == Segmentation.DicColors.Values.ToList().Max(r => r.B)

).Value;

}

//удаляембелыесегменты

segments.RemoveAll(seg => temp == seg.ColorSegment);

//добавляем сегменты в список, если они меньше размеров мазка

ResultSegments.AddRange(segments.FindAll(s => s.Coord.Count < sizeSmear));

//удаляем сегменты, если они меньше размера мазка

segments.RemoveAll(s => s.Points.Count < sizeSmear);

//заполняем холст исходными значениями

Canvas = newBitmap(curPict);

//очищаемизвестныесегменты

foreach (Segment resultSegment in ResultSegments)

{

foreach (Point point in resultSegment.Points)

{

Canvas.SetPixel(point.Y, point.X, Color.White);

}

foreach (Point point in resultSegment.Coord)

{

Canvas.SetPixel(point.Y, point.X, Color.White);

}

}

//количество цветов для кластеризации

сolorCount = segments.Count + i + 1;

//прогрессбар

int k = 99 \* ResultSegments.Sum(s => s.Points.Count + s.Coord.Count) / (\_width \* \_height);

if (k < 100)

{

sender.ReportProgress(k, null);

}

else

{

sender.ReportProgress(99, null);

}

//увеличиваемразмеркисти

if (Math.Abs(segments.Sum(s => s.Coord.Count) - stepPoints) < sizeSmear / 5)

{

sizeSmear = sizeSmear \* 6 / 5;

}

//количествонесегментированныхпикселей

stepPoints = segments.Sum(s => s.Coord.Count);

//отмена выполнения операции

if (sender.CancellationPending)

{

e.Cancel = true;

segments = newList<Segment>();

}

//увеличиваемсчетчикитераций

i++;

} while (segments.Count != 0);

//формируеммазки

foreach (Segment resultSegment in ResultSegments)

{

Smears.Add(newSmear(resultSegment));

}

//возвращаемчисломазков

return Smears.Count;

}

}

publicpartialclassMainForm : Form

{

//входноеизображение

privateBitmap \_currentPicture;

public MainForm()

{

InitializeComponent();

}

privatevoid ImagesPaint()

{

Graphics gVectors = Graphics.FromImage(ImageProcesing.Vectors);

Graphics gResult = Graphics.FromImage(ImageProcesing.Result);

Graphics gCanvas = Graphics.FromImage(ImageProcesing.Canvas);

gVectors.Clear(Color.White);

gCanvas.Clear(Color.White);

gResult.Clear(Color.White);

//рисуемпримитивныеконтуры

foreach (Segment resultSegment inImageProcesing.ResultSegments.OrderByDescending(s => s.Points.Count))

{

for (var i = 0; i < resultSegment.Coord.Count; i++)

{

resultSegment.Coord[i] = newPoint(resultSegment.Coord[i].Y, resultSegment.Coord[i].X);

}

gResult.FillPolygon(newSolidBrush(resultSegment.ColorSegment), resultSegment.Coord.ToArray());

gResult.DrawPolygon(newPen(resultSegment.ColorSegment), resultSegment.Coord.ToArray());

}

//рисуемвекторыимазки

foreach (Smear smear inImageProcesing.Smears.OrderByDescending(s => s.Points.Count))

{

Point vPoint1 = newPoint(smear.Vector.First().X \* 5, smear.Vector.First().Y \* 5);

Point vPoint2 = newPoint(smear.Vector.Last().X \* 5, smear.Vector.Last().Y \* 5);

gVectors.DrawLine(newPen(Color.Black), vPoint2, vPoint1);

gVectors.FillEllipse(newSolidBrush(Color.Red), vPoint1.X - 2, vPoint1.Y - 2, 4, 4);

gCanvas.FillPolygon(newSolidBrush(smear.Color), smear.NodePoints.ToArray());

gCanvas.DrawPolygon(newPen(smear.Color), smear.NodePoints.ToArray());

}

//рисуемошибку

for (int i = 0; i < \_currentPicture.Width; i++)

{

for (int j = 0; j < \_currentPicture.Height; j++)

{

Color canvasColor = ImageProcesing.Canvas.GetPixel(i, j);

Color inputColor = \_currentPicture.GetPixel(i, j);

Color errorColor = Color.FromArgb(255, Math.Abs(canvasColor.R - inputColor.R), Math.Abs(canvasColor.G - inputColor.G), Math.Abs(canvasColor.B - inputColor.B));

ImageProcesing.Error.SetPixel(i, j, errorColor);

}

}

}

privatevoid RunAlg\_Click(object sender, EventArgs e)

{//если объект выполняет операцию

if (bWorker.IsBusy)

{//отменяем её

bWorker.CancelAsync();

}

else

{//запускаем задачу

bWorker.RunWorkerAsync();

runAlg.BackgroundImage = Properties.Resources.cancel;

HelpMessage.SetToolTip(this.runAlg, "Отмена");

}

}

privatevoid OutImageList\_SelectedIndexChanged(object sender, EventArgs e)

{

switch (outImageList.SelectedIndex)

{

case 0:

outputImage.Image = ImageProcesing.Canvas;

break;

case 1:

outputImage.Image = ImageProcesing.Result;

break;

case 2:

outputImage.Image = ImageProcesing.Vectors;

break;

case 3:

outputImage.Image = ImageProcesing.Error;

break;

case 4:

outputImage.Image = \_currentPicture;

break;

default:

outputImage.Image = ImageProcesing.Canvas;

break;

}

}

privatevoid BWorker\_DoWork(object sender, DoWorkEventArgs e)

{

e.Result = ImageProcesing.MainMetod(newBitmap(\_currentPicture), (int)rankFilter.Value, (int)smearSize.Value, bWorker, e);

}

privatevoid BWorker\_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)

{

progress.Value = 100;

runAlg.BackgroundImage = Properties.Resources.run;

HelpMessage.SetToolTip(this.runAlg, "Разбитьизображениенамазки");

if (e.Error != null)

{//выводим сообщение об ошибке

MessageBox.Show(e.Error.Message);

}

elseif (e.Cancelled)

{//в случае отмены операции выводим сообщение

MessageBox.Show("операция была отменена!");

}

else

{

//выводим информацию

labelNumbSegments.Text = "Всего мазков: " + e.Result;

ImagesPaint();

OutImageList\_SelectedIndexChanged(null, e);

MessageBox.Show("Готово!");

}

}

privatevoid BWorker\_ProgressChanged(object sender, ProgressChangedEventArgs e)

{

progress.Value = e.ProgressPercentage;

}

privatevoid OutputImage\_DragDrop(object sender, DragEventArgs e)

{

int x = this.PointToClient(newPoint(e.X, e.Y)).X;

int y = this.PointToClient(newPoint(e.X, e.Y)).Y;

if (x >= outputImage.Location.X && x <= outputImage.Location.X + outputImage.Width && y >= outputImage.Location.Y && y <= outputImage.Location.Y + outputImage.Height)

{

string[] files = (string[])e.Data.GetData(DataFormats.FileDrop);

\_currentPicture = (Bitmap)Image.FromFile(files[0]);

outputImage.Image = \_currentPicture;

//устанавливаем размер мазка по умолчанию

smearSize.Value = (\_currentPicture.Width \* \_currentPicture.Height) /

(\_currentPicture.Width + \_currentPicture.Height) / 2;

}

}

privatevoid OutputImage\_DragEnter(object sender, DragEventArgs e)

{

e.Effect = DragDropEffects.Move;

}

privatevoid OutputImage\_Click(object sender, EventArgs e)

{

//считывание с файла

\_currentPicture = null;

openImageFromFile.Filter = "image files (\*.bmp)|\*.bmp|All files (\*.\*)|\*.\*";

openImageFromFile.FilterIndex = 2;

openImageFromFile.RestoreDirectory = true;

System.IO.Stream sr;

if (openImageFromFile.ShowDialog() == DialogResult.OK)

{

try

{

if ((sr = openImageFromFile.OpenFile()) != null)

{

\_currentPicture = newBitmap(sr);

outputImage.Image = \_currentPicture;

smearSize.Value = (\_currentPicture.Width \* \_currentPicture.Height) /

(\_currentPicture.Width + \_currentPicture.Height) / 2;

}

}

catch (Exception ex)

{

MessageBox.Show("Error: Could not read file from disk. Original error: " + ex.Message);

}

}

}

privatevoid OutputImage\_Paint(object sender, PaintEventArgs e)

{

if (outputImage.Image == null)

{

StringFormat drawFormatTitle = newStringFormat();

drawFormatTitle.Alignment = StringAlignment.Center;

drawFormatTitle.LineAlignment = StringAlignment.Center;

StringFormat drawFormatText = newStringFormat();

drawFormatText.Alignment = StringAlignment.Center;

drawFormatText.LineAlignment = StringAlignment.Center;

float x = outputImage.Height / 2;

float y = outputImage.Width / 3 + 30;

Image backgroundImage = Properties.Resources.loadImage;

e.Graphics.DrawImage(backgroundImage, x - backgroundImage.Height / 2, y + backgroundImage.Height / 3);

e.Graphics.DrawString("Загрузитьизображение", newFont("Bebas Neue Book", 25), newSolidBrush(Color.Gray), x, y - 10, drawFormatTitle);

e.Graphics.DrawString("Нажмитеилиперетащитефайлысюда", newFont("Bebas Neue Book Regular", 11, FontStyle.Italic), newSolidBrush(Color.Gray), x, y + 20, drawFormatText);

}

}

privatevoid ImageSave\_Click\_1(object sender, EventArgs e)

{

saveImage.DefaultExt = "bmp";

saveImage.Filter = "Image files (\*.bmp)|\*.bmp|All files (\*.\*)|\*.\*";

if (saveImage.ShowDialog() == DialogResult.OK)

{

outputImage.Image.Save(saveImage.FileName, ImageFormat.Bmp);

}

}

privatevoid ClearImage\_Click(object sender, EventArgs e)

{

//очищаемизображение

outputImage.Image = null;

}

privatevoid MainForm\_Load(object sender, EventArgs e)

{

//выводпосказок

HelpMessage.SetToolTip(this.runAlg, "Разбитьизображениенамазки");

HelpMessage.SetToolTip(this.ImageSave, "Сохранитьизображение");

HelpMessage.SetToolTip(this.ClearImage, "Очиститьизображение");

}

}