POLITECHNIKA POZNAŃSKA
WYDZIAŁ ELEKTRYCZNY

Szymon Miękus

Rozpoznawanie obrazu z gry w warcaby oraz wizualizacja stanu gry na komputerze

1. Wprowadzenie

1.1 Wstęp

Celem naszego projektu było stworzenie aplikacji która pozwalała na rozpoznawanie obrazu oraz wizualizację stanu gry na komputerze. Zespół w którym pracowałem składał się z trzech osób. Chcąc zrealizować projekt szybko i sprawnie musieliśmy odpowiednio rozdzielić poszczególne zadania.
Podczas pierwszego spotkania wyłoniliśmy w projekcie trzy zasadnicze części:

- przechwytywanie i rozpoznawanie obrazu z kamery
- logika gry określająca czy wykonano prawidłowy ruch
- aplikacja WPF łącząca wszystkie te elementy w jedną całość

Moim zadaniem które otrzymałem było właśnie przechwytywanie i rozpoznawanie obrazu z kamery. Zdecydowałem się na jego realizację ponieważ było to dla mnie coś nowego. Zawsze chciałem tego spróbować jednak brak czasu nie pozwalał mi na to.

1.2 Założenia

W toku realizacji musieliśmy również przyjąć odpowiednie założenia. Plansza na której odbywa się rozgrywka ma wymiary 8 x 8 czyli składa się z 64 pól. Pola pomalowane są na dwa kolory czarny oraz biały. W naszym przypadku gracze poruszają się po czarnych polach szachownicy. Gracze używają kolorowych pionków do poruszania się pomiędzy polami. Każdy z nich posiada na starcie po 12 pionów tego samego koloru. Piony mają kolisty kształt. W celu poprawnej interpretacji sytuacji występującej na szachownicy wprowadziliśmy cztery kolory pionów:

- czerwony
- zielony
- niebieski
- żółty

Gracze toczą pojedynek pionami czerwonymi oraz zielonymi, wykonując przy tym serię bić. W przypadku dotarcia którymś z pionów do przeciwległego końca szachownicy pion który się tam znalazł zostaje staje się damką. W naszym przypadku następuje wtedy zmiana koloru np. z czerwonego na zielony lub z niebieskiego na żółty. Damka w przeciwieństwie do piona może poruszać się o większą liczbę pól.
Takie o to założenia przyjęliśmy odnośnie mojej części ponieważ przechwytywanie obrazu z kamery oraz jego właściwa interpretacja jest kluczowym elementem w realizacji tego oto projektu. Jak ostatecznie okazało były one całkiem słuszne, gdyż wszystko w ten sposób działało prawidłowo.

2. Realizacja

2.1 Wykorzystane narzędzia

W celu realizacji powierzonego mi zadania skorzystałem z biblioteki *Emgu CV* oraz kamery internetowej. Biblioteka *Emgu CV* jest pewnego rodzaju nakładką na Open CV, zawiera ona szereg funkcji wykorzystywanych podczas obróbki obrazu co w naszym przypadku jest kluczową rolą. Kamera internetowa z której korzystałem w celach nie posiadała niestety jakoś specjalnie dobrych parametrów. Obraz który z niej uzyskiwałem był w rozdzielczości 640x480 oraz maksymalnie można było uzyskać do 30 klatek na sekundę. Mimo wszystko jak się później okazało była to wystarczająca ilość. Największą wadą kamery była jej wrażliwość na zmiany oświetlenia, ponieważ niespodziewanie potrafiła przełączyć się w zupełnie inny tryb pracy, co utrudniało mi zadanie.

2.2 Pierwsze kroki

Pierwszym krokiem w celu realizacji powierzonego mi zadanie było zapoznanie się z biblioteką Emgu CV. Zadanie było o tyle trudniejsze ponieważ nigdy wcześniej z niej nie korzystałem. Bibliotekę wpierw należało pobrać a następnie zainstalować. Do celów testowych utworzyłem prosty program testowy oparty o projekty w Windows Forms. Po dodaniu niezbędnych referencji, podpięciu kamery oraz uruchomieniu udało mi się uzyskać następujący efekt (rys 2.1).

Rysunek 2.1: Pierwszy obraz z kamery

Po stronie kodu wyglądało to następująco:

```
private Capture capture;
private bool captureInProgress;

public CameraCapture()
{
    InitializeComponent();
}

        private void btnStart_Click(object sender, EventArgs e)
        {
            #region if capture is not created, create it now
            if (capture == null)
            {
                try
                {
                    capture = new Capture();
                }
                catch (NullReferenceException excpt)
                {
                    MessageBox.Show(excpt.Message);
                }
            }
            #endregion

            if (capture != null)
            {
                if (captureInProgress)
                {
                    btnStart.Text = "Start!"; //
                    Application.Idle -= ProcessFrame;
                }
                else
                {

                    btnStart.Text = "Stop";
                    Application.Idle += ProcessFrame;
                }

                captureInProgress = !captureInProgress;
            }
        }

        private void ProcessFrame(object sender, EventArgs arg)
        {
            #region pierwsze testy
            Image<Bgr, Byte> ImageFrame = capture.QueryFrame().Copy();  //line 1


            CamImageBox.Image = ImageFrame;
            #endregion

         }
```

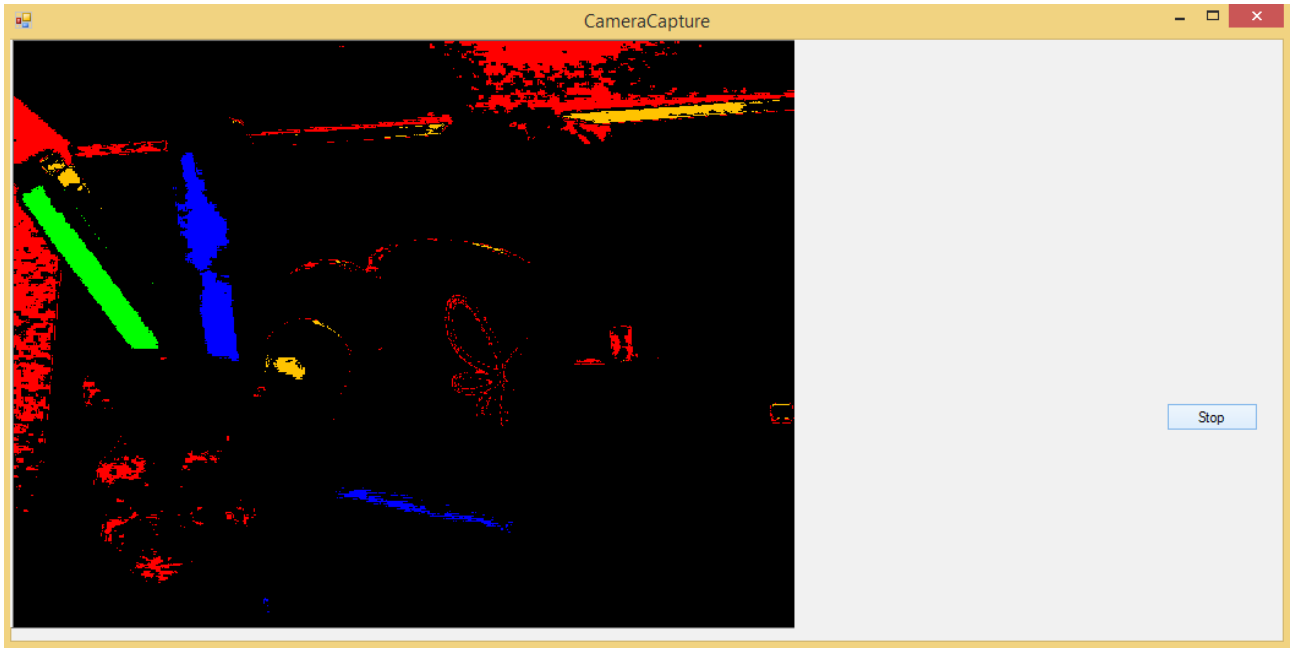
W pierwszej linii następuje deklaracja zmiennej „capture”, która w późniejszym etapie będzie odpowiedzialna za przechwytywanie obrazu prosto z kamery.
Fragment kody znajdujący się poniżej przedstawia konstruktor który po uruchomieniu uruchamia metodę „InitializeComponent”.
Kolejny fragment kodu odpowiedzialny jest za wykonanie akcji po wciśnięciu przycisku *start* który znajduje się na formie programu. W przypadku gdy obiekt *capture* jeszcze nie istniej tworzy, ustawia flagę *captureInProgress* na wartość *true* oraz uruchamia metodę *ProgressFrame*. To w niej mieści się cała logika odpowiedzialna za odczyt obrazu z kamery oraz jego wyświetlanie na formie. *ImageFrame* otrzymuje pojedynczą klatkę z kamery a następnie przekazuję ją do *CamImageBox*.

2.3 Przetwarzanie obrazu

Po tym jak udało się bez większych problemów przechwytywać obraz z kamery, kolejnym krokiem było jego przetwarzanie na bardziej interesującą nas postać. Otóż należało przetworzyć go w taki sposób aby z wszystkich widocznych kolorów otrzymać tylko sześć. Etap ten był dla nas kluczowy gdyż logika która zostanie zaimplementowana będzie korzystała z danych które będę jej zwracał. Pojedyncza klatka którą otrzymuję z kamery ma rozmiar 640x480 pixeli. Na każdy z nich składają się trzy składowe barwy, czyli czerwona, zielona i niebieska. Każda z składowych zapisana jest na ośmiu bitach co w efekcie świadczy o tym, że informacja zawarta na jednym pixelu zapisana jest na 24 bitach. W efekcie daje nam to ponad 16 mln możliwych kolorów. Nas natomiast interesuje tylko sześć czyli:

- czerwony
- zielony
- niebieski
- żółty
- biały
- czarny

Jednak największym problemem jest tutaj to iż każdy element czy to telefon, kubek, pionek do gry, pozornie jednego koloru w zależności od kąta patrzenia na niego składa się z całej masy odcieni koloru na który jest zabarwiony. Fakt ten nieco utrudnił mi zadanie. Musiałem wówczas wprowadzić zakresy kolorów. Wówczas z całej puli dostępnych kolorów wybierałem tylko pewien podzbiór odpowiadający danemu kolorowi. Na rysunku nr 2.2 widać przykład rozpoznawania kolorów na przykładzie zielonego i niebieskiego markera, żółtego nakrętki oraz kubka z czerwonym logo. Niestety tak jak wcześniej wspominałem podczas zmiany oświetlenia kamera przełączała się w inny tryb co automatycznie powodowało utrudnienia z rozpoznawaniem poszczególnych elementów.



Rysunek 2.2: Filtrowanie obrazu

```
#region wykrywanie koloru pierwszy test
            Image<Bgr, Byte> ImageFrame = capture.QueryFrame().Copy();

            Bgr pixel;

            for (int i = 0; i < ImageFrame.Height; i++)
            {
                for (int j = 0; j < ImageFrame.Width; j++)
                {

                    pixel = ImageFrame[i, j];
                    double b = pixel.Blue;
                    double g = pixel.Green;
                    double r = pixel.Red;

                    if (r > 128 && r < 255 && g > 0 && g < 110 && b > 0 && b < 97)
                    {
                        pixel.Blue = 0;
                        pixel.Green = 0;
                        pixel.Red = 255; // czerwony
                        ImageFrame[i, j] = pixel;
                    }
                    else if (r > 0 && r < 102 && g > 128 && g < 255 && b > 0 && b < 119)
                    {
                        pixel.Blue = 0;
                        pixel.Green = 255;
                        pixel.Red = 0; //zielony
                        ImageFrame[i, j] = pixel;
                    }
                    else if (r > 194 && r < 255 && g > 165 && g < 240 && b > 32 && b < 140)
                    {
                        pixel.Blue = 0;
                        pixel.Green = 191;
                        pixel.Red = 255; // złóty
                        ImageFrame[i, j] = pixel;
                    }
                    else if (r > 0 && r < 65 && g > 0 && g < 105 && b > 70 && b < 255)
                    {
                        pixel.Blue = 255;
```

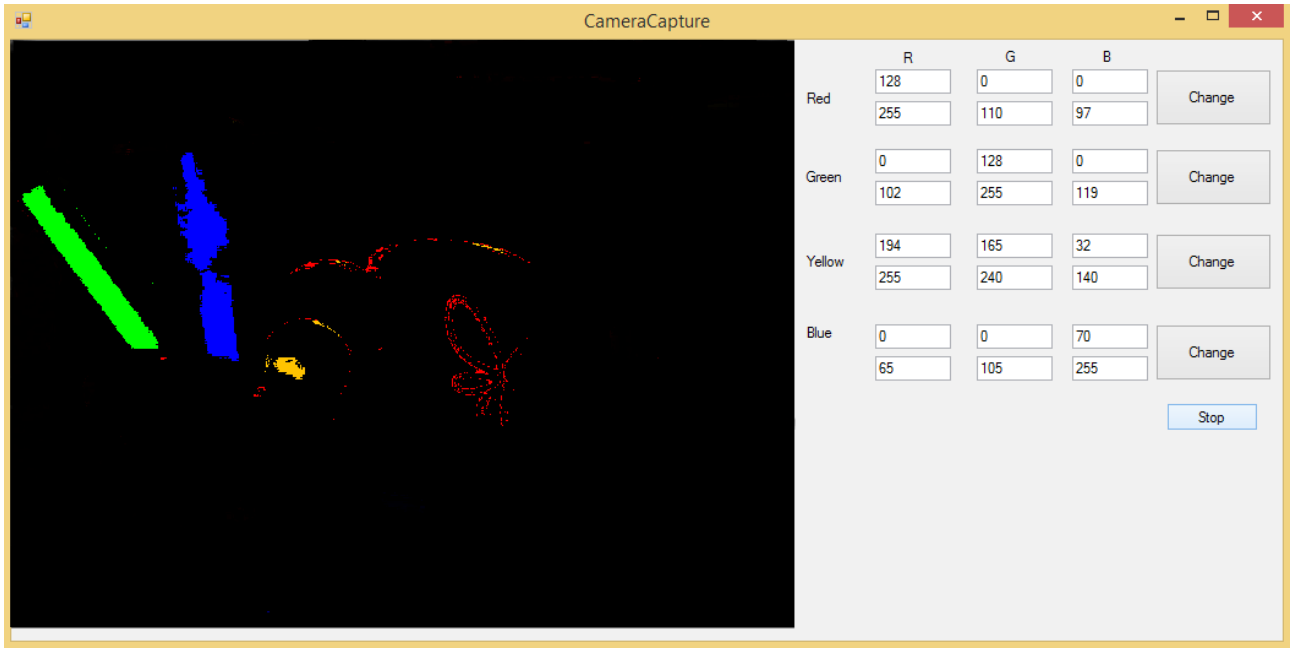
```
                pixel.Green = 0;
                pixel.Red = 0; // niebieski
                ImageFrame[i, j] = pixel;
            }
            else
            {
                pixel.Blue = 0;
                pixel.Green = 0;
                pixel.Red = 0;
                ImageFrame[i, j] = pixel;
            }
        }
    }

    CamImageBox.Image = ImageFrame;
    #endregion
```

Po stronie kodu wyglądało to następująco. Pojedyncza klatka obrazu trafiała do *ImageFrame* następnie zostawała poddana filtrowaniu. Dwie pętle *for* przechodziły pod całej klatce pixel po pixelu. Na samym wstępie pojedynczy pixel zostawał przypisany do zmiennej pixel a następnie pobierane z niego były trzy składowe rgb. W kolejnym kroku sprawdzano czy wartości zapisane w pojedynczym pixelu pokrywają się z tymi które były zapisane w instrukcji warunkowej *if*. Jeżeli tak składowe składowe pixela były nadpisywane nową wartością. Po wykonaniu się pętli *for* tak odpowiednio przetworzony obraz był przekazywany do kontrolki odpowiedzialnej za przedstawienie go na formie.

2.3 Przetwarzanie obrazu kolejny krok

W pierwszej fazie działało to całkiem względnie jednak szybko przekonałem się, że w celu uzyskiwania lepszych rezultatów musiałem dodać element odpowiedzialny za konfigurację. Przypisywanie wartości na sztywno nie sprawdzało się ponieważ przy najmniejszej zmianie oświetlenia kamera przechodziła w inny tryb pracy co powodowało znaczne problemy z poprawnym działaniem jako całości. W tym celu dodałem na formie kontrolki dzięki którym mogłem dokonywać zmiany wartości poszczególnych składowych w momencie działania aplikacji. Efekt który wówczas udało mi się uzyskać był o wiele lepszy od poprzedniego, szczególnie w sytuacji gdy zmianie ulegały warunki oświetlenia które powodowały zmianę trybu w jakim pracowała kamera. Obraz po przefiltrowaniu zawierał o wiele mniej szumów niż wcześniej co automatycznie lepiej rzutowało na przyszłe efekty pracy. Rysunek 2.3 prezentuje otrzymany przeze mnie efekt który jest całkiem zadowalający.



Rysunek 2.3: Filtrowanie obrazu po dodaniu możliwości konfiguracji w czasie działania aplikacji

Po stronie kodu wyglądało to w następujący sposób:

```
#region red
public int redRMaxVal;
public int redRMinVal;
public int redGMaxVal;
public int redGMinVal;
public int redBMaxVal;
public int redBMinVal;
#endregion

#region green
public int greenRMaxVal;
public int greenRMinVal;
public int greenGMaxVal;
public int greenGMinVal;
public int greenBMaxVal;
public int greenBMinVal;
        #endregion

#region yellow
public int yellowRMaxVal;
public int yellowRMinVal;
public int yellowGMaxVal;
public int yellowGMinVal;
public int yellowBMaxVal;
public int yellowBMinVal;
        #endregion

#region blue
public int blueRMaxVal;
public int blueRMinVal;
public int blueGMaxVal;
public int blueGMinVal;
public int blueBMaxVal;
public int blueBMinVal;
        #endregion


    #region red
```

```
            redRMaxVal = Convert.ToInt32(redRMax.Text);
            redRMinVal = Convert.ToInt32(redRMin.Text);
            redGMaxVal = Convert.ToInt32(redGMax.Text);
            redGMinVal = Convert.ToInt32(redGMin.Text);
            redBMaxVal = Convert.ToInt32(redBMax.Text);
            redBMinVal = Convert.ToInt32(redBMin.Text);
            #endregion

            #region green
            greenRMaxVal = Convert.ToInt32(greenRMax.Text);
            greenRMinVal = Convert.ToInt32(greenRMin.Text);
            greenGMaxVal = Convert.ToInt32(greenGMax.Text);
            greenGMinVal = Convert.ToInt32(greenGMin.Text);
            greenBMaxVal = Convert.ToInt32(greenBMax.Text);
            greenBMinVal = Convert.ToInt32(greenBMin.Text);
            #endregion

            #region yellow
            yellowRMaxVal = Convert.ToInt32(yellowRMax.Text);
            yellowRMinVal = Convert.ToInt32(yellowRMin.Text);
            yellowGMaxVal = Convert.ToInt32(yellowGMax.Text);
            yellowGMinVal = Convert.ToInt32(yellowGMin.Text);
            yellowBMaxVal = Convert.ToInt32(yellowBMax.Text);
            yellowBMinVal = Convert.ToInt32(yellowBMin.Text);
            #endregion

            #region blue
            blueRMaxVal = Convert.ToInt32(blueRMax.Text);
            blueRMinVal = Convert.ToInt32(blueRMin.Text);
            blueGMaxVal = Convert.ToInt32(blueGMax.Text);
            blueGMinVal = Convert.ToInt32(blueGMin.Text);
            blueBMaxVal = Convert.ToInt32(blueBMax.Text);
            blueBMinVal = Convert.ToInt32(blueBMin.Text);
            #endregion

Image<Bgr, Byte> ImageFrame = capture.QueryFrame().Copy();


            int wysokosc = ImageFrame.Height;
            int szerokosc = ImageFrame.Width;
            Bgr pixel;

            for (int i = 0; i < ImageFrame.Height; i++)
            {
                for (int j = 0; j < ImageFrame.Width; j++)
                {
                    pixelgray = ImageFrame[i, j];
                    pixel = ImageFrame[i, j];
                    double b = pixel.Blue;
                    double g = pixel.Green;
                    double r = pixel.Red;

if (r > redRMinVal && r < redRMaxVal && g > redGMinVal && g < redGMaxVal && b > redBMinVal
&& b < redBMaxVal)
                    {
                        pixel.Blue = 0;
                        pixel.Green = 0;
                        pixel.Red = 255; // czerwony
                        ImageFrame[i, j] = pixel;
                    }
                    else if (r > greenRMinVal && r < greenRMaxVal && g > greenGMinVal && g <
greenGMaxVal && b > greenBMinVal && b < greenBMaxVal)
                    {
                        pixel.Blue = 0;
```

```
                        pixel.Green = 255;
                        pixel.Red = 0; //zielony
                        ImageFrame[i, j] = pixel;
                    }
                    else if (r > yellowRMinVal && r < yellowRMaxVal && g > yellowGMinVal &&
g < yellowGMaxVal && b > yellowBMinVal && b < yellowBMaxVal)
                    {
                        pixel.Blue = 0;
                        pixel.Green = 191;
                        pixel.Red = 255; // złóty
                        ImageFrame[i, j] = pixel;
                    }
                    else if (r > blueRMinVal && r < blueRMaxVal && g > blueGMinVal && g <
blueGMaxVal && b > blueBMinVal && b < blueBMaxVal)
                    {
                        pixel.Blue = 255;
                        pixel.Green = 0;
                        pixel.Red = 0; // niebieski
                        ImageFrame[i, j] = pixel;
                    }
                    else
                    {
                        pixel.Blue = 0;
                        pixel.Green = 0;
                        pixel.Red = 0;
                        ImageFrame[i, j] = pixel;
                    }
                }
            }

            CamImageBox.Image = ImageFrame;
            #endregion
```

Na samym początku definiowane są zmienne odpowiadające poszczególnym składowym kolorów które poddawane są filtrowaniu. Następnie w konstruktorze są one inicjalizowane przez przypisanie im wartości znajdujących się w polach konfiguracyjnych które znajdują się na formie aplikacji. Wcześniej na sztywno przypisane wartości zostały zastąpione przez zmienne które można konfigurować w trakcie działania aplikacji.

2.4 Zwracane wartości

Filtrowanie obrazu w celu rozpoznania kolorów służyło czemuś większemu. Aby logika która zostanie później zaimplementowana działa poprawnie należało przesyłać jej zbiór danych na których mogłaby działać. W tym celu opracowaliśmy pewien model działania. Obraz który przechwytywany jest z kamery jest w rozdzielczości 640x480. Plansza do gry posiada wymiary 8x8 czyli jest kwadratem. Wówczas obraz który otrzymywałem przemianowałem na rozdzielczość 480x480. Następnie podzieliłem go na 64 regiony o wymiarach 60x60 pixeli każdy. Pozwoliło mi to na uzyskanie odpowiednika każdego z pól na szachownicy.
W celu zebrania wszystkiego w całość przyjąłem że na koniec filtrowania zwracane będzie tablica dwuwymiarowa o wielkości 8x8 oraz przetworzona klatka obrazu.
Pojedynczy element tablic był obiektem klasy *FieldCounter*.

```
public class FieldCounter
    {
        public int white;
        public int black;
        public int red;
        public int green;
        public int yellow;
        public int blue;
        public int undefined;
        public FieldCounter()
        {
            this.white = 0;
            this.black = 0;
            this.green = 0;
            this.red = 0;
            this.yellow = 0;
            this.blue = 0;
            this.undefined = 0;

        }
    }
```

Podczas filtrowania obrazu niestety nie udało się całkowicie wyeliminować szumów kamery. Wobec tego podczas filtrowania gdy dopasowano wartość koloru do któregoś z wyżej wymienionych zwiększano jego wartość o jeden. Ostatecznie dane pole było tego koloru którego wartość była największa.

```
public Tuple<FieldCounter[,], Image<Bgr, Byte>> getFileStateTabel(Image<Bgr, Byte>
boardPlaceImage)
        {

            FieldCounter[,] fieldCounterTable;
            fieldCounterTable = new FieldCounter[8, 8];



            for (int boardHeight = 0; boardHeight < 8; boardHeight++)
            {
                for (int boardWidth = 0; boardWidth < 8; boardWidth++)
                {
                    int imin = 60 * boardWidth;
                    int imax = (60 * boardWidth) + 60;
                    int jmin = 60 * boardHeight;
                    int jmax = (60 * boardHeight) + 60;
                    int iminRefresh = imin;
                    fieldCounterTable[boardHeight, boardWidth] = new FieldCounter();
                    for (; jmin < jmax; jmin++)
                    {
                        imin = iminRefresh;
                        for (; imin < imax; imin++)
                        {

                            Bgr pixel = boardPlaceImage[jmin, imin];
                            double b = pixel.Blue;
                            double g = pixel.Green;
                            double r = pixel.Red;

                            if (r > redRMinVal && r < redRMaxVal && g > redGMinVal && g <
redGMaxVal && b > redBMinVal && b < redBMaxVal)
                            {
                                pixel.Blue = 0;
                                pixel.Green = 0;
                                pixel.Red = 255; // czerwony
```

```
                                boardPlaceImage[jmin, imin] = pixel;
                                fieldCounterTable[boardHeight, boardWidth].red++;
                            }
                            else if (r > greenRMinVal && r < greenRMaxVal && g >
greenGMinVal && g < greenGMaxVal && b > greenBMinVal && b < greenBMaxVal)
                            {
                                pixel.Blue = 0;
                                pixel.Green = 255;
                                pixel.Red = 0; //zielony
                                boardPlaceImage[jmin, imin] = pixel;
                                fieldCounterTable[boardHeight, boardWidth].green++;
                            }
                            else if (r > yellowRMinVal && r < yellowRMaxVal && g >
yellowGMinVal && g < yellowGMaxVal && b > yellowBMinVal && b < yellowBMaxVal)
                            {
                                pixel.Blue = 0;
                                pixel.Green = 191;
                                pixel.Red = 255; // złóty
                                boardPlaceImage[jmin, imin] = pixel;
                                fieldCounterTable[boardHeight, boardWidth].yellow++;
                            }
                            else if (r > blueRMinVal && r < blueRMaxVal && g > blueGMinVal
&& g < blueGMaxVal && b > blueBMinVal && b < blueBMaxVal)
                            {
                                pixel.Blue = 255;
                                pixel.Green = 0;
                                pixel.Red = 0; // niebieski
                                boardPlaceImage[jmin, imin] = pixel;
                                fieldCounterTable[boardHeight, boardWidth].blue++;
                            }
                            else if (r > 220 && r < 255 && g > 220 && g < 255 && b > 220 &&
b < 255)
                            {
                                pixel.Blue = 255;
                                pixel.Green = 255;
                                pixel.Red = 255;//bialy
                                boardPlaceImage[jmin, imin] = pixel;
                                fieldCounterTable[boardHeight, boardWidth].white++;
                            }
                            else if (r > 0 && r < 60 && g > 0 && g < 60 && b > 0 && b < 60)
                            {
                                pixel.Blue = 0;
                                pixel.Green = 0;
                                pixel.Red = 0; //czarny
                                boardPlaceImage[jmin, imin] = pixel;
                                fieldCounterTable[boardHeight, boardWidth].black++;
                            }
                            else
                            {
                                pixel.Blue = 194;
                                pixel.Green = 255;
                                pixel.Red = 0; //czarny
                                boardPlaceImage[jmin, imin] = pixel;
                                fieldCounterTable[boardHeight, boardWidth].undefined++;
                            }
                        }
                    }
                }
            }
            Tuple<FieldCounter[,], Image<Bgr, Byte>> result = new Tuple<FieldCounter[,],
Image<Bgr, Byte>>(fieldCounterTable, boardPlaceImage);
            return result;
        }
```

Funkcja którą widać powyżej zajmuje się filtrowaniem obrazu według wcześniej ustalonego schematu działania. Dopasowuje wartość składowych rgb do wzorca, a następnie zwiększa licznik dopasowanego koloru o jeden. Obraz który jest do niej przekazywany jest odpowiednio dzielony przez ustawianie odpowiednich wartości pętli *for*. Funkcja ta zwraca obiekt typu *Tuple*, który pozwala na zwrócenie dwóch wartości.

```
Image<Bgr, Byte> ImageFrame = capture.QueryFrame().Copy();  //line 1

Image<Bgr, Byte> ImageFrameClone = ImageFrame.Copy(new Rectangle(0, 0, 480, 480));

FieldState[,] fieldStareTable;
fieldStareTable = new FieldState[8, 8];
Tuple<FieldCounter[,], Image<Bgr, Byte>> result = getFileStateTabel(ImageFrameClone);
ImageFrameClone = result.Item2;
            bool playerMakeMove = false;

            for (int i = 0; i < 8; i++)
            {
                for (int j = 0; j < 8; j++)
                {
                    int[] fieldValues = { result.Item1[i,j].white, result.Item1[i,j].black,
result.Item1[i,j].red, result.Item1[i,j].green, result.Item1[i,j].blue,
result.Item1[i,j].yellow, result.Item1[i,j].undefined };
                    int dominationColor = fieldValues.Max();

                    if (result.Item1[i, j].white == dominationColor)
                    {
                        fieldStareTable[i, j] = FieldState.Empty;
                    }
                    else if (result.Item1[i, j].black == dominationColor)
                    {
                        fieldStareTable[i, j] = FieldState.Empty;
                    }
                    else if (result.Item1[i, j].red == dominationColor)
                    {
                        fieldStareTable[i, j] = FieldState.RedPawn;
                    }
                    else if (result.Item1[i, j].green == dominationColor)
                    {
                        fieldStareTable[i, j] = FieldState.GreenPawn;
                    }
                    else if (result.Item1[i, j].yellow == dominationColor)
                    {
                        fieldStareTable[i, j] = FieldState.YellowPawn;
                    }
                    else if (result.Item1[i, j].blue == dominationColor)
                    {
                        fieldStareTable[i, j] = FieldState.BluePawn;
                    }
                    else if(result.Item1[i, j].undefined == dominationColor)
                    {
                        playerMakeMove = true;
                        break;
                    }
                }
                if (playerMakeMove == true)
                {
                    break;
                }
            }
```

Klatka obrazu zostaje przypisana do *ImageFrame* w rozmiarze 640x480.
Następnie jest przycinana do wielkości 480x480, oraz przekazywana do funkcji zwracającej przetworzony obraz oraz tablicę z rozkładem wartości kolorów w poszczególnych polach.
Tak otrzymana tablica przetworzona jest przez pętlę *for* która wykrywa co jest dominującym kolorem w poszczególnych polach szachownicy. Czy jest to jeden z zdefiniowanych sześciu kolorów czy może *undefined.* W przypadku jego wykrycia dalsze sprawdzanie tablicy nie ma sensu, ponieważ świadczy to o tym iż w chwili obecnej wykonywany jest przez użytkownika ruch. Wówczas wartość flagi *playerMakeMove* ustawiana jest na *true*.

2.5 Kalibracja kamery

W celu zapewnienia poprawnego działania całości należy po każdym uruchomieniu aplikacji odpowiednio skalibrować kamerę. Zła kalibracja spowoduje to iż wartości zwracane będą niepoprawne, co poskutkuje złym działaniem całej aplikacji.
Po uruchomieniu w zakładce kalibracja widoczny będzie podgląd z kamery oraz będą na nim zaznaczone linie pomocnicze ułatwiające kalibrację. Aby wszystko działo właściwie należy sprawić, aby widoczne linie pokrywały się z obrysem poszczególnych pól na szachownicy.
Gdy już zadanie to będzie wykonane wszystko powinno działać w sposób poprawny. Obraz będzie przechwytywany z kamery następnie przetwarzany oraz zwracany wraz z tablicą wartości kolorów w poszczególnych polach szachownicy.
Tak odpowiednio wypełniona tablica wędruje do modułu odpowiedzialnego za logikę gdzie wiadomość ta jest następnie przetwarzana oraz wizualizowana na szachownicy.

3. Podsumowanie

3.1 Wnioski z przeprowadzonych prac

Wizualizacja obrazu z kamery oraz jego przetwarzanie było jednym z kluczowych elementów projektu. Bez niego wykonanie dalszego przebiegu prac było praktycznie niemożliwe gdyż cała logika opiera się właśnie na nim. Niemniej jednak całość odpowiedzialną za ten proces udało mi się zrealizować szybko i sprawnie bez większych problemów.
Zapoznanie się z biblioteką Emgu CV również nie zajęło mi sporo czasu. W szybkim tempie przyswoiłem odpowiednie umiejętności do posługiwania się jej funkcjami. Można powiedzieć, że w ten sposób nauczyłem się czegoś zupełnie nowego. Nie wykluczone że może znajdzie to zastosowanie w przyszłych projektach które będę realizował czy to w ramach studiów czy też pracy zawodowej.