16.06.2008

Miłosz Muszyński
milosz@interkam.net

Politechnika Warszawska, Wydział Elektryczny

Projekt Indywidualny, 2007/2008

Projekt wykonany pod opieką dr. inż. Witolda Czajewskiego

Spis treści

1. Opis projektu

Celem projektu jest napisanie aplikacji, tworzącej fotomozaikę z określonego przez użytkownika obrazu. Obraz jest dzielony na określoną przez użytkownika liczbę części oraz tworzony ze zdjęć znajdujących się w bazie danych.

2. Możliwe rozwiązania

a. Język i środowisko programowania

Aplikacja jest możliwa do napisania we wszystkich dostępnych językach programowania. Najbardziej uzasadnione wydaje się napisanie aplikacji w języku PHP (ze względu na możliwość udostępnienia jej użytkownikom w sieci Internet oraz szybkość działania), Java (bardzo dobre wsparcie programowania obiektowego) lub w środowisku .NET (Visual Basic, C#, J#) jako aplikacji webowej lub desktopowej (duża ilość gotowych rozwiązań, dobra obsługa programowania obiektowego).

b. Określanie podobieństwa między obrazami

Jednym z najważniejszych elementów zadania jest porównywanie obrazów w celu określenia podobieństwa. Jest to szczególnie istotne, ponieważ efekt końcowy jest zależny od podobieństwa wstawionego zdjęcia do zastępowanego fragmentu obrazu.

Do określenia podobieństwa między obrazami można wykorzystać jedną z wielu znanych metod, między innymi: maksimum (odnalezienie różnicy między największą i najmniejszą wartością koloru w przetwarzanym obrazie i porównaniu jej z wartością uzyskaną dla drugiego obrazu), różnic bezwzględnych (metoda maksimum z podzieleniem wyniku przez liczbę pikseli w obrazie), błąd średniokwadratowy (zsumowanie kwadratów różnic kolorów pikseli i podzielenie przez liczbę pikseli), znormalizowany błąd średniokwadratowy (metoda błędu średniokwadratowego po normalizacji).

Oprócz tego, aby polepszyć efekt końcowy można zastosować dzielenie części obrazu i podstawianego zdjęcia na obszary. Wówczas porównywane są części obrazu, dzięki czemu uzyskany efekt jest lepszy dla obrazów, składających się z kilku fragmentów, różniących się od siebie pod względem kolorystycznym).

3. Wybór rozwiązania

a. Język i środowisko programowania

Jako język programowania wybrałem środowisko .NET 3.5 oraz język C# 3.0. Kierowałem się przede wszystkim lepszym wsparciem dla programowania obiektowego, niż w języku PHP 5, oraz ułatwieniami dla programisty i dużą liczbą gotowych komponentów, co przeważyło nad zaletami języka Java.

Ponieważ nie zależy mi na przenośności kodu, napisanie programu w środowisku .NET wydaje się lepszym rozwiązaniem także ze względu na większą szybkość działania programu, co jest związane z zapewnieniem lepszej maszyny wirtualnej. Podczas pisania aplikacji korzystałem również z klas generycznych, które w środowisku .NET są znacznie wygodniejsze w użyciu, niż w języku Java.

Język C# wybrałem ze względu na to, że posiadam większe doświadczenie w pisaniu aplikacji w tym języku, niż w pozostałych językach platformy .NET.

Zdecydowałem się na napisanie aplikacji desktopowej, ze względu na większą elastyczność, np. podczas komunikacji z użytkownikiem, brak konieczności przeładowywania strony, oraz większą szybkość działania aplikacji.

b. Określanie podobieństwa między obrazami

Pierwszym rozwiązaniem, które próbowałem wykorzystać była metoda maksimum, ponieważ stwierdziłem, że jest ona najbardziej oczywista, oraz najprostsza w implementacji. Wydawało się również, że będzie najszybsza. Jednak metoda ta okazała się bardzo mało dokładna – nie zapewniała dokładnego porównywania wszystkich kolorów.

Ostatecznie zdecydowałem się na wykorzystanie metody błędu średniokwadratowego, ponieważ uzyskiwane w ten sposób efekty okazały się zdecydowanie lepsze, niż w przypadku metody maksimum. Jednocześnie czas działania aplikacji pogorszył się nieznacznie. Najwięcej czasu zajmuje określenie kolorów każdego obrazka, które musi zostać wykonane niezależnie od wybranej metody porównywania obrazów.

Zdecydowałem się również na podzielenie każdej części obrazu na cztery fragmenty, ponieważ nie spowodowało to pogorszenia wydajności, natomiast pozwoliło na znaczne poprawienie wyników, szczególnie w przypadku małej liczby zdjęć, z których ma się składać określony przez użytkownika obraz.

4. Użyte oprogramowanie

a. Środowisko programowania

Ponieważ aplikacja została napisana w języku C# w środowisku Microsoft .NET 3.5, najbardziej oczywistym rozwiązaniem, jeśli chodzi o środowisko programowania wydaje się Microsoft Visual Studio 2008.

Środowisko to pozwala programiście na znaczne przyspieszenie pracy, zwłaszcza dzięki umieszczaniu kontrolek na formach z wykorzystaniem metody Drag & Drop, oraz udostępnieniu technologii IntelliSense, która znacznie przyspiesza pracę poprzez podpowiadanie składni. Visual Studio 2008 umożliwia też łatwe dodawanie referencji i nawigację pomiędzy klasami projektu.

b. Dodatkowe biblioteki

Początkowym założeniem było wykorzystanie otwartej biblioteki FreeImage w celu ułatwienia przetwarzania obrazów. W związku z trudnościami wynikającymi z niezgodności wersji środowiska .NET ostatecznie jednak, w projekcie nie zostały wykorzystane żadne zewnętrzne biblioteki do przetwarzania obrazu.

Podczas pisania aplikacji okazało się, że elementy wchodzące w skład frameworka .NET 3.5 są zupełnie wystarczające dla potrzeb projektu – pozwalają na łatwą zmianę wielkości (rozciąganie) obrazów, przycinanie, pobieranie fragmentów obrazu, oraz poszczególnych pikseli. Dlatego podczas pisania aplikacji wykorzystane zostały jedynie klasy udostępnione przez Microsoft w przestrzeni System.Drawing.

5. Proces realizacji

Najwięcej czasu zajęło przygotowanie projektu aplikacji. W początkowych założeniach zdjęcia składowe miały mieć stałą wielkość i jeden format. Było to jednak rozwiązanie mało elastyczne, dlatego ostatecznie zdecydowałem się na zdjęcia w dowolnym formacie i dowolnej rozdzielczości. Nie narzucam też stosunku szerokości do wysokości zdjęć, jednak w tym przypadku – jeżeli zdjęcie ma stosunek szerokości do wysokości różny od 1.33 zostaje ono rozciągnięte. Rozwiązanie to okazało się bardzo wygodne i łatwe w implementacji.

Podczas implementacji najtrudniej było poradzić sobie z dwoma głównymi problemami, których nie udało się do końca wyeliminować: zbyt długim czasem działania oraz brakiem pamięci.

Pierwszy problem jest związany z przetwarzaniem zdjęć składowych „w locie” i wstawianiu do odpowiednich części obrazka. Problem ten częściowo udało się wyeliminować, zmieniając błędne założenie, aby dla każdego obrazka, z którego składa się duży obraz podany przez użytkownika, wczytywać wszystkie zdjęcia składowe i znajdować najbardziej odpowiednie. Wiązało się to z niepotrzebnym powtarzaniem tych samych czynności kosztem pewnej oszczędności pamięci. Ostatecznie jednak w projekcie zostało użyte bardziej wydajne rozwiązanie, polegające na

odczytaniu wszystkich zdjęć składowych na początku przetwarzania aplikacji i zapisanie średnich kolorów. To rozwiązanie znacznie przyspieszyło działanie aplikacji.

Obecnie czas wykonywania mozaiki najsilniej zależy od dwóch parametrów: ilości zdjęć składowych, które należy przetworzyć oraz ilości kwadratów, na które zostanie podzielone zdjęcie określone przez użytkownika.

Drugi problem jest związany z przechowywaniem w pamięci bitmap (jest to najszybsze rozwiązanie). Problem ten udało się częściowo rozwiązać poprzez ręczne wywoływanie metody Dispose() dla obrazów w niektórych momentach. Ciągle jednak możliwe jest wystąpienie wyjątku polegającego na braku pamięci. Optymalnym rozwiązanie jest wówczas nieblokowanie działania całej aplikacji, a jedynie jednej z funkcjonalności (np. niewyświetlenie podglądu, ale umożliwienie użytkownikowi zapisania wygenerowanej mozaiki na dysku) i jest to najczęstsza reakcja aplikacji na wystąpienie tego błędu.

Bardzo łatwe okazało się z kolei zapisywanie i odczytywanie obrazów w różnych formatach, ze względu na obsługę przez środowisko. Zapis jest obecnie realizowany tylko w postacji pliku JPEG, ponieważ format ten oferuje największą kompresję, przy braku znacznego pogorszenia jakości. Obecnie dla zdjęcia o wielkości 5000x3800 pikseli wielkość obrazka to 4,3 MB. Dla porównania w przypadku bitmapy (BMP, 24-bpp) było to ponad 30 MB.

6. Podsumowanie

Można przyjąć, że głównymi miarami jakości aplikacji generującej fotomozaikę są: podobieństwo wygenerowanego obrazu do obrazu źródłowego, oraz prędkość działania. Oba czynniki są związane z zastosowanym algorytmem porównywania obrazów, przy czym zwiększenie jakości porównywania obrazów wiąże się ze spadkiem szybkości działania aplikacji. Jako optymalne rozwiązanie należy więc uznać zastosowanie rozwiązań pozwalających na wygenerowanie jak najdokładniejszego obrazu przy niezbyt długim czasie wykonywania. Obie własności są jednak bardzo umowne, więc ciężko jest stwierdzić, czy prędkość lub podobieństwo wygenerowanego obrazu są odpowiednie, czy powinny zostać jeszcze poprawione.

Problem generowania fotomozaiki można rozwiązać, nie wykorzystując skomplikowanych algorytmów porównywania obrazów i nie jest on wówczas trudny do rozwiązania, a uzystkiwane efekty wydają się dość dobre. Najważniejszym czynnikiem, mającym wpływ na jakość rozwiązania jest liczba zdjęć składowych w określonym przez użytkownika obrazie, liczba i jakość zdjęć składowych w bazie oraz docelowa wielkość wygenerowanego obrazu. Im większa liczba zdjęć składowych, tym obraz jest bardziej podobny do oryginału, natomiast im większa rozdzielczość, tym bardziej można powiększyć uzyskany wynik w celu przyjrzenia się zdjęciom składowym. Jednak pierwszy i drugi z czynników – liczba zdjęć składowych w obrazie oraz w bazie, mają bardzo duży wpływ na prędkość działania aplikacji – im jest ich więcej, tym wolniej działa aplikacja. Z kolei zwiększanie rozdzielczości wiąże się ze znacznym wzrostem wykorzystywanej pamięci. Określenie zbyt dużej rozdzielczości wiąże się z możliwością wystąpienia błędu spowodowanego zbyt małą ilością wolnej pamięci.

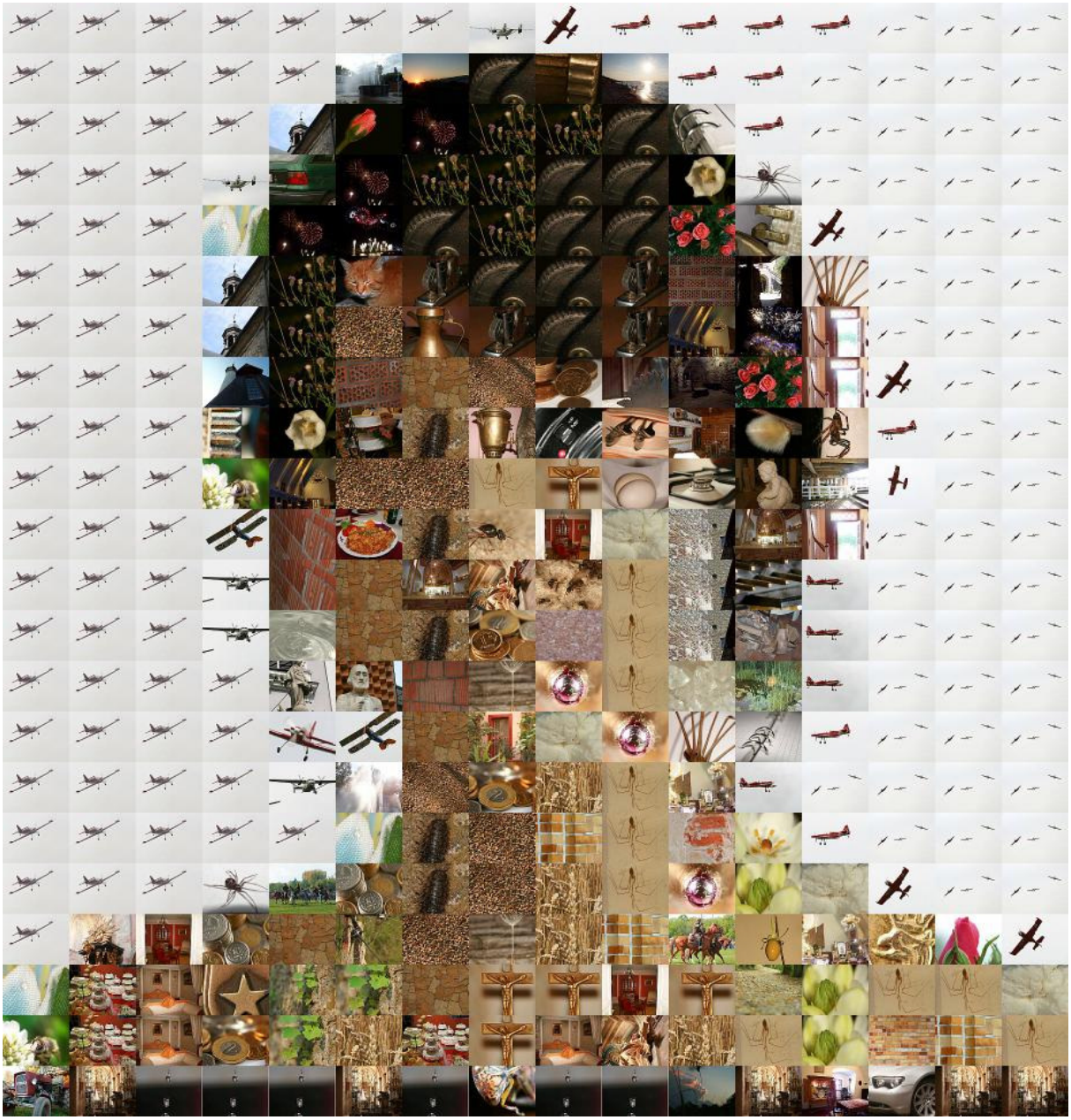
W związku z tym można stwierdzić, że problem Fotomozaiki jest prosty do rozwiązania, jednak wiąże się z dużą złożonością pamięciową oraz obliczeniową.

7. Przykłady działania aplikacji

Przykład działania aplikacji – zdjęcie pobrane z serwisu CGSociety.org (autor: Max Edwin Wahyudi):

Poniżej prezentuję wyniki działania aplikacji dla tego zdjęcia.

Mozaika dla powiększenia 100% i wysokości małego zdjęcia 44px (czas generowania: 30 sekund):



Powyższa grafika jest mało podobna do oryginału, co wynika głównie z małej ilości zdjęć składowych, oraz z niedoskonałego algorytmu porównywania obrazów. Z powodu zbyt małej ilości obrazków składowych, szczegóły zdjęcia są odwzorowane bardzo słabo.

Mozaika przy powiększeniu 200% (poniżej: mozaika pomniejszona do szerokości strony) i wysokości małego zdjęcia 44px (czas generowania: 45 sekund):

Powyższa mozaika wygląda znacznie lepiej niż poprzednia, co jest związane z większą ilością małych kwadratów, z czego wynika dokładniejsze odwzorowanie szczegółów. W dalszym ciągu widoczny jest problem zbyt małej ilości zdjęć składowych do wygenerowania fotomozaiki, przypominającej oryginał w satysfakcjonującym stopniu. Można też zaobserwować problem wynikający z podziału zdjęć składowych na mniejsze elementy, co szczegółowo opisałem poniżej.

Mozaika przy powiększeniu 400% (poniżej: mozaika pomniejszona do szerokości strony) i wysokości małego zdjęcia 44px (czas generowania: 70 sekund):

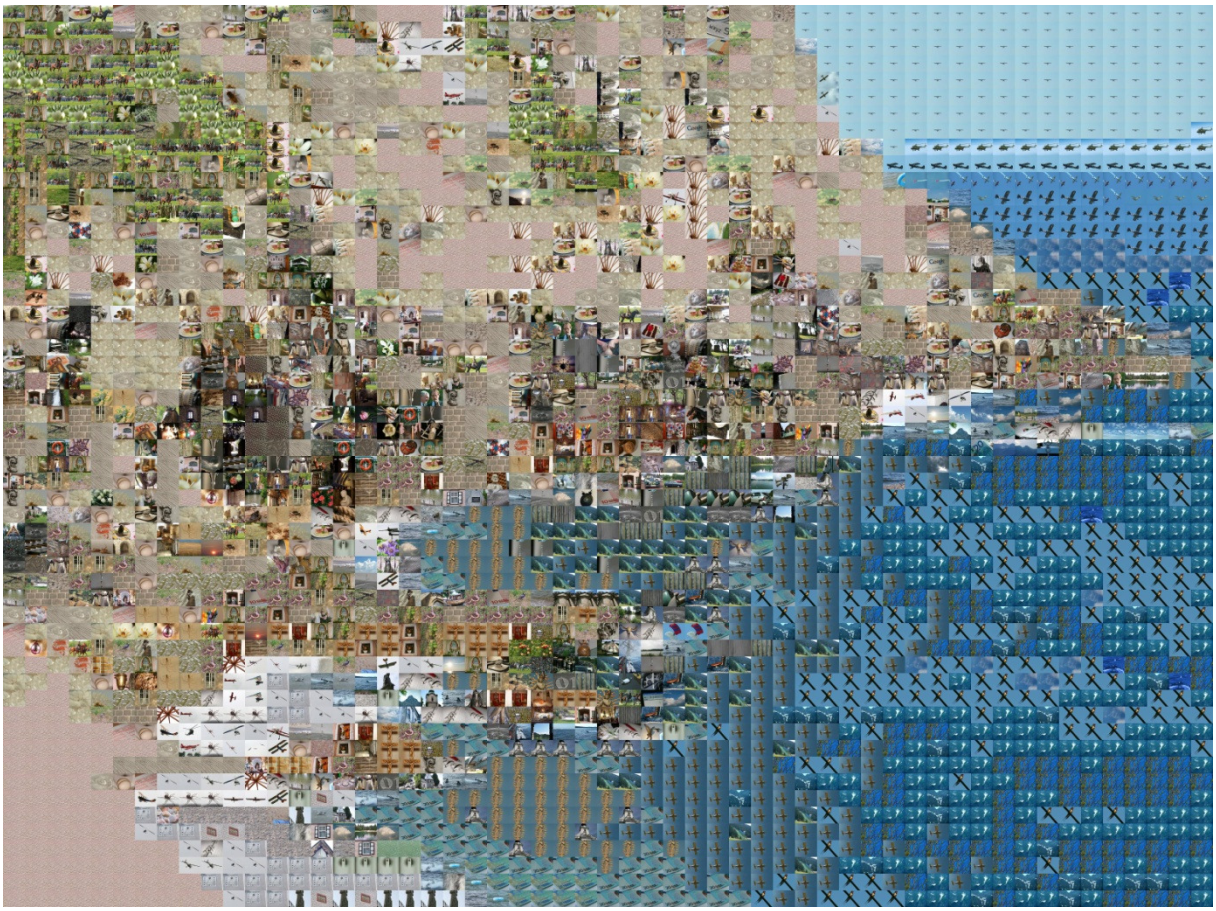
Duża ilość składowych zdjęć pozwoliła na dosyć dokładne odwzorowanie szczegółów. W tym przypadku dosyć widoczna jest z kolei niedoskonałość algorytmu znajdującego najbardziej podobne zdjęcie małe do fragmentu dużego zdjęcia. M.in. drugim rzędzie z dołu widać serię „nie pasujących" obrazków (biały kolor po lewej i prawej stronie, ciemnoczerwony w środku). Powodem wstawienia tego obrazka jest podział zdjęcia na 9 fragmentów. W przypadku tego zdjęcia składowego, fragmenty białego tła obrazka są analizowane w połączeniu z częścią faktycznego zdjęcia (kolor ciemnoczerwony), przez co średnia jest dla programu najbardziej podobna do oryginału. Błąd można poprawić poprzez podział obrazka na więcej elementów (co jednak negatywnie wpłynie na prędkość działania aplikacji), lub zastosować inny podział, niż na równe fragmenty, co byłoby jednak bardzo kłopotliwe w implementacji.

Poniżej prezentuję wyniki działania aplikacji dla zdjęcia autorstwa p. Artura Kowalczyka, pobranego z serwisu: http://darmowe-zdjecia.org/.

Fotomozaika wygenerowana na podstawie poniższego zdjęcia przy powiększeniu 100% i wysokości składowej: 44px (czas generowania – ok. 30 sekund):

Widać na tym przykładzie niedoskonałości algorytmu wyszukującego zdjęcie najbardziej podobne do oryginału – ponownie problem dotyczy zdjęć składowych, dla których podział na 9 elementów nie jest wystarczający (wyraźnie wyróżniają się pewne elementy zdjęć, który po uśrednieniu w kwadracie zostają uznane za podobne do przedstawionego oryginalnie elementu dużego zdjęcia.

Fotomozaika wygenerowana na podstawie poniższego zdjęcia przy powiększeniu 200% (grafika pomniejszona do szerokości strony) i wysokości składowej: 44px (czas generowania – ok. 60 sekund):



Na tym przykładzie podobieństwo do oryginału jest znacznie większe, niż w poprzednim przypadku, co wynika z zastosowania większej liczby obrazków składowych. Podobnie jak w poprzednich przykładach widać tu potrzebę zmiany algorytmu wyszukującego zdjęcie składowe najbardziej podobne do oryginalnego fragmentu zdjęcia oraz wykorzystania bardziej zróżnicowanej kolorystycznie bazy zdjęć składowych.

8. Propozycje zmian

a. Prędkość działania aplikacji

Istnieją dwa podstawowe sposoby na przyspieszenie działania aplikacji. Pierwszym z nich jest jest zapisanie w bazie danych średnich kolorów zdjęć składowych. Wiązało by się to z brakiem konieczności wyliczania średnich „w locie”. Wadą takiego rozwiązania jest jednak konieczność napisania dodatkowej aplikacji (lub funkcjonalności), dodającej zdjęcia do bazy. Alternatywnym rozwiązaniem jest przeszukiwanie katalogu jednorazowo, po uruchomieniu aplikacji i indeksowanie znajdujących się w nim zdjęć.

Drugim istotnym sposobem na przyspieszenie działania aplikacji jest zastąpienie wykorzystanych w programie metod GetPixel() oraz SetPixel() na ich odpowiedniki, działające na tablicy bitów. Jest to rozwiązanie znacznie trudniejsze w implementacji i mogące powodować pewne zagrożenie (konieczość korzystania ze wskaźników – możliwość wystąpienia błędów ochrony pamięci lub „mazania po pamięci" w przypadku błędu w aplikacji), jednak takie rozwiązanie znacznie przyspieszyłoby działanie aplikacji.

b. Dokładność wygenerowanej mozaiki

Generowana przez program mozaika, przy odpowiednio dużej ilości zdjęć składowych wydaje się dobra i wierna oryginałowi. Polepszyć mogłoby ją zwiększenie standardowej rozdzielczości, jednak wymagałoby to rozwiązania problemu z ograniczoną ilością pamięci. Jednym ze spososób może być zapisywanie danych do plików lub bazy danych zamiast do pamięci, to jednak wiązałoby się z kolei ze znacznym zmniejszeniem wydajności aplikacji.

Drugim rozwiązaniem może być zastosowanie wydajniejszego algorytmu do porównywania obrazów oraz podzielenie każdej części obrazu na większą liczbę fragmentów. Miałoby to jednak sens tylko przy niewielkiej liczbie fragmentów i dużej liczbie zdjęć składowych w bazie danych. Dla losowych zdjęć i przy dużej liczbie fragmentów, nie zmieniłoby to znacząco wyniku.

Ważnym powodem różnic pomiędzy prezentowanymi w poprzednim punkcie mozaikami w stosunku do oryginalnych grafik jest zbyt mała różnorodność obrazków składowych (grafik jest 1500, jednak wiele z nich jest podobnych do innych. Aby wyniki działania aplikacji były bardziej satysfakcjonujące, należy udostępnić aplikacji katalog z większą liczbą bardziej zróżnicowanych fotografii składowych. Wpłynie to negatywnie na czas działania aplikacji (przeszukiwanie katalogu w poszukiwaniu najlepszego zdjęcia), jednak będzie miało pozytywny wpływ na efekt jej działania.

9. Opis implementacji

a. Struktura projektu

Projekt został podzielony na dwa projekty składowe:

- PhotoProcessing – projekt, którego zadaniem jest przechowywanie obrazów oraz przetwarzanie ich, funkcjonujący jako biblioteka dll. Projekt składa się z następujących klas:
    - BigPhoto – klasa statyczna, przechowująca obraz podany przez użytkownika, oraz odpowiadająca za odpowiednie przycięcie obrazu, jeżeli jest to konieczne, podział obrazu na małe obrazki, które później są zastępowane zdjęciami składowymi, oraz wstawienie zdjeć składowych w odpowiednie miejsca obrazu użytkownika.
    - SmallPhoto – klasa zawierająca zarówno mały obrazek, jak również zdjęcie składowe, zawierająca dodatkowo tablicę średnich kolorów oraz odpowiadająca

za wypełnienie tej tablicy, a także analizę podobieństwa obrazów oraz zastąpienie obrazka zdjęciem składowym.
    - ComponentPhotoList – klasa statyczna, zawierająca listę zdjęć składowych (obiektów klasy SmallPhoto), odpowiadająca za odczyt tej listy z odpowiedniego katalogu oraz zawierająca metodę pozwalającą na wyciągnięcie średniego koloru z fragmentu zdjęcia.
- UserInterfaces – projekt, odpowiadający za komunikację z użytkownikiem, zawierający:
    - FormMain – forma główna, zawierająca także obsługę importu zdjęcia, eksportu wygenerowanej mozaiki do pliku, oraz obsługę przycisków na formie poprzez wywołanie odpowiednich metod i nadanie wartości polom klas projektu PhotoProcessing.
    - FormViewFullSize – forma pozwalająca na wyświetlenie zdjęcia użytkownika lub mozaiki w pełnych rozmiarach,

b. Odczyt obrazu użytkownika z pliku

Za tę funkcjonalność odpowiada metoda btnPickPhoto_Click(). Odczyt obrazu użytkownika z pliku odbywa się w następujący sposób:

- Użytkownik jest proszony o podanie pliku do odczytu poprzez otwarcie odpowiedniej kontrolki.
- Metoda odczytuje z formy i zapisuje do odpowiednich zmiennych powiększenie i wysokość zdjęcia składowego.
- Wybrane przez użytkownika zdjęcie jest po przeskalowaniu zapisywane do pola klasy statycznej BigPhoto i wyświetlane na formie.
- Wywoływana jest metoda odpowiadająca za przycięcie obrazu użytkownika, jeżeli jest to konieczne.
- W przypadku wystąpienia błędów wyświetlane są stosowne komunikaty w postaci alertów MessageBox.
- Włączany jest przycisk do wygenerowania mozaiki.

Cała definicja metody wygląda następująco:

```
private void btnPickPhoto_Click(object sender, EventArgs e)
{
   DialogResult result = DialogResult.None;
   try
   {
     result = dlgImportBigPhoto.ShowDialog();
   }
   catch (Exception)
   {
     result = DialogResult.None;
     return;
   }
   switch (result)
   {
     case DialogResult.OK:
     case DialogResult.Yes:

       int zoomIn = 400;
       bool parseResult = Int32.TryParse(txtZoomIn.Text, out zoomIn);
       this.txtZoomIn.Text = zoomIn.ToString();

       int smallPhotoHeight = 88;
       parseResult = Int32.TryParse(txtLittlePhotoHeight.Text, out smallPhotoHeight);
```

```
        this.txtLittlePhotoHeight.Text = smallPhotoHeight.ToString();

        try
        {
          Image image = new Bitmap(dlgImportBigPhoto.FileName);
             Size size = new Size((int)(image.Width * (float)(zoomIn / 100)),
             (int)(image.Height * (float)(zoomIn / 100)));
          BigPhoto.SmallPhotoHeight = smallPhotoHeight;
          if (BigPhoto.Image != null)
            BigPhoto.Image.Dispose();
          BigPhoto.Image = new Bitmap(image, size);
        }
        catch (Exception)
        {
             MessageBox.Show("Wystąpił błąd podczas próby otwarcia pliku.", "BŁĄD",
             MessageBoxButtons.OK, MessageBoxIcon.Error);
          return;
        }

        try
        {
          if (this.pcbMosaic.Image != null)
            this.pcbMosaic.Image.Dispose();
          BigPhoto.Image = BigPhoto.CropImage();
        }
        catch (Exception)
        {
             MessageBox.Show("Wybrany obraz ma nieprawidłowy rozmiar.", "BŁĄD",
             MessageBoxButtons.OK, MessageBoxIcon.Error);
          return;
        }

        this.btnGenerate.Enabled = true;

        this.txtSourcePath.Text = dlgImportBigPhoto.FileName;

        this.pcbMosaic.Image = BigPhoto.Image;
        this.resizePhotoToShow(this.pcbMosaic);
        this.centerPhotoToShow(this.pcbMosaic);
        this.pcbMosaic.Refresh();

        break;
      default:
        break;
    }
}
```

c. Wstępne przygotowanie obrazu

Funkcjonalność polega na sprawdzeniu, jakie powinny być wymiary obrazka i przycięciu jeżeli jest to konieczne. Jest to realizowane przez metody GetProperSize() oraz CropImage() znajdujące się w klasie statycznej BigPhoto.

Metoda GetProperSize() zwraca nową wysokość i szerokość obrazu. Ciało metody wygląda następująco:

```
private static Size getProperSize()
{
  int newHeight = Image.Height;
  int newWidth = Image.Width;

  if (Image.Height % SmallPhotoHeight != 0)
  {
   newHeight = SmallPhotoHeight * (int)Math.Floor((decimal)(Image.Height /
   SmallPhotoHeight));
  }
  if (Image.Width % SmallPhotoWidth != 0)
  {
   newWidth = SmallPhotoWidth * (int)Math.Floor((decimal)(Image.Width / SmallPhotoWidth));
  }

  return new Size(newWidth, newHeight);
```

```
}
```

Metoda CropImage() odpowiada za przycięcie obrazka do odpowiedniej wielkości w taki sposób, aby dostosować wielkość do wielkości zdjęć składowych określonej przez użytkownika. Ciało tej metody wygląda następująco:

```
public static Bitmap CropImage()
{
  Size size = getProperSize();

  int xMin = (int)Math.Floor((float)((Image.Width - size.Width) / 2));
  int yMin = (int)Math.Floor((float)((Image.Height - size.Height) / 2));

  Rectangle cropArea = new Rectangle(xMin, yMin, size.Width, size.Height);

  BigPhoto.Image = BigPhoto.Image.Clone(cropArea, PixelFormat.Format24bppRgb);

  return BigPhoto.Image;
}
```

d. Przygotowanie zdjęć składowych

Przygotowanie zdjęć składowych jest wykonywane w metodzie getComponentPhotoList(), która wykonuje następujące czynności:

- Pobiera listę plików z katalogu Photos, znajdującego się w głównym katalogu programu.
- Zwiększa maksymalną wartość paska postępu o liczbę znalezionych plików
- Próbuje tworzyć bitmapę z każdego napotkanego pliku
- Jeżeli uda się stworzyć bitmapę, dodaje ją do statycznej listy zdjęć składowych
- Zwiększa wskaźnik paska postępu po każdym odczytanym pliku

Kod metody wygląda następująco:

```
public static void GetComponentPhotoList(ref ProgressBar progressBar)
{
    string[] fileArray = new
    string[Directory.GetFiles(AppDomain.CurrentDomain.BaseDirectory + "/Photos").Count()];
  progressBar.Maximum += fileArray.Count();
  fileArray = Directory.GetFiles(AppDomain.CurrentDomain.BaseDirectory + "/Photos");
  SmallPhotoList = new List<SmallPhoto>();

  foreach (string fileName in fileArray)
  {
    try
    {
      SmallPhoto smallPhoto = new SmallPhoto(new Bitmap(fileName));
      SmallPhotoList.Add(smallPhoto);
      progressBar.Value++;
    }
    catch(Exception)
    {
      progressBar.Value++;
    }
  }
}
```

e. Podział obrazu użytkownika na fragmenty

Funkcjonalność ta jest realizowana przez metodę getRectangles() znajdującą się w klasie statycznej BigPhoto.

Metoda nadaje odpowiednią wartość maksymalną paskowi postępu, równą liczbie elementów, na który obraz został podzielony, a następnie iteruje po obrazie użytkownika, za każdym razem przesuwając się o określoną przez użytkownika wysokość zdjęcia składowego, a po zakończeniu kolumny o szerokość zdjęcia składowego. Każdy prostokąt jest dodawany do listy małych obrazków, które następnie są zastępowane zdjęciami. Przy każdym obiegu jest również zwiększana wartość wyświetlana na pasku postępu. Wynikiem zwracanym przez metodę jest lista małych fragmentów obrazu.

Ciało tej metody przestawia się następująco:

```
public static List<SmallPhoto> getRectangles(ProgressBar progressBar)
{
  int xCount = Image.Width / SmallPhotoWidth;
  int yCount = Image.Height / SmallPhotoHeight;

  progressBar.Minimum = 0;
  progressBar.Maximum = xCount * yCount;
  progressBar.Value = 0;

  int x = 0;
  int y = 0;

  ComponentPhotoList.GetComponentPhotoList(ref progressBar);
  List<SmallPhoto> list = new List<SmallPhoto>();

  while (x < xCount)
  {
    while (y < yCount)
    {
          Rectangle area = new Rectangle(x * SmallPhotoWidth, y * SmallPhotoHeight,
          SmallPhotoWidth, SmallPhotoHeight);
          list.Add(new SmallPhoto(Image.Clone(area, PixelFormat.Format24bppRgb), new
          Point(x * SmallPhotoWidth, y * SmallPhotoHeight)));
      progressBar.Value += 1;
      y++;
    }
    y = 0;
    x++;
    progressBar.Refresh();
  }

  return list;
}
```

f. Wstawienie zdjęć składowych do obrazu użytkownika

Funkcjonalność realizowana przez metodę Replace() znajdującą się w klasie SmallPhoto. Metoda korzysta z metody checkSimmilarity(), która zwraca wynik funkcji odwrotnie proporcjonalnej do podobieństwa obrazów (zwracającej zero w przypadku obrazów identycznych). Po wywołaniu tej metody Replace() zastępuje fragment małego obrazka najbardziej odpowiednim zdjęciem składowym.

Metoda checkSimmilarity() zwraca sumę kwadratów różnic poszczególnych składowych kolorów (czerwonej, zielonej, niebieskiej) obu obrazów:

```
  return Math.Abs(
          Math.Pow(
            Math.Pow((this.averageColor[0].R - smallPhoto.averageColor[0].R), 2)
            + Math.Pow((this.averageColor[0].G - smallPhoto.averageColor[0].G), 2)
            + Math.Pow((this.averageColor[0].B - smallPhoto.averageColor[0].B), 2), 1) +
          Math.Pow(
              Math.Pow((this.averageColor[1].R - smallPhoto.averageColor[1].R), 2)
            + Math.Pow((this.averageColor[1].G - smallPhoto.averageColor[1].G), 2)
            + Math.Pow((this.averageColor[1].B - smallPhoto.averageColor[1].B), 2), 1) +
          Math.Pow(
              Math.Pow((this.averageColor[2].R - smallPhoto.averageColor[2].R), 2)
            + Math.Pow((this.averageColor[2].G - smallPhoto.averageColor[2].G), 2)
            + Math.Pow((this.averageColor[2].B - smallPhoto.averageColor[2].B), 2), 1) +
          Math.Pow(
              Math.Pow((this.averageColor[3].R - smallPhoto.averageColor[3].R), 2)
            + Math.Pow((this.averageColor[3].G - smallPhoto.averageColor[3].G), 2)
            + Math.Pow((this.averageColor[3].B - smallPhoto.averageColor[3].B), 2), 1)
      );
```

Natomiast opisana wyżej metoda Replace() wygląda następująco:

```
public void Replace()
{
  try
  {
    double maxSimilarity = double.MaxValue;
    SmallPhoto bestImage = ComponentPhotoList.SmallPhotoList.First();

    foreach (SmallPhoto smallPhoto in ComponentPhotoList.SmallPhotoList)
    {
      double similarity = checkSimilarity(smallPhoto);
      if (similarity < maxSimilarity)
      {
        maxSimilarity = similarity;
        bestImage = smallPhoto;
      }
    }

    this.Image = new Bitmap(bestImage.Image, this.Image.Size);

  }
  catch
  {
    return;
  }
}
```

g. Zapis wygenerowanej mozaiki do pliku

Zapis mozaiki do pliku jest realizowany przez metodę btnSave_Click(), znajdującą się w klasie formy FormMain. Metoda wyświetla odpowiednią kontrolkę do określenia wyjściowej nazwy pliku, oraz – w przypadku potwierdzenia – zapisuje plik w formacie JPEG do pliku o określonej przez użytkownika nazwie.

Cała funkcja jest bardzo prosta i została zaimplementowana w następujący sposób:

```
private void btnSave_Click(object sender, EventArgs e)
{
  DialogResult result = dlgExportMosaic.ShowDialog();
  switch (result)
  {
    case DialogResult.OK:
    case DialogResult.Yes:

      BigPhoto.Image.Save(dlgExportMosaic.FileName, ImageFormat.Jpeg);
```

```
            break;
        default:
            break;
    }
}
```

h. Interfejs użytkownika

Interfejs użytkownika jest realizowany przez dwie klasy, reprezentujące oddzielne formy:

- FormMain – forma główna, składająca się z kontrolek służących użytkownikowi do określenia powiększenia oraz wysokości zdjęcia składowego (szerokość to zawsze 4/3 wysokosci), zawierająca również pogląd wczytanego obrazu, a później wygenerowanej mozaiki, oraz przyciski służące do wczytania i zapisania pliku, oraz wygenerowania mozaiki. W czasie generowania mozaiki jest wyświetlany na niej pasek postępu. Forma zawiera też przycisk otwierający podgląd zdjęcia lub mozaiki w pełnej wielkości.
- FormViewFullSize – wyświetla podgląd zdjęcia lub wygenerowanej mozaiki w pełnej wielkości. Jeżeli obraz nie mieści się na ekranie, są wyświetlane paski przewijania.

10. Kod źródłowy

Pełny kod źródłowy aplikacji znajduje się w załączniku do sprawozdania – source.rar.

11. Bibliografia

1. Serwis foto.otwarty.pl (zdjęcia składowe).
2. Michał Kowalczyk – „Sortowanie obrazów”
3. Serwis pl.wikipedia.org – przykładowe obrazy
4. Biblioteka MSDN – msdn.microsoft.com