



Universitatea
Transilvania
din Brașov
FACULTATEA DE MATEMATICĂ
ȘI INFORMATICĂ

PROCESAREA DIGITALĂ A IMAGINIILOR

Cubul rubik

Realizat de: Edvesă Alexandru, Cîrstea Andrei

Brașov
Anul 3 2023-2024

Cuprins

1 Introducere	3
1.1 Descrierea temei	3
1.2 Obiectivul proiectului	3
1.3 Contribuții personale	3
2 Tehnologii folosite	3
3 Modul de Funcționare	4
4 Ghid de folosire	4
4.1 Încărcarea imaginilor	4
4.2 Indicații pentru rezolvare	6
5 Structura proiectului	7
5.1 Foldere	7
5.1.1 Images	7
5.1.2 Solver	8
5.1.3 Tools	8
6 Algoritmi folosiți	9
6.1 Filtrul Gaussian	9
6.1.1 Implementarea Algoritmului	9
6.1.2 Formula Nucleului Gaussian	10
6.2 Canny pentru imagini color	11
6.2.1 Calculul Gradientilor	11
6.2.2 Non-Maximum Suppression	11
6.2.3 Hysteresis Thresholding	12
6.3 Harris	13
6.3.1 Procesul General al Algoritmului	13
6.3.2 Calculul Gradientilor	13
6.3.3 Calculul Produselor Derivatelor	14
6.3.4 Netezirea Gaussiană	14
6.3.5 Calculul Răspunsului Harris (CRF)	14
6.3.6 Aplicarea Thresholding și Non-Maximum Suppression	14
6.3.7 Verificarea Maximului Local și a Răspunsului Cel Mai Puternic în Vecinătate	14
6.3.8 Filtrarea Colturilor Feței Cubului Rubik	14
6.3.9 Calculul Aspect Ratio	15
6.3.10 Calculul Consecvenței Unghiurilor	15
6.3.11 Calculul Ariei Cvadrilaterului	16
6.3.12 Ordonarea Colturilor	16
6.4 Transformarea proiectivă	16
6.4.1 Prezentare	16
6.4.2 Implementarea algoritmului	17
6.5 Thresholding pe culori	19
6.5.1 Aplicarea Binarizării pe Culori pentru Toate Fețele Cubului	19
6.5.2 Aplicarea Binarizării pe Culori	20
6.5.3 Verificarea Intervalului de Culoare	20
6.6 Obținerea șirului de culori	21
6.6.1 Prezentare	21
6.6.2 Implementare	21
7 Concluzii și referințe	23

1 Introducere

1.1 Descrierea temei

Proiectul propus oferă o modalitate simplificată de a rezolva cubul Rubik prin posibilitatea utilizatorului de a încărca imagini separate pentru fiecare față a cubului. Aplicația îndrumă apoi utilizatorul printr-o succesiune de mutări, oferind o ghidare pas cu pas către soluție, facilitând astfel procesul de rezolvare și făcând experiența utilizatorului mai accesibilă și intuitivă.

1.2 Obiectivul proiectului

Scopul principal al acestui proiect este de a exemplifica o situație practică în care pot fi aplicati algoritmi de procesare digitală a imaginilor.

Cubul Rubik, fiind un subiect popular la nivel mondial, este platforma ideală pentru a evidenția utilitatea acestor algoritmi într-un context captivant. Astfel, proiectul își propune să aducă la cunoștință unui public a cărui număr crește constant, beneficiile și aplicabilitatea acestor tehnologii în cadrul unuia dintre cele mai îndrăgite puzzle-uri din lume.

1.3 Contribuții personale

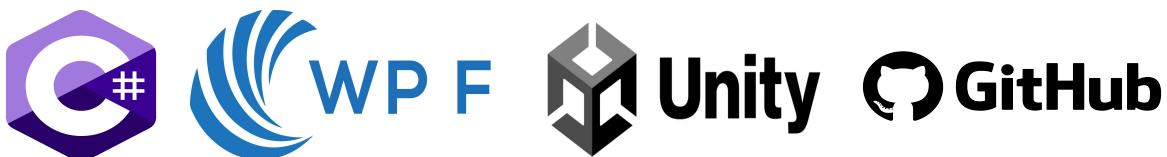
Algoritmii utilizați în etapa de prelucrare a imaginilor sunt în totalitate implementați de catre noi, fără a se baza pe soluții preexistente.

Interfața de încărcare a imaginilor a fost modificată, permitând utilizatorului să selecteze pentru fiecare poza față corespunzătoare a cubului.

Interfața de rezolvare[2] a cubului a fost modificată pentru a reflecta configurația reală a cubului pe baza imaginilor, iar funcționalitatea de prezentare a rezolvării a fost și ea modificată pentru a permite urmărirea pașilor în mod detaliat. De asemenea a fost adăugată posibilitatea de a vizualiza informații utile despre mutarea următoare(culoare, număr de rotații și direcția rotației).

2 Tehnologii folosite

- **C#** - întreaga parte de cod a acestei aplicații a fost realizată în limbajul C#, acoperind atât implementarea algoritmilor de prelucrare a imaginilor, cât și partea de rezolvare a cubului.
- **WPF(Windows Presentation Foundation)** - pentru a crea interfața destinată încărcării imaginilor de către utilizator.
- **Unity** - a fost folosit pentru dezvoltarea interfeței grafice care are scopul de a reprezenta cubul și de a permite utilizatorului să urmărească pașii de rezolvare a acestuia.
- **Github** - pentru a facilita colaborarea asupra proiectului, având în vedere că acesta a fost dezvoltat de două persoane. Prin intermediul platformei, am putut lucra simultan la aplicație, gestionând eficient versiunile, urmărind modificările și asigurând o sincronizare eficientă a codului.



3 Modul de Funcționare

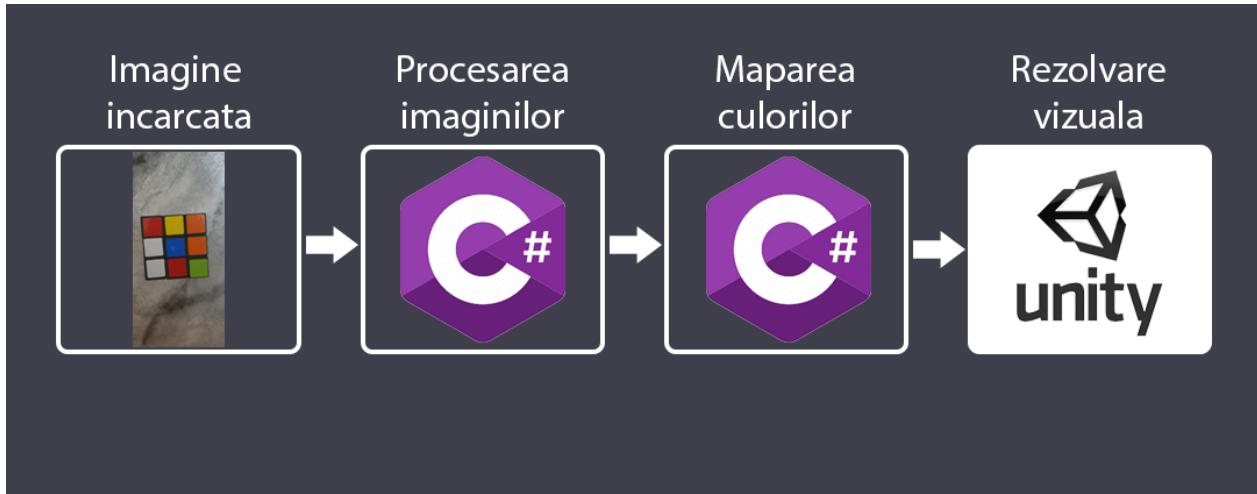


Figure 1: Reprezentare grafică a modului de funcționare

Programul preia imaginile încărcate de utilizator și le supune unui proces detaliat de procesare a imaginilor pentru a analiza fațetele Cubului Rubik. Fiecare imagine a cubului trece prin următoarele faze de procesare:

1. **Aplicarea Filtrului Gaussian:** Inițial, se aplică un filtru Gaussian pentru netezirea imaginii și reducerea zgomotului.
2. **Detectarea Marginilor cu Canny:** Se utilizează Operatorul Canny pentru a detecta marginile cubului.
3. **Detectarea Colțurilor cu Harris:** Se aplică Operatorul Harris pentru a identifica colțurile fațetelor cubului.
4. **Filtrarea Colțurilor:** Se selectează colțurile care probabil formează conturul fiecarei fațe a cubului.
5. **Corectia de Perspectivă și Decupare:** Se aplică corecții de perspectivă și se decupează fațetele cubului din imagine.
6. **Binarizare pe Culori în Spațiul HSV:** Se aplică thresholding pe culori în spațiul HSV pentru fiecare față a cubului, pentru a separa culorile stickerelor.

La finalul acestor faze de procesare, sunt generate imagini binarizate pentru fiecare dintre cele șase culori ale cubului. Aceste imagini sunt organizate în șase foldere, câte unul pentru fiecare față a cubului, fiecare folder conținând câte șase imagini în tonuri de gri, reprezentând fiecare culoare detectată.

Output-ul generat este apoi utilizat de algoritmii de mapare pentru a crea un șir de caractere ce reprezintă starea actuală a cubului Rubik. Acest șir de caractere este trimis către un executabil realizat în Unity 3D, responsabil cu aplicarea algoritmului de rezolvare și afișarea pașilor soluției.

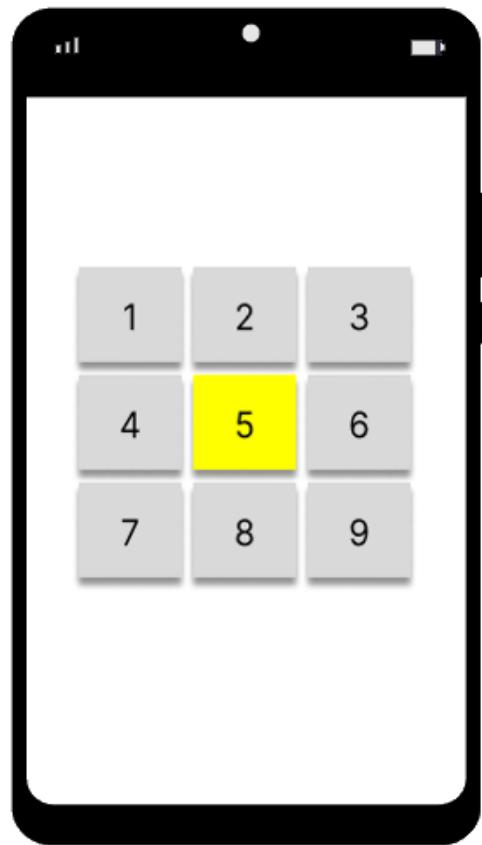
4 Ghid de folosire

4.1 Incărcarea imaginilor

- **Realizarea pozelor**

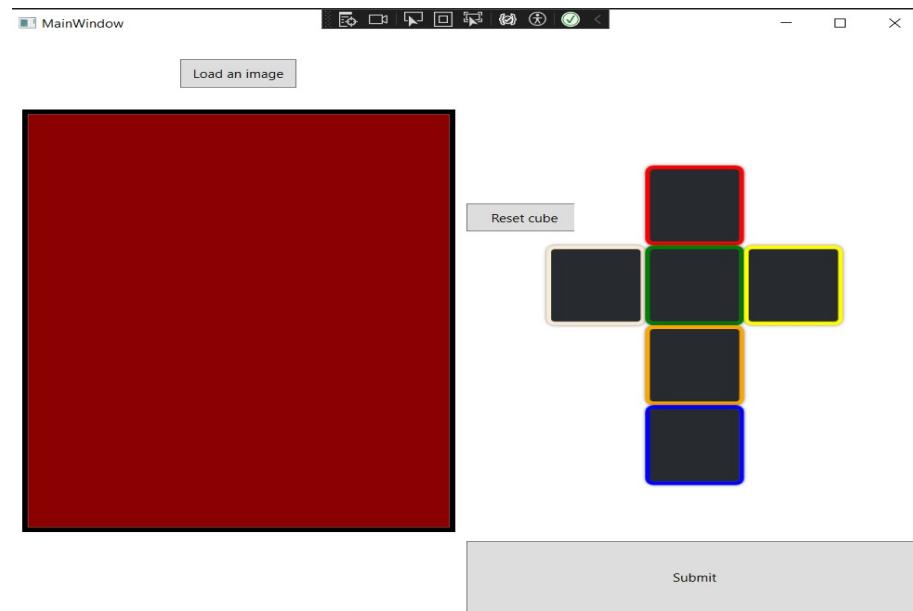
Utilizatorul face poză la fiecare față a cubului conform indicațiilor din figura următoare.

1	2	3
4	5	6
7	8	9
1	2	3
4	5	6
7	8	9
1	2	3
4	5	6
7	8	9



- **Incărcarea pozelor**

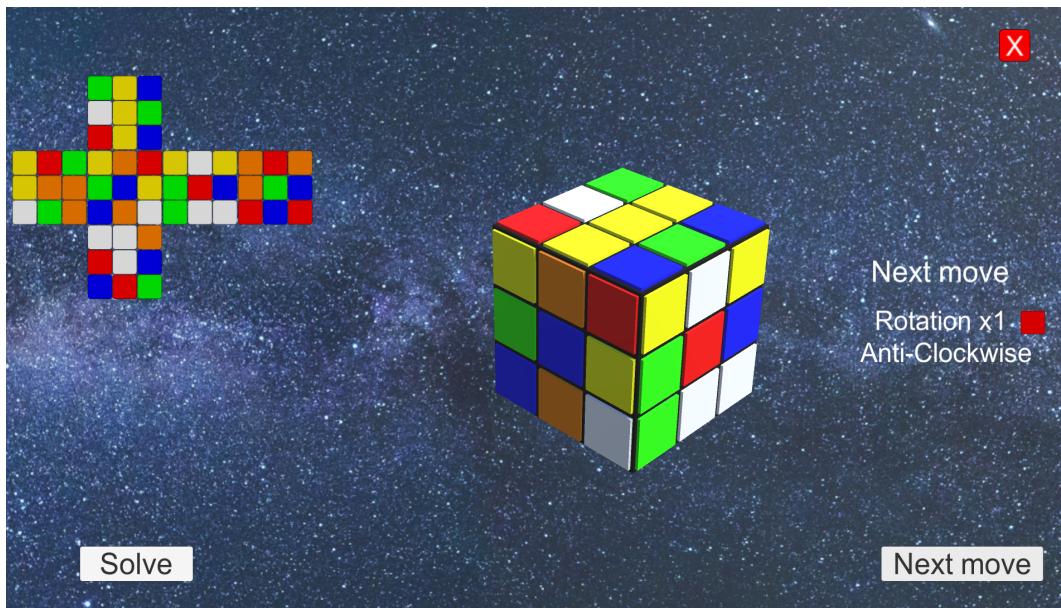
Fiecare poză se introduce în spațiul cu contur corespunzător culorii de pe pătratul din mijlocul feței respective.



4.2 Indicații pentru rezolvare

Cubul afișat pe ecran este în aşa fel construit încât să fie identic cu cel din realitate pentru care au fost introduse pozele.

Pentru a iniția procesul de rezolvare, utilizatorul trebuie să apese butonul "Solve" aflat în colțul stânga jos al ecranului. După apăsare, pe ecran vor fi afișate informații referitoare la următoarea mutare, iar un buton "Next Move" va apărea pe ecran. Acest buton, dacă este apăsat, va simula mutarea pe care utilizatorul trebuie să o facă. Acest ciclu se repetă până când cubul este complet rezolvat, oferind astfel o experiență interactivă pentru utilizator.



5 Structura proiectului

5.1 Foldere

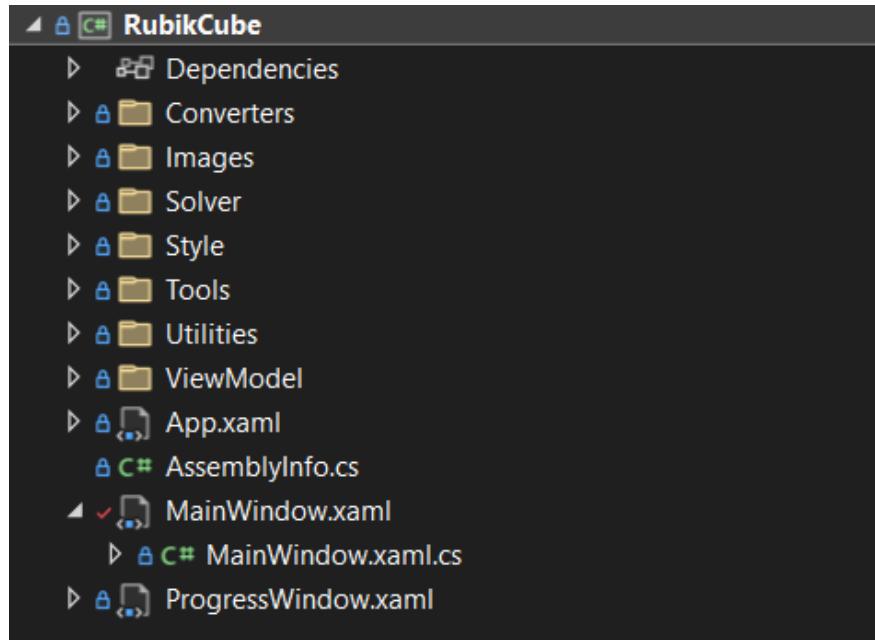


Figure 2: Structura proiectului

Proiectul a fost împărțit în mai multe foldere pentru a ajuta la organizarea codului și a resurselor folosite. Dintre toate, următoarele prezintă importanță deosebită în structura proiectului:

5.1.1 Images

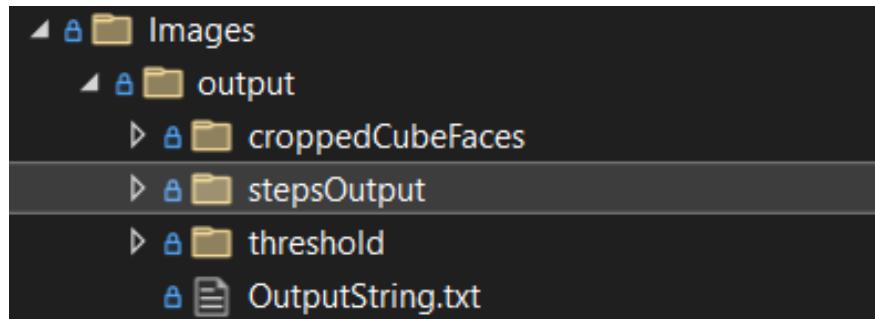


Figure 3: Folderul dedicat imaginilor

Folderul dedicat imaginilor este locația principală pentru salvarea imaginilor intermediare în timpul procesării.

5.1.2 Solver

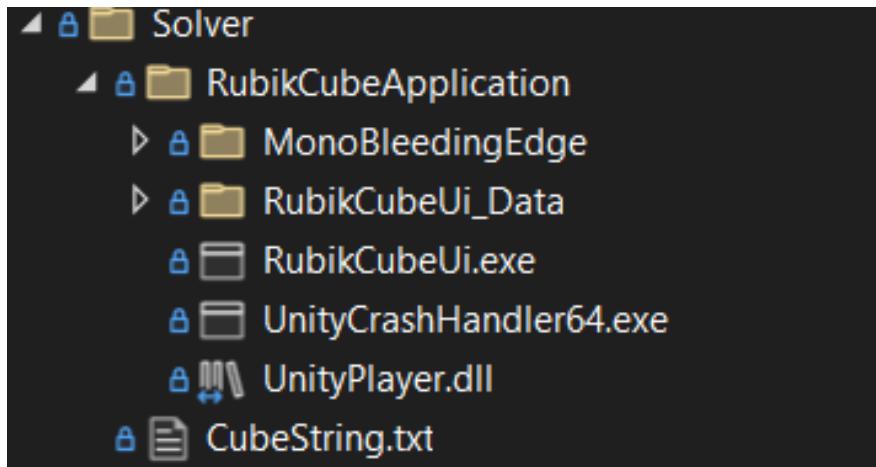


Figure 4: Folderul executabilului de rezolvare

Folderul dedicat rezolvării cubului rubik conține executabilul unity, împreună cu sirul de caractere necesar pentru ca acesta să ruleze corect.

5.1.3 Tools

În folderul dedicat algoritmilor de procesare a imaginilor se regasesc module numite sugestiv care grupează algoritmi în diferitele categorii ale procesării imaginilor.

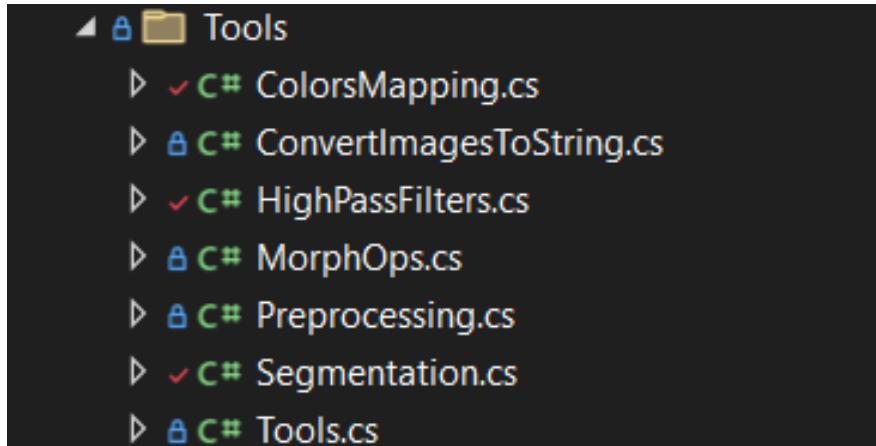


Figure 5: Folderul dedicat algoritmilor de procesare

6 Algoritmi folosiți

6.1 Filtrul Gaussian

```
public static Image<Bgr, byte> ApplyGaussianBlur(Image<Bgr, byte> image, int kernelSize, double sigma)
{
    int width = image.Width;
    int height = image.Height;
    Image<Bgr, byte> blurredImage = new Image<Bgr, byte>(width, height);

    double[,] kernel = GenerateGaussianKernel(kernelSize, sigma);

    int kernelRadius = kernelSize / 2;

    for (int channel = 0; channel < 3; channel++)
    {
        for (int y = kernelRadius; y < height - kernelRadius; y++)
        {
            for (int x = kernelRadius; x < width - kernelRadius; x++)
            {
                double sum = 0.0;

                for (int ky = -kernelRadius; ky <= kernelRadius; ky++)
                {
                    for (int kx = -kernelRadius; kx <= kernelRadius; kx++)
                    {
                        int pixelValue = (int)image.Data[y + ky, x + kx, channel];
                        sum += pixelValue * kernel[ky + kernelRadius, kx + kernelRadius];
                    }
                }

                blurredImage.Data[y, x, channel] = (byte)Math.Min(Math.Max(sum, 0), 255);
            }
        }
    }

    return blurredImage;
}
```

Figure 6: Funcția ApplyGaussianBlur

Functia `ApplyGaussianBlur` aplică un filtru Gaussian unei imagini pentru a reduce zgomotul și a netezi imaginea. Funcția primește ca parametri o imagine color (`Image<Bgr, byte> image`), dimensiunea nucleului de conoluție (`int kernelSize`) și deviația standard a distribuției Gaussiene (`double sigma`). Imaginea este procesată pe fiecare canal de culoare (BGR) separat.

6.1.1 Implementarea Algoritmului

1. Se generează un nucleu Gaussian folosind funcția `GenerateGaussianKernel`, care va fi explicitată separat.
2. Pentru fiecare pixel din imagine (excluzând marginile unde nucleul de conoluție nu se încadrează complet), se aplică nucleul Gaussian:
 - Se calculează suma ponderată a valorilor pixelilor din vecinătatea fiecărui pixel, folosind valorile din nucleul Gaussian ca ponderi.
 - Valoarea fiecărui pixel este actualizată cu această sumă ponderată.

6.1.2 Formula Nucleului Gaussian

Nucleul Gaussian este generat de funcția `GenerateGaussianKernel`. Procesul include următorii pași:

1. Inițializarea unui nucleu ca o matrice bidimensională de dimensiunea `kernelSize × kernelSize`.
2. Calculul fiecărui element al nucleului folosind formula Gaussiană fără constantă de normalizare:

$$G(x, y) = e^{-0.5 \left(\frac{(x-\text{mean})^2}{\sigma^2} + \frac{(y-\text{mean})^2}{\sigma^2} \right)}$$

unde x și y sunt coordonatele fiecărui element în nucleu, `mean` este centrul nucleului, iar σ este deviația standard.

3. Normalizarea nucleului astfel încât suma tuturor valorilor sale să fie egală cu 1. Aceasta se realizează prin împărțirea fiecărui element al nucleului la suma totală a tuturor elementelor.

```
public static double[,] GenerateGaussianKernel(int kernelSize, double sigma)
{
    double[,] kernel = new double[kernelSize, kernelSize];
    double mean = kernelSize / 2;
    double sum = 0.0;

    for (int x = 0; x < kernelSize; ++x)
    {
        for (int y = 0; y < kernelSize; ++y)
        {
            kernel[x, y] = Math.Exp(-0.5 * (Math.Pow((x - mean) / sigma, 2.0) + Math.Pow((y - mean) / sigma, 2.0)))
                / (2 * Math.PI * sigma * sigma);

            sum += kernel[x, y];
        }
    }

    // Normalize the kernel
    for (int x = 0; x < kernelSize; ++x)
        for (int y = 0; y < kernelSize; ++y)
            kernel[x, y] /= sum;

    return kernel;
}
```

Figure 7: Funcția `GenerateGaussianKernel`

Nucleul Gaussian astfel generat și normalizat este utilizat pentru a aplica filtrul Gaussian asupra imaginii.

6.2 Canny pentru imagini color

```
public static Image<Gray, byte> ApplyCannyEdgeDetectorForColor(Image<Bgr, byte> colorImage, double t1, double t2)
{
    var blurredImage = ApplyGaussianBlur(colorImage, 3, 1);

    var gradients = CalculateGradients(blurredImage);

    var suppressedGradients = NonMaximumSuppression(gradients);

    var edgeImage = HysteresisThresholding(suppressedGradients, t1, t2);

    return edgeImage;
}
```

Figure 8: Funcția ApplyCannyEdgeDetectorForColor

Funcția `ApplyCannyEdgeDetectorForColor` implementează algoritmul Canny pentru detectarea marginilor într-o imagine color. Procesul implică mai mulți pași cheie:

1. Aplicarea filtrului Gaussian pentru netezirea fiecărui canal de culoare al imaginii. Aceasta reduce zgomotul și pregătește imaginea pentru detectarea marginilor.
2. Calculul magnitudinii și direcției gradientului pentru fiecare pixel, folosind metoda `CalculateGradients`.
3. Pasul de Non-maximum Suppression pentru a păstra doar contururile cele mai puternice și a elimina cele slabe sau false.
4. Aplicarea Hysteresis Thresholding pentru a stabili care margini să fie păstrate în imaginea finală.

6.2.1 Calculul Gradientilor

Metoda `CalculateGradients` calculează gradientul fiecărui pixel din imagine. Gradientul este un vector care indică direcția celei mai mari variații a intensității și este calculat folosind operatorii Sobel. Acești operatori sunt matrici care detectează schimbările de intensitate în direcțiile orizontală și verticală.

Operatorii Sobel Sunt folosite două matrici Sobel, una pentru detectarea schimbărilor pe orizontală (`sobelX`) și alta pentru verticală (`sobelY`):

$$\text{sobelX} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad \text{sobelY} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Calculul Magnitudinii și Direcției Gradientului Pentru fiecare pixel, metoda aplică aceste matrici pentru a obține valorile gradientului pe axe X și Y (`gradientX`, `gradientY`). Magnitudinea (`Magnitude`) și direcția (`Direction`) gradientului sunt calculate astfel:

$$\text{Magnitude} = \sqrt{\text{gradientX}^2 + \text{gradientY}^2}, \quad \text{Direction} = \arctan 2(\text{gradientY}, \text{gradientX}) \times \frac{180}{\pi}$$

Acstea valori sunt folosite în pașii ulteriori ai detectării marginilor în algoritmul Canny.

6.2.2 Non-Maximum Suppression

Metoda `NonMaximumSuppression` este o componentă esențială în detectarea marginilor folosind Operatorul Canny. Scopul acestei metode este de a subția marginile detectate în etapa precedentă, astfel încât să păstreze doar contururile cele mai puternice și să eliminate variațiile minore de intensitate, care nu contribuie la structura principală a imaginii.

Procesul de Suprimare

1. Imaginea este parcursă pixel cu pixel, exclusiv o margine de 10 pixeli pentru a evita erorile de la limite.
2. Pentru fiecare pixel, se calculează magnitudinea și direcția gradientului său.
3. Se determină direcția marginii (orizontală, verticală, +45 grade, -45 grade) pe baza direcției gradientului.
4. Se compară magnitudinea gradientului pixelului curent cu magnitudinea gradientilor din direcția marginii. Acest lucru se realizează folosind o interpolare între valorile gradientului vecinilor direcți.

Interpolarea Gradientului Funcția `Interpolate` este folosită pentru a estima magnitudinea gradientului între doi pixeli adiacenți, oferind o comparație mai precisă. Formula de interpolare este:

$$\text{Interpolated Gradient} = \text{grad1} \times (1 - \text{weight}) + \text{grad2} \times \text{weight}$$

unde `grad1` și `grad2` sunt magnitudinile gradientilor vecinilor, iar `weight` este un factor bazat pe tangenta unghiului direcției gradientului.

Aplicarea Non-Maximum Suppression Dacă magnitudinea gradientului pixelului curent este mai mică decât magnitudinea interpolată, atunci acesta este considerat un non-maxim și este suprimat (setat la zero). În caz contrar, pixelul este păstrat, iar valoarea sa este ajustată în intervalul [0, 255] pentru a fi reprezentată corect în imaginea finală.

Rezultatul Non-Maximum Suppression Rezultatul acestei metode este o imagine unde marginile sunt mai subțiri și mai definite, ceea ce facilitează pasul următor de aplicare a Hysteresis thresholding în algoritmul Canny.

6.2.3 Hysteresis Thresholding

Funcția `HysteresisThresholding` este folosită în etapa finală a detectării marginilor cu Operatorul Canny. Această tehnică utilizează două praguri: un prag inferior (`lowThreshold`) și un prag superior (`highThreshold`). Scopul este de a identifica marginile puternice și de a elimina cele slabe sau neînsemnante, pentru a asigura că marginile detectate sunt semnificative și conecțate.

Procesul de Hysteresis Thresholding

1. Imaginea rezultată din Non-Maximum Suppression este analizată pixel cu pixel.
2. Fiecare pixel este evaluat în funcție de pragurile stabilite:
 - Dacă magnitudinea gradientului unui pixel este mai mare sau egală cu pragul superior, acesta este marcat ca margine puternică (`strong`).
 - Dacă magnitudinea este între pragurile inferior și superior, este marcat ca margine slabă (`weak`).
 - Dacă este sub pragul inferior, este eliminat (setat la zero).
3. Într-un al doilea pas, se examinează fiecare pixel marcat ca margine slabă. Dacă are un vecin care este o margine puternică, este convertit în margine puternică. Altfel, este eliminat.

Funcția IsStrongEdgeNeighbor Această funcție auxiliară verifică dacă un pixel are în vecinătatea sa un pixel marcat ca margine puternică. Dacă găsește un astfel de vecin, returnează `true`; altfel, `false`.

6.3 Harris

```
public static List<Point> DetectHarrisCorners(Image<Gray, byte> image, double k, double threshold)
{
    // Calculate gradients Ix and Iy
    var gradients = CalculateGradients(image);
    var Ix = gradients.Item1;
    var Iy = gradients.Item2;

    // Produs derivate
    var gradientProducts = CalculateGradientProducts(Ix, Iy);
    var Ix2 = gradientProducts.Item1;
    var Iy2 = gradientProducts.Item2;
    var IxIy = gradientProducts.Item3;

    int kernelSize = 5;
    double sigma = 1.5;
    var Ix2Smoothed = ApplyGaussianSmoothing(Ix2, kernelSize, sigma);
    var Iy2Smoothed = ApplyGaussianSmoothing(Iy2, kernelSize, sigma);
    var IxIySmoothed = ApplyGaussianSmoothing(IxIy, kernelSize, sigma);

    var harrisResponse = CalculateHarrisResponse(Ix2Smoothed, Iy2Smoothed, IxIySmoothed, k);

    var corners = ApplyThresholdingAndNonMaximumSuppression(harrisResponse, threshold, 20);

    return corners;
}
```

Figure 9: Funcția DetectHarrisCorners

Funcția `DetectHarrisCorners` implementează algoritmul Harris pentru detectarea colțurilor într-o imagine.

6.3.1 Procesul General al Algoritmului

1. Calculul gradientilor: Se determină gradientii I_x și I_y ai imaginii utilizând metoda `CalculateGradients`.
2. Produsul derivatelor: Se calculează produsele gradientilor (I_x^2 , I_y^2 , $I_x I_y$) prin metoda `CalculateGradientProducts`.
3. Netezirea: Se aplică o netezire Gaussiană asupra fiecărui produs al derivatelor, folosind metoda `ApplyGaussianSmoothing`.
4. Răspunsul Harris: Se calculează un răspuns Harris pentru fiecare pixel, reprezentând potențialul acestuia de a fi un colț.
5. Aplicarea pragului și suprimarea non-maximelor: Se identifică punctele care depășesc un anumit prag și care sunt maxime locale, interpretate ca fiind colțuri ale imaginii.

Algoritmul Harris este eficient în detectarea colțurilor datorită sensibilității sale la variațiile de intensitate din diferite direcții.

6.3.2 Calculul Gradientilor

Metoda `CalculateGradients` calculează gradientii pe axe X și Y (I_x și I_y) ai imaginii folosind operațorii Sobel. Aceasta este o etapă fundamentală pentru determinarea schimbărilor de intensitate în imagine, necesară pentru detectarea colțurilor.

6.3.3 Calculul Produselor Derivatelor

Metoda `CalculateGradientProducts` calculează pătratele valorilor gradientilor (Ix^2 și Iy^2) și produsul lor ($IxIy$). Aceste valori sunt folosite pentru a construi o matrice de autocorelație necesară în calculul răspunsului Harris.

6.3.4 Netezirea Gaussiană

`ApplyGaussianSmoothing` aplică un filtru Gaussian asupra Ix^2 , Iy^2 și $IxIy$ pentru a netezi variațiile bruște de intensitate. Această netezire ajută la stabilizarea detectării colțurilor în prezența zgomotului.

6.3.5 Calculul Răspunsului Harris (CRF)

Metoda `CalculateHarrisResponse` evaluează potențialul fiecărui pixel de a fi un colț. Se folosește formula:

$$\text{Răspunsul Harris} = \det(M) - k \times (\text{trace}(M))^2$$

unde M este matricea de autocorelație a gradientilor, $\det(M)$ este determinantul acestei matrice, iar $\text{trace}(M)$ este urma (suma elementelor de pe diagonala principală). Parametrul k este un factor de sensibilitate.

6.3.6 Aplicarea Thresholding și Non-Maximum Suppression

După calculul răspunsului Harris, se aplică un prag pentru a identifica punctele semnificative care pot fi colțuri. Acestea sunt punctele cu un răspuns Harris mai mare decât o valoare de prag specificată. De asemenea, se aplică Non-Maximum Suppression pentru a asigura că în vecinătatea unui colț puternic nu sunt selectate alte puncte ca fiind colțuri.

6.3.7 Verificarea Maximului Local și a Răspunsului Cel Mai Puternic în Vecinătate

Functiile `IsLocalMaximum` și `IsStrongestResponseWithinDistance` sunt folosite pentru a verifica dacă un pixel este maxim local și dacă are cel mai puternic răspuns într-o anumită vecinătate. Aceasta asigură că punctele selectate ca colțuri sunt distincte și bine definite.

6.3.8 Filtrarea Colțurilor Feței Cubului Rubik

Metoda `FilterCubeFaceCorners` filtrează colțurile detectate de către operatorul Harris pentru a identifica cele patru care pot forma fața unui Cub Rubik. Procesul include următorii pași:

1. Verificarea dacă sunt cel puțin patru colțuri detectate de operatorul Harris.
2. Calcularea combinărilor posibile de patru colțuri și evaluarea fiecăreia pentru a determina cea mai probabilă față a cubului.
3. Pentru fiecare grup de patru puncte, se calculează aria, aspect ratio și consistența unghiurilor.
4. Se alege combinația cu cea mai mare arie și scorul cel mai mic bazat pe aspect ratio și consistența unghiurilor.

```

private static List<Point> FilterCubeFaceCorners(List<Point> corners)
{
    double maxArea = 0;

    if (corners.Count < 4)
        return corners;

    List<Point> bestQuadrilateral = null;
    double bestScore = double.MaxValue;

    foreach (var quadrilateral in GetCombinations(corners, 4))
    {
        var quadrilateral_aux = OrderCorners(quadrilateral);
        double area = CalculateQuadrilateralArea(quadrilateral_aux);
        double aspectRatio = CalculateAspectRatio(quadrilateral_aux);

        double angleConsistency = CalculateAngleConsistency(quadrilateral_aux);

        double score = Math.Abs(1 - aspectRatio) + angleConsistency / 100;

        if (area > maxArea)
            if (score < bestScore)
            {
                bestScore = score;
                bestQuadrilateral = quadrilateral;
                maxArea = area;
            }
    }

    return bestQuadrilateral;
}

```

Figure 10: Funcția FilterCubeFaceCorners

6.3.9 Calculul Aspect Ratio

`CalculateAspectRatio` determină raportul de aspect al unui cvadrilater format din patru colțuri. Aceasta măsoară lungimea medie a laturilor opuse și calculează raportul dintre lățime și înălțime. Un raport de aspect apropiat de 1 indică o formă pătratică, așteptată în cazul unei fețe de cub.

6.3.10 Calculul Consecvenței Unghiurilor

`CalculateAngleConsistency` evaluează cât de apropriate sunt unghiurile cvadrilaterului de 90 de grade. Acest lucru ajută la identificarea cvadrilaterelor cu unghiuri drepte, care sunt mai susceptibile de a forma o față a cubului Rubik.

6.3.11 Calculul Ariei Cvadrilaterului

`CalculateQuadrilateralArea` calculează aria unui cvadrilater folosind coordonatele celor patru colțuri. O arie mai mare sugerează un cvadrilater mai probabil să reprezinte o față a cubului.

6.3.12 Ordonarea Colțurilor

`OrderCorners` aranjează colțurile unui cvadrilater într-o ordine specifică (sus-stânga, sus-dreapta, jos-dreapta, jos-stânga). Aceasta facilitează calculele ulterioare, cum ar fi corecția de perspectivă și decuparea imaginii.

6.4 Transformarea proiectivă

6.4.1 Prezentare

Acest algoritm este dedicat corectării perspectivei unei fețe a cubului. Beneficiind de punctele determinate în algoritmul anterior, acesta efectuează o rotație și extrage exclusiv fata cubului, eliminând detaliile din fundal. Astfel, se obține o imagine centrală și corectată a feței respective, contribuind la precizia și coerența procesului de identificare și interpretare a fețelor cubului.

```
public static Image<Bgr, byte> ProjectionTransformation(Image<Bgr, byte> inputImage, List<Point> inputPoints, List<Point> outputPoints)
{
    Image<Bgr, byte> result = new Image<Bgr, byte>(inputImage.Width, inputImage.Height);

    Matrix pMatrix = CalculatePMatrix(inputPoints);
    Matrix pSecondMatrix = CalculatePMatrix(outputPoints);

    Matrix bMatrix = CalculateBVector(pSecondMatrix, TransformPointToMatrix(outputPoints[3]));
    Matrix bSecondMatrix = CalculateBVector(pMatrix, TransformPointToMatrix(inputPoints[3]));

    Matrix aMatrix = CalculateAMatrix(bMatrix, bSecondMatrix, inputPoints, outputPoints);

    for (int y = 0; y < result.Height; y++)
    {
        for (int x = 0; x < result.Width; x++)
        {
            Matrix currentMatrix = MultiplyMatrices(aMatrix, TransformPointToMatrix(new Point(x, y)));

            double xC = currentMatrix[0][0] / currentMatrix[2][0];
            double yC = currentMatrix[1][0] / currentMatrix[2][0];

            int x0 = (int)xC;
            int y0 = (int)yC;

            int x1 = x0 + 1;
            int y1 = y0 + 1;

            double xr = xC - x0;
            double yr = yC - y0;

            if (x0 >= 0 && y0 >= 0 && x1 < result.Width && y1 < result.Height)
            {
                result.Data[y, x, 0] = (byte)(CalculateResultForOneChannel(x0, x1, y0, y1, xr, yr, inputImage, 0) + 0.5f);
                result.Data[y, x, 1] = (byte)(CalculateResultForOneChannel(x0, x1, y0, y1, xr, yr, inputImage, 1) + 0.5f);
                result.Data[y, x, 2] = (byte)(CalculateResultForOneChannel(x0, x1, y0, y1, xr, yr, inputImage, 2) + 0.5f);
            }
        }
    }

    return result;
}
```

Figure 11: Funcția `ProjectionTransformation`

Funcția primește ca parametrii o imagine color(`Image<Bgr, byte> inputImage`), o listă de puncte care indică unde se află colțurile feței cubului în imaginea inițială(`List<Point> inputPoints`) și o a doua listă de puncte (`List<Point> outputPoints`) care reprezintă poziția din imaginea rezultat unde se va adăuga imaginea transformată.

6.4.2 Implementarea algoritmului

- Se transformă fiecare listă de puncte într-o matrice, apelând funcția **CalculatePMatrix**. Matrice care este calculată după formula:

$$P = \begin{bmatrix} lista[0].X & lista[1].X & lista[2].X \\ lista[0].Y & lista[1].Y & lista[2].Y \\ 1 & 1 & 1 \end{bmatrix}$$

2. Calcularea vectorilor de ponderi

- Se apelează funcția **CalculateBVector** cu lista de puncte ca parametru.
- Datorită faptului că cele două pătrate, delimitate de cele două liste de puncte, sunt nedegenerate, primele trei puncte din fiecare listă ($lista[0]$, $lista[1]$, $lista[2]$) sunt liniar independente. Din această afirmație rezultă că matricile P și P' sunt inversabile. Astfel vectorii de ponderi b și b' pot fi calculați după urmatoarele formule:

$$\begin{aligned} b &= P^{-1} * P_4 \\ b' &= (P')^{-1} * P'_4 \end{aligned}$$

Unde P_4 reprezintă matricea:

$$P_4 = \begin{bmatrix} lista[3].X \\ lista[3].Y \\ 1 \end{bmatrix}$$

3. Compunerea matricei de transformare

- Se apelează funcția **CalculateAMatrix** cu vectorii de ponderi și listele de puncte ca parametrii.
- Prin această matrice de transformare A , dorim maparea patrulaterului determinat de punctele din imaginea initială pe patrulaterul determinat de punctele din a două listă în imaginea rezultat. Această matrice se poate calcula după formula:

$$A = h_4 * \left(\frac{b'_1}{b_1} P'_1 \frac{b'_2}{b_2} P'_2 \frac{b'_3}{b_3} P'_3 \right) P^{-1}$$

Unde $h_4 \neq 0$ arbitrar. De obicei se consideră $h_4 = 1$.

4. Maparea punctelor

Fiecare pixel din imaginea sursă (x, y) , considerat în coordonate omogene $(x, y, 1)^T$ se mapează pe un pixel (x', y') din imaginea rezultat astfel:

$$\begin{bmatrix} \hat{x}' \\ \hat{y}' \\ h \end{bmatrix} = A * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

De unde prin împărțirea la h se obțin coordonatele din imaginea rezultat $x' = x'/h$, $y' = y'/h$ pe care se va plasa pixelul din imaginea sursă aflat la coordonatele (x, y) . Prin transformarea directă de la imaginea sursă către imaginea rezultat, o parte din pixeli nu vor avea corespondent. O soluție la această problemă

este să pornim de la imaginea rezultat către imaginea sursă folosind transformarea inversă, dată de A^{-1} . Însă în loc să calculăm matricea A și după să o inversăm, putem utiliza toate calculele de mai sus, dar schimbăm semnificația punctelor care determină vârfurile dreptunghiului în modul următor:

- În loc să considerăm P_1, P_2, P_3, P_4 din imaginea sursă și P'_1, P'_2, P'_3, P'_4 din imaginea rezultat, vom considera invers(P_1, P_2, P_3, P_4 din imaginea rezultat și P'_1, P'_2, P'_3, P'_4 din imaginea sursă).
- Prin această metodă matricea A va reprezenta exact transformarea inversă.



Figure 12: Imagine inițială

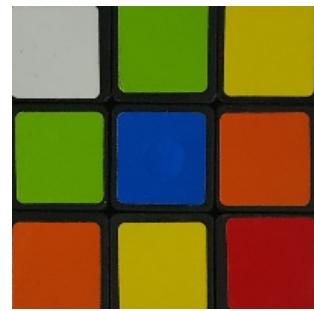


Figure 13: Imagine corectată

6.5 Thresholding pe culori

6.5.1 Aplicarea Binarizării pe Culori pentru Toate Fețele Cubului

```
public static IDictionary<string, IDictionary<string, Image<Gray, byte>>> ApplyColorThresholdingToAllSides(
    IDictionary<string, Image<Bgr, byte>> sidesImages,
    IDictionary<string, (Hsv lower, Hsv upper)> colorHsvRanges)
{
    var thresholdedImages = new Dictionary<string, IDictionary<string, Image<Gray, byte>>>();

    foreach (var sideImagePair in sidesImages)
    {
        IDictionary<string, Image<Gray, byte>> colorMasks = new Dictionary<string, Image<Gray, byte>>();
        Image<Hsv, byte> hsvImage = Tools.ConvertToHSV(sideImagePair.Value);

        foreach (var colorRange in colorHsvRanges)
        {
            Image<Gray, byte> mask = ApplyColorThresholding(hsvImage, colorRange.Value.lower, colorRange.Value.upper);
            colorMasks.Add(colorRange.Key, mask);
        }

        thresholdedImages.Add(sideImagePair.Key, colorMasks);
    }

    return thresholdedImages;
}
```

Figure 14: Funcția ApplyColorThresholdingToAllSides

Metoda `ApplyColorThresholdingToAllSides` aplică binarizarea pe culori pe imaginile tuturor fețelor cubului Rubik pentru a separa fiecare culoare a stickerelor. Procesul implică următoarele etape:

1. Inițializarea unui dicționar pentru a stoca imaginile cu prăguirea aplicată.
2. Conversia fiecărei imagini a feței în spațiul de culori HSV folosind `ConvertToHSV`.
3. Aplicarea prăguirii pe culori pe imaginea HSV pentru fiecare interval de culoare HSV definit, folosind `ApplyColorThresholding`.
4. Stocarea măștilor de culoare rezultate în dicționarul corespunzător fiecărei fețe.

Acest proces ajută la identificarea și separarea precisă a fiecărei culori de pe fiecare față a cubului.

6.5.2 Aplicarea Binarizării pe Culori

```
public static Image<Gray, byte> ApplyColorThresholding(Image<Hsv, byte> hsvImage, Hsv lowerBound, Hsv upperBound)
{
    Image<Gray, byte> mask = new Image<Gray, byte>(hsvImage.Width, hsvImage.Height);

    for (int y = 0; y < hsvImage.Height; y++)
    {
        for (int x = 0; x < hsvImage.Width; x++)
        {
            Hsv pixel = hsvImage[y, x];
            if (IsInRange(pixel, lowerBound, upperBound))
            {
                mask[y, x] = new Gray(255);
            }
            else
            {
                mask[y, x] = new Gray(0);
            }
        }
    }

    return mask;
}

1 reference | CirsteaAndrei, 5 days ago | 1 author, 2 changes
public static bool IsInRange(Hsv color, Hsv lowerBound, Hsv upperBound)
{
    return (color.Hue >= lowerBound.Hue && color.Hue <= upperBound.Hue &&
            color.Saturation >= lowerBound.Saturation && color.Saturation <= upperBound.Saturation &&
            color.Value >= lowerBound.Value && color.Value <= upperBound.Value);
}
```

Figure 15: Funcția ApplyColorThresholding

Metoda `ApplyColorThresholding` creează o mască binară pe baza unei imagini HSV și a unor limite inferioare și superioare de culoare HSV. Procesul include:

1. Crearea unei imagini binare (mască) de aceeași dimensiune cu imaginea HSV.
2. Parcursarea fiecărui pixel din imaginea HSV și aplicarea prăguirii:
 - Dacă valorile HSV ale pixelului se încadrează în intervalul definit, pixelul este marcat ca alb (255) în mască.
 - Altfel, este marcat ca negru (0).

6.5.3 Verificarea Intervalului de Culoare

`IsInRange` este o funcție auxiliară care verifică dacă o culoare HSV dată se încadrează într-un interval HSV specificat. Aceasta compară valorile de nuanță (Hue), saturatie (Satuation) și valoare (Value) ale culorii cu limitele intervalului, returnând `true` dacă culoarea este în interval și `false` în caz contrar.

6.6 Obținerea șirului de culori

6.6.1 Prezentare

Acest algoritm transformă fețele cubului într-un sir de litere. Aceste litere reprezintă inițialele fiecărei culori existente pe cub. Sirul este construit în aşa fel încât pozițiile literelor din acesta corespund cu poziția culorilor de pe cub.

```
public static string ConvertCubeToString()
{
    SolveRedCase();

    string[] cubeFaces = GetAllSubfolders(imageFolderPath);
    string cubeString = InstantiateCubeString();

    foreach (string cubeFace in cubeFaces)
    {
        string currentFaceString = ConvertFaceToString(cubeFace);

        int test = orderedFaces.IndexOf(cubeFace.Split("\\").Last());
        cubeString = InsertInCubeString(cubeString, currentFaceString, orderedFaces.IndexOf(cubeFace.Split("\\").Last()));
    }

    return cubeString.ToLower();
}
```

Figure 16: Funcția ConvertCubeToString

6.6.2 Implementare

1. Cazul roșu

- Se apelează funcția **SolveRedCase**
- Din cauza poziției culorii roșu în spațiul HSV a fost necesară tratarea a două cazuri diferite. În primul caz culoarea are valoarea nuanței (Hue) în intervalul [0, 5], iar în al doilea caz în intervalul [160, 180]. Astfel pentru culoarea roșu se generează două imagini în loc de una cum se întâmplă la restul culorilor.
Funcția **SolveRedCase** tratează această problemă prin combinarea celor două imagini. Acest lucru se realizează prin reunirea pixelilor albi a celor două imagini.

```
private static Image<Gray, byte> CombineTwoImage(Image<Gray, byte> firstImage, Image<Gray, byte> secondImage)
{
    Image<Gray, byte> combinedImage = new Image<Gray, byte>(firstImage.Size);

    for (int i = 0; i < combinedImage.Height; i++)
    {
        for (int j = 0; j < combinedImage.Width; j++)
        {
            if (firstImage.Data[i, j, 0] == 255)
            {
                combinedImage.Data[i, j, 0] = firstImage.Data[i, j, 0];
            }
            else
            {
                if (secondImage.Data[i, j, 0] == 255)
                {
                    combinedImage.Data[i, j, 0] = secondImage.Data[i, j, 0];
                }
                else
                {
                    combinedImage.Data[i, j, 0] = 0;
                }
            }
        }
    }

    return combinedImage;
}
```

Figure 17: Funcția CombinedTwoImage

2. Transformarea imaginilor în string

Pentru fiecare fată a cubului se apelează funcția **ConvertFaceToString**. În interiorul acestei funcții este apelată funcția **ConvertImageToString**. Această funcție împarte fiecare imagine rezultată din procesul de threshold, în trei părți atât pe orizontală, cât și pe verticală. Astfel, imaginea este împărțită într-o grilă de 3x3, iar pentru fiecare grilă dacă numărul de pixeli albi este mai mare decât numărul de pixeli negri se inserează culoarea asociată imaginii curente în sirul feței pe poziția dată de grila curentă.

După ce se generează sirul pentru fața curentă acesta se inserează pe poziția corespunzătoare în sirul complet al cubului. Această poziție este dată de indexul feței într-o listă predefinită în care este specificată ordinea corectă a fețelor cubului.

```
private static string ConvertFaceToString(string faceName)
{
    string[] imagesForCurrentFace = LoadImagesFromFolder(faceName);
    string currentFaceString = "NNNnnnNNN";

    foreach (string image in imagesForCurrentFace)
    {
        string currentImageName = "" + image.Split("\\").Last().Split(".")[0][0];

        if (IsImageFile(image))
        {
            Image<Gray, byte> currentImage = new Image<Gray, byte>(image);

            currentFaceString = ConvertImageToString(currentImage, currentFaceString, currentImageName);
        }
    }

    return currentFaceString;
}
```

Figure 18: Funcția ConvertFaceToString

```
private static string ConvertImageToString(Image<Gray, byte> imageToConvert, string currentFaceString, string currentColorName)
{
    string result = currentFaceString;

    int incremenFactor = (int)(imageToConvert.Width / 3);

    for (int i = 0; i < imageToConvert.Height - incremenFactor + 1; i += incremenFactor)
    {
        for (int j = 0; j < imageToConvert.Width - incremenFactor + 1; j += incremenFactor)
        {
            if (IsValidCurrentSquare(i, j, incremenFactor, imageToConvert))
            {
                result = InsertInString(result, i, j, incremenFactor, currentColorName);
            }
        }
    }

    return result;
}
```

Figure 19: Funcția ConvertImageToString

7 Concluzii și referințe

Concluzii

În cadrul acestui proiect, am abordat dezvoltarea unei aplicații complexe de procesare a imaginilor pentru rezolvarea unui Cub Rubik. Prin intermediul acestui proiect, am consolidat și aplicat cunoștințe fundamentale din domeniul procesării imaginilor digitale.

Am implementat cu succes un sistem care detectează și extrage fețele cubului din imaginile furnizate de utilizator, urmat de aplicarea unor praguri pentru identificarea culorilor. De asemenea, am adaptat o interfață grafică pentru afișarea soluțiilor, care a contribuit la o mai bună interactivitate și ușurință în utilizarea aplicației.

Prin acest proiect, am demonstrat cum teoria procesării imaginilor poate fi aplicată într-un context practic și util. Rezultatele obținute validează abordările teoretice studiate și deschid calea pentru îmbunătățiri ulterioare. Acest proiect a fost, de asemenea un mod interactiv de a învăța algoritmi de procesare a imaginilor.

Link proiect github - <https://github.com/AlexE10/RubikCube>

Referințe

1. Lector Dr. Plajer Ioana Cristina, *Procesarea digitală a imaginilor (Curs)*.
2. *Unity UI*. Disponibil online la <https://www.youtube.com/watch?v=JN9vx0veZ-c> Adaptat nevoilor proiectului.