**РОССИЙСКИЙ ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ Г.В. ПЛЕХАНОВА**

Курсовая работа

по курсу «Теория разработки современных программных комплексов»

на тему «Создание 3D модели местности по 2D фотографии»

**Выполнил студент группы ДКО-131б:**

*Полевик Ю.С. ДКО-13186*

**Руководитель:**

*Чапкин Н.С.*

Москва 2015

Оглавление

[Введение 3](#_Toc440641588)

[*Цель задачи:* 3](#_Toc440641589)

[Задача 5](#_Toc440641590)

[*Постановка задачи:* 5](#_Toc440641591)

[*Реализация задачи:* 5](#_Toc440641592)

[1.Обработка 2D изображения фильтрами 5](#_Toc440641593)

[2. Деление 2D изображения на сегменты (объекты) 13](#_Toc440641594)

[3. Нахождение земли, заднего фона и областей сегментов 19](#_Toc440641595)

[4. Построение 3D модели с наложением текстуры (получение 3-ей координаты у объектов) 20](#_Toc440641596)

[5. Интерфейс и использованные сторонние библиотеки 21](#_Toc440641597)

[*Заключение* 24](#_Toc440641598)

[Список литературы 25](#_Toc440641599)

[Листинг 25](#_Toc440641600)

[1. Реализованные фильтры программы: 25](#_Toc440641601)

[2. Сегментация изображения 34](#_Toc440641602)

[3. Классы вершин, рёбер 40](#_Toc440641603)

[4. Анализатор 43](#_Toc440641604)

[5. Main программы 48](#_Toc440641605)

[6. Реализация форм 48](#_Toc440641606)

[6.1. 3DModelForm 48](#_Toc440641607)

[6.2. MainMenuForm 54](#_Toc440641608)

[6.3. SettingsForm 56](#_Toc440641609)

# Введение

## *Цель задачи:*

Используя одну 2D фотографию, необходимо построить её 3D модель местности.

На входе подаётся одна фотография, представляющая двумерное изображение какой-либо местности, содержащая в себе только x и y координаты каждого пикселя относительно фотографии в целом и их цвет. На выходе должна быть построена 3D модель данной местности, включающая в себя расположение объектов относительно друг друга и их ориентацию в трехмерном пространстве относительно точки, из которой делался фотоснимок. Для её построения необходимы координаты глубины объектов на изображении, но из одной 2D фотографии их получить невозможно, т.к. их не существует. Из чего можно сделать вывод, что входных данных недостаточно для автоматизации построения 3D модели (имеется ввиду построение модели не для одной конкретной фотографии, а для любой, которую предложит пользователь).

Для нахождения недостающих данных для построения 3D модели программа опирается на монокулярные признаки глубины.

Восприятие глубины, или дифференциальное восприятие расстояния, является результатом работы зрительных и слуховых анализаторов. Что касается зрения, то восприятие глубины основано на монокулярных и бинокулярных признаках. Бинокулярные признаки отражают совместную деятельность обоих глаз. Монокулярными являются признаки, которые могут восприниматься не только двумя, но и одним глазом, что сопоставимо с нашим 2D изображением местности.

К монокулярным признакам относят: окклюзию, знания о размерах того или иного объекта, параллакс движения, линейную перспективу, градиент текстуры и затенение.

Так как наша задача научить программу автоматически распознавать объекты и их расположение в пространстве, то признаки, основанные на человеческом факторе, нам не подходят. Рассмотрим подробнее некоторые монокулярные признаки.

**Окклюзия (интерпозиция)** – это признак, который обеспечивает возникновение эффекта глубины, когда один наблюдаемый предмет частично закрывает другой. Несмотря на то, что большая часть белого треугольника отсутствует, зрительная система «достраивает» изображение, чтобы воссоздать общую смысловую структуру, причем одновременно возникает иллюзия удаленности треугольника от трех черных кругов.



**Параллакс движения** – это монокулярный источник информации о глубине и взаимном расположении объектов, возникающий в результате перемещения наблюдателя или наблюдаемых объектов (например, при повороте головы).

**Линейная перспектива** – это планомерное уменьшение величины удаленных предметов и расстояний между ними, создающее впечатление глубины при восприятии плоскостных изображений.

Используя данные признаки, при обработке фотографии можно получить больше информации, необходимой для реализации поставленной задачи.

# Задача

## *Постановка задачи:*

- Имея только одну 2D фотографию построить её 3D модель местности.

- Нельзя использовать сторонние ресурсы с уже доступной реализацией необходимых алгоритмов (за исключением вывода 3D модели на экран, ввода данных и других деталей, связанных с интерфейсом программы).

## *Реализация задачи:*

Единого рабочего алгоритма решения поставленной задачи на данный момент в мире не существует, так как она считается нетривиальной в связи с отсутствием необходимого количества информации во входных данных (максимально достигнутый процент успешного выполнения машиной поставленной задачи - 60%). Программа включает в себя множество алгоритмов, связанных с компьютерным зрением, объединённых вместе для совместной реализации (некоторые будут реализованы в будущем в более поздних версиях программы).

Работу программы условно можно разделить на четыре части:

1. Обработка 2D изображения фильтрами
2. Деление 2D изображения на сегменты (объекты)
3. Нахождение земли, заднего фона и областей сегментов
4. Построение 3D модели с наложением текстуры (получение 3-ей координаты у объектов)

### 1.Обработка 2D изображения фильтрами

#### 1.1. GetBytes и GetBitmap

При получении 2D изображения местности, программа записывает его как Bitmap. Так как напрямую работать с пикселями в Bitmap нельзя, в программе реализованы два метода работы с фотографией: GetBytes и GetBitmap. GetBytes выделяет напрямую место под массив байтов, по 8 бит на каждый цветовой канал пикселя (24 в сумме на пиксель). GetBitmap конвертирует массив байтов обратно в Bitmap, что необходимо для вывода изображения. У Bitmap есть способ получения и замены пикселя напрямую, но он считается устаревшим и не оптимальным по скорости и ресурсам программы.

Таким образом, используя GetBytes, мы получаем из Bitmap-объекта массив байтов, в котором значения каналов (R, G, B) каждого пикселя записываются подряд. Информация об одном пикселе изображения занимает 3 элемента массива. Метод GetBitmap, в свою очередь, проверяет, делится ли входящий массив байтов на 3, чтобы избежать ошибок конвертации.

#### 1.2. «Свёртка»

Свёртка - это операция вычисления нового значения заданного пикселя, при которой учитываются значения окружающих его соседних пикселей. Её используют для большинства существующих фильтров обработки изображений (например, размытие, повышение резкости и т.д.). Главным элементом свёртки является ядро свёртки. Оно представляет собой матрицу, обычно квадратную, размера n\*n, где n – нечётное число (в программе используется стандартный размер ядра свёртки 3\*3).

Важным элементом ядра считается его якорь (я), который находится в центре.

[ ][ ][ ]

[ ][я][ ]

[ ][ ][ ]

Данное ядро поочерёдно проходит по всем пикселям изображения, где каждый пиксель попадает на место якоря. Соседние пиксели также накрываются ядром.

Далее вычисляется сумма произведений значения пикселя со значением соответствующего элемента ядра (который его покрывает). Полученную сумму делят на сумму всех элементов ядра и записывают как новое значения пикселя, лежащего в якоре. Таким образом, в зависимости от значений элементов ядра, получаются различные эффекты.

Например, имеется данный набор пикселей и ядро свёртки:

[8][3][3][ ][ ][ ][ ][ ] [0][0][0]

[9][7][2][ ][ ][ ][ ][ ] [0][0][0]

[1][6][3][ ][ ][ ][ ][ ] [0][1][0]

[ ][ ][ ][ ][ ][ ][ ][ ]

Посчитаем новое значение пикселя, где ранее было значение 7, после наложения ядра свёртки:

(8\*0+3\*0+3\*0+9\*0+7\*0+2\*0+1\*0+6\*1+3\*0)/1 = 6

Результат:

[8][3][3][ ][ ][ ][ ][ ]

[9][6][2][ ][ ][ ][ ][ ]

[1][6][3][ ][ ][ ][ ][ ]

[ ][ ][ ][ ][ ][ ][ ][ ]

#### 1.3. «GrayImage»

Данный метод переводит наше изображение, имеющее изначально 3 цветовых канала, в градацию серого. Это необходимо для реализации некоторых алгоритмов, которые работают только с серыми оттенками. Например, детектор границ Кэнни (Canny). Также, в последующей доработке программы, изображение в серых тонах можно использовать для дополнительного анализа расположений объектов на изображении (опираясь на «палитру яркости» пикселей).

Данный алгоритм работает по формуле, которая домножает на определённый коэффициент значение каждого канала пикселя и суммирует их:

GrayColor = R\*0.299 + G\*0.587 + B\*0.114;

Полученное число записывается в каждый канал пикселя (R, G, B). Такое преобразование проводится для всех пикселей изображения.

Важно отметить, что в программе все преобразования над пикселями изображения проводятся с помощью редактирования не Bitmap-объекта, а массива байтов, который содержит в себе всю информацию о пикселях (см. пункт 1.1. GetBytes и GetBitmap).

#### 1.4. «Фильтр Гаусса»

Фильтр Гаусса (Gaussian filter) используется для обработки изображения с целью снижения уровня шума. Его главной идеей является усреднение значения пикселей вокруг главного пикселя по закону Гаусса. Т.к. в нашей программе уже реализована свёртка, то можно воспользоваться ей.

Гауссов фильтр убирает с изображения артефакты, избавляется от шумов, но также делает изображение более размытым. Но нам достаточно этого для выполнения поставленной задачи.

Использованные значения элементов свёртки для фильтрации по Гауссу в программе:

[0][1][0]

[1][4][1]

[0][1][0]

Фильтр Гаусса использует оператор Собеля и детектор границ Кэнни.

#### 1.5. «Оператор Собеля»

Реализация Собеля в программе несколько изменена с целью получения нужных для задачи данных. Далее будет пояснение самого алгоритма, а после его реализация в программе.

***Работа алгоритма:***

Оператор Собеля - это дискретный дифференциальный оператор, вычисляющий градиент яркости изображения в каждой точке. Так находится направление наибольшего увеличения яркости и величина её изменения в этом направлении. Результат показывает, насколько «резко» или «плавно» меняется яркость изображения в каждой точке.

Производная функции в точке - это скорость изменения функции в данной точке, т.е. предел отношения приращения функции к приращению ее аргумента при стремлении приращения аргумента к нулю. Получается, что, в нашем случае, производная — это отношение значения приращения пикселя по y к значению приращению пикселя по x:

Оператор Собеля использует два ядра 3x3, с которыми свёртывает исходное изображение для вычисления приближенных значений производных по горизонтали и по вертикали:

по горизонтали по вертикали

[-1][0][1] [-1][-2][-1]

[-2][0][2] [ 0][ 0][ 0]

[-1][0][1] [ 1][ 2][ 1]

Работая с изображением I, мы работаем с функцией двух переменных I(x,y). В нашем случае, более правильно будет говорить не о производной, а о градиенте изображения.

Градиент - вектор, показывающий направление возрастания некоторой величины, значение которой меняется от одной точки пространства к другой.

Итак, градиент для каждой точки изображения - двумерный вектор, компонентами которого являются производные яркости изображения по горизонтали и вертикали:

В каждой точке изображения градиентный вектор ориентирован в направлении наибольшего увеличения яркости, а его длина соответствует величине изменения яркости.

Таким образом, с помощью Собеля мы можем получить вектор для каждой точки, задающийся длиной и направлением (угол между вектором и осью x), вычисляющиеся по формулам:

длина

направление

Оператор Собеля высчитывает неточное приближение градиента изображения, но достаточного для практического применения в детекторе границ Кэнни.

***Реализация в программе:***

В программе данный алгоритм служит для нахождения границ различных объектов на изображении. Предполагается, что граничные пиксели отличаются от соседних резкой сменой яркости. В пример можно привести кирпичный дом, стоящий на зелёной траве. При проверке пикселя, являющегося самым нижним пикселем дома, будет заметен перепад от коричневого цвета к зелёному. Оператор выделит этот пиксель, как, предположительно, граничный (нахождение градиентов).

В реализации есть настраиваемый пользователем коэффициент «rangeLimit», который задаёт какое-то конкретное значение, определяющее достаточно ли высокая смена яркости у пикселя или нет. Чем больше данный коэффициент, тем меньше границ находит алгоритм и тем тоньше становятся найденные границы.

Если лимит был превышен, то в массив результата вносится три подряд значения «0» (три – т.к. все данные о пикселях хранятся в массиве байта, занимая три элемента; см. пункт 1.1. метод GetBytes). В противном случае вносится три подряд значения «255».

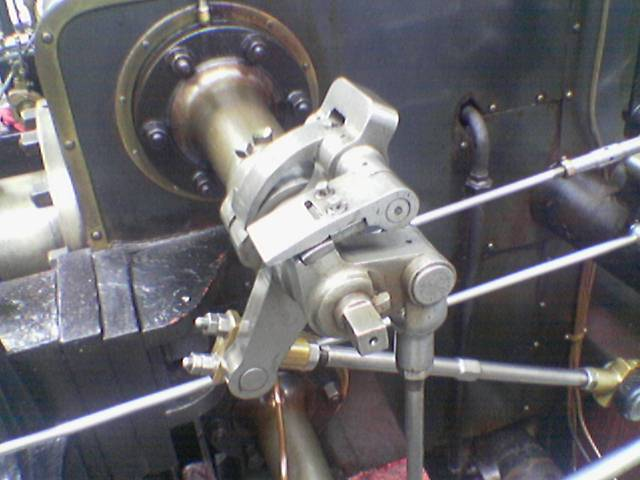
Таким образом, результатом работы оператора Собеля будет изображение, содержащее только границы найденных объектов.

Плюсом такой реализации будет то, что пользователь может сам настроить лимитное значение, что нам и нужно для настройки полученных результатов, по которым в дальнейшем выстраивается 3D модель. Алгоритм всё также способен получать необходимые величины (длину градиентного вектора и направление), которые уже после обрабатываются детектором границ Кэнни, получая более качественный результат.

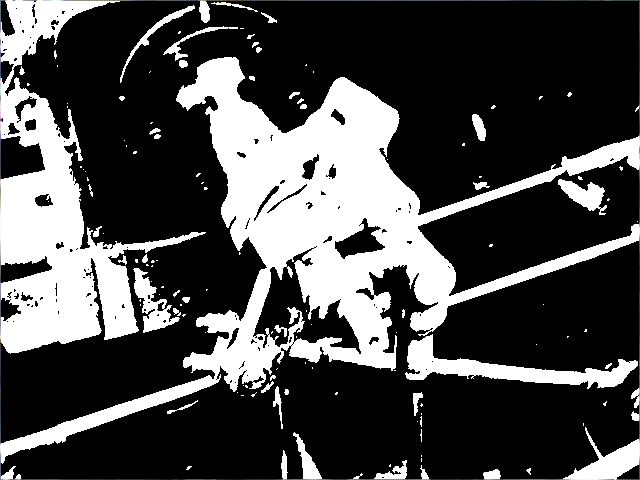
***Сравнение работы оператора Собеля с методами Робертса и Превитта:***

При выборе метода нахождения границ объектов на изображении было проведено сравнение различных методов по их результатам и оценивание качества полученных границ.

Исходное изображение:

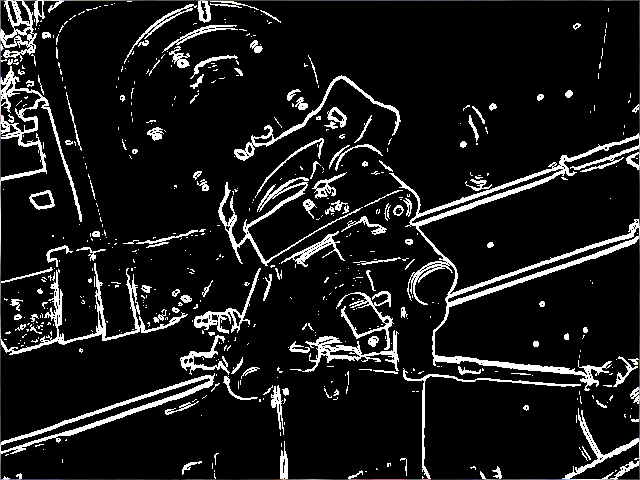
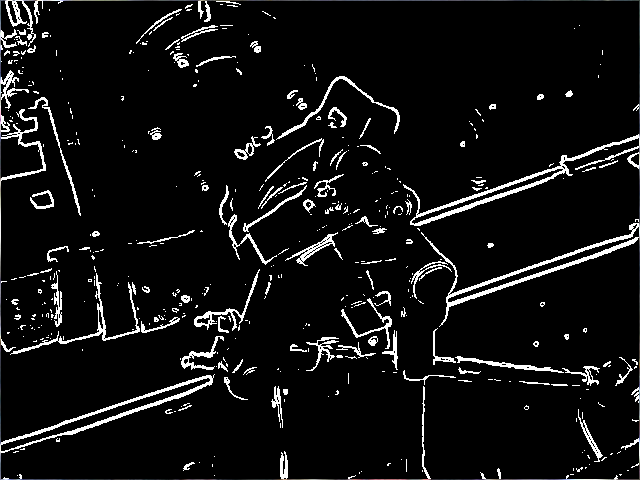


Результат, полученный методом Робертса:



Результат, полученный методом Превитта:

Результат, полученный оператором Собеля:



Рассматривая полученные варианты, можно сделать вывод, что оператор Собеля получает более качественный результат, по сравнению с методом Превитта. А результат, полученный методом Робертса, не соответствует поставленной задаче (вывод только границ объектов), здесь же присутствует их частичная сегментация.

#### 1.6. «Детектор границ Кэнни (Canny)»

Данный детектор был изобретён Кэнни ещё в1986-ом году, но он до сих пор является одним из лучших детекторов.

Алгоритм Кэнни ориентирован на работу с изображениями в градиенте серого. Существует доработка метода, позволяющая объединять работу алгоритма по трём цветовым каналам при помощи Якобиана. Это позволяет устранить проблему потери части информации цветного изображения, которая необходима для нахождения границы между пикселями, обладающими одинаковой светимостью.

Например, для такой картинки (левая) стандартный детектор Кэнни ничего не выделит (средняя). Но с доработкой при помощи матрицы Якоби Кэнни определит пиксели как разные (справа).



Эта доработка хороша, когда поставлена задача получения максимально качественных границ. В нашем случае такая качественность не требуется.

Работа детектора границ Кэнни делится на следующие шаги:

— Удаление шум и лишних деталей из изображения

— Расчёт градиента изображения

— Преобразование краёв в более тонкие

— Связывание краёв в контур

Сначала, используя фильтр Гаусса, детектор границ Кэнни удаляет артефакты и шумы, сглаживая изображение, так как он восприимчив к шумам.

Далее, пользуясь полученными данными оператором Собеля, Кэнни анализирует данные, пропуская их через 4 фильтра, чтобы распознать горизонтальные, вертикальные и диагональные границы.

Оператор Собель передаёт два нужных для этого парметра каждого пикселя: длину градиентного вектора (границы) и угол направления.

Угол направления границы Кэнни округляет до одного из четырех углов, представляющих вертикаль, горизонталь и две диагонали (в нашей программе это углы: 0, 45, 90 и 135 градусов).

Происходит проверка, превышает ли длина градиентного вектора каждого пикселя длины определённых соседних пикселей (соседи выбираются в зависимости от угла направления). Реализация происходит через проверку 4-ёх условий:

- если угол направления градиента равен 0, точка будет считаться границей, если её интенсивность больше чем у точки выше и ниже рассматриваемой точки

- если угол направления градиента равен 90 градусам, точка будет считаться границей, если её интенсивность больше чем у точки слева и справа рассматриваемой точки

- если угол направления градиента равен 45 градусам, точка будет считаться границей, если её интенсивность больше чем у точек находящихся в верхнем правом и нижнем левом углу от рассматриваемой точки

- если угол направления градиента равен 135 градусам, точка будет считаться границей, если её интенсивность больше чем у точек находящихся в верхнем левом и нижнем правом углу от рассматриваемой точки

Таким образом, получается двоичное изображение, содержащее тонкие края объектов.

Детектор границ Кэнни до сих пор является одним из лучших алгоритмов для нахождения границ объектов на изображении, а также опирается на уже реализованные в программе методы. Поэтому было принято решение реализации именно этого алгоритма.

### 2. Деление 2D изображения на сегменты (объекты)

Для реализации главной задачи курсовой работы, сегментация играет очень важную роль. При помощи сегментирования изображения, программа получает основную часть информации, необходимую для создания 3D модели, отделяя предположительно разные объекты изображения друг от друга.

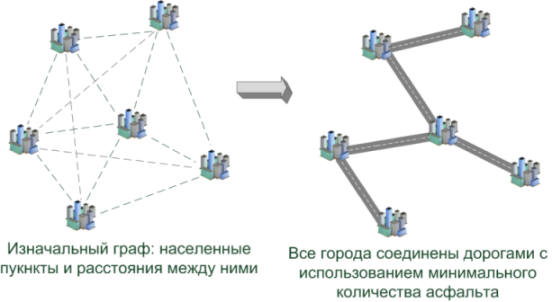
За основу был взят алгоритм сегментирования с помощью графов, но итоговая реализация отличается и была написана конкретно для решения поставленной задачи.

#### 2.1. «Efficient Graph-Based Image Segmentation»

Данный алгоритм был написан на языке программирования C++ в 2004 году. Его авторами являются Pedro F. Felzenszwalb и Daniel P. Huttenlocher. Алгоритм до сих пор остается популярным, демонстрируя неплохие результаты в плане производительности.

Сам алгоритм опирается на алгоритм Краскала и Disjoint-set data structure.

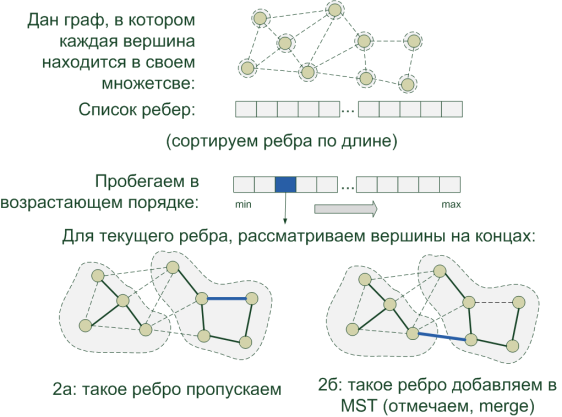
Алгоритм Краскала строит каркас — минимальное дерево данного графа MST (minimum spanning tree). Главной его задачей является решение проблем, подобных таким: на карте расположено несколько населенных пунктов, необходимо соединить все из них друг с другом таким образом, чтобы суммарная длина дорог была минимальной.



В текущей постановке задачи вершины графа представляют собой города, а ребра - дороги между городами. Нам известны расстояния между парами городов – это вес ребер. Нужно найти MST, т.е. такое дерево в графе (без циклов), чтобы сумма ребер была минимальной и при этом все вершины были достижимы. В нашем же случае за города выступают пиксели, за дороги – ребра с перепадом яркости (их весами) между пикселями, которые они соединяют.

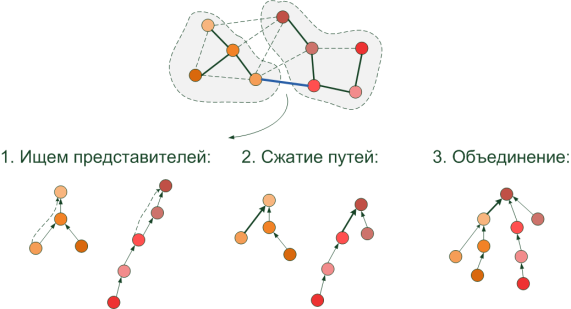
Изначально каждая вершина (пиксель) является единственным представителем соответствующего «оптимального» множества. В ходе алгоритма будет происходить объединение этих множеств в одно единственное, образуя дерево вершин, где для каждого пикселя указан его предок (некий схожий по цвету пиксель в том же сегменте). Для этого сортируются все имеющиеся в графе ребра в порядке возрастания длины (перепада яркости).

Далее, мы начинаем идти по рёбрам в порядке возрастания и соединять соседние пиксели похожего цвета (у них перепад яркости будет маленьким) в один сегмент, если они ещё не соединены (для этого происходит проверка самого верхнего пикселя - корня дерева, «представителя» данного сегмента на текущий момент). Т.к. все ребра меньшей длины уже рассмотрены, то дешевле, чем этим ребром, объединить сегменты не получится. Такое ребро заносится в список ребер, использованных при построении дерева MST, а множества объединяются в одно.



Если встреченные вершины уже принадлежат одному множеству, то значит, что они уже участвуют в некотором образованном подмножестве микро-MST, внутри их подграфа сумма ребер уже минимальна. Такое ребро пропускается.

Также, для того, чтобы каждый раз не бегать далеко от пикселя к корню, после того, как «представитель» будет успешно обнаружен, устанавливается прямая ссылка из пикселя сразу на него.



В случае задач, решающихся алгоритмом Краскала, цикл объединения продолжается до получения одного единственного множества, равного искомому MST, в котором будут присутствовать все вершины графа. В итоге работы алгоритма все вершины объединяются в одно единственное множество вершин (MST) и список ребер, использованных для его объединения, который представляет собой дерево минимальной суммарной длинны, объединяющее все вершины.

Для реализации понятия «множество» в алгоритме Краскала применяется структура данных Disjoint-set data structure. Работа с ней оптимизирована на выполнение двух основных операций:

- узнать для некоторой вершины, какому множеству в текущий момент она принадлежит

- объединить эти множества в одно

В ходе выполнения алгоритма сегментации образуется дерево вершин, где для каждого пикселя указан его предок (некий схожий по цвету пиксель в том же сегменте).

Таким образом, на промежуточном этапе мы будем иметь несколько разрозненных сегментов (подмножеств пикселей), с минимальным суммарным весом ребер внутри: сегменты будут объединены ребрами минимальной длины, т.е. с минимальными «перепадами интенсивностей» между соседними пикселями. Поэтому соседние пиксели внутри одного сегмента будут схожи по цвету.

Для определения различия между сегментами, с каждым уже построенным сегментом мы ассоциируем некоторую величину – максимальный перепад интенсивностей внутри него, т.е. самое длинное ребро в MST внутри сегмента:

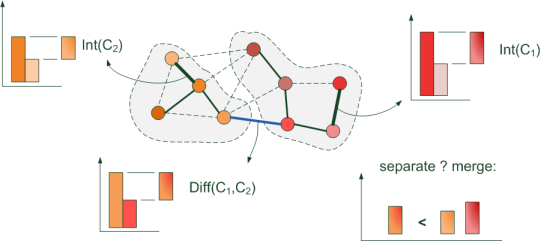
https://habrastorage.org/getpro/habr/post_images/6c3/5a9/2fd/6c35a92fd5bb3900af76e0c2c5860683.png

Достаточно просто сохранять длину ребра, добавляемого при объединении составляющих «подсегментов». Ведь на момент объединения длина добавляемого ребра была больше, чем в уже построенных MST каждого «подсегмента», т.к. ребра обрабатываются в возрастающем порядке.

Получаем решающее правило для сегментов:

https://habrastorage.org/getpro/habr/post_images/fdf/7a5/a0f/fdf7a5a0f558d400ba3612959c2d3803.png

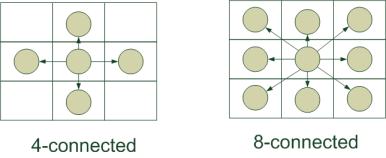
Получается что, для того чтобы сегменты «объединились», перепад интенсивностей на их границе должен быть меньше максимального перепада внутри каждого из объединяемых сегментов:



Для построения графа по изображению и вычисления, какие пиксели считаются соседними, данный алгоритм предлагает 2 варианта:

- каждый пиксель соединяем с соседними сверху / снизу / слева / справа

- дополнительно к предыдущему варианту каждый пиксель соединяем с соседними, находящимися по диагонали



В качестве «расстояния» для обработки цветных изображений авторы алгоритма берут простую разницу цвета пикселей:

https://habrastorage.org/getpro/habr/post_images/4e5/c2f/b7c/4e5c2fb7c5adadc434f491a6f37d4c78.png

В альтернативном варианте, авторы предлагают рассматривать не только близлежащие пиксели, но и дотягиваться до продолжения сегмента, находящегося на некотором удалении. Для этого в качестве длины ребра авторы предлагают выбрать Евклидово расстояние, зависящее как от положения пикселей (x,y), так и от их цвета (r, g, b): https://habrastorage.org/getpro/habr/post_images/f4a/c23/e91/f4ac23e91b6fd7bbfa7a208034a75f55.png

Это позволит избавиться от чрезмерной сегментации, и находить один объект, даже если он разделён другим тонким сегментом. Для построении графа, авторы предлагают каждый пиксель соединять с 10 (20, 30) соседними. Это дает более качественную сегментацию, но требует больше вычислительных ресурсов.

#### 2.2. Реализованный алгоритм сегментации

Реализованный в программе алгоритм имеет некоторые отличия от описанного выше. Его можно поделить на несколько шагов:

1. Создание списка рёбер (каждая вершина, не учитывая «граничные», входит в 12 рёбер)
2. Объединение пикселей в сегменты
3. Повторное объединение малых сегментов в большие, чтобы избавиться от излишней сегментации (реализовано два способа: проверка ведётся по «большим» или «малым» рёбрам)

Соединение пикселей в сегменты происходит на основе сравнения цены рёбер с выбранным лимитом (пользователь сам может его настроить). Это позволяет вручную задавать необходимые параметры, тем самым подстраивая работу фильтров для любого изображения, влияя на качество итоговой 3D модели.

Сравнение, приведённое в исходном алгоритме (через максимальные длины рёбер во всём сегменте) был реализован, но оказался слишком затратным и некачественным, потому не используется в конечном алгоритме сегментации. В алгоритме реализовано повторное прохождение по рёбрам («большим» или «малым») с целью устранения слишком малых сегментов. Для этого используется вторая лимитная переменная, которую пользователь сам может настроить при необходимости.

Также реализовано, но не используется сравнение цены рёбер с коэффициентом, зависимым от настроенного лимита и уже достигнутого количества пикселей в сегменте на момент сравнения, т.к. это сильно сказывалось на качестве сегментации.

Была проделана колоссальная работа по оптимизации алгоритма. Сначала получение сегментов изображения занимало несколько часов работы (точное время получить не удалось). После первой оптимизации, алгоритм стал работать по 20-25 минут на изображение. Далее удалось достичь работы в 2-3 минуты. Но в итоговых оптимизациях, его работа стала занимать около 5 секунд! Стоит отметить, что основным тестовым изображением, было изображение размером (400х600). Время выполнения алгоритма увеличивается в зависимости от размера изображения и объектов на нём, т.к. напрямую зависит от количества получаемых рёбер и сегментов.

***Сравнение алгоритмов***

Было принято решение использовать сегментацию на основе метода работы с графами, т.к. такая структура построения алгоритма считается выигрывающей по производительности, но несколько проигрывающей по качеству, например, пирамидному методу сегментации изображения.   
 Реализованный алгоритм сегментации в программе является уникальным (созданным вручную). Он несколько проигрывает взятому за основу алгоритму в производительности (стоит отметить, что «Efficient Graph-Based Image Segmentation» написан на языке C++, являющимся более производительным языком программирования изначально), но тестирование его работы показало, что полученные результаты достаточно качественны, а время выполнения сегментирования изображения максимально оптимизировано и подходят для решения поставленной задачи.

### 3. Нахождение земли, заднего фона и областей сегментов

После сбора всей возможной информации об изображении, программа переходит к её анализу. Как и говорилось ранее, получение z-координаты там, где её не существует, невозможно. Единственным решением является опора на монокулярные признаки глубины, используя которые можно научить программу правильно «видеть» местность на изображении. Т.к. в задаче требуется автоматизация процесса переделывания 2D фотографии в 3D модель, должна присутствовать огромная база, сравнимая с человеческим мышлением (т.к. большую часть информации о местности перед нами мы берём из того, что «знаем» размеры объекта).

Анализатор программы разделяет все сегменты на три вида:

1. Земля
2. Задний фон
3. Все остальные сегменты объектов

Сначала ищется сегмент земли, т.к. на нём строится вся 3D модель и задаётся количество z-координат. Производится поиск корня (Root-а) самого большого сегмента в нижнем ряду пикселей и принадлежащий соответствующему сегменту элемент.

После формируется граница земли (граница сегмента) и вся область, входящая в неё. Деление сегментов на области необходимо в случае, если наш сегмент разделён на части другими сегментами. После земля добавляется в список обработанных областей - у неё будет индекс 0 (хранение списка всех пикселей, являющихся частью границы земли, происходит через лист листов, где каждый элемент листа является листом с вершинами).

Похожая обработка происходит с задним фоном. Разница заключается в том, что производится поиск корня (Root-а) самого большого сегмента в верхнем ряду пикселей (так как теперь мы ищем «небо») и принадлежащий соответствующему сегменту элемент. В списке обработанных областей у заднего фона индекс 1.

После обработки земли и заднего фона, обрабатываются все остальные выделенные сегменты, поочерёдно добавляясь в список обработанных областей.

Нахождение границ сегмента происходит через метод **FindBorder**. Он заключается в прохождении всей границы, поочерёдно цепочкой добавляя вершины в лист. Метод выкидывает пиксельные «дорожки», тупики, которые могут помешать построение 3D модели.

Метод **FormSingleArea** находит все пиксели, принадлежащие проверенной области, включая тупиковые «дорожки» пикселей. Но так как граница их не обрабатывает, то при моделировании они игнорируются. Все проверенные пиксели вносятся в массив «проверенных пикселей», чтобы при обработке пикселя, входящего в уже проверенную область, она не обрабатывалась ещё раз.

Таким образом, мы получаем готовый список списков пикселей, являющихся границами различных областей сегментов, отделённые от всех землю и задний фон.

Эти данные получает метод **Set3DModel**, который вычисляет z-координату, строит 3D модель и накладывает текстуры.

### 4. Построение 3D модели с наложением текстуры (получение 3-ей координаты у объектов)

Последний этап работы программы выполняет поставленную задачу курсовой работы. На этом этапе при помощи полученных данных вычисляется z-координата объектов, строится их модель и накладывается текстура.

Т.к. наша модель постоянно отрисовывается, чтобы не рассчитывать области на каждом шаге и не высчитывать заново z-координаты, в программе присутствует флажок, активирующийся при первом расчёте.

Отображение 3D моделей в SharpGL, координаты текстуры и хранение вершин в матрице отличаются. Для их соответствия вводится ещё один массив, содержащий информацию о готовых к отрисовке вершинах.

Земля и задний фон отрисовываются отдельно от остальных областей. При отрисовывании земли высчитываем её самую дальнюю вершину, чтобы определить максимальное значение z-координаты.

Задний фон получает самую дальнюю z-координату земли (т.к. является самым дальним объектом 3D модели).

Для всех других объектов создаются свои новые вершины и заносятся в новую матрицу. X-координата приравнивается к бесконечности, чтобы можно было точно определить, переписали ли мы координаты для данной вершины или нет.

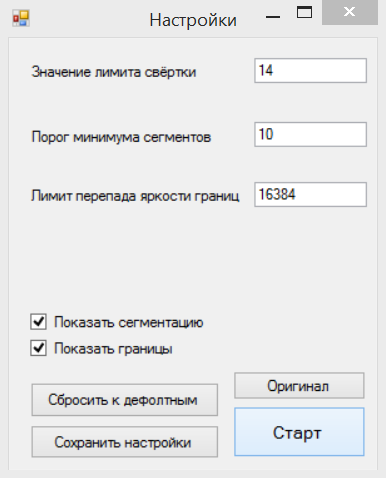
После заполнения матрицы программа проходит по всем граничным вершинам, проверяя, высчитали ли мы ей новые координаты для построения модели и наложения текстуры.

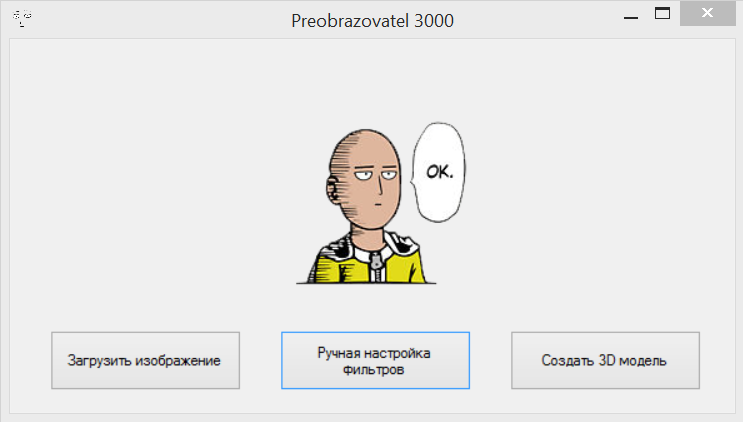
В конечном итоге, для всех списков граничных вершин (кроме земли и заднего фона) производится отрисовка на 3D модель, используя высчитанные новые координаты.

Когда алгоритм завершается, программа выводит полученную 3D модель на экран через подключённый интерфейс SharpGL.

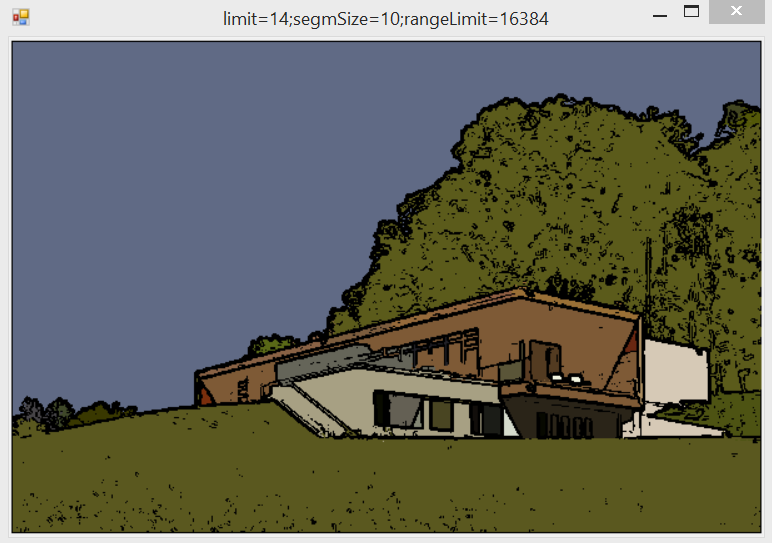
### 5. Интерфейс и использованные сторонние библиотеки

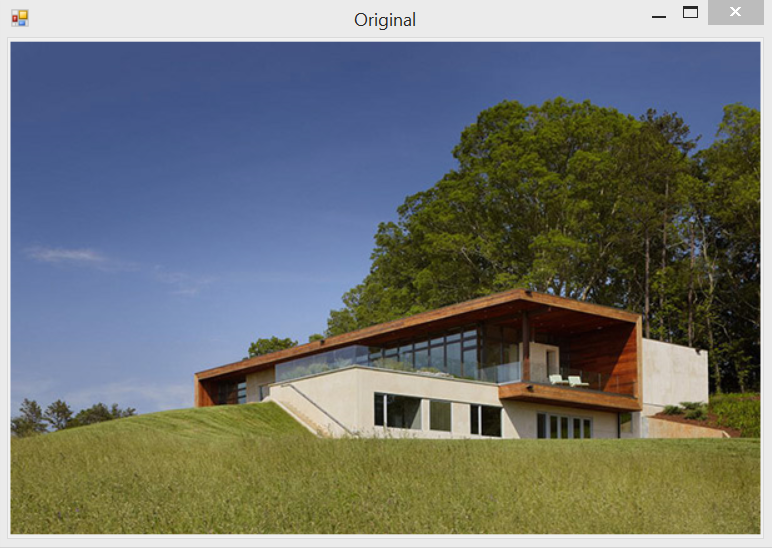
Для реализации интерфейса программы были использованы формы (Windows Forms).





Для вывода работы фильтров использовался элемент управления PictureBox.





Для работы с 3D моделью и для её вывода, к программе подключён SharpGL (реализация библиотеки OpenGL на C#).

OpenGL (Open Graphics Library) - спецификация, определяющая независимый от языка программирования платформонезависимый программный интерфейс для написания приложений, использующих двумерную и трёхмерную компьютерную графику.



## *Заключение*

Поставленная задача является нетривиальной, т.к. чтобы получить 3D модель местности, разработчики восстанавливают трехмерную информацию, комбинируя несколько (два и более) снимков одного и того же объекта. Восстановление трехмерной модели сцены всего из одного фотоснимка до сих пор не имеет 100% качественной реализации.

Определённые успехи были достигнуты в проекте Стэндфордского университета "Make3D", реализованным профессором Ashutosh Saxena, профессором Andrew Y. Ng, и другими членами Стэндфордской команды 3D реконструкции (в практической реализации удалось добиться 60% удовлетворительных результатов).

Данная проблема основывается на нехватке данных для построения 3D модели, т.к. 2D изображение не содержит нужной информации.

Данную проблему можно решить созданием анализатора, который будет обрабатывать огромное количество возможных условий для определения расположения объекта относительно пространства. Другими словами, программу следует обучить «видеть» предметы на изображении.

Альтернативным вариантом реализации задачи может быть подключение к программе человеческого фактора, то есть создания подходящего интерфейса, с помощью которого пользователь сможет указывать программе на объекты, а она обрабатывать полученные данные для заполнения недостающей информации.

# Список литературы

1. <http://habrahabr.ru/post/111175/> - статья «Знакомимся с OpenGL»
2. <http://robocraft.ru/blog/computervision/264.html> - «OpenCV шаг за шагом»
3. <http://www.algorithmist.ru/2011/03/colorful-image-edge-detection.html> - доработка детектора границ Кэнни
4. <http://habrahabr.ru/post/81279/> - «Эффективная сегментация изображений на графах»
5. <http://www.psyworld.info/monokulyarnye-priznaki-glubiny> - «Монокулярные признаки восприятия глубины»

# Листинг

## Реализованные фильтры программы:

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Drawing.Imaging;

using System.Linq;

using System.Runtime.InteropServices;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace CurseWork\_2D3D

{

// методом Кэнни находим лайнарт

public class Filters

{

private int height;

private int width;

public Bitmap \_Photo;

private int limit;

// для Кэнни

private byte[] DlBytes;

private byte[] AtanBytes;

public Filters(Bitmap Photo2D)

{

\_Photo = Photo2D;

height = Photo2D.Height; // высота фото

width = Photo2D.Width; // ширина фото

limit = MainMenuForm.\_trueRangeLimit;

//AtanMass = new int[height\*width];

/\*

// Всё домножается на 3, т.к. мы проверяем по 3-ём палитрам

double[,] kernel = {{-3, 0, 3}, {-6, 0, 6}, {-3, 0, 3}};

foto2 = Filters.Svertka(foto1, height, width, kernel);\*/

}

public Filters(Bitmap Photo2D, int limitR)

{

\_Photo = Photo2D;

height = Photo2D.Height; // высота фото

width = Photo2D.Width; // ширина фото

limit = limitR;

/\*

// Всё домножается на 3, т.к. мы проверяем по 3-ём палитрам

double[,] kernel = {{-3, 0, 3}, {-6, 0, 6}, {-3, 0, 3}};

foto2 = Filters.Svertka(foto1, height, width, kernel);\*/

}

////////////////////////////////////////////////////////////////////////////

// Два важных метода для конвертации

// получение байтов из Битмапа

public static byte[] GetBytes(Bitmap input)

{

int count = input.Height\*input.Width\*3; // размер нашего изображения

BitmapData inputD = input.LockBits(new Rectangle(0, 0, input.Width, input.Height), ImageLockMode.ReadOnly,

PixelFormat.Format24bppRgb); // выделяем память

var output = new byte[count];

Marshal.Copy(inputD.Scan0, output, 0, count); // копируем себе в массив

input.UnlockBits(inputD); // разблокировка памяти

return output;

}

// получение Битмапа из байтов

public static Bitmap GetBitmap(byte[] input, int width, int height)

{

if (input.Length%3 != 0)

return null;

// проверяем сможем ли мы сконвертировать обратно (должно делиться на 3, так хранятся цветные пиксели)

var output = new Bitmap(width, height);

BitmapData outputD = output.LockBits(new Rectangle(0, 0, width, height), ImageLockMode.ReadWrite,

PixelFormat.Format24bppRgb); // выделяем память

Marshal.Copy(input, 0, outputD.Scan0, input.Length);

output.UnlockBits(outputD); // разблокировка памяти

return output;

}

////////////////////////////////////////////////////////////////////////////

// Перевод цветного изображения в градацию серого

public static Bitmap GrayImage(Bitmap ImageBitmap)

{

byte[] ImageBytes = GetBytes(ImageBitmap);

for (int i = 0; i < ImageBytes.Length; i += 3)

{

byte GrayColor = (byte) (ImageBytes[i]\*0.299 + ImageBytes[i + 1]\*0.587 + ImageBytes[i + 2]\*0.114);

ImageBytes[i] = GrayColor;

ImageBytes[i+1] = GrayColor;

ImageBytes[i+2] = GrayColor;

}

ImageBitmap = GetBitmap(ImageBytes, ImageBitmap.Width, ImageBitmap.Height);

return ImageBitmap;

}

// алгоритм свёртки для фильтров

public Bitmap Svertka(Bitmap foto, int height, int width, double[,] kernel) // принимает параметры ядра

{

// переводим наше изображение в байты

byte[] inputBytes = GetBytes(foto);

// создаём массив для итога с нужным размером

byte[] outputBytes = new byte[inputBytes.Length];

int kernelWidth = kernel.GetLength(0);

int kernelHeight = kernel.GetLength(1);

double sumR;

double sumG;

double sumB;

double sumKernel;

// проходим по изображению, не обрабатывая края

for (int x = 0; x < width; x++)

{

for (int y = 0; y < height; y++)

{

sumR = 0;

sumG = 0;

sumB = 0;

sumKernel = 0;

// проходим по ядру

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

{

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

{

int positionX = x + (i - (kernelWidth/2));

int positionY = y + (j - (kernelHeight/2));

// не обрабатываются края (при их категоричности доработаю)

if ((positionX < 0) || (positionX >= width) || (positionY < 0) || (positionY >= height))

continue;

// т.к. всё лежит подряд в массиве, то и умножаем позицию на 3, получаем 3 палитры подряд

byte r = inputBytes[3\*(width\*positionY + positionX) + 0];

byte g = inputBytes[3\*(width\*positionY + positionX) + 1];

byte b = inputBytes[3\*(width\*positionY + positionX) + 2];

double kernelValue = kernel[i, j];

sumR += r\*kernelValue;

sumG += g\*kernelValue;

sumB += b\*kernelValue;

sumKernel += kernelValue;

}

}

// Нельзя делить на ноль

if (sumKernel <= 0)

sumKernel = 1;

// Нельзя выйти за цветовые пределы

sumR = sumR/sumKernel;

if (sumR < 0)

sumR = 0;

if (sumR > 255)

sumR = 255;

sumG = sumG/sumKernel;

if (sumG < 0)

sumG = 0;

if (sumG > 255)

sumG = 255;

sumB = sumB/sumKernel;

if (sumB < 0)

sumB = 0;

if (sumB > 255)

sumB = 255;

// Записываем результат в цвет пикселя

outputBytes[3\*(width\*y + x) + 0] = (byte) sumR;

outputBytes[3\*(width\*y + x) + 1] = (byte) sumG;

outputBytes[3\*(width\*y + x) + 2] = (byte) sumB;

}

}

// Конвертируем полученные байты обратно в Битмап

return GetBitmap(outputBytes, width, height);

}

// Фильтр Гаусса

private Bitmap Gauss(Bitmap ourPhoto)

{

// Использует определенные значения свёртки

double[,] kernel = { { 0, 1, 0 }, { 1, 4, 1 }, { 0, 1, 0 } };

ourPhoto = Svertka(ourPhoto, ourPhoto.Height, ourPhoto.Width, kernel);

// Ещё возможный вариант

// double[,] kernel = { { 1, 2, 1 }, { 2, 4, 2 }, { 1, 2, 1 } };

// Всё домножено на 3, т.к. смотрим по 3-ём палитрам сразу'

//double[,] kernelX = { { -3, 0, 3 }, { -6, 0, 6 }, { -3, 0, 3 } };

//double[,] kernelY = { { -3, -6, -3 }, { 0, 0, 0}, { 3, 6, 3 } };

//Bitmap fotoAfterSvertka = Svertka(\_foto, height, width, kernelY);

return ourPhoto;

}

// Собель

public Bitmap SobelCanny(Bitmap photo)

{

int width = photo.Width;

int height = photo.Height;

// избавимся от шумов Гауссом

//double[,] kernel = { { 0, 1, 0 }, { 1, 4, 1 }, { 0, 1, 0 } };

photo = Gauss(photo);

// Оператор Собеля

double[,] kernelX = { { -1, 0, 1 }, { -2, 0, 2 }, { -1, 0, 1 } };

double[,] kernelY = { { -1, -2, -1 }, { 0, 0, 0 }, { 1, 2, 1 } };

// лимит для вычисления границ

//int limit = 128\*128;// (photo.Height \* photo.Width);

// переводим наше изображение в байты

byte[] inputBytes = GetBytes(photo);

// создаём итоговый массив с нужным размером

byte[] outputBytes = new byte[inputBytes.Length];

// создаём массив для итоговых длин с нужным размером

//byte[] outputBytesD = new byte[inputBytes.Length/3];

// создаём массив для итоговых углов с нужным размером

//byte[] outputBytesT = new byte[inputBytes.Length/3];

// вдруг потом понадобится другой размер

int kernelWidth = kernelX.GetLength(0);

int kernelHeight = kernelX.GetLength(1);

double sumRx;

double sumGx;

double sumBx;

double sumRy;

double sumGy;

double sumBy;

double sumKernelx;

double sumKernely;

// проходим по изображению, не обрабатывая края

for (int x = 0; x < width; x++)

{

for (int y = 0; y < height; y++)

{

sumRx = 0;

sumGx = 0;

sumBx = 0;

sumRy = 0;

sumGy = 0;

sumBy = 0;

sumKernelx = 0;

sumKernely = 0;

// проходим по ядру (по горизонтали)

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

{

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

{

int positionX = x + (i - (kernelWidth/2));

int positionY = y + (j - (kernelHeight/2));

// не обрабатываются края (при их категоричности доработаю)

if ((positionX < 0) || (positionX >= width) || (positionY < 0) || (positionY >= height))

continue;

// т.к. всё лежит подряд в массиве, то и умножаем позицию на 3, получаем 3 палитры подряд

byte rX = inputBytes[3\*(width\*positionY + positionX) + 0];

byte gX = inputBytes[3\*(width\*positionY + positionX) + 1];

byte bX = inputBytes[3\*(width\*positionY + positionX) + 2];

double kernelValueX = kernelX[i, j];

sumRx += rX\*kernelValueX;

sumGx += gX\*kernelValueX;

sumBx += bX\*kernelValueX;

sumKernelx += kernelValueX;

}

}

// проходим по ядру (по вертикали)

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

{

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

{

int positionX = x + (i - (kernelWidth / 2));

int positionY = y + (j - (kernelHeight / 2));

// не обрабатываются края (при их категоричности доработаю)

if ((positionX < 0) || (positionX >= width) || (positionY < 0) || (positionY >= height))

continue;

// т.к. всё лежит подряд в массиве, то и умножаем позицию на 3, получаем 3 палитры подряд

byte rY = inputBytes[3 \* (width \* positionY + positionX) + 0];

byte gY = inputBytes[3 \* (width \* positionY + positionX) + 1];

byte bY = inputBytes[3 \* (width \* positionY + positionX) + 2];

double kernelValueY = kernelY[i, j];

sumRy += rY \* kernelValueY;

sumGy += gY \* kernelValueY;

sumBy += bY \* kernelValueY;

sumKernely += kernelValueY;

}

}

// Лайтовая версия КэнниСобеля - вводим настраиваемый лимит

if (sumRx\*sumRx + sumRy\*sumRy > limit || sumGx\*sumGx + sumGy\*sumGy > limit ||

sumBx\*sumBx + sumBy\*sumBy > limit)

{

outputBytes[3\*(width\*y + x) + 0] = 0;

outputBytes[3\*(width\*y + x) + 1] = 0;

outputBytes[3\*(width\*y + x) + 2] = 0;

}

else

{

outputBytes[3\*(width\*y + x) + 0] = 255;

outputBytes[3\*(width\*y + x) + 1] = 255;

outputBytes[3\*(width\*y + x) + 2] = 255;

}

/\*

// чуть более расширенные данные для Кэнни, пока не надо

outputBytes[3 \* (width \* y + x)] = (byte)Math.Sqrt(sumGx \* sumGx + sumGy \* sumGy);

outputBytes[3 \* (width \* y + x) + 1] = (byte)Math.Sqrt(sumGx \* sumGx + sumGy \* sumGy);

outputBytes[3 \* (width \* y + x) + 2] = (byte)Math.Sqrt(sumBx \* sumBx + sumBy \* sumBy);

//outputBytesD[(width \* y + x)] = (byte)Math.Sqrt(sumRx \* sumRx + sumRy \* sumRy);

//outputBytesT[(width \* y + x)] = (byte)Math.Atan(sumRx / sumRy);

\*/

}

}

//DlBytes = outputBytesD;

//AtanBytes = outputBytesT;

// Конвертируем полученные байты обратно в Битмап

return GetBitmap(outputBytes, width, height);

}

//Алгоритм Кэнни (Canny)

public Bitmap CannyFilter(Bitmap photo)

{

Bitmap photoForCanny = photo;

byte[] photoBytes = GetBytes(photo);

byte[] atans = {0, 45, 90, 135};

// избавимся от шумов фильтром Гаусса

//double[,] kernel = { { 0, 1, 0 }, { 1, 4, 1 }, { 0, 1, 0 } };

//photo = Svertka(photo, height, width, kernel);

// Обрабатывает

photoForCanny = Filters.GrayImage(photoForCanny);

SobelCanny(photoForCanny);

// Считаем максимальные перепады яркости и берём соответствующие углы

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

{

/\*

int max = 0;

// тут смотрим, по какой палитре максимальный перепад

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

{

if (DlBytes[i + j] > DlBytes[i + max])

max = j;

}

\*/

// дальше надо найти к какому углу из 4-ёх соответствующий угол ближе

int minDif = Math.Abs(AtanBytes[i] - atans[0]);

int numberAtan = 0;

for (int numb = 1; numb < 4; numb++)

{

// Упростить?? Лишнее вычисление?

if (minDif > Math.Abs(AtanBytes[i] - atans[numb]))

{

minDif = Math.Abs(AtanBytes[i] - atans[numb]);

numberAtan = numb;

}

}

//AtanMass[i] = atans[numberAtan];

AtanBytes[i] = atans[numberAtan];

//DlBytes[i] = DlBytes[i];

// ????????

//AtanBytes[i+1] = (byte)atans[numberAtan];

//AtanBytes[i+2] = (byte)atans[numberAtan];

}

int sizeStr = height\*3;

byte[] endBytes = new byte[width\*height\*3];

int pixelNumb = height+1;

// Строим границы

for (int j = 1; j < width-1; j++)

{

for (int y = 3; y < sizeStr-3; y+=3)

{

int x = j\*sizeStr;

// если угол направления градиента равен нулю, точка будет считаться границей, если её интенсивность больше чем у точки выше и ниже рассматриваемой точки

if (AtanBytes[pixelNumb] == 0)

{

if (DlBytes[pixelNumb] >= DlBytes[pixelNumb - height] && DlBytes[pixelNumb] >= DlBytes[pixelNumb + height])

{

endBytes[x + y] = 255;

endBytes[x + y + 1] = 255;

endBytes[x + y + 2] = 255;

}

}

// если угол направления градиента равен 90 градусам, точка будет считаться границей, если её интенсивность больше чем у точки слева и справа рассматриваемой точки

if (AtanBytes[pixelNumb] == 90)

{

if (DlBytes[pixelNumb] >= DlBytes[pixelNumb - 1] && DlBytes[pixelNumb] >= DlBytes[pixelNumb + 1])

{

endBytes[x + y] = 255;

endBytes[x + y + 1] = 255;

endBytes[x + y + 2] = 255;

}

}

// если угол направления градиента равен 45 градусам, точка будет считаться границей, если её интенсивность больше чем у точек находящихся в верхнем правом и нижнем левом углу от рассматриваемой точки

if (AtanBytes[pixelNumb] == 45)

{

if (DlBytes[pixelNumb] >= DlBytes[pixelNumb - height + 1] && DlBytes[pixelNumb] >= DlBytes[pixelNumb + height - 1])

{

endBytes[x + y] = 255;

endBytes[x + y + 1] = 255;

endBytes[x + y + 2] = 255;

}

}

// если угол направления градиента равен 135 градусам, точка будет считаться границей, если её интенсивность больше чем у точек находящихся в верхнем левом и нижнем правом углу от рассматриваемой точки

if (AtanBytes[pixelNumb] == 135)

{

if (DlBytes[pixelNumb] >= DlBytes[pixelNumb - height - 1] && DlBytes[pixelNumb] >= DlBytes[pixelNumb + height + 1])

{

endBytes[x + y] = 255;

endBytes[x + y + 1] = 255;

endBytes[x + y + 2] = 255;

}

}

pixelNumb++;

}

}

/\*

for (int b = 0; b < endBytes.Length; b++)

{

if (endBytes[b] != 255)

endBytes[b] = 0;

}\*/

return GetBitmap(endBytes, photo.Width, photo.Height);

}

## Сегментация изображения

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Runtime.Versioning;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace CurseWork\_2D3D

{

public class Segmentation

{

private int \_width;

private int \_height;

private byte[] \_photo;

public double limit;

public int segmSize;

List<Versh> versh;

public static List<Rib> ribs;

public static List<Rib> smallRibs;

public static Versh[,] v2d;

private byte[,] r;

private byte[,] g;

private byte[,] b;

public Segmentation(Bitmap photo2D)

{

limit = MainMenuForm.\_trueSegmLimit;

segmSize = MainMenuForm.\_trueSegmSize;

\_width = photo2D.Width;

\_height = photo2D.Height;

versh = new List<Versh>(\_width \* \_height);

ribs = new List<Rib>(\_width \* \_height \* 6);

smallRibs = new List<Rib>(\_width \* \_height);

\_photo = Filters.GetBytes(photo2D);

r = new byte[\_height, \_width];

g = new byte[\_height, \_width];

b = new byte[\_height, \_width];

}

public Segmentation(Bitmap photo2D, double limit, int segmSize)

{

this.limit = limit;

this.segmSize = segmSize;

\_width = photo2D.Width;

\_height = photo2D.Height;

versh = new List<Versh>(\_width \* \_height);

ribs = new List<Rib>(\_width \* \_height \* 6);

smallRibs = new List<Rib>(\_width \* \_height);

\_photo = Filters.GetBytes(photo2D);

r = new byte[\_height, \_width];

g = new byte[\_height, \_width];

b = new byte[\_height, \_width];

}

public void SortRebr()

{

int index = 0;

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

{

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

{

r[i, j] = \_photo[index++];

g[i, j] = \_photo[index++];

b[i, j] = \_photo[index++];

}

}

v2d = new Versh[\_height, \_width];

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

{

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

{

v2d[x, y] = new Versh(x, y, r[x, y], g[x, y], b[x, y]);

}

}

int[,] rebr = { { 0, 1 }, { 0, 2 }, { 0, 3 }, { 1, 0 }, { 2, 0 }, { 3, 0 } };

for (int x = 0, limitX = \_height - 3; x < limitX; x++)

{

for (int y = 0, limitY = \_width - 3; y < limitY; y++)

{

for (int i = 0; i < 6; i++)

{

int positionX = x + rebr[i, 0];

int positionY = y + rebr[i, 1];

//тут мы знаем, что следующий пиксель существует, потому что границы в цикле ширина-3, высота-3

CreateRib(x, y, positionX, positionY, r, g, b, v2d, ribs);

}

}

for (int y = \_width - 3; y < \_width; y++)

{

for (int i = 0; i < 6; i++)

{

int positionX = x + rebr[i, 0];

int positionY = y + rebr[i, 1];

// если следующий пиксель не существует

if ((positionX >= \_height) || (positionY >= \_width))

continue;

CreateRib(x, y, positionX, positionY, r, g, b, v2d, ribs);

}

}

}

for (int x = \_height - 3; x < \_height; x++)

{

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

{

for (int i = 0; i < 6; i++)

{

int positionX = x + rebr[i, 0];

int positionY = y + rebr[i, 1];

// если следующий пиксель не существует

if ((positionX >= \_height) || (positionY >= \_width))

continue;

CreateRib(x, y, positionX, positionY, r, g, b, v2d, ribs);

}

}

}

// сортировка по дистанции (цене ребра)

ribs.Sort((first, second) => first.\_dist.CompareTo(second.\_dist));

foreach (Versh versh1 in v2d)

{

versh.Add(versh1);

}

}

private void CreateRib(int x1, int y1, int x2, int y2, byte[,] r, byte[,] g, byte[,] b, Versh[,] v2d, List<Rib> list)

{

double dist = Math.Sqrt(Math.Pow((x1 - x2), 2) + Math.Pow((y1 - y2), 2) +

Math.Pow((r[x1, y1] - r[x2, y2]), 2) +

Math.Pow((g[x1, y1] - g[x2, y2]), 2) +

Math.Pow((b[x1, y1] - b[x2, y2]), 2));

Rib rib = new Rib(v2d[x1, y1], v2d[x2, y2], dist);

ribs.Add(rib);

list.Add(rib);

}

public Bitmap Segment()

{

int[,] maxSize = new int[\_height \* \_width, 2];

// соединяем по пикселям

for (int k = 0, ribCount = ribs.Count/\*так он один раз вычислит и не будет делать это на каждом шаге\*/; k < ribCount; k++)

{

Rib r = ribs[k];

if (r.\_dist > limit)

break;

//если у них один корень, то они уже в одном сегменте - тогда их не надо соединять снова

if (r.\_firstV.Root == r.\_secondV.Root)

continue;

//если же они в разных сегментах, то мы должны оценить, должны ли мы соединять сегменты.

//Одинокий пиксель - сегмент из одной вершины

//если вершины в разных сегментах, и вес ребра меньше лимита - соединяем сегменты

if (r.\_dist < limit)

r.\_firstV.MergeSegment(r.\_secondV);

}

//для каждого сегмента ищем самое большое ребро внутри него (идём от больших к меньшим, т.к. ищем больший)

for (int k = ribs.Count - 1; k >= 0; k--)

{

if (ribs[k].\_firstV.Root == ribs[k].\_secondV.Root && ribs[k].\_firstV.MaxDist < ribs[k].\_dist)

ribs[k].\_firstV.MaxDist = ribs[k].\_dist;

}

// Вторая проверка, уже с образовавшимися сегментами

RemoveSmallSegments();

// MergeSegments();

///////////////////////////////////////////////////////////////////////////////

// Красим сегменты в зависимости от цвета корней

byte[] mimimi = new byte[\_photo.Length];

int index = 0;

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

{

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

{

mimimi[index++] = v2d[i, j].Root.\_r;

mimimi[index++] = v2d[i, j].Root.\_g;

mimimi[index++] = v2d[i, j].Root.\_b;

}

}

Bitmap end = Filters.GetBitmap(mimimi, \_width, \_height);

//////////////////////////////////////////////////////////////////////

//end = Filters.GrayImage(end);

//Bitmap zyuzyu = Filters.GetBitmap(\_foto, \_width, \_height);

//Application.Run(new Form1(zyuzyu, zyu));

return end;

//new Form1(zyuzyu, end).Show();

}

// очень умная шикарная штука, но она осложнила программу и добавила неточностей(

private void MergeSegments()

{

//теперь объединяем уже существующие сегменты по какой-то хитрой формуле - МАГИЯ

double ksegm = 2;

for (int k = 0; k < ribs.Count; k++)

{

Rib r = ribs[k];

//если у них один корень, то они уже в одном сегменте - тогда их не надо соединять снова

if (r.\_firstV.Root == r.\_secondV.Root)

continue;

//если же они в разных сегментах, то мы должны оценить, должны ли мы соединять сегменты

//одинокий пиксель - сегмент из одной вершины

//поправка от количества пикселей

double t1 = ksegm / r.\_firstV.VershCount;

double t2 = ksegm / r.\_secondV.VershCount;

//перепады внутри сегментов с поправкой от количества пикселей внутри сегмента

double m1 = r.\_firstV.MaxDist + t1;

double m2 = r.\_secondV.MaxDist + t2;

//меньшая из больших дистанций внутри сегментов, с поправкой (та которая с количеством пикселей)

double minMax = Math.Min(m1, m2);

if (r.\_dist < minMax)

{

r.\_firstV.MergeSegment(r.\_secondV);

Versh root = r.\_firstV.Root;

//после объединения ищем новый наибольший перепад

//т.к. рёбра упорядочены по возрастанию веса, а мы идём с конца - первое попавшееся ребро из нашего сегмента будет самым большим

GetMaxDist(root);

}

}

}

// принадлежит методу сверху, ищет самый больший перепад в сегменте

private static void GetMaxDist(Versh root)

{

for (int i = ribs.Count - 1; i >= 0; i--)

{

//если оба принадлежат тому сегменту, который мы только что слепили - т.е. если у них тот же корень

//то это ребро нам подходит

if (ribs[i].\_firstV.Root == root && ribs[i].\_secondV.Root == root)

{

root.MaxDist = ribs[i].\_dist;

break;

}

}

}

// проходим второй раз либо по всем рёбрам, лбо по маленьким (реализовано два варианта)

private void RemoveSmallSegments()

{

int[,] rebr = { { 0, 1 }, { 1, 0 } };

for (int x = 0, limitX = \_height - 1; x < limitX; x++)

{

for (int y = 0, limitY = \_width - 1; y < limitY; y++)

{

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

{

int positionX = x + rebr[i, 0];

int positionY = y + rebr[i, 1];

//тут мы знаем, что следующий пиксель существует, потому что границы в цикле ширина-3, высота-3

CreateRib(x, y, positionX, positionY, r, g, b, v2d, smallRibs);

}

}

for (int y = \_width - 1; y < \_width; y++)

{

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

{

int positionX = x + rebr[i, 0];

int positionY = y + rebr[i, 1];

// если следующий пиксел не существует

if ((positionX >= \_height) || (positionY >= \_width))

continue;

CreateRib(x, y, positionX, positionY, r, g, b, v2d, smallRibs);

}

}

}

for (int x = \_height - 1; x < \_height; x++)

{

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

{

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

{

int positionX = x + rebr[i, 0];

int positionY = y + rebr[i, 1];

// если следующий пиксел не существует

if ((positionX >= \_height) || (positionY >= \_width))

continue;

CreateRib(x, y, positionX, positionY, r, g, b, v2d, smallRibs);

}

}

}

int minSize = (\_height + \_width) / segmSize;

for (int k = 0, ribCount = ribs.Count; k < ribCount; k++) // для всех рёбер

// for (int k = 0, ribCount = smallRibs.Count; k < ribCount; k++) // для маленьких рёбер

{

Rib r = ribs[k]; // для всех рёбер

// Rib r = smallRibs[k]; // для маленьких рёбер

//если у них один корень, то они уже в одном сегменте - тогда их не надо соединять снова

if (r.\_firstV.Root == r.\_secondV.Root)

continue;

//если вершины в разных сегментах, и вес ребра меньше лимита - соединяем сегменты

if (r.\_secondV.Root.VershCount < minSize)

r.\_firstV.MergeSegment(r.\_secondV);

else if (r.\_firstV.Root.VershCount < minSize)

r.\_secondV.MergeSegment(r.\_firstV);

}

}

}

}

## 3. Классы вершин, рёбер

public class Versh

{

private static int[,] rebr = { { 0, 1 }, { 0, 2 }, { 0, 3 }, { 0, -1 }, { 0, -2 }, { 0, -3 }, { 1, 0 }, { 2, 0 }, { 3, 0 }, { -1, 0 }, { -2, 0 }, { -3, 0 } };

public readonly int \_row;

public readonly int \_column;

public readonly byte \_r;

public readonly byte \_g;

public readonly byte \_b;

private Versh \_parent;

private double \_maxDist;

private int \_count;

public int \_z = -1;

// достаём корень какой-либо вершины, кладём его в \_parent и возвращаем

public Versh Root

{

get

{

//если \_parent = null, то мы уже в корне

if (\_parent == null)

return this;

//пока \_parent - не корень, двигаем его вверх по дереву

while (\_parent.\_parent != null)

\_parent = \_parent.\_parent;

return \_parent;

}

set

{

Versh root = this.Root;

root.\_parent = value.Root;

}

}

public int VershCount

{

get { return Root.\_count; }

set { Root.\_count = value; }

}

public double MaxDist

{

get { return Root.\_maxDist; }

set { Root.\_maxDist = value; }

}

public Versh(int x1, int y1, byte r, byte g, byte b)

{

\_count = 1;

\_row = x1;

\_column = y1;

\_r = r;

\_g = g;

\_b = b;

\_parent = null;

\_maxDist = 0;

}

public void MergeSegment(Versh v)

{

//проверяем, если уже один сегмент

if (this.Root == v.Root)

{

return;

}

VershCount += v.VershCount;

v.Root = this.Root;

}

// public override bool Equals(object obj)

// {

// Versh v = obj as Versh;

// if (v == null)

// return false;

// return \_row == v.\_row && \_column == v.\_column;

// }

public bool isBorderVersh(int height, int width)

{

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

{

for (int j = -1; j <= 1; j++)

{

if (i == 0 && j == 0)

continue;

if (\_row + i < 0 || \_row + i >= height || \_column + j < 0 || \_column + j >= width)

return true;

//если мы сюда пришли, то мы знаем, что соседний к нашему пиксель внутри картинки

if (Segmentation.v2d[\_row + i, \_column + j].Root != Root)

return true;

}

}

return false;

}

}

public class Rib

{

public Versh \_firstV;

public Versh \_secondV;

public double \_dist;

public int index;

public Rib(Versh first, Versh second, double dist)

{

\_firstV = first;

\_secondV = second;

\_dist = dist;

index = 0;

}

}

## Анализатор

public class Analizator

{

public Bitmap \_photo;

public int \_height;

public int \_width;

public Versh[,] \_v2d;

public int[] zCoordinates;

private bool[,] visited;

public static bool iStarted = false;

public Analizator(Bitmap photo)

{

\_photo = photo;

\_height = photo.Height;

\_width = photo.Width;

visited = new bool[\_height, \_width];

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

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

visited[i, j] = false;

\_v2d = Segmentation.v2d;

}

public List<AreaContainer> FormAreas()

{

List<AreaContainer> result = new List<AreaContainer>();

if (\_v2d == null)

{

Segmentation segm = new Segmentation(\_photo);

segm.SortRebr();

segm.Segment();

\_v2d = Segmentation.v2d; //Полученный битмап нам неинтересен, только массив

}

AreaContainer currentArea = new AreaContainer();

// Сначала ищем землю

// Находим самый большой сегмент на нижней строке (его корень)

Versh ground = FindLargeSegment(\_height - 1);

int pixelRow = 0;

int pixelColumn = 0;

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

{

if (\_v2d[\_height - 1, i].Root == ground.Root)

{

pixelRow = \_height - 1;

pixelColumn = i;

break;

}

}

// Формируем границу земли

currentArea.Borders = FindBorder(pixelRow, pixelColumn);

// Формируем всю область земли

FormSingleArea(pixelRow, pixelColumn);

// Добавляем землю в список областей - у неё будет индекс 0

result.Add(currentArea);

// Потом найдём небо

currentArea = new AreaContainer();

Versh sky = FindLargeSegment(0);

pixelRow = 0;

pixelColumn = 0;

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

{

if (\_v2d[0, i].Root == sky.Root)

{

pixelRow = 0;

pixelColumn = i;

break;

}

}

// Формируем границу неба

currentArea.Borders = FindBorder(pixelRow, pixelColumn);

// Формируем всю область неба

FormSingleArea(pixelRow, pixelColumn);

// Добавляем небо в список областей - у него будет индекс 1

result.Add(currentArea);

// А теперь, дамы и господа, добавляем всё остальное

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

{

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

{

bool alreadyDone = false;

if (visited[row, column] == true)

{

continue;

}

// Чтобы избавиться от багов и мелких отростков пикселей

int svoi = 0;

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

{

for (int j = -1; j <= 1; j++)

{

if (row + i < 0 || row + i >= \_height || column + j < 0 || column + j >= \_width)

continue;

if (\_v2d[row, column].Root == \_v2d[row + i, column + j].Root)

svoi++;

}

}

if (svoi < 5)

continue;

currentArea = new AreaContainer();

// Ищет граничные пиксели

currentArea.Borders = FindBorder(row, column);

// Ищет все пиксели области

FormSingleArea(row, column);

// Добавляет в лист листов, где границы всех областей

result.Add(currentArea);

}

}

return result;

}

private void FormSingleArea(int pixelRow, int pixelColumn)

{

// List<Versh> result = new List<Versh>();

int[,] steps = {{0, 1}, {1, 1}, {1, 0}, {1, -1}, {0, -1}, {-1, -1}, {-1, 0}, {-1, 1}};

Stack<Versh> untestedVersh = new Stack<Versh>(1000);

untestedVersh.Push(\_v2d[pixelRow, pixelColumn]);

while (true)

{

Versh currentVersh;

try

{

currentVersh = untestedVersh.Pop();

}

catch (InvalidOperationException e)

//чтобы нормально выйти, если стек вершей пуст (т.е. когда все обработали все верши внутри области)

{

return;

}

if (currentVersh == null)

break;

pixelRow = currentVersh.\_row;

pixelColumn = currentVersh.\_column;

visited[pixelRow, pixelColumn] = true;

for (int k = 0; k < 8; k++)

{

int testRow = pixelRow + steps[k, 1];

int testColumn = pixelColumn + steps[k, 0];

if (testRow < 0 || testRow >= \_height || testColumn < 0 || testColumn >= \_width)

continue;

if (\_v2d[pixelRow, pixelColumn].Root == \_v2d[testRow, testColumn].Root &&

visited[testRow, testColumn] == false)

{

untestedVersh.Push(\_v2d[testRow, testColumn]);

}

}

}

// return result;

}

// Находим поверхность ("землю") 3д модели, а также максимальное количество слоёв

public List<Versh> FindBorder(int firstRow, int firstColumn)

{

while (\_v2d[firstRow, firstColumn].isBorderVersh(\_height, \_width) == false)

firstRow--;

int[,] steps = {{0, 1}, {1, 1}, {1, 0}, {1, -1}, {0, -1}, {-1, -1}, {-1, 0}, {-1, 1}};

List<Versh> borders = new List<Versh>();

List<Versh> trash = new List<Versh>();

borders.Add(\_v2d[firstRow, firstColumn]);

Versh first = \_v2d[firstRow, firstColumn];

Versh current = null;

for (int k = 0; k < 8; k++)

{

int secondColumn = firstColumn + steps[k, 1];

int secondRow = firstRow + steps[k, 0];

if (secondColumn < 0 || secondColumn >= \_width || secondRow < 0 ||

secondRow >= \_height)

continue;

if (\_v2d[secondRow, secondColumn].Root == first.Root &&

\_v2d[secondRow, secondColumn].isBorderVersh(\_height, \_width) &&

\_v2d[secondRow, secondColumn] != first)

{

current = \_v2d[secondRow, secondColumn];

borders.Add(current);

break;

}

}

while (current != first)

{

bool foundNext = false;

int currentRow = current.\_row;

int currentColumn = current.\_column;

for (int k = 0; k < 8; k++)

{

int newRow = currentRow + steps[k, 0];

int newColumn = currentColumn + steps[k, 1];

if (newRow == first.\_row && newColumn == first.\_column)

{

return borders;

}

if (newColumn < 0 || newColumn >= \_width || newRow < 0 ||

newRow >= \_height)

continue;

if (\_v2d[newRow, newColumn].Root == first.Root)

if (\_v2d[newRow, newColumn].isBorderVersh(\_height, \_width))

if (borders.Contains(\_v2d[newRow, newColumn]) == false)

if (trash.Contains(\_v2d[newRow, newColumn]) == false) //для выхода из тупиков

{

current = \_v2d[newRow, newColumn];

borders.Add(current);

foundNext = true;

break;

}

}

if (foundNext == false)

{

trash.Add(borders[borders.Count - 1]);

borders.RemoveAt(borders.Count - 1);

current = borders[borders.Count - 1];

}

}

// // Создадим землю 3д модели

// for (int k = lastY, range = 0; k >= firstY; k--, range++)

// {

// for (int x = 0; x < \_width; x++)

// {

// // Оптимизировать??

// if (\_v2d[k, x].Root == Ground.Root)

// \_v2d[k, x].\_z = range;

// }

// }

return borders;

}

// Нахождение заднего фона изображения

public void FindBackground()

{

// Найдём самый большой сегмент, который касается верхнего края

Versh Background = FindLargeSegment(0);

// Т.к. это фон, координата z максимально возможно отдалена от пользователя

int z = zCoordinates.Length;

// Зададим z координату всем вершинам сегмента

setZ(Background, z);

}

// Нахождение самого большого сегмента в определённом ряду

// Пока работает только для строк

public Versh FindLargeSegment(int range)

{

int max = 1;

Versh maxSegm = \_v2d[range, 0];

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

{

if (\_v2d[range, x].VershCount > max)

{

max = \_v2d[range, x].VershCount;

maxSegm = \_v2d[range, x];

}

}

return maxSegm.Root;

}

// метод, задающий координату Z всему сегменту

public void setZ(Versh ourSegm, int z)

{

foreach (Versh v in \_v2d)

{

if (v.Root == ourSegm.Root)

v.\_z = z;

}

}

}

public class AreaContainer

{

//границы области

public List<Versh> Borders = new List<Versh>();

}

class DrawingVertex

{

public float x;

public float y;

public float z;

public float textx;

public float texty;

}

## Main программы

static class Program

{

/// <summary>

/// Главная точка входа для приложения.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new MainMenuForm());

}

}

## Реализация форм

### 3DModelForm

public partial class Form1 : Form

{

//private double limit = 14;

//private double segmSize = 10;

private Bitmap \_photo;

private string \_fileName;

private int \_height;

private int \_width;

// середина модели для отрисовки

private float srHeight;

private float srWidth;

private bool haveData;

private bool polygonsFormed;

private DrawingVertex[,] \_drawingVertices;

private List<AreaContainer> areaContainers;

public Form1(Bitmap photo, string name)

{

\_photo = photo;

\_height = \_photo.Height;

\_width = \_photo.Width;

\_fileName = name;

haveData = false;

polygonsFormed = false;

InitializeComponent();

// середина модели для отрисовки

srHeight = \_height / 2;

srWidth = \_width / 2;

}

private float rtri = 0;

Texture texture = new Texture();

private void openGLControl1\_OpenGLDraw(object sender, SharpGL.RenderEventArgs args)

{

OpenGL glWind = this.openGLControl1.OpenGL; // для удобства работы с окном вывода

//надо накладывать 3д текстуру!

//glWind.Enable(OpenGL.GL\_TEXTURE\_2D);

//texture.Create(glWind, "testFoto2.jpg");

glWind.Enable(OpenGL.GL\_TEXTURE\_2D);

texture.Create(glWind, \_fileName/\*"testFoto10.jpg"\*/); // задаём текстуру

glWind.Clear(OpenGL.GL\_COLOR\_BUFFER\_BIT | OpenGL.GL\_DEPTH\_BUFFER\_BIT); // чистим цвета и глубины

glWind.LoadIdentity(); // сброс системы координат к начальной позиции

glWind.Translate(0.0f, -0.5f, -3.0f); // по сути двигаем перо, которым рисуем (f - float)

glWind.Rotate(rtri, 0, 1, 0); // вращение системы координат (угол поворота, координаты вектора вращения)

Set3DModel(glWind);

glWind.Flush();

rtri += 5;

}

public void Set3DModel(OpenGL glWind)

{

bool alreadyDone = false;

Analizator analizator = new Analizator(\_photo);

// List<Versh> Ground = analizator.FindGround();

if (haveData == false)

{

// Преобразованные пиксели в вершины для рисования

\_drawingVertices = new DrawingVertex[\_height, \_width];

// Получим лист листов с границами всех областей

areaContainers = analizator.FormAreas();

haveData = true;

}

MakePolygons(areaContainers, glWind);

}

public void MakePolygons(List<AreaContainer> areas, OpenGL glWind)

{

// float earthYcoef = 1f;

// float earthZcoef = 1f;

// float groundShift = 0;

float farthestZ = 1;

float sdvig = -0.5f;

// areas[0] - земля, areas[1] - небо

// Сначала преобразуем землю

glWind.Begin(OpenGL.GL\_POLYGON);

texture.Bind(glWind);

glWind.Color(1.0f, 1.0f, 1.0f); // Задаём цвет в RGB

// Отрисовываем землю

foreach (Versh versh in areas[0].Borders)

{

if (\_drawingVertices[versh.\_row, versh.\_column] == null)

{

DrawingVertex drawingVertex = new DrawingVertex();

drawingVertex.x = (float)versh.\_column / \_width + sdvig;

drawingVertex.y = ((float) \_height - versh.\_row)/\_height; //\* earthYcoef;

drawingVertex.z = ((float) versh.\_row - \_height)/\_height; //\* earthZcoef;

drawingVertex.textx = (float) versh.\_column/\_width;

drawingVertex.texty = ((float) \_height - versh.\_row)/\_height;

\_drawingVertices[versh.\_row, versh.\_column] = drawingVertex;

}

float x = \_drawingVertices[versh.\_row, versh.\_column].x;

float y = \_drawingVertices[versh.\_row, versh.\_column].y;

float z = \_drawingVertices[versh.\_row, versh.\_column].z;

if (z < farthestZ)

{

farthestZ = z;

}

float textx = \_drawingVertices[versh.\_row, versh.\_column].textx;

float texty = \_drawingVertices[versh.\_row, versh.\_column].texty;

glWind.TexCoord(textx, -texty);

glWind.Vertex(x, y, z);

}

glWind.End();

//потом небо

glWind.Begin(OpenGL.GL\_POLYGON);

texture.Bind(glWind);

glWind.Color(1.0f, 1.0f, 1.0f); // задаём цвет в RGB

int minRow = \_height;

int minColumn = \_width;

foreach (Versh versh in areas[1].Borders)

{

if (\_drawingVertices[versh.\_row, versh.\_column] == null)

{

if (minRow == \_height) //ищем координаты Верша, соответствующего самой высокой границе ОБЛАСТИ (не сегмента) земли

{ //сравнение стоит,чтобы не считать заново для каждой вершины небы

foreach (Versh groundVersh in areas[0].Borders)

{

if (groundVersh.\_row < minRow)

{

minRow = groundVersh.\_row;

minColumn = groundVersh.\_column;

}

}

}

//из этих двух высот получаем величину смещения, чтобы приклеить небо к сплющенной земле

DrawingVertex drawingVertex = new DrawingVertex();

drawingVertex.x = (float) versh.\_column/\_width+sdvig;

drawingVertex.y = ((float) \_height - versh.\_row)/\_height;

drawingVertex.z = \_drawingVertices[minRow, minColumn].z;

//drawingVertex.y -= Math.Abs(groundShift\*(drawingVertex.z/farthestZ));

drawingVertex.textx = (float)versh.\_column / \_width;

drawingVertex.texty = ((float)\_height - versh.\_row) / \_height;

\_drawingVertices[versh.\_row, versh.\_column] = drawingVertex;

}

float x = \_drawingVertices[versh.\_row, versh.\_column].x;

float y = \_drawingVertices[versh.\_row, versh.\_column].y;

float z = \_drawingVertices[versh.\_row, versh.\_column].z;

float textx = \_drawingVertices[versh.\_row, versh.\_column].textx;

float texty = \_drawingVertices[versh.\_row, versh.\_column].texty;

glWind.TexCoord(textx, -texty);

glWind.Vertex(x, y, z);

}

glWind.End();

//потом нацепим на них всё остальное -- но сначала просто сделаем вертексы

for (int areaIndex = 2; areaIndex < areas.Count; areaIndex++)

{

List<Versh> borders = areas[areaIndex].Borders;

foreach (Versh versh in borders)

{

if (\_drawingVertices[versh.\_row, versh.\_column] != null)

break;

\_drawingVertices[versh.\_row, versh.\_column] = new DrawingVertex();

\_drawingVertices[versh.\_row, versh.\_column].x = float.PositiveInfinity; //этим мы показываем, что вертекс ещё не обработали

}

}

//а теперь мы идём по картнке снизу вверх и клеим все области к земле или областям под ними

//сначала пройдём по самой нижней строке, потому что она крепится не к земле, а ставится с z=0

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

{

int row = \_height - 1;

DrawingVertex lowerDrawingVertex = \_drawingVertices[row, column];

if (lowerDrawingVertex == null || double.IsPositiveInfinity(lowerDrawingVertex.x) == false)

continue;

lowerDrawingVertex.x = (float) column/\_width + sdvig;

lowerDrawingVertex.y = 0;

lowerDrawingVertex.z = 0;

lowerDrawingVertex.textx = (float) column/\_width;

lowerDrawingVertex.texty = ((float) \_height - row)/\_height;

foreach (AreaContainer area in areas)

{

List<Versh> borders = area.Borders;

if (borders.Contains(Segmentation.v2d[row, column]))

//в borders лежат пиксели-границы (в виде Versh)

{

foreach (Versh versh in borders)

{

DrawingVertex drawingVertex = \_drawingVertices[versh.\_row, versh.\_column];

drawingVertex.x = (float) versh.\_column/\_width +sdvig;

drawingVertex.y = ((float) \_height - versh.\_row)/\_height; //- groundShift\*((float) (\_height - minRow)/\_height);

drawingVertex.z = 0;

//drawingVertex.y -= Math.Abs(groundShift \* (drawingVertex.z / farthestZ));

drawingVertex.textx = (float) versh.\_column/\_width;

drawingVertex.texty = ((float) \_height - versh.\_row)/\_height;

}

}

}

}

for (int row = \_height - 2; row >= 0; row--)

{

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

{

DrawingVertex drawingVertex = \_drawingVertices[row, column];

if (\_drawingVertices[row, column] == null || double.IsPositiveInfinity(drawingVertex.x) == false)

continue;

//теперь мы знаем, что вертекс существует, н не был обработан

//находим границу вершей, в котором лежит текущий

List<Versh> border = null;

foreach (AreaContainer area in areas)

{

List<Versh> areaBorder = area.Borders;

if (areaBorder.Contains(Segmentation.v2d[row, column]))

border = areaBorder;

}

int[] steps = new[] {0, -1, 1};

foreach (int step in steps)

{

if (column + step < 0 || column + step >= \_width)

continue;

if (\_drawingVertices[row, column + step] == null ||

double.IsPositiveInfinity(\_drawingVertices[row, column + step].x) == true)

continue;

//теперь мы знаем, что данный нижний вертекс существует и уже был обработан

//поэтому заберём у него координату - приклеим нашу область к нему

foreach (Versh versh in border)

{

drawingVertex = \_drawingVertices[versh.\_row, versh.\_column];

drawingVertex.x = (float) versh.\_column/\_width + sdvig;

drawingVertex.y = ((float) \_height - versh.\_row)/\_height;

// groundShift\*((float) (\_height - minRow)/\_height);

drawingVertex.z = \_drawingVertices[row, column + step].z;

//drawingVertex.y -= Math.Abs(groundShift \* (drawingVertex.z / farthestZ));

drawingVertex.textx = (float) versh.\_column/\_width;

drawingVertex.texty = ((float) \_height - versh.\_row)/\_height;

}

}

}

}

for (int i = 2; i < areas.Count; i++)

{

glWind.Begin(OpenGL.GL\_POLYGON);

texture.Bind(glWind);

glWind.Color(1.0f, 1.0f, 1.0f); // задаём цвет в RGB

foreach (Versh versh in areas[i].Borders)

{

float x = \_drawingVertices[versh.\_row, versh.\_column].x;

float y = \_drawingVertices[versh.\_row, versh.\_column].y;

float z = \_drawingVertices[versh.\_row, versh.\_column].z;

float textx = \_drawingVertices[versh.\_row, versh.\_column].textx;

float texty = \_drawingVertices[versh.\_row, versh.\_column].texty;

glWind.TexCoord(textx, -texty);

glWind.Vertex(x, y, z);

}

glWind.End();

}

}

### MainMenuForm

public MainMenuForm()

{

\_trueSegmLimit = 14;

\_trueSegmSize = 10;

\_trueRangeLimit = 16384;

InitializeComponent();

}

private void openFileDialog1\_FileOk(object sender, CancelEventArgs e)

{

}

private void button1\_Click(object sender, EventArgs e)

{

if (loadedIt == true)

return;

OpenFileDialog openFileDialog1 = new OpenFileDialog();

openFileDialog1.InitialDirectory = "C:\\Users\\SONY\\Documents\\Visual Studio 2013\\Projects\\2D3D\\2D3D\\bin\\Debug";

openFileDialog1.Filter = "jpg files (\*.jpg)|\*.jpg|All files (\*.\*)|\*.\*";

openFileDialog1.FilterIndex = 2;

openFileDialog1.RestoreDirectory = true;

if (openFileDialog1.ShowDialog() == DialogResult.OK)

{

loadedIt = false;

try

{

newWorkForMe = new Bitmap(openFileDialog1.OpenFile());

fileName = openFileDialog1.SafeFileName;

////////////////////////////////////////////////////////////////////////////////////////////

//Form3 filtresForm = new Form3(newWorkForMe);

//filtresForm.Show();

////////////////////////////////////////////////////////////////////////////////////////////

loadedIt = true; //флаг о том, что новая картинка загружена

madeIt = false; //флаг о том, что он еще не сделал 3д модель

}

catch (Exception ex)

{

loadedIt = false;

MessageBox.Show("Ошибка при чтении файла. " + ex.Message);

}

}

}

private void button2\_Click(object sender, EventArgs e)

{

if (loadedIt == true && madeIt == false)

{

Form filtres = new SettingsForm(newWorkForMe);

filtres.Show();

}

else

{

}

}

private void button3\_Click(object sender, EventArgs e)

{

// // глупая защита

// if (Analizator.iStarted)

// return;

// Analizator.iStarted = true;

if (loadedIt == true)

{

if (madeIt == false)

{

// сегментация

//Segmentation seg = new Segmentation(newWorkForMe);

//seg.SortRebr();

//endWork = seg.Segment();

// после этого v2d точно существует

// анализатор сегментов и поиск их примерного расположения

}

Form1 frm = new Form1(newWorkForMe, fileName);

frm.Show();

}

}

}

### SettingsForm

public partial class SettingsForm : Form

{

private Bitmap \_photo;

private Bitmap \_photoEdge;

private Bitmap \_photoEnd;

private string settings = "";

private double segmLimit;

private int rangeLimit;

int segmSize;

public SettingsForm(Bitmap newWork)

{

\_photo = newWork;

InitializeComponent();

label1.Text = "Значение лимита свёртки";

label3.Text = "Порог минимума сегментов";

label5.Text = "Лимит перепада яркости границ";

label2.Text = "";

label4.Text = "";

label6.Text = "";

segmLimit = MainMenuForm.\_trueSegmLimit;

segmSize = MainMenuForm.\_trueSegmSize;

rangeLimit = MainMenuForm.\_trueRangeLimit;

textBox1.Text = segmLimit.ToString();

textBox2.Text = segmSize.ToString();

textBox3.Text = rangeLimit.ToString();

}

private void textBox1\_TextChanged(object sender, EventArgs e)

{

}

private void textBox2\_TextChanged(object sender, EventArgs e)

{

}

private void button1\_Click(object sender, EventArgs e)

{

try

{

segmLimit = double.Parse(textBox1.Text);

label2.Text = "";

}

catch (FormatException)

{

label2.Text = "Неверный формат, введите double";

return;

}

try

{

segmSize = int.Parse(textBox2.Text);

label4.Text = "";

}

catch (FormatException)

{

label4.Text = "Неверный формат, введите int";

return;

}

if (segmSize == 0)

{

label4.Text = "Не может быть равным 0!";

return;

}

try

{

rangeLimit = int.Parse(textBox3.Text);

label6.Text = "";

}

catch (FormatException)

{

label6.Text = "Неверный формат, введите int";

return;

}

// Если надо отображать и сегментацию, и границы

if (checkBox2.Checked && checkBox1.Checked)

{

Segmentation seg = new Segmentation(\_photo, segmLimit, segmSize);

seg.SortRebr();

\_photoEnd = seg.Segment();

int height = \_photo.Height;

int width = \_photo.Width;

// к сегментации добавим обведённые границы

Filters filt = new Filters(\_photo, rangeLimit);

\_photoEdge = filt.SobelCanny(\_photo);

//\_photoEdge = filt.Sobel(\_photo);

byte[] segmentByte = Filters.GetBytes(\_photoEnd);

byte[] edgeByte = Filters.GetBytes(\_photoEdge);

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

{

if (edgeByte[i] == 0)

segmentByte[i] = 0;

}

\_photoEnd = Filters.GetBitmap(segmentByte, width, height);

}

else

{

// если надо отобразить сегментацию

if (checkBox2.Checked)

{

Segmentation seg = new Segmentation(\_photo, segmLimit, segmSize);

seg.SortRebr();

\_photoEnd = seg.Segment();

}

// Если надо отобразить границы

if (checkBox1.Checked)

{

int height = \_photo.Height;

int width = \_photo.Width;

Filters filt = new Filters(\_photo, rangeLimit);

\_photoEnd = filt.SobelCanny(\_photo);

//\_photoEnd = Filters.GrayImage(\_photo);

}

}

settings = "limit=" + segmLimit + ";segmSize=" + segmSize + ";rangeLimit=" + rangeLimit;

new Picture(\_photoEnd, settings).Show();

}

private void button2\_Click(object sender, EventArgs e)

{

settings = "Original";

new Picture(\_photo, settings).Show();

}

// Сохранение настроек для модели

private void button3\_Click(object sender, EventArgs e)

{

MainMenuForm.\_trueSegmLimit = segmLimit;

MainMenuForm.\_trueSegmSize = segmSize;

}

// Сбросить настройки к дефолтным

private void button4\_Click(object sender, EventArgs e)

{

MainMenuForm.\_trueSegmLimit = 14;

MainMenuForm.\_trueSegmSize = 10;

textBox1.Text = MainMenuForm.\_trueSegmLimit.ToString();

textBox2.Text = MainMenuForm.\_trueSegmSize.ToString();

}

private void checkBox1\_CheckedChanged(object sender, EventArgs e)

{

}

}