

Dokumentacja Podpowiadarki do puzzli - Puzzle Matcher

1. Wstęp

Spis treści

Wstęp	1
Spis treści	1
Opis	2
Motywacja	2
Podział i krótki przebieg prac:	3
Podział	3
Adam Godziński:	3
Dawid Korach:	3
Przebieg pracy	4
Podsumowanie prac	6
Funkcje programu Puzzle Matcher	7
Rozpoznawanie puzzli.	7
Układanie puzzli z pomocą obrazu na którym są one ułożone	9
Zapisywanie wyników pracy programu.	9
Wybrane Technologie	10
Architektura i algorytmy	11
Architektura	11
Diagram zależności	11
Diagramy klas	12
Algorytmy	14
Algorytm rozpoznawania puzzli.	14
Algorytm wykrywania konfiguracji w jakiej trzeba ułożyć puzzle.	15
Wykrywanie punktów wspólnych i obliczenie dla nich średniej pozycji.	15
Algorytm układania puzzli.	16
Algorytm generowania obrazu	19
Interesujący problemy i błędy oraz jak z nimi się uporaliśmy	20
Jak obrobić zdjęcie tak aby lepiej wykrywa krawędzie?	20
Problemy z dopasowywaniem puzzli do obrazu	21
Instrukcja Użytkownika	22
Bibliografia	31

1.1. Opis

Puzzle Matcher to rozwijany przez dwóch studentów: Adama Godzińskiego oraz Dawida Koracha, program, którego głównym zadaniem jest przeanalizowanie i ułożenie w poprawny sposób porozrzucanych puzzli. Aplikacja korzysta z wielu autorskich rozwiązań i algorytmów, takich jak: [algorytm do wykrywania konfiguracji puzzli](#) czy [algorytm układania puzzli](#). Program zapewnia: podgląd wykrytych puzzli, wygenerowany obraz z ułożonymi puzzlami oraz instrukcję jak ułożyć puzzle. Oprócz tego minimalistyczny, łatwy w obsłudze interfejs użytkownika oraz możliwość zapisania wyników są mocnymi stronami programu.

1.2. Motywacja

Projekt został wybrany głównie z powodu tematu z którym się on oczywiście pokrywa, a jest to mianowicie: przetwarzanie obrazów. Jako że mieliśmy wcześniej już do czynienia z tym zagadnieniem i podobała się nam praca nad nim, postanowiliśmy kolejny raz do niego przysiąść. Dodatkowo sam projekt jest swego rodzaju wyzwaniem, trudno nam było przypomnieć sobie czy istnieje już taki system oraz czy jesteśmy go w stanie zaimplementować.

2. Podział i krótki przebieg prac:

2.1. Podział

Podsumowując wykonane przez nas zadania można podzielić w następujący sposób:

Adam Godziński:

- Rozpoznawanie puzzli.
- Numerowanie wykrytych puzzli.
- Wykrywanie konfiguracji w jakiej trzeba ułożyć puzzle.
- Wykrywanie punktów wspólnych.
- Układanie puzzli.
- Testowanie i debug.
- Generowanie ostatecznego obrazu.

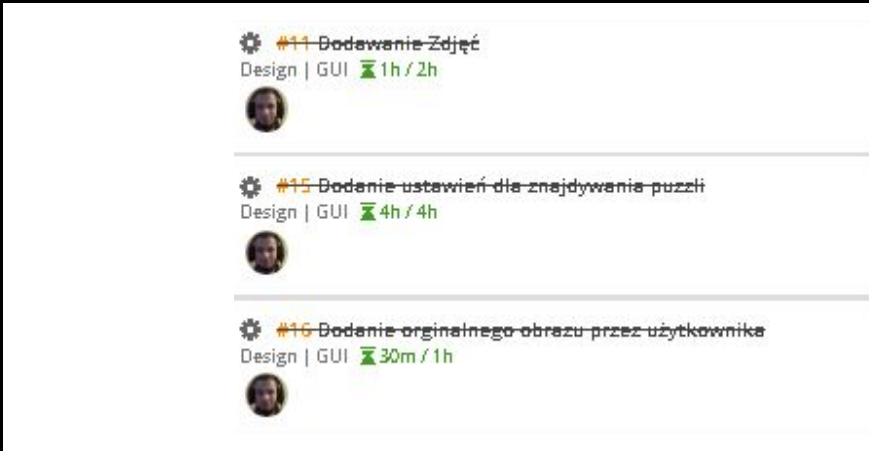
Dawid Korach:

- Wykonanie interfejsu użytkownika.
- Wczytywanie obrazu.
- Rozpoznawanie puzzli.
- Wyświetlanie w jaki sposób ponumerowane puzzle zostały ułożone.
- Zapisywanie do wyniku układania do pliku.
- Wykonanie podglądu które puzzle zostały wykryte.
- Testowanie i debug.

2.2. Przebieg pracy

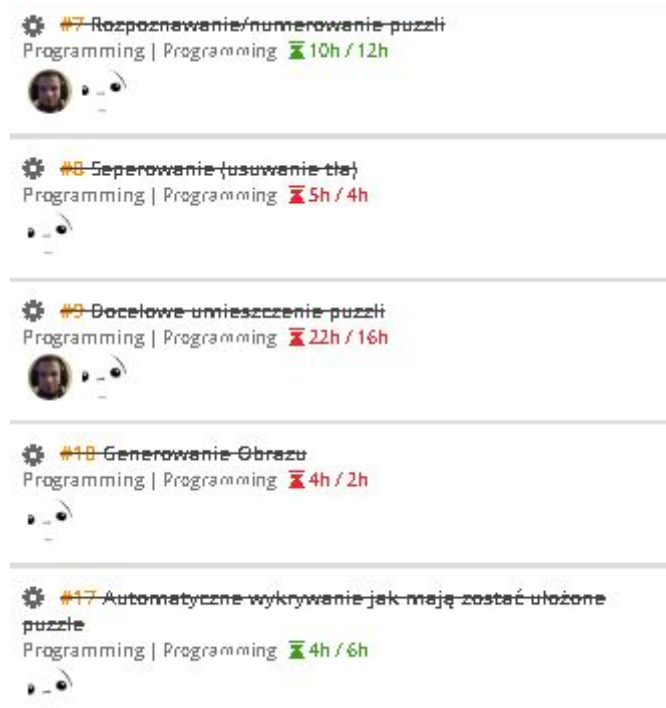
Na początku podzieliliśmy się zadaniami.

Adam Godziński zajmował się pracami związanymi z wstępnym przetwarzaniem obrazu. Natomiast Dawid Korach zajmował się projektowaniem programu i wykonaniem interfejsu użytkownika a także samym wczytywaniem obrazu do programu.

 <p>The screenshot shows a list of tasks from the Hack&plan application. Each task is preceded by a gear icon and a task ID in a yellow box. The tasks are: #11 Dodawanie Zdjęć (Design GUI, 1h / 2h), #15 Dodanie ustawień dla znajdowania puzzli (Design GUI, 4h / 4h), and #16 Dodanie oryginalnego obrazu przez użytkownika (Design GUI, 30m / 1h). Each task has a small circular profile picture of a person to its left.</p>
<p>Rys. 1 - Lista zadań z Hack&plan, związanych z interfejsem użytkownika, którą zajmował się Dawid Korach. Zadania zostały wykonane w początkowym etapie rozwoju aplikacji.</p>
<p>Źródło: Prywatna tablica sum-like na stronie https://app.hacknplan.com</p>

Podstawy Teleinformatyki; Rok: 3; Semestr: 5

Wraz z progresem prac Dawid wykonał projekt GUI i zaczął pracować wraz z Adamem nad algorytmami wykrywania i rozpoznawania puzzli. Podobnie było również z samym algorytmem układającym. Po dojściu do punktu gdzie puzzle już mogły być ułożone skupiliśmy się na przedstawieniu tego jak mają zostać ułożone przez nasz program. Adam zrobił generowanie ostatecznego obrazu, a Dawid podpowiedzi do układania (rozłożenie ponumerowanych puzzli).



Rys. 2 - Lista zadań z Hack&plan.

Źródło: Prywatna tablica sum-like na stronie <https://app.hacknplan.com>

Następnie rozpoczęliśmy testowanie oraz wynajdywanie problemów a także rozwiązywanie ich. Dawid dodał dodatkowe opcje w programie takie jak zapisywanie wyniku oraz podgląd tego które puzzle zostały zaznaczone. Adam dodał jeszcze usprawnienie które typuje w jakiej konfiguracji powinny zostać te puzzle ułożone.



Rys. 3 - Lista zadań z Hack&plan.

Źródło: Prywatna tablica sum-like na stronie <https://app.hacknplan.com>

2.3. Podsumowanie prac

Category Metrics

	Tasks				Hours			
	Estimated	Pending	Closed	Progress	Estimated	Logged	Unused	Remaining
Programming	7	0	7	100%	44h	49h 30m	- 5h 30m	0
Design	5	0	5	100%	18h	16h 1m	1h 59m	0
Documentation	3	0	3	100%	12h	19h 55m	- 7h 55m	0
	15	0	15	100%	74h	85h 26m	- 11h 26m	0

Category Distribution



Rys. 4 - Tabela z wykresami podsumowująca czas pracy nad projektem.

Źródło: Prywatna tablica scrum-like na stronie <https://app.hacknplan.com>

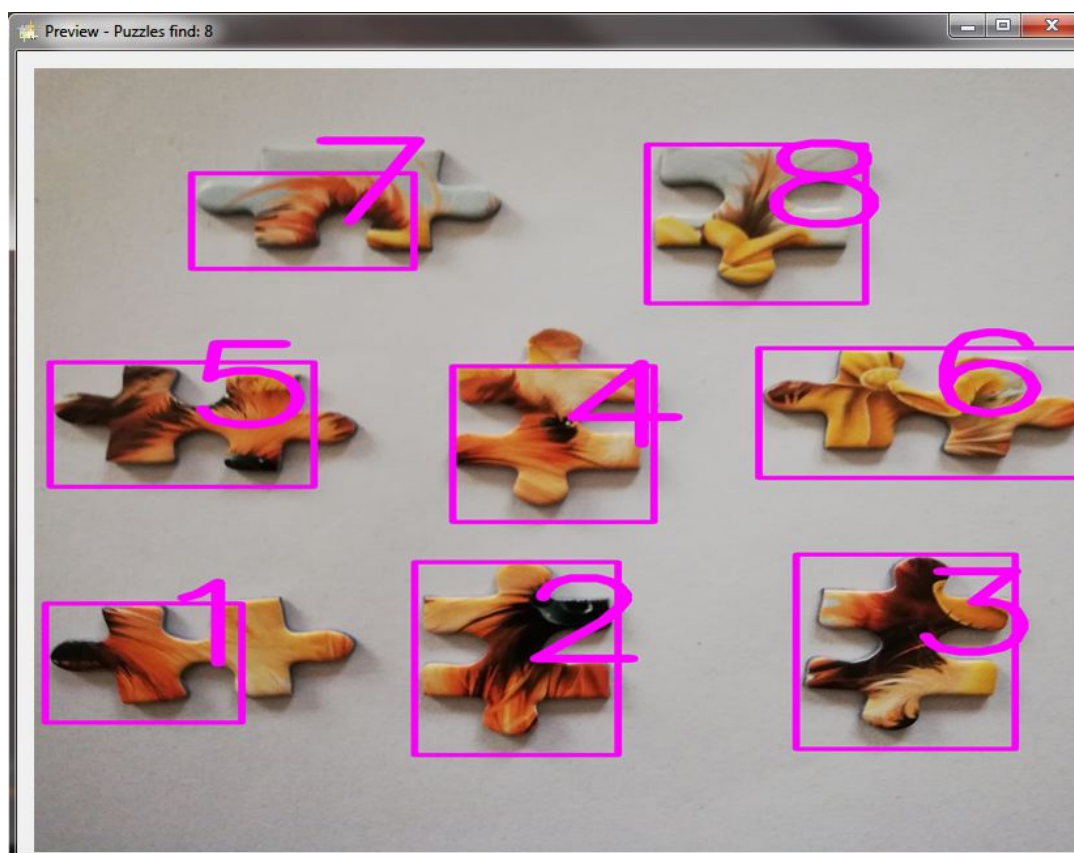
Całość projektu zajęła nam około 85 godzin. Podczas pracy okazało się, że nieraz źle oszacowaliśmy przewidywaną ilość godzin na wykonanie zadania. Łączny błąd wynosi około 11 godzin. Czas poświęcony pisaniu logiki programu to około 60% całego czasu w projekcie. Ciekawą rzeczą jest fakt, że nad dokumentacją spędziliśmy więcej czasu, a niżeli przy interfejsie graficzny.

3. Funkcje programu Puzzle Matcher

Tytuł projektu już sam dyktuje funkcjonalność. Podstawowymi funkcjami naszego programu są:

3.1. Rozpoznawanie puzzli.

Aplikacja potrafi rozpoznać z większą lub mniejszą dokładnością puzzle znajdujące się na ekranie. Ta funkcjonalność oparta jest na algorytmie do rozpoznawania puzzli - opisanego w [dalszej części dokumentacji](#)



Rys. 5 -Przykładowe rozpoznanie puzzli.

Źródło: Obraz własny.



Rys. 6 - Przykładowe rozpoznanie puzzli prostokątnych.

Źródło: Obraz własny.

Dodatkowo puzzle są numerowane według kolejności w której zostały wykryte. Pozwala to na późniejsze ich ułożenie.

3.2. Układanie puzzli z pomocą obrazu na którym są one ułożone

Program mając do dyspozycji dodatkowy obraz z już ułożonymi puzzlami, potrafi je złożyć.

Puzzle są dopasowywane na podstawie wykrytych cech (SURF). Następnie cechy są dopasowywane względem ułożonego obrazka. Kolejno na podstawie wykrytych cech wspólnych, liczymy średnie pozycje puzzli względem oryginalnego obrazu i układamy na podstawie wykrytej/zadanej konfiguracji.

Ta funkcjonalność oparta jest na o [algorytm generowania obrazu](#).



Rys. 7 - Obraz z ułożonymi puzzlami.

Źródło: Obraz własny

3.3. Zapisywanie wyników pracy programu.

Na ekranie końcowy jest duży przycisk ("Zapisz wszystko do folderu"), który pozwala na zapisanie zdjęć w wskazanej przez użytkownika lokalizacji.

4. Wybrane Technologie



Źródło:

https://upload.wikimedia.org/wikipedia/commons/thumb/3/32/OpenCV_Logo_with_text.svg/1200px-OpenCV_Logo_with_text.svg_version.svg.png

Do przetwarzania obrazów wybraliśmy znaną nam już bibliotekę OpenCV. Jednak nie było jedyny powód. OpenCv jest jedną z najbardziej popularnych bibliotek służących do przetwarzania, ma dobrą dokumentację oraz działa sprawnie.



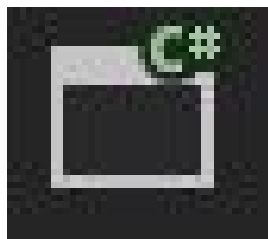
Źródło: <https://jaki-jzyk-programowania.pl/img/csharp.png>



Źródło: http://www.emgu.com/wiki/index.php/Main_Page

Na początku chcieliśmy aby użytym przez nas językiem programowania był Python, jako że to właśnie w nim po raz pierwszy nauczyliśmy się używać OpenCV. Jednak po rozpoczęciu prac stwierdziliśmy że do wykonania interfejsu użytkownika lepszy będzie C#.

Zmusiło nas to do użycia wrappera OpenCV dla C# czyli EmguCV.



Źródło: Obraz pochodzi z programu VisualStudio 2017

Ze względu na brak konieczności silnego rozdzielania logiki programu od jego interface'u jako model aplikacji wybraliśmy WinForms. Dodatkowo, każdy z nas miał pewne doświadczenie w pisaniu programów w oparciu o ten model, co niekoniecznie można było powiedzieć o np.: WPF.



Źródło:

https://upload.wikimedia.org/wikipedia/commons/thumb/6/61/Visual_Studio_2017_logo_and_wordmark.svg/2000px-Visual_Studio_2017_logo_and_wordmark.svg.png

Jako środowisko programistyczne wybraliśmy Visual Studio, ponieważ dzięki zainstalowanym wcześniej dodatkom, integracji z repozytorium wersji (GitHub) oraz z czasem już w nim poświęconym, zapewniliśmy wysoką wydajność i jakość pisanego przez nas kodu.



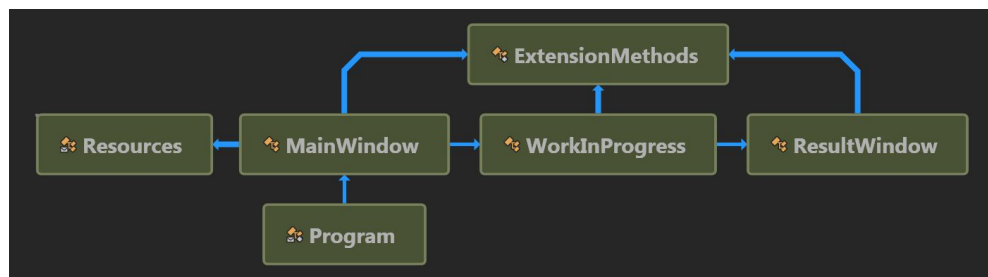
Źródło:
<http://www.aisoftwarellc.com/media/image?path=NuGet%20Logo%202.png>

Aby połączyć i zintegrować technologie takie jak: OpenCV, EmguCV oraz WinForms użyliśmy Menadżera Pakietów NuGet. Dzięki takiemu rozwiązaniu, byliśmy w stanie zrobić to szybko i łatwo.

5. Architektura i algorytmy

5.1. Architektura

Diagram zależności



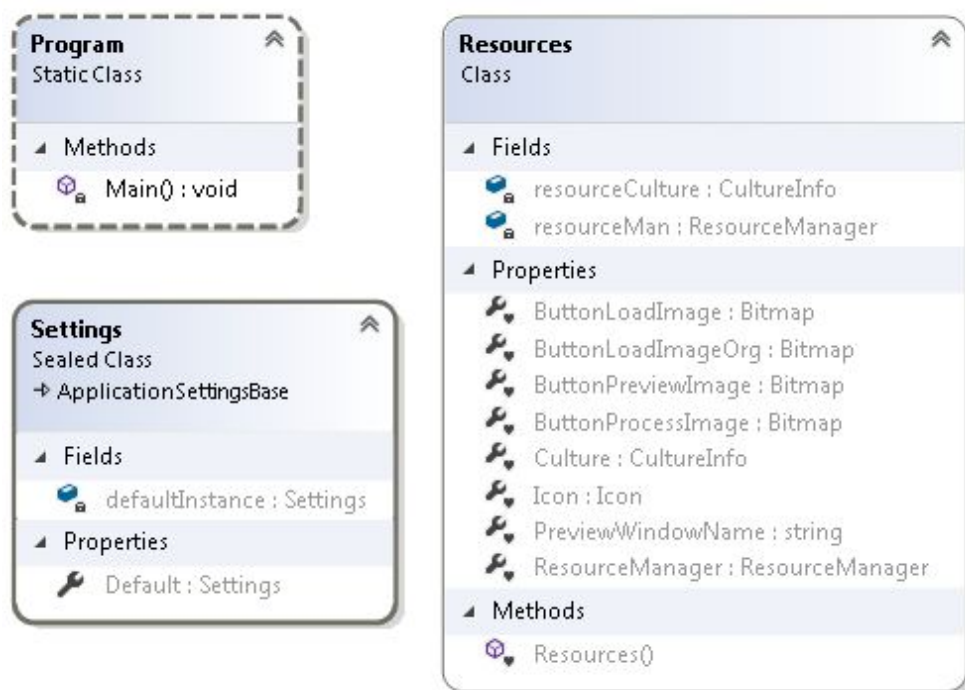
Rys. 8 - Program aby wyświetlić okno główne korzysta z klasy *MainWindow*. Z pomocą klasy *Resources* jest zaopatrywana w poprawny wygląd elementów graficznych. Parametry konfiguracyjne są zapisywane w klasie *ExtensionMethods*.

Następnie program przechodzi do okna na bazie klasy *WorkInProgress*. To w niej są wykonywane wszystkie obliczenia. Klasa ta jest silnie uzależniona od klasy *ExtensionMethods*, ponieważ to w niej znajdują się wszystkie wywoływane metody oraz dane wejściowe do wszystkich algorytmów.

Po zakończeniu obliczania, otwiera się okno korzystające z klasy *ResultWindow*. Ponieważ wyniki są zapisane w *ExtensionMethods*, to musi z niej skorzystać.

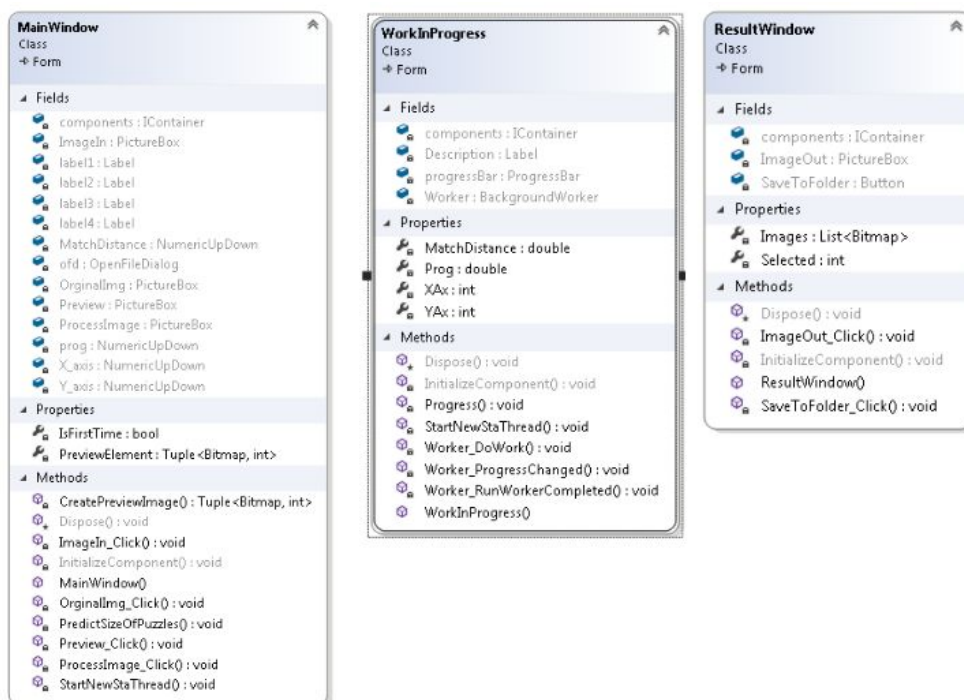
Źródło: Obraz własny z programu VisualStudio 2017.

Diagramy klas



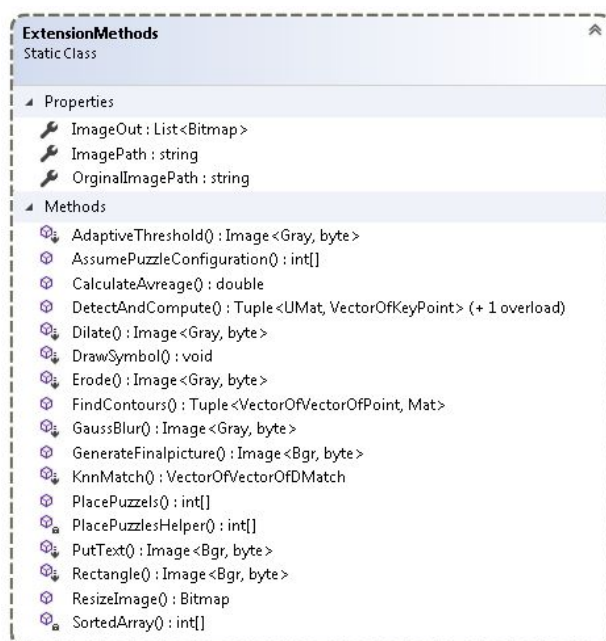
Rys. 9 -Wnętrze klas: *Program*, *Settings* oraz *Resources*.

Źródło: Obraz własny wygenerowany za pomocą VisualStudio 2017.



Rys. 10 - Wnętrze klas: *MainWindow*, *WorkInProgress* oraz *ResultWindow*.

Źródło: Obraz własny.



Rys. 11 - Klasa *ExtensionMethods* wygenerowany za pomocą VisualStudio 2017.

Źródło: Obraz własny.

5.2. Algorytmy

Algorytm rozpoznawania puzzli.

Na początku użytkownik musi wybrać obraz na którym znajdują się nie ułożone puzzle. Następnie po wczytaniu zdjęcia tworzymy jego czarno-białą kopię na której wykonujemy wstępną obróbkę. Obróbka oznacza lekkie rozmycie obrazu za pomocą rozmycia gaussowskiego. Potem używane jest progowanie adaptatywne aby łatwiej było znaleźć krawędzie puzzli. Kolejnym krokiem jest dylatacja za pomocą której “uwypuklamy” krawędzie aby jeszcze łatwiej było je znaleźć. Mając to wszystko używamy OpenCV do odnajdywania konturów. Wykryte są zapisywane do listy.

Oczywiście chociaż po obróbce pozbyliśmy się większości szumów, cały czas mogło coś pozostać, dodatkowo oprócz konturu puzzla mogły zostać znalezione dodatkowe w samych puzzlach. Dlatego następnym krokiem algorytmu jest obliczanie średniej wielkości wykrytych konturów.

Wielość konturów wyznaczmy za pomocą funkcji z OpenCV `ContourArea`. Następnie iterujemy przez listę i na każdym konturze używamy wcześniej wspomnianej funkcji a wynik sumujemy na zmiennej którą następnie dzielimy przez liczbę wykrytych konturów. Mając średnią przechodzimy do filtrowania znalezionych konturów i odrzucamy te które są poniżej tej wartości oraz dodatkowego stałego progu. Ten próg jest wartością którą użytkownik może modyfikować. Dzięki temu program jest w stanie znaleźć puzzle niezależnie od rozdzielczości zdjęcia.

Po prawidłowym odfiltrowaniu konturów zostały te największe wrzucamy je do tablicy i oznaczamy jako puzzle. Następnie oznaczamy je przy pomocy “bounding rectangle” i wrzucamy do listy oraz przypisujemy im numer.

Kolejnym krokiem jest odseparowanie puzzla od całego obrazu. Robimy to określając “bounding rectangle” jako ROI czyli Region of Interest (pl. region zainteresowania). W ten sposób odseparowany puzzle wrzucamy do listy dzięki której będziemy mogli przetwarzać obrazy puzzli pojedynczo.

Algorytm wykrywania konfiguracji w jakiej trzeba ułożyć puzzle.

Na wstępie wspomnę że ten algorytm na pewno da się jeszcze poprawić jako że nie zawsze podawany przez niego wynik jest prawidłowy. Zostały stworzone dwie wersje tego algorytmu pierwszy był trochę bardziej rozbudowany ale wymagał więcej obliczeń, a jako że używamy tego algorytmu w tworzeniu podglądu musieliśmy go ograniczyć tworząc drugą wersję której działanie opiszemy. Na początku algorytm bierzmy wykryte puzzle i odczytujemy ich pozycje. Mając je oblicza średnią wartość dla zmiennej X i Y reprezentujących odpowiednio pozycje na szerokości i wysokości. Mając ją liczy ile pozycji puzzli jest poniżej średniej. Następnie podejmuje decyzje. Jeżeli średnia liczba puzzli dla pozycji X jest równa tej dla Y zakłada że jest tyle samo puzzli w pionie co i w poziomie (3x3, 4x4). Jeżeli liczba pozycji X jest większa niż Y, dzieli on liczbę pozycji X przez sumę znalezionych puzzli i wynik tej operacji jest liczbą puzzli w pionie. Podobnie jest dla odwrotnej sytuacji, to znaczy gdy liczba pozycji Y jest większa od X wtedy dzielimy Y przez sumę puzzli i otrzymujemy liczbę puzzli w poziomie.

Wykrywanie punktów wspólnych i obliczenie dla nich średniej pozycji.

W momencie kiedy użytkownik wybrał już zdjęcie na którym zostały znalezione puzzle oraz obraz na którym jest ułożona cała układanka użytkownik naciska przycisk "Process Image" i wtedy rozpoczynamy układanie puzzli.

Pierwszym krokiem jest wykrycie punktów kluczowych. Robimy to używając algorytmu SURF (speeded up robust feature) zaimplementowanego w OpenCV. W skrócie jest algorytm służący do wykrywania cech i deskryptorów na zdjęciach. Służy też do innych celów takich jak klasyfikacja czy rejestracja obrazów. Z jego pomocą odnajdujemy i zapisujemy cechy w postaci punktów kluczowych(ang. keypoints). Tą czynność najpierw wykonujemy na obrazie z ułożoną układanką jako że będziemy dopasowywać każdy puzzle oddzielnie. Następnie jeszcze przed jakimkolwiek działaniem inicjujemy obiekt klasy BFMatcher dzięki któremu będzie możliwe dopasowywanie. Dzięki niemu jesteśmy w stanie znaleźć i narysować (zapałki) matches czyli dopasowane cechy. Teraz algorytm zaczyna obrabiać każdy puzzle po kolei. Najpierw wykrywa na nim cechy za pomocą SURFa a następnie odnalezione cechy porównuje z cechami oryginalnego obrazu dzięki BFMatcher. W ten sposób otrzymuje dopasowane cechy Teraz musimy lekko odfiltrować te cechy. Podczas testów okazało się że choć zazwyczaj większość cech jest dobrze dopasowana to zdarzają się złe jako że są cechy które pasowały niemal wszędzie. Pozostawienie złych cech może mieć konsekwencje podczas wyliczania średniej pozycji puzzle. Dlatego w następnym kroku sprawdzamy wynik podobieństwa dla dopasowanych cech i odrzucamy te których wynik jest za mały według domyślnego progu - ustalonego przez użytkownika.

Po odrzuceniu mało dokładnych cech algorytm zapisuje pozycję punktu kluczowego. Są to pozycje dopasowanych cech puzzli do ułożonego obrazka. Te ułożone pozycje są sumowane , a następnie liczona jest z nich średnia. Dzięki temu otrzymaliśmy średnie pozycje puzzli względem ułożonego zdjęcia. Wyniki są zapisywane do tablic oddzielnie dla współrzędnych X oraz dla współrzędnych Y.

Algorytm układania puzzli.

Opisywany algorytm układa puzzle dzięki wcześniej obliczonym średnim pozycją.(wspomniane przy algorytmie Wykrywanie punktów wspólnych i obliczenie dla nich średniej pozycji) dodatkowo algorytm musi wiedzieć jak ułożyć puzzle, to znaczy ,że musi znać konfigurację w jakiej je ułożyć. Na przykład mając 16 puzzli chcemy by puzzle były ułożone w konfiguracji 4x4. Został napisany algorytm który wstępnie to przewiduje (Algorytm wykrywanie konfiguracji). Zanim przejdę do działania algorytmu wspomnę ,że rozważając jak to zrobić w końcu wpadłem na pomysł by przedstawić ułożenie puzzli ze znanymi nam na tablicy dwuwymiarowej:

Puzzle

1. X100 Y80	2. X400 Y900	3. X200 Y 100
4. X150 Y200	5. X240 Y600	6. X500 Y800

	100	150	200	240	400	500
80	1					
100			3			
200		4				
600				5		
800						6
900					2	

1 3 6
4 5 2

Rys. 12 - Notatki Adama Godzińskiego na temat układania puzzli. Obraz przedstawia ponumerowane prostokąty odpowiadające puzzlą wraz z ich współrzędnymi.

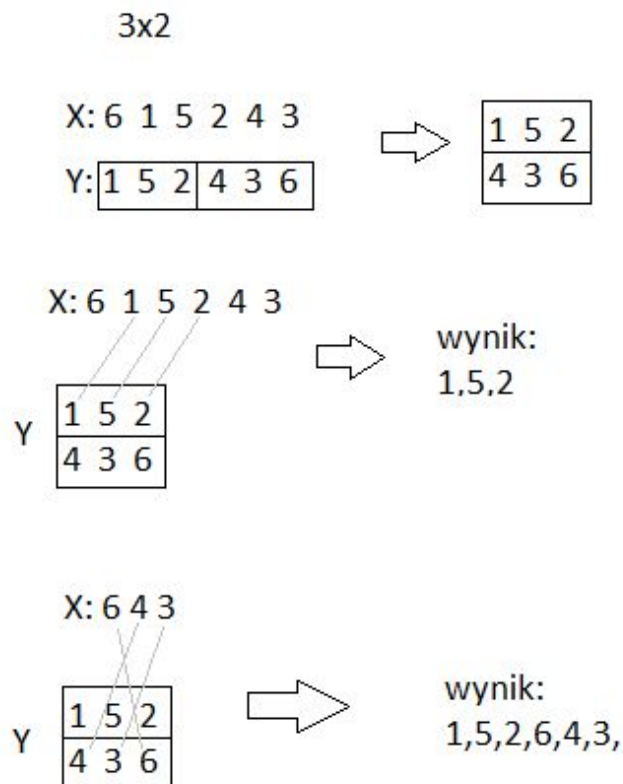
Źródło: Obraz własny

Obraz domyślnie ustawiony w konfiguracji 3x2. Po wrzuceniu wszystkich pozycji do tablicy, zauważyłem że dość naturalnie można stwierdzić jak powinny zostać ułożone. Obraz ten w przybliżeniu pokazuje w jaki sposób działa algorytm układający do którego omawiania zaraz przejdę.

Najpierw chcemy dostać tablicę w której zamiast współrzędnych puzzli będą odpowiadające im numery identyfikacyjne. Na samym początku algorytm tworzy kopię tablic średnich punktów ułożenia puzzli.

Następnie kopie są sortowane od najmniejszej do największej. Dzięki temu dostajemy tablice w której współrzędne są ułożone od lewej do prawej dla X i od góry do dołu dla Y. Następnie posiadając uporządkowaną tablicę oraz oryginalną tablicę tworzymy kolejną na której będę wiedział jak rozkładają się średnie pozycje po identyfikatorze nadanym puzzlom. Inaczej mówiąc zapisuje które puzzle mają kolejne pozycje (przykład dla $X=[10,40,80]$ dostaje $X_p=[1,3,2]$). Teraz kiedy jestem w stanie zidentyfikować puzzle zabieram się do układania.

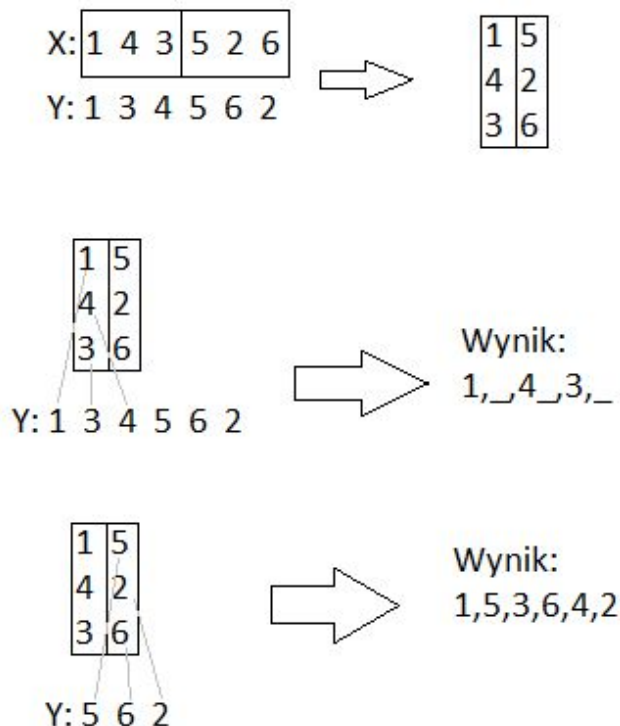
Puzzle są układane w następującym porządku od lewej do prawej a potem od góry do dołu i tak też jest zapisywany wynik. Oznaczmy liczbę puzzli w poziomie jako "W" a w pionie jako "H". Algorytm inaczej się zachowuje w innej konfiguracji puzzli. Kiedy W jest większe od H (na przykład 3x2) a inaczej kiedy $W < H$ (na przykład 2x5).



Rys. 13 - Notatki Adama Godzińskiego na temat układania puzzli. Rysunek przedstawia zasadę działania algorytmu układania puzzli, gdy $W > H$.

Źródło: Obraz własny

Gdy $W > H$, algorytm wybiera z tablicy indeksów dla współrzędnej Y pierwsze W indeksów aby podzielić je na H części. Dzięki temu wiemy jak rozłożyć się puzzle w poziomie. Teraz musimy się dowiedzieć jak je ułożyć względem X. Aby to zrobić przeszukuje od początku tablicę X względem podzielonej wcześniej tablicy Y. Szukam tych samych indeksów, a znaleziony zapisuje jako wynik. Sytuacja w której $W < H$ przedstawia się następująco:



Rys. 14 - Notatki Adama Godzińskiego na temat układania puzzli. Rysunek przedstawia zasadę działania algorytmu układania puzzli, gdy $W < H$.

Źródło: Obraz własny

Gdy $W < H$, algorytm bierze pierwsze H elementów tablicy X by podzielić je na W części. Następnie iterujemy przez tablicę T oraz kolejne segmenty podzielonej tablicy X i w momencie znalezienia tych samych indeksów wynik jest zapisywany. W tym przypadku jednak nie będziemy w stanie ładnie dopisywać do wyniku kolejnych znalezionych indeksów ponieważ wtedy dostalibyśmy układ ułożenia puzzli od góry do dołu i potem od prawej do lewej. Z powodu że zapisuje wyniki w tablicy jestem w stanie tak ją zamienić na taką która zwróci mi dobrą reprezentację wyniku.

Została jeszcze opcja gdy $W=H$, wtedy można użyć dowolnego ze sposobów , w przypadku tego algorytmu używany jest $W>H$,jako że nie wymaga przekształcania tablicy wyników.

Algorytm generowania obrazu

Algorytm służy do wygenerowania wyniku. Mając tablicę dzięki której wiemy w jakiej kolejności ułożyć puzzle oraz listę z obrazami przedstawiającymi pojedyncze puzzle jesteśmy w stanie skleić ostateczny obraz. Na początku sumuje długości i szerokości wszystkich obrazków puzzli. Następnie oblicza z nich średnią dzięki której wiemy jakie będą na jakie wymiary przeskalować wszystkie puzzle oraz oczywiście jaka będzie wielkość ostatecznego obrazu. Potem algorytm zaczyna iterować przez obrazy znajdując te które powinny zostać kolejno ustawione. Następnie przeskalowuje obrazy puzzli na takie ze średnimi wymiarami oraz wrzuca je do końcowego obrazu.



Rys. 15 - Obraz wynikowy.

Źródło: Obraz własny

Chociaż na początku może się wydawać że skalowanie według średniej może mocno zdeformować puzzle to jak na razie podczas testów ani razu nam się to nie przytrafiło. Z kolei widać, że między fragmentami są przerwy. Jako że obrazy puzzle są prostokątami nie da się uniknąć pozostawienia jakiś kawałków tła, przynajmniej bez większej obróbki wszystkich puzzli. Jednak jako że naszym celem jest raczej podpowiedzenie jak ułożyć puzzle uznaliśmy że taki wynik jest zupełnie wystarczający.

6. Interesujące problemy i błędy oraz jak z nimi się uporaliśmy

6.1. Jak obrobić zdjęcie tak aby lepiej wykrywa krawędzie?

Po tym jak rozpoczęliśmy pracę nad wykrywaniem puzzli natknęliśmy się na problem że używana przez nas funkcja wykrywająca nie znajduje żadnych puzzli obojętnie jak byśmy się starali. Doszliśmy szybko do wniosku że musimy jakoś obrobić zdjęcie aby niemal zawsze znajdowało nam te puzzle. Na początku użyliśmy progowania dylatacji i erozji. Przyniosło to poprawę, ale nie dość zadowalającą, ponieważ przy różnych obrazach musieliśmy dobierać inne wartości. Zajrzeliśmy do pracy naszych poprzedników i zauważyliśmy że używają oni progowania adaptacyjnego. Również użyliśmy tej funkcjonalności, wyniki tym razem uznaliśmy za zadowalające. Jakiś czas później po rozmowie z naszymi kolegami uznaliśmy że możemy jeszcze użyć rozmycia gaussowskiego. Po użyciu zaczęliśmy jeszcze trochę kombinować metodą prób i błędów. Koniec końców zmniejszyliśmy wartość dylatacji jak również erozji. Od tego momentu nie mieliśmy problemu poza jednym ciekawym aczkolwiek oczywistym przypadkiem, mianowicie niemal białym puzzlem na białym tle.

6.2. Problemy z dopasowywaniem puzzli do obrazu

Problem o którym tutaj wspomnę pojawił się jeszcze kilka razy, choć stosunkowo jego rozwiązanie sprowadza się do jednego a mianowicie spojrzenia do innej dokumentacji niż tej dla C# i Emgu CV , najczęściej tej Pythonowej. Jednakże to ten przypadek zajął nam najwięcej czasu na zrozumienie jak to w zasadzie powinno być zapisane. Na stronie emgu jest przykład dla implementacji rozpoznawania cech oraz dopasowywania cech wspólnych, problem jest jednak taki że tam wszystko dzieje się na matkach a u nas na obrazach (Image) . Oczywiście na początku próbowaliśmy jakoś przekopiować obrazy na maty i użyć tego kodu ze strony. Niestety kod nie działał po kilku próbach zaczęliśmy szukać po internecie oraz w dokumentacji dla Pythona. Próbowaliśmy paru podejść , jednak ta końcowa która w końcu zadziała była swego rodzaju średnią testowanych metod. Ostatecznie częściowo używamy maty ale nie wpływa to na złożoność kodu . Przy okazji przetestowaliśmy inne algorytmy wykrywania cech takie jak FAST oraz ORB które okazały się gorsze od używanego przez nas SURFa a także zauważyliśmy że dopasowane cechy (matches) mają taki atrybut jak dystans dzięki któremu mogliśmy później odfiltrować słabo dopasowane cechy.

7. Instrukcja Użytkownika

- 7.1. Uruchomić program dwukrotnym szybkim naciśnięciem lewego przycisku myszy na jego ikonie lub zaznaczyć ikonę i wcisnąć [Enter].



Rys. 16 - Grafika odzwierciedlająca sposób uruchomienia programu.

Źródło: Obraz własny.

Wtedy pojawi się okno główne programu:

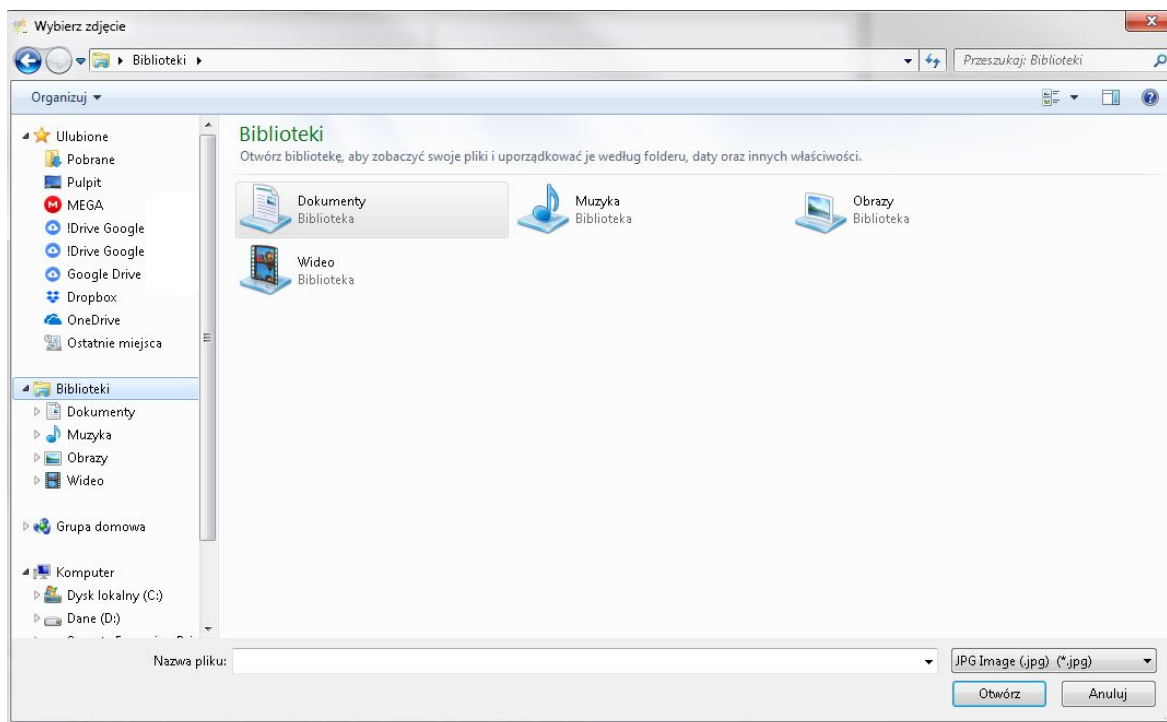


Rys. 17 - Okno główne programu.

Źródło: Obraz własny.

7.2. Nacisnąć na element oznaczony numerem 1. Dzięki niemu będziesz w stanie wybrać obraz na którym są puzzle do ułożenia.

7.3. Wybrać z dysku proszony obraz.



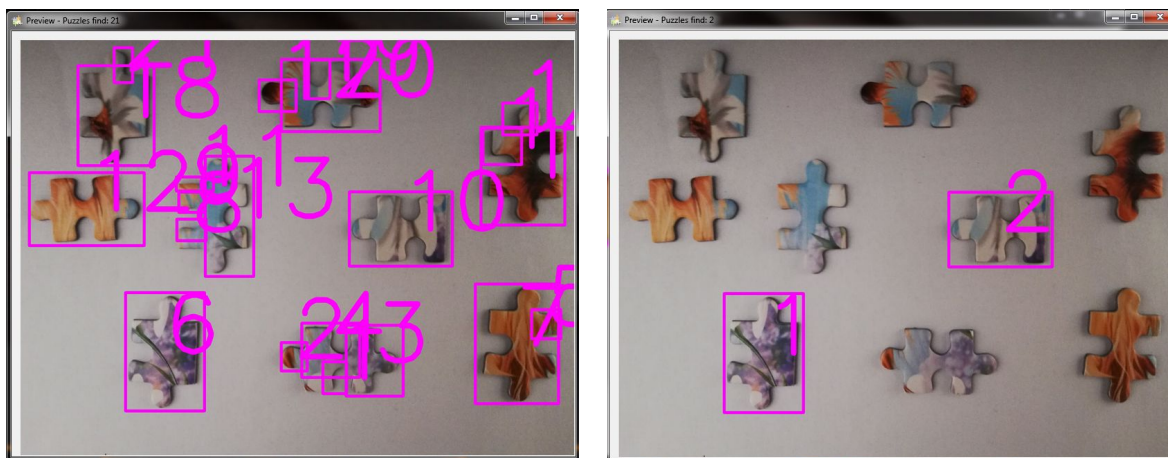
Rys. 18 - Okno programu do wybraniu pliku typu - obraz.

Źródło: Obraz własny.

7.4. Nacisnąć na element oznaczony numerem 2. Pozwala on na podgląd zaznaczonych puzzli.

7.5. Zweryfikować poprawność podglądu puzzli.

W zależności od poprawności rozpoznania puzzli, odpowiednio zmienić wartość "*Wielkość konturu*" oraz inne pola zaznaczone zielonym kolorem nad przyciskiem z numerem czwartym.



Rys. 19 - Po lewej i prawej stronie skrajne przypadki źle wygenerowanego podglądu znalezionych puzzli.

Źródło: Obraz własny.

Źle dobrane parametry powodują nie wychwycenie wszystkich puzzli, obcinanie ich do mniejszego rozmiaru, a nawet traktowanie jednego puzzla jako większej ich ilości. W przypadku zbyt dużej wartości parametrów, ilość znajdowanych puzzli gwałtownie się zmniejsza. Poniżej przykład dla optymalnej wartości parametrów. Ze względu na inne czynniki (jakość wykonania zdjęcia), algorytmowi nie udało się dokładnie określić wszystkich puzzli.

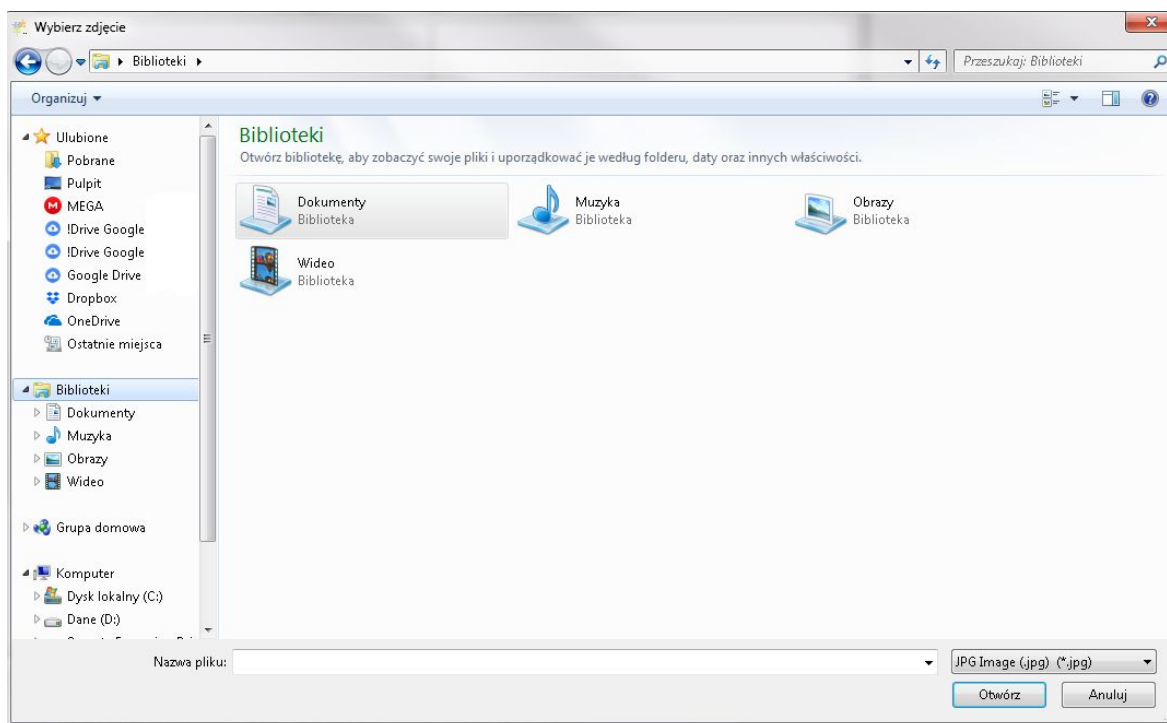


Rys. 20 - Przykład podglądu znalezionych puzzli. UWAGA: obrazek może się drastycznie różnić w porównaniu z poglądem wygenerowanym przez użytkownika.

Źródło: Obraz własny.

Jeżeli wszystko zostało poprawnie rozpoznane, to przejść do punktu 7.6. W przeciwnym razie, przejść do punktu 7.4.

- 7.6. Nacisnąć na element oznaczony numerem 3. Do ułożenia puzzli jest wymagane zdjęcie z ułożoną układanką, w tym miejscu możesz je dodać do programu
- 7.7. Wybrać z dysku proszony obraz.



Rys. 21 -Okno programu do wybraniu pliku typu - obraz.

Źródło: Obraz własny.

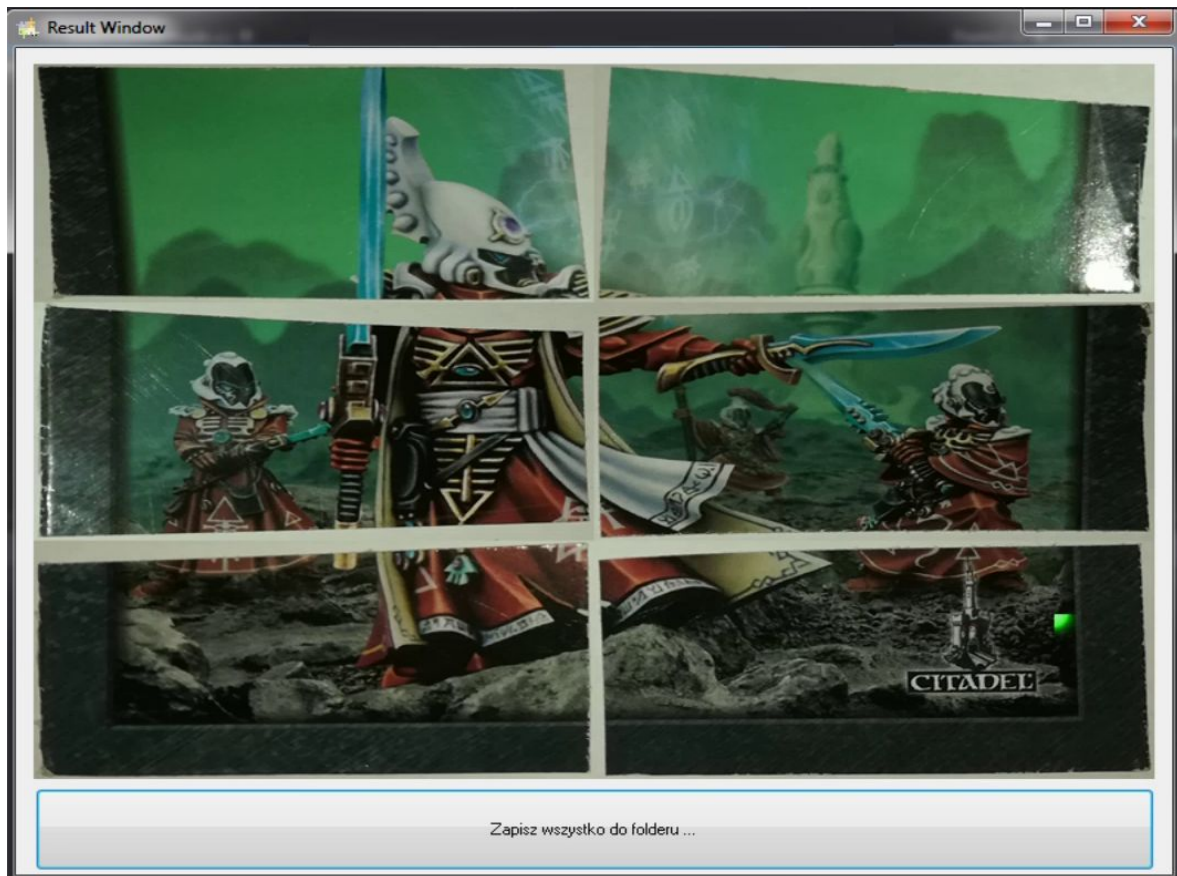
- 7.8. Nacisnąć na element oznaczony numerem 4. Rozpocznie on proces przetwarzania wybranych zdjęć.
- 7.9. Czekać, aż program wykona swoją pracę. Kiedy skończy pasek stanu powinien być pełny.



Rys. 22 -Okno programu informujące o postępach procesu przetwarzania zdjęć. W tym wypadku proces się zakończył.

Źródło: Obraz własny.

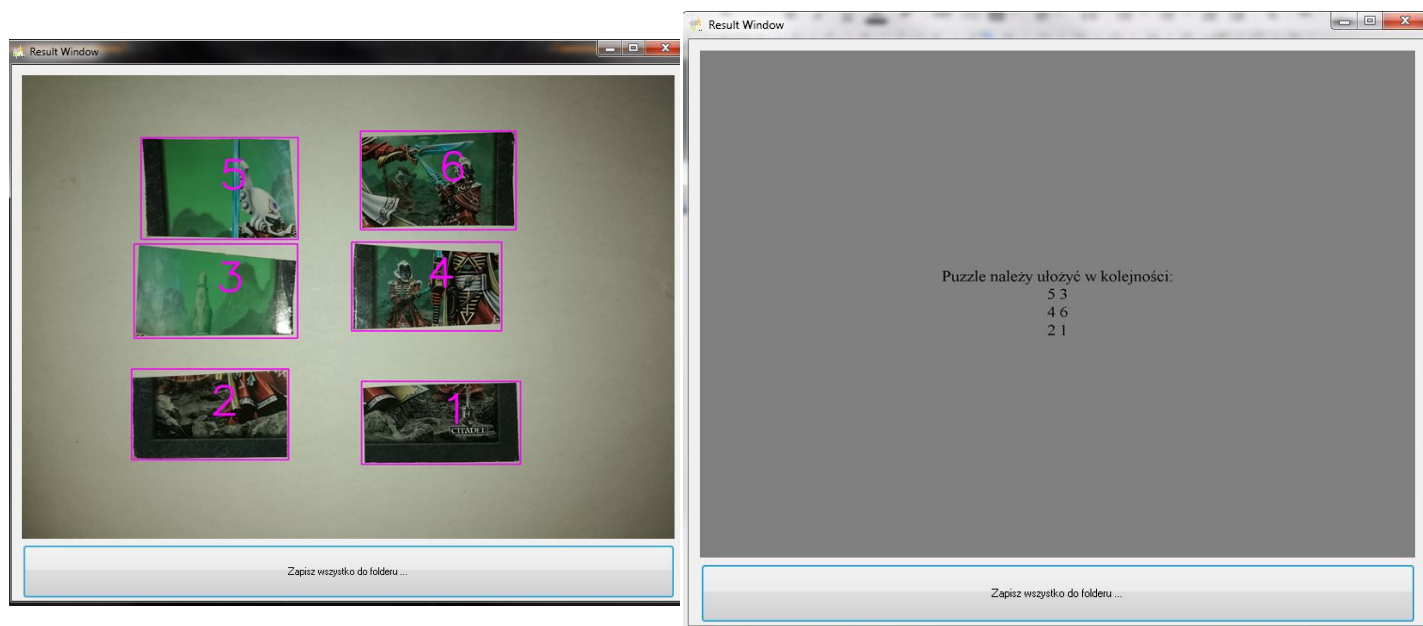
Wtedy program otworzy okno wynikowe:



Rys. 23 - Wygenerowany obraz ułożonych puzzli przez program.

Źródło: Obraz własny.

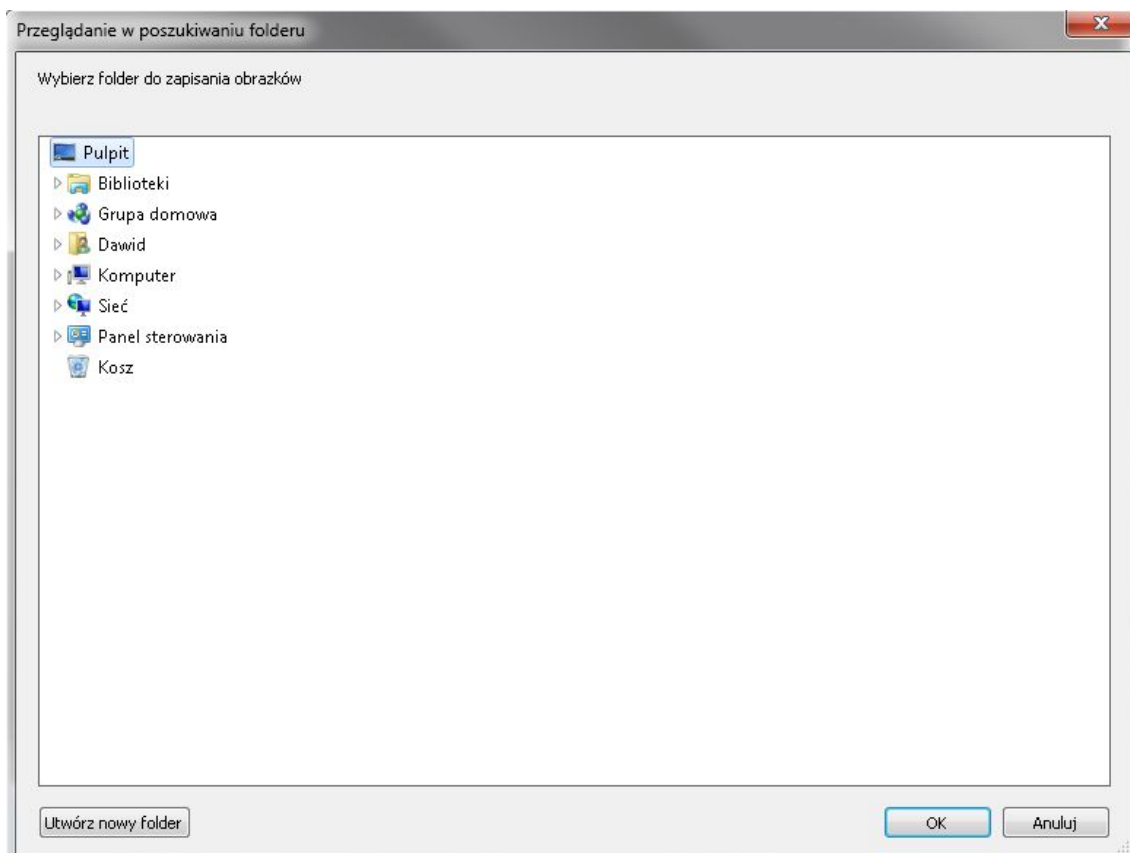
- 7.10. Klikając w jakiekolwiek miejsce w środku wyświetlanego obrazka przejdziemy do następnych obrazków.



Rys. 24 - Po lewej stronie obraz z zaznaczonymi i ponumerowanymi puzzlami.
Po prawej instrukcja dla użytkownika, jak ułożyć puzzle zgodnie z oznaczeniami z obrazka po lewej stronie.

Źródło: Obraz własny.

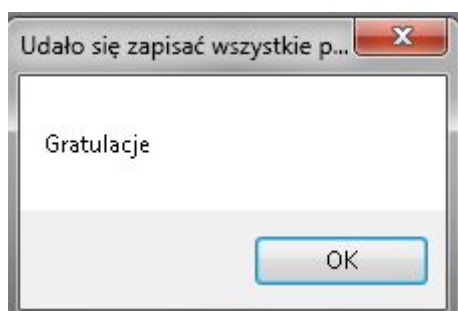
- 7.11. Klikając przycisk *“Zapisz wszystko do folderu”*, pojawi się okno wyboru folderu zapisu.



Rys. 25 - Okno wyboru folderu do zapisu zdjęć.

Źródło: Obraz własny.

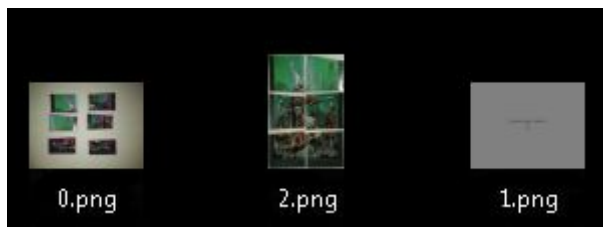
Po wybraniu folderu i kliknięciu przycisku *“OK”*, pojawi się komunikat informujący o stanie powodzenia zapisu.



Rys. 26 - Komunikat informujący o zakończeniu zapisywania.

Źródło: Obraz własny.

- 7.12. Wszystkie pliki zapiszą się do wskazanego folderu z rozszerzeniem *“.png”*.



Rys. 27 -Obraz przedstawia zapisane pliki na pulpicie. Zalecane jest zapisywanie do konkretnego folderu.. UWAGA: pliki mogą się znacząco różnić w porównaniu z plikami wygenerowanymi przez użytkownika.

Źródło: Obraz własny.

- 7.13. Program można bezpiecznie zamknąć

8. Bibliografia

- http://www.emgu.com/wiki/index.php/Main_Page
Strona z poradnikami do emgu CV.
- <https://msdn.microsoft.com/en-us/library/>
Strona odnosząca się do technologii Microsoftowych w tym C#.
- https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_tutorials.html
Dokumentacja i poradniki od OpenCV dla Pythona.
- <https://romovs.github.io/>
Blog z ciekawymi kodami do EmguCV dotyczącymi dopasowywania cech.
- <https://www.codeproject.com/>
Strona z opisami implementacji dla różnych języków programowania w tym dla C# i emguCV.