

**Московский авиационный институт  
(национальный исследовательский университет)  
Факультет информационных технологий и прикладной  
математики**

**Кафедра 806 «Вычислительной математики и  
программирования»**

**Курсовая работа**

**по дисциплине: «Объектно-ориентированное  
программирование»**

**по теме: «Создание приложения в Windows Forms».**

Студент: К. Ю. Астахов

Преподаватель: Н. П. Аносова

Группа: М8О-212Б

Дата:

Оценка:

**Москва, 2021**

## Задание

Написать приложение с графическим интерфейсом, реализованным при помощи Windows Forms на языке программирования C#, использующее при проектировании решения одного или нескольких паттернов проектирования.

Для выполнения этого задания мною был выбран алгоритм изменения размеров изображения, который учитывает содержимое. Он носит название Seam Carving. В программе реализована возможность выделить определенные участки изображения, которые будут в последствии удалены в первую очередь. Также, благодаря паттерну Command, есть возможность отменять редактирование, нажав сочетание клавиш Ctrl+Z и снова применить отмененное действие, нажав сочетание Ctrl+U. Используемые паттерны проектирования: MVC, Command.

## Описание алгоритма

Весь алгоритм состоит из таких составных частей:

1. Нахождение энергии каждой точки. На этом этапе необходимо узнать какие части изображения являются более важными, а какие — менее важными, исходя из этих данных в последствии будет меняться размер изображения.
2. Нахождение такой вертикальной цепочки пикселей, чтобы суммарная энергия пикселей, которые входят в эту цепочку была минимальной. Цепочка пикселей — это такой набор пикселей, что в каждой строке выбрано ровно по одному пикслю, и соседние пиксели в ней соединены сторонами или углами. Если будет найдена такая цепочка, то можно будет ее удалить из изображения, при этом минимально затронув информационное наполнение изображения.
3. Когда цепочка с минимальной энергией найдена, то остается ее только удалить, если надо уменьшить изображение, или растянуть, если надо увеличить изображение.

Рассмотрим каждый пункт более детально.

### 1) Вычисление энергии точек

Первым делом необходимо решить какие участки изображения важны, а какие не

очень.

Для решения этой задачи авторы алгоритма предлагают для каждого пикселя посчитать его «энергию» — то есть какой-то условный показатель, который будет показывать, важен ли данный пикセル в данном изображении, или не очень.

Способов это сделать много: от самых простых (например посчитать изменения цвета по сравнению с соседними) до достаточно сложных (например сделать анализ акцентирования внимания человека). По словам самих авторов, они проверили множество функций энергии, и одна из самых простых функций дала один из лучших результатов. Поэтому воспользуемся сейчас этой функцией. Вот она:

$$e_1(I) = \frac{\partial f}{\partial x} I + \frac{\partial f}{\partial y} I$$

Иначе говоря, энергия пикселя равна изменению цвета соседних пикселей по сравнению с данным пикслем. То есть чем больше разница в цвете между данным пикслем и соседними — тем больше его энергия.

Пусть точки изображения характеризуются своей интенсивностью. Есть следующее изображение размером 3x3 точки:

9	1	9
7	4	6
4	8	2

Сначала посчитаем по модулю разницу интенсивностей между пикслем и его соседями (справа и снизу), а потом вычислим энергию этого пикселя как среднее арифметическое этих значений.

Для второго пикселя: разница между ним и соседом справа — 8, соседом снизу — 3. Среднее арифметическое —  $(8+3)/2 = 5.5$ . Но поскольку с целыми числами работать удобнее и быстрее, а такая точность излишняя, то отбросим дробную часть. То есть энергия второго пикселя равна 5.

В результате получим следующую матрицу энергий пикселей:

5	5	3
3	3	4
4	6	0

Если же пиксели характеризуются не просто интенсивностью, а значением интенсивности отдельно для R, G, B, то энергия пикселя равна сумме энергий по каждой из компонент. Пример карты энергий для изображения:



Нахождение энергии реализуется с помощью следующей функции:

### Листинг 1: C# FindEnergy

```
1 private void FindEnergy()
2 {
3     for (int i = 0; i < image.Height; i++)
4     {
5         for (int j = 0; j < image.Width; j++)
6         {
7             energy[i, j] = 0;
8
9             for (int k = 0; k < 3; k++)
10            {
11                 int sum = 0, count = 0;
12
13                 // Если пиксель не крайний снизу, то добавляем в sum разность
14                 // между текущим
15                 // пикселям и соседом снизу
16                 if (i != image.Height - 1)
17                 {
18                     count++;
19                     sum += Math.Abs(image.At<Vec3b>(i, j)[k] - image.At<Vec3b>(i +
20
21                     // Если пиксель не крайний справа, то добавляем в sum разность
22                     // между поточным
23                     // пикселям и соседом справа
24                     if (j != image.Width - 1)
25                     {
26                         count++;
27                         sum += Math.Abs(image.At<Vec3b>(i, j)[k] - image.At<Vec3b>(i,
28
29                         // В массив energy добавляем среднее арифметическое разностей
30                         // пикселя с соседями
31                         // по kтой- компоненте то ( есть по R, G или B)
```

```

31         if (count != 0)
32             energy[i, j] += sum / count;
33     }
34 }
35 }
36 }
```

Здесь `energy` — массив, в который записываются энергии и `image` — массив, в котором хранится изображение.

## 2) Нахождение цепочки с минимальной суммарной энергией.

Сначала создадим новый массив, который по размеру равен массиву с энергией пикселей. В этот массив для каждого пикселя запишем сумму элементов минимальной цепочки пикселей, которая начинается у верхнего края изображения и заканчивается на данном пикселе. Если формализовать вычисление этого массива, то получим следующую формулу:

$$s[i, j] = \begin{cases} e[i, j] & i = 0 \\ e[i, j] + \min(s[i - 1, j - 1], s[i - 1, j], s[i - 1, j + 1]) & i \neq 0 \end{cases}$$

, где `s` — массив сумм, а `e` — массив энергий.

Мой вариант реализации вычисления массива сумм:

### Листинг 2: C# FindSum

```

1 private void FindSum()
2 {
3     FindEnergy();
4     // Для верхней строчки значение sum и energy будут совпадать
5     for (int j = 0; j < image.Width; j++)
6         sum[0, j] = energy[0, j];
7
8     // Для всех остальных пикселей значение элемента (i, j) массива sum будут
9     // равны
10    // energy[i, j] + MIN (sum[i-1, j-1], sum[i-1, j], sum[i-1, j+1])
11    for (int i = 1; i < image.Height; i++)
12    {
```

```

12         for (int j = 0; j < image.Width; j++)
13     {
14         sum[i, j] = sum[i - 1, j];
15         if (j > 0 && sum[i - 1, j - 1] < sum[i, j]) sum[i, j] = sum[i - 1,
16             j - 1];
17         if (j < image.Width - 1 && sum[i - 1, j + 1] < sum[i, j]) sum[i, j
18 ] = sum[i - 1, j + 1];
19         sum[i, j] += energy[i, j];
20     }
21 }

```

Сначала найдем какой пиксел из нижней строчки изображения принадлежит этой цепочке: элемент, что мы ищем, будет иметь наименьшее значение в массиве сумм среди элементов нижней строчки, так как в данном массиве записаны значения сумм элементов минимальных цепочек от верхнего края до данного пикселя. Цепочки, которые нас интересуют заканчиваются на пикселе из последнего ряда. Соответственно для всего рисунка минимальная цепочка будет выбрана как минимальная из всех минимальных цепочек, которые заканчиваются на пикселях из нижней строчки.

После выполнения всех этих операций получим набор пикселей, которые можно изменить с минимальными потерями для изображения. Программная реализация алгоритма поиска цепочки:

### Листинг 3: C# FindVerticalPixels

```

1 private int[] FindVerticalPixels()
2 {
3     FindSum();
4     // Номер последней строчки
5     int last = image.Height - 1;
6     // Выделяем память под массив результатов
7     int[] res = new int[image.Height];
8
9     // Ищем минимальный элемент массива sum, который находится в нижней
строке
10    // и записываем результат в res[last]
11    res[last] = 0;

```

```

12     for (int j = 1; j < image.Width; j++)
13     {
14         if (sum[last, j] < sum[last, res[last]])
15             res[last] = j;
16     }
17
18     // Теперь вычисляем все элементы массива от предпоследнего до первого .
19     for (int i = last - 1; i >= 0; i--)
20     {
21         // prev - номер пикселя цепочки из предыдущей строки
22         // В этой строке пикселями цепочки могут быть только (prev-1), prev
23         или (prev+1),
24         // поскольку цепочка должна быть связанной
25         int prev = res[i + 1];
26
27         // Здесь мы ищем, в каком элементе массива sum, из тех, которые мы
28         можем удалить,
29         // записано минимальное значение и присваиваем результат переменной
30         res[i]
31         res[i] = prev;
32         if (prev > 0 && sum[i, res[i]] > sum[i, prev - 1]) res[i] = prev - 1;
33         if (prev < image.Width - 1 && sum[i, res[i]] > sum[i, prev + 1]) res[i]
34     ] = prev + 1;
35     }
36
37     return res;
38 }

```

Данная функция реализует поиск вертикальной цепочки пикселей. Для уменьшения изображения по горизонтали можно искать цепочки горизонтальных пикселей.

### 3) Уменьшение рисунков.

Если необходимо уменьшить ширину картинки на 1 пикセル, то находим вертикальную цепочку, как это описывается выше, и удаляем ее из изображения. В моем случае эта операция реализована с помощью следующей функции:

#### Листинг 4: C# RemoveVerticalSeam

```

1 || private void RemoveVerticalSeam(int[] col)

```

```
2    {
3        for (int i = 0; i < image.Height; i++)
4        {
5            for (int j = col[i]; j < image.Width; j++)
6            {
7                image.At<Vec3b>(i, j) = image.At<Vec3b>(i, j + 1);
8                mask.Data[i, j] = mask.Data[i, Math.Min(j + 1, image.Width - 1)];
9            }
10        }
11
12        image = image.ColRange(0, image.Width - 1);
13    }
```

Для случая уменьшения изображения по горизонтали также реализована соответствующая функция.

Примеры работы программы:





Интересно то, что этот алгоритм можно применять и для увеличения рисунков. Для этого также можно находить цепочку пикселей, содержащую наименьшее количество информации. После этого она может быть растянута. Но в таком случае это мало чем будет отличаться от простого растяжения изображения.

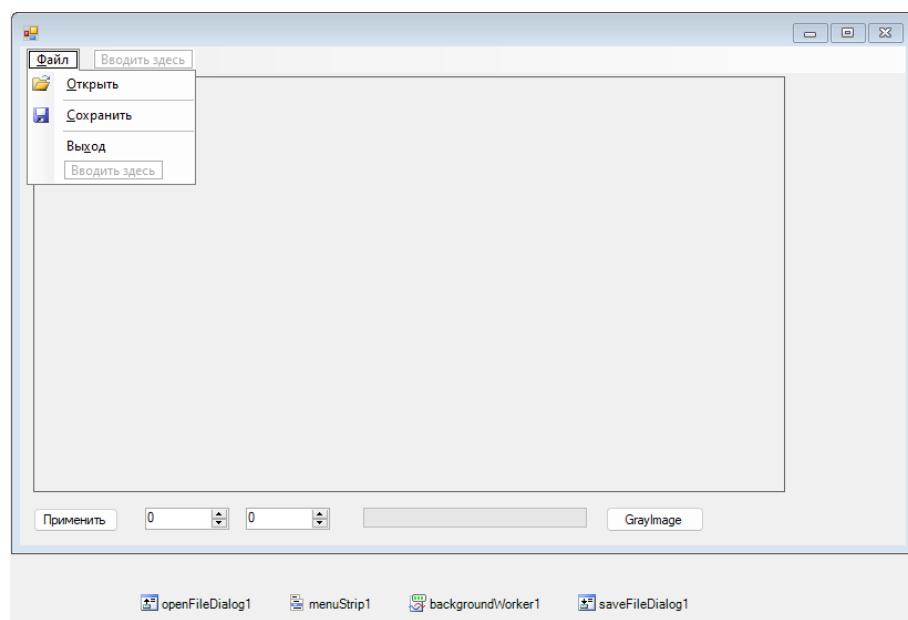
Авторы алгоритма предлагают следующие действия: растянуть изображение сохранив пропорции, после чего применить алгоритм уменьшения изображения. Этот план действий дает приемлемый результат.

Этот же алгоритм можно применять для удаления объектов с изображения путем присвоения энергии некоторых пикселей отрицательных значений.



## Реализация интерфейса

Программа имеет главное окно, созданное при помощи Form1, в котором находится pictureBox1, куда загружаются картинки, которые необходимо открыть. В нем находится menuStrip1, содержащий пункты меню для открытия и сохранения картины. В программе есть возможность открывать изображения путем перетаскивания их на pictureBox1.



Для реализации открытия и сохранения изображений на форме есть openFileDialog1 и saveDialog1, а так же backgroundWorker для обработки рисунков в отдельном потоке для продолжения работы интерфеса и progressBar1, отображающий процесс обработки.

Так же на форме есть кнопки «Применить» и «GrayImage», которые начинают обработку изображения и показывают серую карту энергии соответственно. Счетчики показывают текущий размер изображения в пикселях.

Для работы backgroundWorker были переопределены следующие обработчики событий:

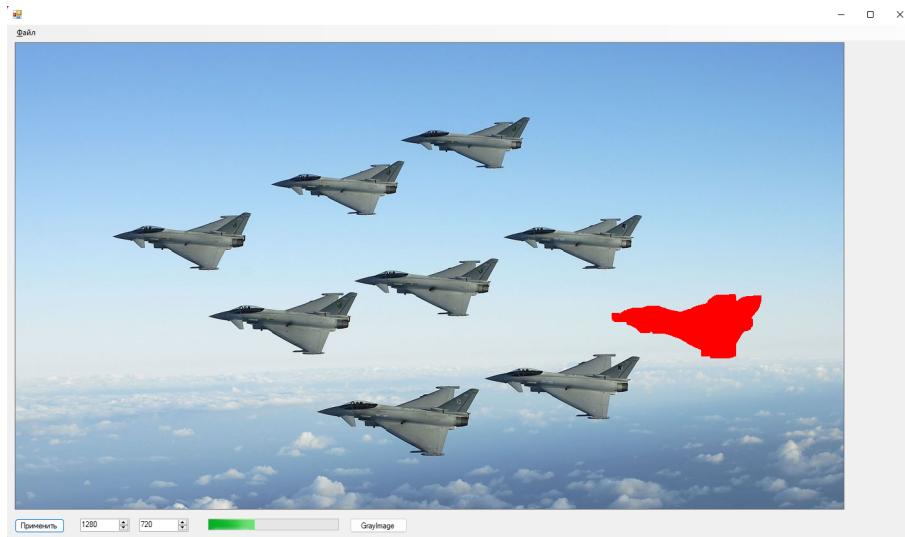
#### Листинг 5: C# backgroundWorker1

```
1 private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
2 {
3     BackgroundWorker bg = sender as BackgroundWorker;
4
5     e.Result = new Bitmap(controller.DeleteLines(controller.GetImage(), (int)
6 numericUpDown1.Value,
7         (int)numericUpDown2.Value, bg));
8 }
9
10 private int Clamp(int i)
11 {
12     if (i > 100)
13         return 100;
14     return i;
15 }
16 private void backgroundWorker1_ProgressChanged(object sender,
17 ProgressChangedEventArgs e)
18 {
19     progressBar1.Value = Clamp(e.ProgressPercentage);
20 }
21 private void backgroundWorker1_RunWorkerCompleted(object sender,
22 RunWorkerCompletedEventArgs e)
23 {
```

```
23     Bitmap b = (Bitmap) e.Result;
24     pictureBox1.Image = b;
25     numericUpDown1.Value = b.Width;
26     numericUpDown2.Value = b.Height;
27
28     pictureBox1.Size = b.Size;
29     Size = b.Size + deltaPb_Form1;
30
31     controller.LoadImage((Bitmap) pictureBox1.Image.Clone());
32     controller.CreateMask(pictureBox1.Image);
33     progressBar1.Visible = false;
34 }
```

Они нужны для корректной работы progressBar и корректного отображения результата.

На pictureBox есть возможность помечать красным цветом объекты, которые в последствии будут стерты с изображения.



В результате работы программы получается следующее изображение:



Код для отрисовки помеченной области выглядит следующим образом:

**Листинг 6: C# DrawMap**

```
1  private void pictureBox1_MouseMove(object sender, MouseEventArgs e)
2  {
3
4      if (pictureBox1.Image == null || backgroundWorker1.IsBusy)
5      {
6          return;
7      }
8
9      if ((e.Button & MouseButtons.Left) != 0)
10     {
11         using (Graphics g = Graphics.FromImage(pictureBox1.Image))
12         {
13             g.FillRectangle(brush, e.X - 4, e.Y - 4, 9, 9);
14             controller.AddRectangle(new Point(e.X - 4, e.Y - 4), 9, 9);
15         }
16         pictureBox1.Invalidate();
17     }
18 }
19
20 public void DrawMap(ImgMask mask)
21 {
22     using (Graphics g = Graphics.FromImage(pictureBox1.Image))
```

```

23    {
24        for (int i = 0; i < mask.GetDataSize().Width; ++i)
25        {
26            for (int j = 0; j < mask.GetDataSize().Height; ++j)
27            {
28                if (mask.Data[j, i] == 1)
29                    g.FillRectangle(brush, i, j, 1, 1);
30
31            }
32        }
33    }
34    pictureBox1.Invalidate();
35 }
36 private void pictureBox1_MouseUp(object sender, MouseEventArgs e)
37 {
38
39    if (pictureBox1.Image == null) return;
40
41    pictureBox1.Image = (Bitmap)controller.GetImage().Clone();
42    controller.LoadMask((Bitmap)pictureBox1.Image.Clone());
43
44    DrawMap(controller.GetMask());
45
46 }
```

## Реализация паттернов проектирования

В основе проектирования программы стоят два паттерна проектирования: Command и MVC.

Паттерн MVC позволяет разбить программу на три основных компонента: Model (класс Model), View (класс Form1) и Controller (класс Controller). Здесь пользователь общается с программой с помощью формы, необходимые действия связанные со сменой интерфейса и внутреннего устройства совершаются через обращение к контроллеру, а основные вычислительные алгоритмы выполняются при помощи модели и вспомогательных классов. UML диаграмма выглядит следующим образом:

