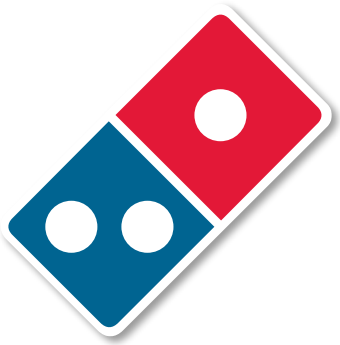
PRZETWARZANIE CYFROWE OBRAZÓW

DOKUMENTACJA PROJEKTU, 24.01.2023 ARKADIUSZ DAWID, 300199

1. TEMAT PROJEKTU

Dla indywidualnie wybranej klasy obrazów dobrać, zaimplementować i przetestować odpowiednie procedury wstępnego przetwarzania, segmentacji, wyznaczania cech oraz identyfikacji obrazów cyfrowych. Powstały w wyniku projektu program powinien poprawnie rozpoznawać wybrane obiekty dla reprezentatywnego zestawu obrazów wejściowych.

Jako docelowy obiekt do wykrywania wybrałem logo Domino's Pizza™, które prezentuje rysunek 1.



Rys. 1: Logo Domino's

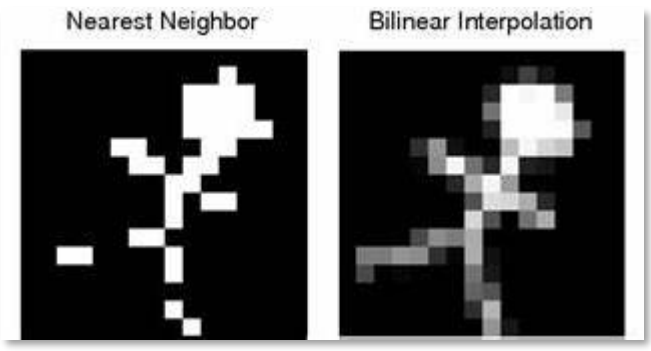
2. POTOK PRZETWARZANIA

Działanie programu składa się z pięciu głównych kroków. Są to: przetwarzanie wstępne, konwersja przestrzeni barw, segmentacja, wyznaczenie cech i detekcja logo. Kroki te przedstawia rysunek poniżej.

Rys. 2: Potok przetwarzania

3. PRZETWARZANIE WSTĘPNE

Pierwszym krokiem przetwarzania wstępnego jest przeskalowanie obrazka do wymiarów 800px na 600px. W tym celu zaimplementowałem dwa algorytmy – interpolację *Nearest-Neighbor* oraz interpolację dwuliniową. Porównanie efektów działania algorytmów przedstawia rysunek 3. Ze względu na lepsze efekty algorytmu interpolacji bilinearnej, to właśnie jego ostatecznie używam w przetwarzaniu. Rysunek 4 przedstawia obrazek po przeskalowaniu.



Rys. 3: Porównanie algorytmów skalowania

Rys. 4: Obrazek po przeskalowaniu z użyciem interpolacji dwuliniowej

Kolejnym krokiem jest zastosowanie filtru dolnoprzepustowego w celu usunięcia szumu z obrazka. Zastosowałem filtr o rozmiarze okna analizy równym 5. Kod realizujący ten filtr przedstawia rysunek 5, a efekt działania przedstawia rysunek 6.

```
4   void filterLow(cv::Mat& mat) {
5       const int KERNEL_SIZE = 5;
6       const int KERNEL_SIZE_SQUARED = KERNEL_SIZE * KERNEL_SIZE;
7       cv::Mat filtered = mat.clone();
8
9       for (int i = 0; i < RESIZED_WIDTH; i++) {
10          for (int j = 0; j < RESIZED_HEIGHT; j++) {
11              int x = i;
12              int y = j;
13
14              if (x < KERNEL_SIZE / 2 || y < KERNEL_SIZE / 2 ||
15                  x >= RESIZED_WIDTH - KERNEL_SIZE / 2 || y >= RESIZED_HEIGHT - KERNEL_SIZE / 2) {
16                  continue;
17              }
18
19              for (int k = 0; k < 3; k++) {
20                  double temp = 0;
21                  for (int l = 0; l < KERNEL_SIZE; l++) {
22                      for (int m = 0; m < KERNEL_SIZE; m++) {
23                          temp += mat.at<cv::Vec3b>(y - KERNEL_SIZE / 2 + m, x - KERNEL_SIZE / 2 + l)[k];
24                      }
25                  }
26                  filtered.at<cv::Vec3b>(j, i)[k] = (uchar)(temp / KERNEL_SIZE_SQUARED);
27              }
28          }
29      }
30      mat = filtered;
31  }
```

Rys. 5: Funkcja realizująca filtr dolnoprzepustowy

Rys. 6: Obrazek po filtracji dolnoprzepustowej

Niestety wraz z usunięciem szumu, filtr dolnoprzepustowy rozmazuje krawędzie na obrazie. Aby temu zaradzić, w kolejnym kroku wykorzystuję filtr górnoprzepustowy, tzn. filtr wyostrzający. Kod realizujący ten filtr przedstawia rysunek 7, a efekt działania przedstawia rysunek 8.

```
33  void filterHigh(cv::Mat& mat) {
34      const int KERNEL_SIZE = 3;
35
36      cv::Mat kernel = (cv::Mat_<double>(KERNEL_SIZE, KERNEL_SIZE) << -1, -1, -1,
37          -1, 9, -1,
38          -1, -1, -1);
39      cv::Mat filtered = mat.clone();
40
41      for (int i = 0; i < RESIZED_WIDTH; i++) {
42          for (int j = 0; j < RESIZED_HEIGHT; j++) {
43              int x = i;
44              int y = j;
45
46              if (x < KERNEL_SIZE / 2 || y < KERNEL_SIZE / 2 ||
47                  x >= RESIZED_WIDTH - KERNEL_SIZE / 2 || y >= RESIZED_HEIGHT - KERNEL_SIZE / 2) {
48                  continue;
49              }
50
51              for (int k = 0; k < 3; k++) {
52                  double temp = 0;
53                  for (int l = 0; l < KERNEL_SIZE; l++) {
54                      for (int m = 0; m < KERNEL_SIZE; m++) {
55                          temp += kernel.at<double>(m, l) *
56                              mat.at<cv::Vec3b>(y - KERNEL_SIZE / 2 + m, x - KERNEL_SIZE / 2 + l)[k];
57                      }
58                  }
59                  filtered.at<cv::Vec3b>(j, i)[k] = (uchar)(std::max(std::min((int)temp, 255), 0));
60              }
61          }
62      }
63      mat = filtered;
64  }
```

Rys. 7: Funkcja realizująca filtr górnoprzepustowy

Rys. 8: Obrazek po filtracji górnoprzepustowej

Zaimplementowałem również funkcję, która redukuje liczbę kolorów do 16 na każdą składową (16*16*16 możliwych kolorów) oraz funkcję realizującą wyrównanie histogramu. W czasie prac okazało się jednak, że redukcja kolorów jest zbędna dla poprawnego działania programu, a wyrównanie histogramu w niektórych przypadkach powodowało zbytnie zmiany w kolorach obrazka co uniemożliwiało poprawne segmentowanie w następnym kroku. Rysunek 9 przedstawia działanie algorytmu wyrównującego histogram.

Rys. 9: Obrazek po wyrównaniu histogramu

4. KONWERSJA RGB NA HSL

Na tym etapie obrazek został już przeskalowany oraz poddany filtracji dolno- i górnoprzepustowej. W ramach przygotowań do segmentacji należy przekonwertować wartości RGB poszczególnych pikseli na HSL. Działanie to wykonuje funkcja przedstawiona na rysunku 10.

```
128  static hsl RGBToHSL(rgb rgb) {
129      hsl hsl;
130      hsl.h = 0;
131      hsl.s = 0;
132      hsl.l = 0;
133
134      float r = (rgb.r / 255.0f);
135      float g = (rgb.g / 255.0f);
136      float b = (rgb.b / 255.0f);
137
138      float min = std::min(std::min(r, g), b);
139      float max = std::max(std::max(r, g), b);
140      float delta = max - min;
141
142      hsl.l = (max + min) / 2;
143
144      if (delta == 0) {
145          hsl.h = 0;
146          hsl.s = 0.0f;
147
148      }
149      else {
150          hsl.s = (hsl.l <= 0.5) ? (delta / (max + min)) : (delta / (2 - max - min));
151          float hue;
152
153          if (r == max) { hue = ((g - b) / 6) / delta; }
154          else if (g == max) { hue = (1.0f / 3) + ((b - r) / 6) / delta; }
155          else { hue = (2.0f / 3) + ((r - g) / 6) / delta; }
156
157          if (hue < 0) { hue += 1; }
158          if (hue > 1) { hue -= 1; }
159
160          hsl.h = (int)(hue * 360);
161      }
162      return hsl;
163  }
```

Rys. 10: Funkcja konwertująca wartości RGB na HSL

5. SEGMENTACJA

Najważniejszym krokiem całego procesu wykrywania logo jest segmentacja. Celem procesu segmentacji jest automatyczne wydzielenie obszarów obrazu, których wygląd jest dla obserwatora spójny. Piksele spójne, to takie które spełniają podane kryterium jednorodności. Typowe kryteria jednorodności to podobny poziom jasności, podobny poziom barwy lub ta sama tekstura uzyskana w wyniku analizy częstotliwościowej. Zasadniczo, proces segmentacji można podzielić na dwie fazy:

- Wydzielanie - określenie czy dany piksel należy do interesującego obiektu czy tez do tła,
- Oznaczanie - łączenie i przyporządkowywanie interesujących pikseli do podzbiorów pikseli, będących potencjalnie obrazami szukanych obiektów.

W programie wykorzystałem algorytm hybrydowy łączący ze sobą algorytm segmentacji przez progowanie oraz metodą rozrostu obszarów. Działa to następująco:

5.1. MACIERZ *states*

Tworzę macierz *states*, która na początku wypełniona jest wartościami *UNVISITED* dla każdego piksela. Wszystkie możliwe wartości to ADDED, VISITED, UNVISITED oraz MISSED. Macierz ta zostanie później wykorzystana do wydzielania segmentów.

5.2. MACIERZ *matOfColors*

Tworzę macierz *matOfColors*, która zawiera tylko białe, niebieskie, czerwone i czarne piksele. Powstaje ona w taki sposób: przechodzimy piksel po pikselu obrazka i zgodnie z regułami opisanymi poniżej przypisujemy im nowy kolor.

- `WHITE: hsl.l >= 0.79 || (hsl.l > 0.72 && hsl.s < 0.35)`
- `RED: (hsl.h < 15 || hsl.h > 315) && hsl.s > 0.43 && hsl.l > 0.32 && hsl.l < 0.79`
- `BLUE: hsl.h < 245 && hsl.h > 180 && hsl.s > 0.35 && hsl.l < 0.79 && hsl.l > 0.22`
- `BLACK: w przeciwnym wypadku`

5.3. LISTA SEGMENTÓW

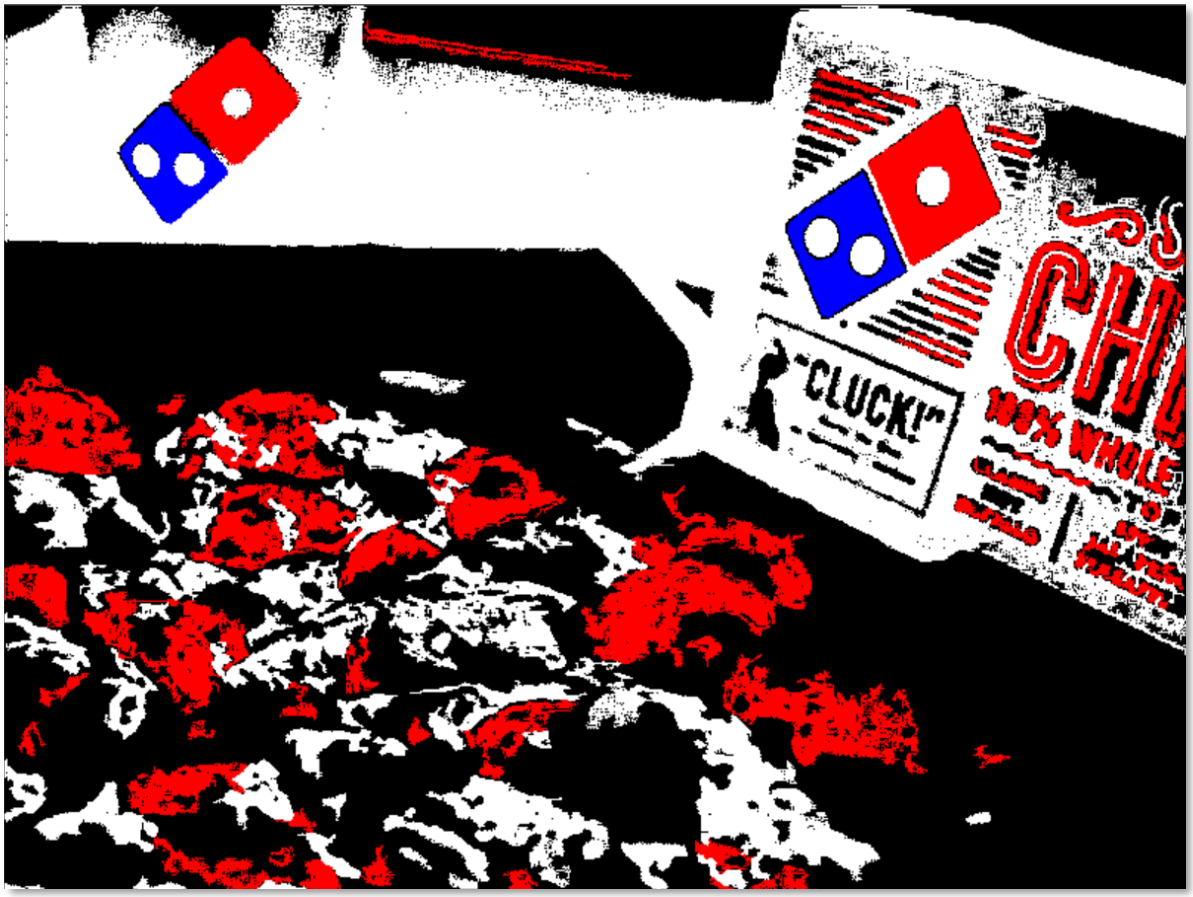
Tworzę listę segmentów typu `vector<vector<pair<int, int>>>`. Każdy segment składa się z listy par, gdzie każda para wskazuje współrzędne piksela należącego do danego segmentu. Rysunek 11 przedstawia funkcję wyznaczającą segmenty metodą rozrostu obszarów.

```
31
32  SegmentVector SegmentationProcessor::segmentation(ColorMat& colorMat, StateMat& stateMat) {
33      SegmentVector segments;
34      int matHeight = colorMat[0].size();
35      for (int i = 1; i < matHeight - 1; i++) {
36          for (int j = 1; j < colorMat.size() - 1; j++) {
37              Color color = colorMat[j][i];
38              if (color == OTHER) {
39                  stateMat[j][i] = MISSED;
40                  continue;
41              }
42              State state = stateMat[j][i];
43              if (state != UNVISITED) { continue; }
44              Color seed = color;
45              std::vector<std::pair<int, int>> segment;
46              std::stack<std::pair<int, int>> pointsStack;
47              pointsStack.push(std::make_pair(j, i));
48              while (!pointsStack.empty()) {
49                  std::pair<int, int> point = pointsStack.top();
50                  State pointState = stateMat[point.first][point.second];
51                  Color pointColor = colorMat[point.first][point.second];
52                  if (pointColor != seed || pointState == ADDED) {
53                      pointsStack.pop();
54                      continue;
55                  }
56                  stateMat[point.first][point.second] = ADDED;
57                  segment.push_back(point);
58                  for (int x = 0; x < 3; x++) {
59                      for (int y = 0; y < 3; y++) {
60                          if (x == 1 && y == 1) { continue; }
61                          int neighbourX = point.second + x - 1;
62                          int neighbourY = point.first + y - 1;
63                          if (neighbourX < 0 || neighbourY < 0 || neighbourX >= matHeight || neighbourY >= colorMat.size()) { continue; }
64                          pointsStack.push(std::make_pair(neighbourY, neighbourX));
65                      }
66                  }
67                  pointsStack.pop();
68              }
69              segments.push_back(segment);
70          }
71      }
72      return segments;
73  }
```

Rys. 11: Funkcja segmentująca

5.4. WSTĘPNA FILTRACJA SEGMENTÓW I WYŚWIETLENIE WYNIKU

Wstępnie filtruję wyznaczone segmenty – odrzucam te, które składają się z liczby pikseli nie przekraczającej liczby SEGMENT_SIZE_THRESHOLD = 100. Wynik tak przefiltrowanej segmentacji przedstawia rysunek 12.



Rys. 12: Wynik segmentacji

5.5. STWORZENIE DESKRYPTORÓW SEGMENTÓW

Na koniec tworzę listę deskryptorów, gdzie na każdy deskryptor składa się:

- Segment (lista pikseli – par współrzędnych)
- Kolor (WHITE, BLUE lub RED)
- BoundingBox (współrzędne rogów, wysokość, szerokość)
- Pole powierzchni (liczba pikseli)
- Współrzędne środka ciężkości

6. WYZNACZENIE CECH I FILTRACJA DESKRYPTORÓW

Ten etap przetwarzania polega na wyznaczeniu cech każdego segmentu i odrzuceniu tych, które nie spełniają danych kryteriów. Jako cechy warte uwagi uznałem momenty geometryczne od M1 do M7 oraz stosunek szerokości do wysokości bounding boxa opisującego dany segment. Wartości modelowe obliczyłem dla niebieskiej i czerwonej części logo oraz białego koła. Przedstawia je tabela 1.

Cecha	BLUE	RED	WHITE
M1	0.229965	0.201113	0.159155
M2	0.000363102	4.72441e-07	1.0888e-08
M3	1.05411e-06	8.05831e-07	1.153e-08
M4	5.80308e-08	5.85783e-08	1.04152e-15
M5	9.84522e-15	1.15908e-14	3.43534e-27
M6	1.10566e-09	4.02584e-11	1.03342e-19
M7	0.0131302	0.0101115	0.00633255
W/H	1.0	1.0	1.0

Tabela 1: Wartości modelowe cech

Na tym etapie przetwarzania wynikiem jest lista deskryptorów segmentów, które spełniają dane cechy. Deskryptory są pogrupowane według odpowiadającego im koloru. Rysunek 13 przedstawia obrazek wejściowy z oznaczonymi bounding boxami segmentów, które spełniły kryteria.

Rys. 13: Segmenty spełniające kryteria

7. DETEKCJA LOGO

Ostatnim krokiem programu jest odpowiednie powiązanie ze sobą segmentów aby móc zdecydować, czy stanowią one części logo. Algorytm jest stosunkowo prosty i wygląda następująco:

1. Znajdź pary czerwono-niebieskie:
    a. Dla każdego czerwonego segmentu znajdź jeden segment niebieski.
    b. Oblicz stosunek szerokości niebieskiego do odległości środków ciężkości czerwonego i niebieskiego.
    c. Jeśli stosunek ten jest bliski wartości BLUE_WIDTH_TO_BLUE_RED_DISTANCE_MODEL = 1.2966841, dodaj punkty segmentu niebieskiego do czerwonego i stwórz nowy segment *redBluePart*.
    d. Dodaj segment *redBluePart* do tablicy *redBlueObjects*.
2. Znajdź białe koła:
    a. Dla każdego segmentu czerwono-niebieskiego znajdź wszystkie segmenty białe, które zawierają się wewnątrz niego.
    b. Jeśli znaleziono dokładnie trzy takie segmenty, dodaj segment czerwono-niebieski do tablicy *redBlueObjectsWithThreeCircles*.

Po wszystkich tych operacjach mamy listę segmentów, które powinny pokrywać się ze znalezionymi logo na obrazku. Zostaje tylko pokolorować ich ramki i wyświetlić końcowy rezultat – rysunek 14.

Rys. 14: Końcowy rezultat – znalezione logo w zielonych ramkach