

**Szegedi Tudományegyetem**  
**Informatikai Intézet**

# **SZAKDOLGOZAT**

**Czinkóczki Tamás**

**2022**

**Szegedi Tudományegyetem  
Informatikai Intézet**

**Képfeldolgozó szűrők alkalmazása  
hatszögmözaikon**

**Szakdolgozat**

*Készítette:*

**Czinkócz Tamás**  
Programtervező informatikus  
hallgató

*Témavezető:*

**Dr. Németh Gábor**  
adjunktus

**Szeged**  
**2022**

# Tartalomjegyzék

Feladatkiírás . . . . .	4
Tartalmi összefoglaló . . . . .	5
Bevezetés . . . . .	6
<b>1. Szomszédsági relációk</b>	<b>7</b>
1.1. Négyzetmozaik . . . . .	7
1.2. Hatszögmozaik . . . . .	8
<b>2. Program alapjai</b>	<b>9</b>
2.1. Képek beolvasása . . . . .	9
2.2. Új kép elkészítése . . . . .	9
2.3. RGB színek . . . . .	10
2.3.1. Színek lekérése . . . . .	10
2.3.2. Szürke árnyalat . . . . .	11
<b>3. Átlagoló szűrés</b>	<b>12</b>
3.1. Matematikai háttér . . . . .	12
3.2. Megvalósítás . . . . .	13
<b>4. Gauss-szűrő</b>	<b>14</b>
4.1. Matematikai háttér . . . . .	14
4.2. Megvalósítás . . . . .	15
<b>5. Unsharp szűrő</b>	<b>16</b>
5.1. Matematikai háttér . . . . .	16
5.2. Megvalósítás . . . . .	17
<b>6. Sobel operátor</b>	<b>19</b>
6.1. Matematikai háttér . . . . .	19
6.2. Megvalósítás . . . . .	20

<b>7. Frei-Chen operátor</b>	<b>21</b>
7.1. Matematikai háttér . . . . .	21
7.2. Megvalósítás . . . . .	22
<b>8. Kiértékelés</b>	<b>23</b>
8.1. Eredmény szűrők nélkül . . . . .	24
8.2. Átlagoló szűrő . . . . .	24
8.3. Gauss-szűrő . . . . .	26
8.4. Unsharp szűrő . . . . .	27
8.5. Sobel és Fei-Chen operátor . . . . .	28
8.6. Felhasználói felület . . . . .	29
<b>9. Telepítési és felhasználói útmutató</b>	<b>30</b>
<b>10. Összefoglalás</b>	<b>32</b>
Nyilatkozat . . . . .	33
Köszönnetnyilvánítás . . . . .	34
Irodalomjegyzék . . . . .	35

# Feladatkiírás

- *A témát kiíró oktató neve:*

Németh Gábor

- *A témát meghirdető tanszék:*

Képfeldolgozás és Számítógépes Grafika Tanszék

- *Típus:*

Szakdolgozat

- *Jelentkezhet:*

1 fő Programtervező informatikus BSc, Műszaki Informatikus BSc vagy Gazdaságinformatikus BSc szakos hallgató

- *A feladat rövid leírása:*

A digitális képfeldolgozásban a képműveletek rendszerint négyzetmozaikos képekre vannak definiálva, azonban az elmúlt időben egyre több kutatásban vizsgálják a hatszögmozaikos mintavételezést is.

A jelentkező feladata gyakran használt szűrők megvalósítása hatszögmozaikú képeken. A feladat része a négyzetmozaikon mintavételezett kép újramintavételezése hatszögmozaikon, valamint a megjelenítést is.

A megvalósítás tetszőleges programozási nyelven és felhasznált függvénykönyvtárral történhet.

- *Előismeretek:*

Nem szükségesek

- *Szakirodalom:*

Főként angol nyelvű, de létezik magyar nyelvű egyetemi jegyzet is

# Tartalmi összefoglaló

- *A téma megnevezése:*

Képfeldolgozó szűrők alkalmazása hatszögmozaikon

- *A megadott feladat megfogalmazása:*

A feladat egy olyan alkalmazás elkészítése, amely a fehasználó által feltöltött képeket hatszögmozaik alapúra alakítja, és ezen különböző képfeldolgozó műveleteket hajt végre, majd az eredményeket megjeleníti és elmenti.

- *A megoldási mód:*

Képek pixeleinek beolvasása, ezeken hatszögrácsnak megfelelően átalakított matematikai műveletek végrehajtása. Ezekből a kész kép kirajzolása, és mentése.

- *Alkalmasztott eszközök, módszerek:*

A feljesztése IntelliJ IDEA-ban történt Java nyelven maven segítségével. A grafikus felület.fxml segítségével lett létrehozva.

- *Elérhető eredmények:*

Az alkalmazás egy publikus GitLab oldalon elérhető, ahonnan bárki letöltheti és használhatja. A program futtatáshoz legalább 9-es JDK-val kell rendelkeznie a számítógépnek.

- *Kulcsszavak:*

képfeldolgozás, hatszögrács, hatszögmozaik, szűrők

# Bevezetés

A szakdolgozatom ötlete Dr. Palágyi Kálmán Digitális topológia és matematikai morfológia kurzusán merült fel bennem, amikor a különböző mozaikracsokról beszélünk. Mindennapi életben a monitorok, kijelzők, fényképezőgépek szenzorjai, ebből adódóan képeinken is négyzetrácsos elrendezést használunk, azonban érdekes lenne belegondolni, hogyan változna mindez meg, ha más mozaikracsot alkalmaznánk. Ezen gondolatok mentén merült fel bennem a programom ötlete, hogy megnézzük hogyan is néznénk ki hatszögmozaikra alakított képeket. Azért esett a hatszögrácsra a választásom, mivel egyre több helyen kerül elő napjainkban ennek alkalmazása. Népszerű például a digitális képek esetében, de körökre osztott stratégiai, valamint a társasjátékok területén is egyre többször használják fel.

Ha hatszögrácsokon dolgozunk érdemes odafigyelni a hatszögek egymáshoz való viszonyára. Ellentétben a négyzetráccsal, itt nem kell külön megvizsgálnunk a lap, valamint az él szomszédokat, mivel hatszögek esetében csak lapszomszédságról beszélünk. Ebből adódóan távolságszámításkor is figyelembe kell venni, hogy egy hatszög minden szomszédja egy távolságra van tőle. Erre különösen olyan képletek esetében kell odafigyelni, melyek távolságálapúak.

A továbbiakban arról fogok írni, hogy milyen funkciókkal rendelkezik a programom, ezeket hogyan valósítottam meg, valamint hogy milyen változtatásokat hajtottam végre az egyes matematikai műveleteken, hogy a kapott eredmények igazodjanak a hatszögrácschoz.

Először a fontosabb fogalmakkal, szomszédsági relációkkal, négyzet- és hatszögmozaikkal fogunk megismерkedni. Ezt követően a programom felépítéséről és néhány olyan függvényről fogok beszélni, melyeket a program használata során többször is felhasználok. Végül az egyes műveleteket fogom bemutatni, ahol először a működésükről és a matematikai hátterükről fogok beszélni, majd végül bemutatom, hogy hogyan valósítottam meg őket hatszögmozaikon a programomban.

# 1. fejezet

## Szomszédsági relációk

Amennyiben egy 2D síkot szeretnénk szabályos poligonokkal lefedni hézagok és átfedések nélkül, arra három szabályos mozaik létezik. A szabályos háromszög, a négyzet, és a szabályos hatszög. Ezek a poligonok feleltethetőek meg a képek egyes elemeinek, melyeket pixelnek hívunk.

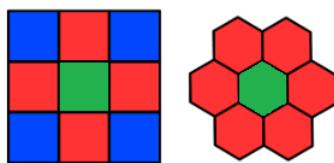
Ezek a poligonok szomszédsági relációban állnak egymással. Amennyiben az egyes poligonok közös éssel rendelkeznek úgy 1-szomszédságról, amennyiben csak közös csúccsal rendelkeznek, úgy 2-szomszédságról beszélünk. Ezek mellett megkülönböztetünk valódi szomszédságot is annak függvényében, hogy az adott képelemnek hány valódi szomszédja van. [3]

### 1.1. Négyzetmozaik

Az egyes négyzeteknek 1- és 2-szomszédai a szakirodalom számosságuk miatt 4- és 8-szomszédoknak nevezi valódi szomszédság esetén. Ezekből adódóan egyes négyzetek távolsága 4-szomszédság esetében egy egység lesz, míg 8-szomszédság esetében ezt  $\sqrt{2}$  egység.

## 1.2. Hatszögmozaik

Hatszögmozaik esetében minden elem hat mások elemmel osztozik csúcsán, és ugyanazzal a hat elemmel osztozik élen is, így az 1- és a 2-szomszédság egybeesik. Ezek miatt hatszögmozaik esetében csak 6-szomszédságról beszélünk. Mivel a hatszög minden szomszédja osztozik vele közös élen, ezért az egyes szomszédjai távolsága minden esetben egy egység lesz.



1.1. ábra. A piros elemek a zöld 1-szomszédai, a kék elemek ugyanezen elem 2-szomszédai

## 2. fejezet

# Program alapjai

Ebben a fejezetben azt fogom bemutatni, hogy a programom milyen részekből épül fel, mi történik a futása során, és milyen, az algoritmusok futását elősegítő, függvényekkel rendelkezik.

### 2.1. Képek beolvasása

A program megírásához a Java nyelvre esett a választásom, ugyanis szerettem volna egy platform független alkalmazást készíteni, hogy minél több ember tudja használni akadályok nélkül, így a továbbiakban Java nyelv beli kódrészletekre fogok hivatkozni.

A beolvasott képeket *BufferedImage* típusú változóban tárolom el, ennek méretét ezt követően csökkentem, ezáltal kevesebb pixelből fog állni a bemeneti képem. Erre azért van szükség, mert a későbbiekben az újrarajzolt képen az egyes pixeleknek megfelelő hatszögek nagyobb mérete miatt, a kép mérete is megnő. Ezt követően a képen található pixelek színét elmentem a *Picture* osztály egy kétdimenziós tömbjébe.

### 2.2. Új kép elkészítése

A beolvasott adatokat a *HexaPanel* osztályban fogom feldolgozni. Itt *JPanel* osztályból örökítve, valamint a *Hexagon* osztályban megírt hatszög polinom segítségével fogom megrajzolni az egyes pixeleknek megfelelő hatszögeket az alábbi módon:

```
private Polygon createHexagon() {
    Polygon polygon = new Polygon();
    for (int i = 0; i < 6; i++) {
        int xval = (int) (center.x + radius
                           * Math.cos(i * 2 * Math.PI / 6));
        int yval = (int) (center.y + radius
                           * Math.sin(i * 2 * Math.PI / 6));
        polygon.addPoint(xval, yval);
    }
    return polygon;
}
```

Miután létrehoztam egy üres poligon objektumot,  $60^{\circ}$ -os forgatások segítségével élenként megrajzolom a hatszög oldalait.

Ezekből a hatszögekből úgy alkottam összefüggő rácsot, hogy minden páros számú függőleges oszlopot elcsúsztattam lefelé a hatszög magasságának felével. Kirajzolás közben egy új *Hexagon* objektumot hozok létre minden pixelnek, melynek színét beállítom az adott helyen található pixel színére. Végül az elkészült rajzot a *JFrame* osztályból örökített *HexFrame* osztály segítségével jelenítem meg egy új ablakban.

## 2.3. RGB színek

Itt olyan kisebb kódöt fogok bemutatni, amiket a későbbiekk során többször is felhasználok.

### 2.3.1. Színek lekérése

Az elmentett színeket egy *Color* típusú változóban tárolom, azonban ez nem nyúlt lehetőséget az RGB kódban eltárolt alapszínek elérésére beépített parancsal, így erre három parancsot hoztam létre, *getRed*, *getGreen*, *getBlue*, melyek felépítése nagyban megegyezik.

```
public static int getRed(int i, int j) {
    if (i < 0 || i >= height || j < 0 || j >= width) {
        return -1;
    }
    return (pixel[i][j] >> 16) & 0xFF;
}
```

A metódus paraméterében vár két egész számot, melyek a kép egy pixelének koordinátáit lesznek. Ezek után ellenőrzi, hogy a megadott pontok részei-e a képnak. Amennyiben azok, a függvény az adott pixel értékét biteltolás segítségével számolja ki. Mivel az egyes értékek 0-255-ig terjedhetnek, ezért ezek tárolására 8 bitre van szükség színenként, melyeket fordított sorrendben tárol el. Tehát a kék színnél nincs szükség biteltolásra, mivel ez az első 8 biten tárolódik, a zöld színnél 8 bitnyi eltolásra van szükség, a piros színnél pedig 16 bitnyire. Ezek után az **0xFF** segítségével határozzuk meg, hogy a bittől kezdve csak a következő 8 bitet válasszuk ki. Ugyan ezen logika mentén működik a *getRedFilter* a *getBlueFilter* és a *getGreenFilter* is, viszont ezek a fentiekkel ellentétben nem az eredeti *pixel[][]* tömbből kérnek le értékeket, hanem a későbbiekben az átlagoló szűrő által elkészített kép értékeit fogják visszaadni.

### 2.3.2. Szürke árnyalat

A későbbiekben szükséges volt az egyes színeknek megfelelő szürke árnyalatot is megadjam. Ehhez az alábbi *getGray* függvényt használtam, mely a három szín átlagából adja vissza az árnyalatot.

```
public static int getGray(int i, int j) {  
    if (i < 0 || i >= height || j < 0 || j >= width) {  
        return 0;  
    }  
    return (getRed(i, j) + getGreen(i, j)  
        + getBlue(i, j)) / 3;  
}
```

## 3. fejezet

### Átlagoló szűrés

Az átlagoló szűrő egy képsímító művelet, mely a szomszédságot felhasználva számolja ki az egyes pontok színének értékét úgy, hogy minden pontot egy átlag segítségével számol ki a saját és a szomszédos pontok intenzitásából. Hatszögrács esetében 6-szomszédság szerint fogunk számolni.

#### 3.1. Matematikai háttér

Egy viszonylag egyszerű algoritmusról beszélhetünk, mely minden iterációba a környezeti értékekből számít átlagot. Ez a számítás egy pont közvetlen szomszédait figyelembe véve hatszögmozaik esetében a hat szomszédjával és a saját értékével történik, így kilenc helyett csak hét értékkel számolunk. Természetesen ez a művelet nagyobb sugárral is végrehajtható, ekkor a sugárnak megfelelően nagyobb szomszédsági mátrixszal kell dolgoznunk. A képek szélénél a szomszédság vizsgálata miatt problémába ütközhetünk, ugyanis ezek a pontok nem rendelkeznek minden oldalról szomszédokkal. Ennek kezelésére több megoldás is létezik, például az új érték kiszámítható csak az adott értékekből. Másik megoldás lehet, hogy nulla értéket használunk azokon a helyeken, ahol nincs pixel, ekkor a képek szélén található pontok jelentősen sötétebbek lesznek.

### 3.2. Megvalósítás

Ezt a funkciót a *MeanFilter* osztályomban valósítottam meg, ahol két for ciklus segítségével bejárom a kép pixeleit. A függvény futása a felhasználó által megadott sugártól függően három különböző méretű maszkot számol. A pontok kiszámítása egy egységnyi sugár esetén az alábbiak szerint történik:

```
Color mean;

mean = new Color(
    (getRed(i, j) + getRed(i - 1, j) + getRed(i + 1, j)
    + getRed(i, j + 1) + getRed(i, j - 1) +
    getRed(i - 1, j - 1) + getRed(i + 1, j - 1)) / 7,
    (getGreen(i, j) + getGreen(i - 1, j) + getGreen(i + 1, j)
    + getGreen(i, j + 1) + getGreen(i, j - 1) +
    getGreen(i - 1, j - 1) + getGreen(i + 1, j - 1)) / 7,
    (getBlue(i, j) + getBlue(i - 1, j) + getBlue(i + 1, j)
    + getBlue(i, j + 1) + getBlue(i, j - 1) +
    getBlue(i - 1, j - 1) + getBlue(i + 1, j - 1)) / 7);
```

Készítetek egy új *Color* típusú változót, melyben az eredményt fogom tárolni. Ennek a változónak az inicializálása három paraméter alapján történik meg, ami a színek RGB értékeinek felelnek meg. Ezekhez minden pontban lekértem az egyes színek értékét és ezekből számolom ki az adott színérték átlagát. Miután ezt mind a három színre kiszámoltam, a kapott értéket eltárolom a *Picture pixel[][]* tömbjében.

A program futása két és három egységnyi sugár esetében is hasonló futást követ, azonban ekkor nagyobb szomszédságot figyelembe véve több pontot használunk az átlagok kiszámításakor. Az olyan pontokat, amelyek nem rendelkeznek minden oldalukon szomszéddal a program figyelmen kívül fogja hagyni, értéküket nem fogja módosítani.

## 4. fejezet

### Gauss-szűrő

A Gauss-szűrő egy általában zajszűrésre használt algoritmus, melynek alapját a Gauss-eloszlás adja.

#### 4.1. Matematikai háttér

A Gauss-szűrő általában konvolúció segítségével érhetjük le Gauss-eloszlást használva. Kétdimenziós képek esetén ennek képlete:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

ahol  $x$  és  $y$  az adott pont távolsága a kép középpontjától,  $\sigma^2$  pedig a variancia értéke. Ennek használata után a kapott értékekből egy konvolúciós mátrixot kapunk. minden pixel értéke a szomszédos pixelek súlyozott átlagából kerül kiszámításra, ahol az eredeti pixelünk értéke kapja a legnagyobb sújt, mivel ez rendelkezik a legnagyobb értékkel a képlet kiszámítása után, a többi pixel értéke pedig távolságuk függvényében arányosan csökken.

## 4.2. Megvalósítás

A szűrőt a *GaussBlur* osztályban valósítottam meg. Itt először létrehoztam a különböző maszkméretekhez tartozó filtereket egy tömbként, ahol az értékeket hatszögmozaikból leképeztem négyzetrácsra, amiket alkalmazni fogok a képen. Ezek alkalmazásakor figyelembe kell vennem azt is, hogy a filterek nem lóghatnak ki a képből. Ezt követően az egyes filtereket használva végrehajtottam a Gauss-eloszlás, amellyel kiszámoltam az egyes színek értékét.

```
final int pixelIndexOffset = width - radius;
final int centerOffsetX = radius / 2;
final int centerOffsetY = filter.length / radius / 2;

for (int h = height - filter.length / radius + 1,
    w = width - radius + 1, y = 0; y < h; y++) {
    for (int x = 0; x < w; x++) {
        int r = 0;
        int g = 0;
        int b = 0;
        for (int filterIndex = 0, pixelIndex = y * width + x;
            filterIndex < filter.length;
            pixelIndex += pixelIndexOffset) {
            for (int fx = 0; fx < radius; fx++, pixelIndex++,
                filterIndex++) {
                int col = input[pixelIndex];
                int factor = filter[filterIndex];

                r += ((col >>> 16) & 0xFF) * factor;
                g += ((col >>> 8) & 0xFF) * factor;
                b += (col & 0xFF) * factor;
            }
        }
        r /= sum; g /= sum; b /= sum;
        output[x + centerOffsetX + (y + centerOffsetY) * width]
        = (r << 16) | (g << 8) | b | 0xFF000000;
    }
}
```

# 5. fejezet

## Unsharp szűrő

Az unsarp szűrő egy képélesítő maszk, amelyet már a sötétkamrás képelőhívásokkor is használtak, és a digitális képfeldolgozásban is megjelent ennek köszönhetően. Neve a folyamat működésére utal, ahol egy életlen, homályos, negatív képet használunk fel, hogy az eredeti képre maszkot készítsünk. Bármilyen digitális képen szeretnénk ezt alkalmazni, először szinte minden esetben szükséges a képről egy másolatot készíteni. Ezen a bármilyen műveletet végrehajthatunk, ami a képet homályosítja. Ilyen művelet például az átlagoló szűrés és a gauss-szűrés is, de minden zajszűrésre alkalmazott szűrő megfelel. Ezt követően az eredeti, és a homályos képünket egymásra helyezve el tudunk távolítani az elmosódott részeket, miközben az élek erőteljesebben jelennek meg.

### 5.1. Matematikai háttér

Matematikailag egy egyszerű képlettel leírható a művelet során felhasznált kernel:

$$G(x, y) = F(x, y) + (F(x, y) - F_{blured}(x, y)) * amount$$

Itt  $G(x, y)$  az elkészült képet jeletni,  $F(x, y)$  az eredeti kép,  $F_{blured}(x, y)$  az eredeti kép valamilyen módon homályosított verziója, az *amount* pedig általában attól függ, hogy a homályos kép esetén hány pixel értéket érint ez a homályosítás az adott kernelben. Például egy 4-szomszédságot használó szűrés után ez a szám öt lesz, egy 8-szomszédságot használó után pedig kilenc, de ennek az értéknek az ilyen módon történő megadása nem kötelező az operátor szempontjából. Ezek alapján a kernel négyzetrács esetén így néz ki:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \left( \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} / 5 \right) 5 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

5.1. ábra. Kernel kiszámítása négyzetrács esetén

Mivel én hatszögráccson dolgozom a fenti példát alapul véve alakítottam át a kernelem. Mivel hatszögek esetében minden szomszédos hatszög egy egységnyi távolságra van, így a homályos kép minden pontja egyes értékkel fog szerepelni az  $F_{blured}(x, y)$  mátrixban, és ehhez igazodva az *amount* változó értéke is hétre fog növekedni. Ezekből kiindulva az alábbi kernelt kaptam:

$$\begin{array}{c} \text{Input Grid: } \begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \\ + \left( \begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} - \begin{array}{ccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array} / 7 \right) 7 = \begin{array}{ccccccc} -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & 7 & -1 & -1 & -1 & -1 & -1 \end{array} \end{array}$$

5.2. ábra. Kernel kiszámítása hatszögrács esetén

## 5.2. Megvalósítás

A maszkot az *UnsharpFilter* osztályban valósítottam meg. A maszk számításához használt kernelt egy *unsharpFilter* nevű metódusban valósítottam meg.

```
public static int unsharpFilter(int i, int j, String color) {
    switch (color) {
        case "red":
            return control(
                (float) getRedFilter(i, j) +
                ((float) getRedFilter(i, j) -
                (float) getRed(i, j)) * 7) / 7;
```

```
case "green":  
    return controll(  
        (float) getGreenFilter(i, j) +  
        (((float) getGreenFilter(i, j) -  
        (float) getGreen(i, j)) * 7) / 7);  
  
case "blue":  
  
    return controll(  
        (float) getBlueFilter(i, j) +  
        (((float) getBlueFilter(i, j) -  
        (float) getBlue(i, j)) * 7) / 7);  
  
default:  
    return 0;  
}  
}
```

A függvény paraméterben várja az adott pixel helyzetét a kétdimenziós tömbben, valamint egy szöveges patamétert, amely segítségével a pixel egyes RGB komponensének értékét számolja ki. A kernel kiszámításához az átlag szűrőm által elkészített képet használtam fel (*getRedFilter*, *getGreenFilter*, *getBlueFilter*), és készítettem egy *controll* nevű metódust is, amely azt figyeli, hogy az egyes színértékek a 0 és 255 közötti tartományt ne léphessék át. Erre a típuskonverziókor megtörténő kerekítések miatt van szükség.

Az *unsharp* metódusban végül bejárom a kép pixeleit, és kiszámolom az értékeket a *unsharpFilter* segítségével, majd ezeket elmentem a régi értékek helyére a metódus futásának végén.

# 6. fejezet

## Sobel operátor

A Sobel, vagy Sobel–Feldman operátor egy digitális képfeldolgozásban alkalmazott konvolúciós maszk. Ez egy  $3 \times 3$ -as gradiens detektor, amely vertikális és horizontális irány esetében is kiszámolja az adott pontok magnitúdóját, és ezek segítségével tudja megállapítani az élek helyzetét egy képen. Előnye, hogy a maszk mérete miatt a zajokra kevésbé érzékeny.

### 6.1. Matematikai háttér

Az operátor két kernellel dolgozik, melyek segítségével az éleket meghatározza, ezek az alábbi módon néznek ki:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

6.1. ábra. Sobel operátor kerneljei

Ahol  $A$  az eredeti képünk, a két mátrix pedig egy horizontális és egy vertikális maszk. Miután ezek segítségével megkapjuk a  $G_x$  és a  $G_y$  képeket, ezek négyzetösszegeinek gyökéből megkapható a gradiens magnitúdó kép. Ez a fenti művelet azonban más dimenziókra is kiterjeszhető, én ezt felhasználva hoztam létre kerneljeim, mivel hatszögrács esetében három dimenzió mentén tudunk köztük navigálni. Az így létrehozott kerneljeim között a horizontálisat szébtöltöttem kettőre, melyek átlósan haladnak át a képen:

$$\begin{array}{ccc} 1 & 2 & \\ 1 & 0 & 1 \\ -1 & -2 & -1 \end{array} \quad \begin{array}{ccc} 2 & 1 & \\ 1 & 0 & -1 \\ -1 & -2 & -1 \end{array} \quad \begin{array}{ccc} 1 & -1 & \\ 2 & 0 & -2 \\ 1 & 1 & -1 \end{array}$$

6.2. ábra. Sobel operátor kerneljei hatszögetrács esetén

## 6.2. Megvalósítás

Az operátort a *SobelOperator* osztály *sobelDetector* metódusában valósítottam meg. Mivel a képet fekete-fehérben fogjuk feldolgozni, ezért itt az RGB értékek helyett a *getGray* metódussal dolgoztam. Először kiszámolom a három koordináta iránynak megfelelő maszkok szerint az egyes pontok értékét az alábbiek szerint:

```
int color;

int x = -1 * getGray(i - 1, j) + -1 * getGray(i + 1, j) +
        -2 * getGray(i, j + 1) + 2 * getGray(i, j - 1) +
        getGray(i - 1, j - 1) + getGray(i + 1, j - 1);

int y = getGray(i - 1, j) + -2 * getGray(i + 1, j) +
        -1 * getGray(i, j + 1) + getGray(i, j - 1) +
        2 * getGray(i - 1, j - 1) + -1 * getGray(i + 1, j - 1);

int z = 2 * getGray(i - 1, j) + -1 * getGray(i + 1, j) +
        getGray(i, j + 1) + -1 * getGray(i, j - 1) +
        getGray(i - 1, j - 1) + -2 * getGray(i + 1, j - 1);

color = (int) Math.sqrt((x * x) + (y * y) + (z * z));
```

Ezt követően a három értékből kiszámoljuk az egyes pontok intenzitását. Közben folyamatosan nyomon követtem, hogy a kapott értékek közül melyik a legnagyobb, és a legkisebb. Erre azért volt szükség, hogy a későbbiekben hisztogram széthúzást tudjunk alkalmazni, mert előfordulhat, hogy egyes világos vagy sötét képek esetén ezek az értékek nagyon közel vannak egymáshoz, ez által az élek nehezen kivehetőek lennének. Ehhez a *scale* függvényt használtam, amely az intervallum minimum és maximum, valamint az adott maszkok legkisebb és legnagyobb elemeiből számolja ki az értékeket széthúzva, vagy éppen összenyomva. Ezek után a kapott értékek mentésre kerülnek.

## 7. fejezet

# Frei-Chen operátor

A Frei-Chen hasonlóan a Sobel operátorhoz, egy élkereső operátor, amely  $3 \times 3$ -as konvolúciós maszkot használ, azonban itt kilenc konvolúciós maszkot különböztetünk meg, melyek tartalmazzák a bázisvektorokat.

### 7.1. Matematikai háttér

Az operátorhoz használt kernelekből összesen kilenc van, melyek így néznek ki:

$$\begin{aligned} G_1 &= \frac{1}{2\sqrt{2}} \begin{bmatrix} 1 & \sqrt{2} & 1 \\ 0 & 0 & 0 \\ -1 & -\sqrt{2} & -1 \end{bmatrix} & G_2 &= \frac{1}{2\sqrt{2}} \begin{bmatrix} 1 & 0 & -1 \\ \sqrt{2} & 0 & -\sqrt{2} \\ 1 & 0 & -1 \end{bmatrix} & G_3 &= \frac{1}{2\sqrt{2}} \begin{bmatrix} 0 & -1 & \sqrt{2} \\ 1 & 0 & -1 \\ -\sqrt{2} & 1 & 0 \end{bmatrix} \\ G_4 &= \frac{1}{2\sqrt{2}} \begin{bmatrix} \sqrt{2} & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & -\sqrt{2} \end{bmatrix} & G_5 &= \frac{1}{2} \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} & G_6 &= \frac{1}{2} \begin{bmatrix} -1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \end{bmatrix} \\ G_7 &= \frac{1}{6} \begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix} & G_8 &= \frac{1}{6} \begin{bmatrix} -2 & 1 & -2 \\ 1 & 4 & 1 \\ -2 & 1 & -2 \end{bmatrix} & G_9 &= \frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \end{aligned}$$

7.1. ábra. Frei-Chen operátor kerneljei

Ezek közül az első négyet használjuk az élek, a következő négyet a vonalak detektálására, az utolsó pedig az átlag kiszámítására használjuk. Miután a megfelelő maszkokat kiválasztjuk, és rávettítjük a képünkre a kapott eredményekből készült képek négyzetösszegeinek gyökéből megkapható a művelet végeredménye.

Mivel a hatszögrácshoz a fenti kerneleket át kellett alakítanom, és az itt használtak hasonló felépítésűek a Sobel-ben használtakhoz, ezért az ott végrehajtott átalakítások mentén az alábbi maszkokat kaptam:

$$\begin{array}{ccc}
 \begin{matrix} 1 & \sqrt{2} & 1 \\ 0 & 0 & -1 \\ -1 & -\sqrt{2} & -1 \end{matrix} &
 \begin{matrix} \sqrt{2} & 1 & -1 \\ 0 & 0 & -\sqrt{2} \\ 1 & -1 & -1 \end{matrix} &
 \begin{matrix} 1 & -1 & -\sqrt{2} \\ 0 & 0 & -1 \\ \sqrt{2} & 1 & -1 \end{matrix}
 \end{array}$$

7.2. ábra. Frei-Chen kernelek hatszögrács esetén

## 7.2. Megvalósítás

A műveletet a *FreiChen* osztály *FreiChenFilter* metódusában valósítottam meg. A képen található értékeket itt is átalakítottam a *getGray* metódust használva, majd az egyes maszkokat ráhelyeztem a képekre.

```

int color;

int x = -1 * getGray(i - 1, j) + -1 * getGray(i + 1, j) +
       -2 * getGray(i, j + 1) + 2 * getGray(i, j - 1) +
       getGray(i - 1, j - 1) + getGray(i + 1, j - 1);

int y = getGray(i - 1, j) + -2 * getGray(i + 1, j) +
       -1 * getGray(i, j + 1) + getGray(i, j - 1) +
       2 * getGray(i - 1, j - 1) + -1 * getGray(i + 1, j - 1);

int z = 2 * getGray(i - 1, j) + -1 * getGray(i + 1, j) +
       getGray(i, j + 1) + -1 * getGray(i, j - 1) +
       getGray(i - 1, j - 1) + -2 * getGray(i + 1, j - 1);

color = (int) Math.sqrt((x * x) + (y * y) + (z * z));
    
```

Ezt követően kiszámoljuk az egyes pontok értékét a *color* változóba majd, majd itt is hisztogram széthúzást hajtunk végre, ahogy azt a Sobel operátor estében is tettük. Végül a kapott értékeket elmentjük.

## 8. fejezet

### Kiértékelés

Lentebbiekben két példa képen fogom bemutatni a programom futása után kapott eredményeket. Az egyik képem egy színes gyíkot ábrázoló kép, mely különböző stílusokra van bontva, a másik képem pedig a képfeldolgozásban gyakran használt Lena kép lesz.

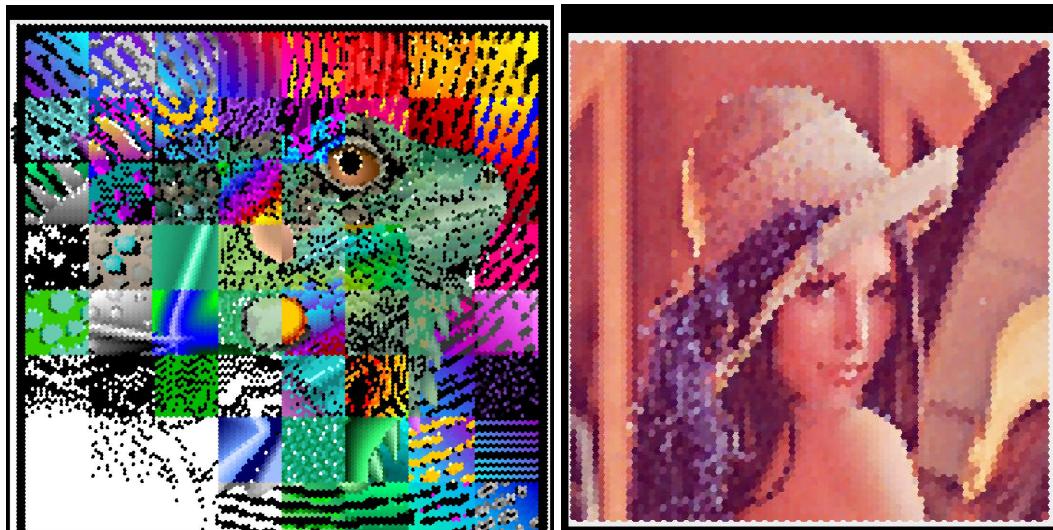


8.1. ábra. Kiindulási gyík és Lena kép

Azért esett ezekre a képekre a választásom, mert a gyík kép esetében érdekes lehet megfigyelni, hogy az adott részletekhez tartozó színek, éles váltások és apró vonalak hogyan fognak módosulni, esetleg eltűnni vagy torzulni az egyes műveletek hatására. Léna esetében pedig egy szinte minden képfeldolgozási területen tesztelt képről van szó, így az eredmények bármikor könnyedén összehasonlíthatóak lesznek a négyzetrácsos képműveletek eredményeivel.

## 8.1. Eredmény szűrők nélkül

Itt azt fogjuk látni, hogy milyen eredményt kapunk, amennyiben a program futásakor nem használunk semmilyen maszkot. A képek szélén találhat fekete keretet a megjelenítéskor megjelenő keret okozza.



8.2. ábra. Hatszögrácsos gyík és Lena kép

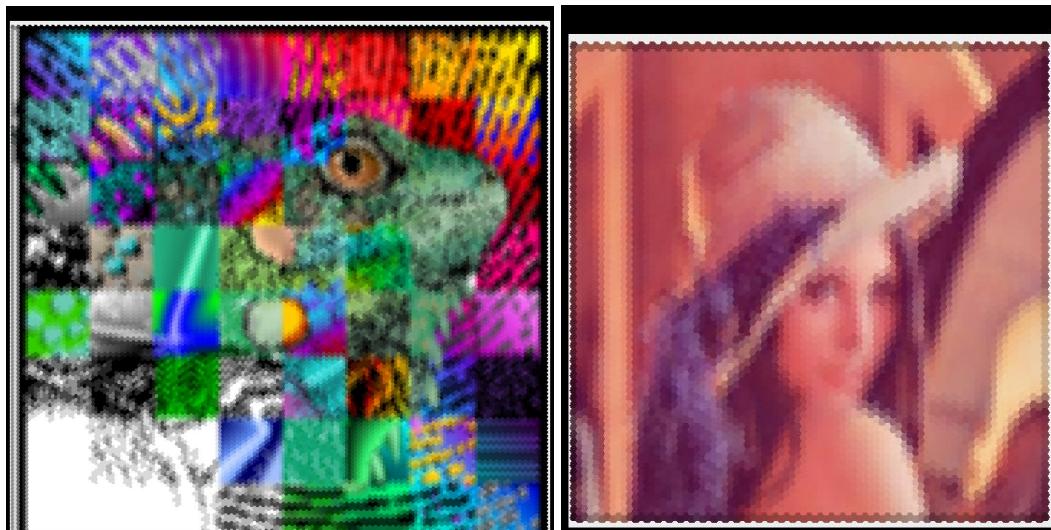
A kép láthatólag jól megőrizte az eredeti színeit, az élek nem mosódtak el, minden össze a hatszögek elhelyezkedéséből adódó elcsuszások vehetők észre. Az apróbb pontok és vékony vonalak esetében észrevehető pixelszétválasztás. Ezt az okozza, hogy a pixelek nagyobb mérete miatt az eredeti kép méretét csökkentettük.

## 8.2. Átlagoló szűrő

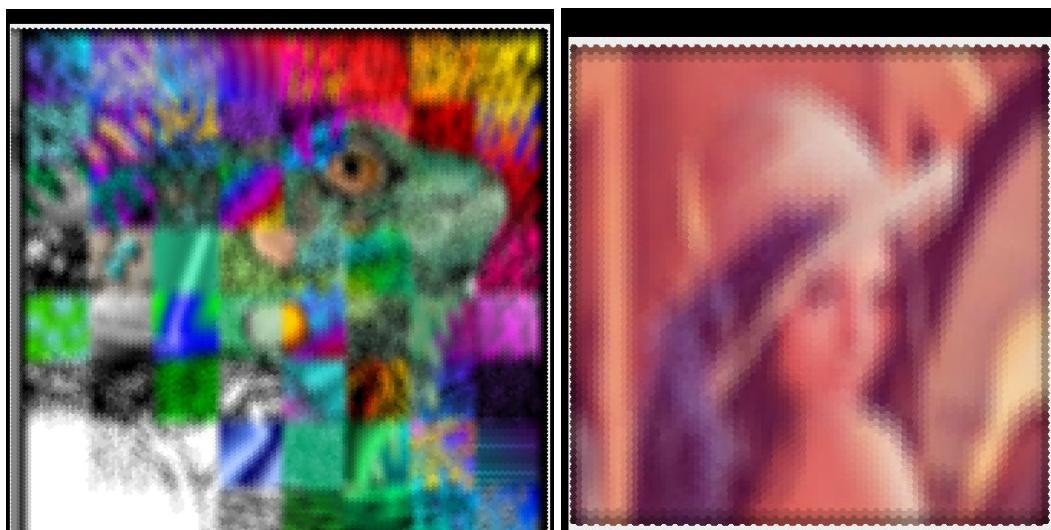
Itt az eredményeket rendre 1, 2 és 3 egységnyi sugár szerint fogjuk egymás alá helyezni. Jól megfigyelhető, hogy a sugár növelésével a képek egyre homályosabbak lesznek, az alakzatok egyre kevésbé kivehetőek. A gyík esetében a bal alsó sarokban lévő pontokon észrevehető, hogy egyre kevésbé látszanak, jól megfigyelhető így a zajszűrés is.

## Képfeldolgozó szűrők alkalmazása hatszögmozaikon

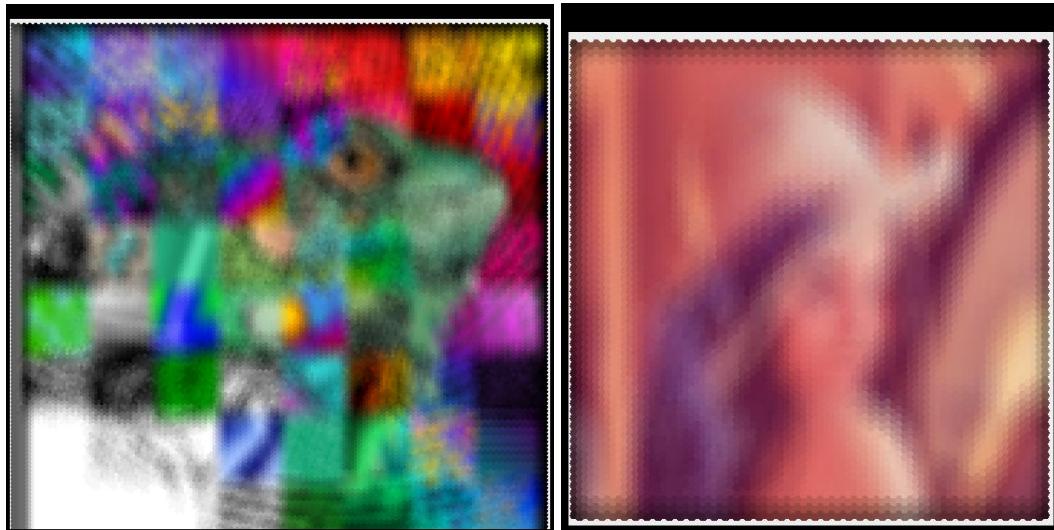
---



8.3. ábra. Egy egységnyi sugarú átlagoló szűrő használata gyík és Lena képen



8.4. ábra. Két egységnyi sugarú átlagoló szűrő használata gyík és Lena képen



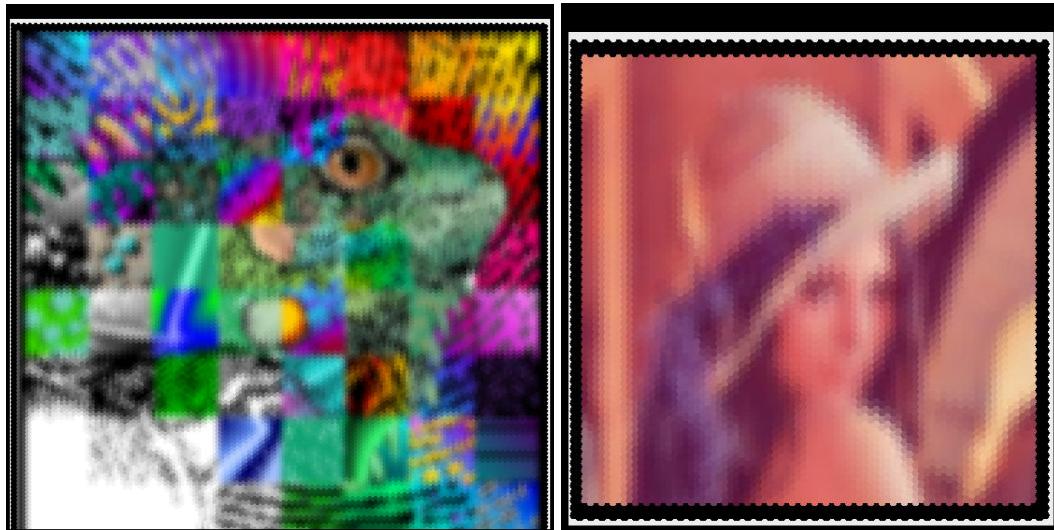
8.5. ábra. Hárrom egységnyi sugarú átlagoló szűrő használata gyík és Lena képen

### 8.3. Gauss-szűrő

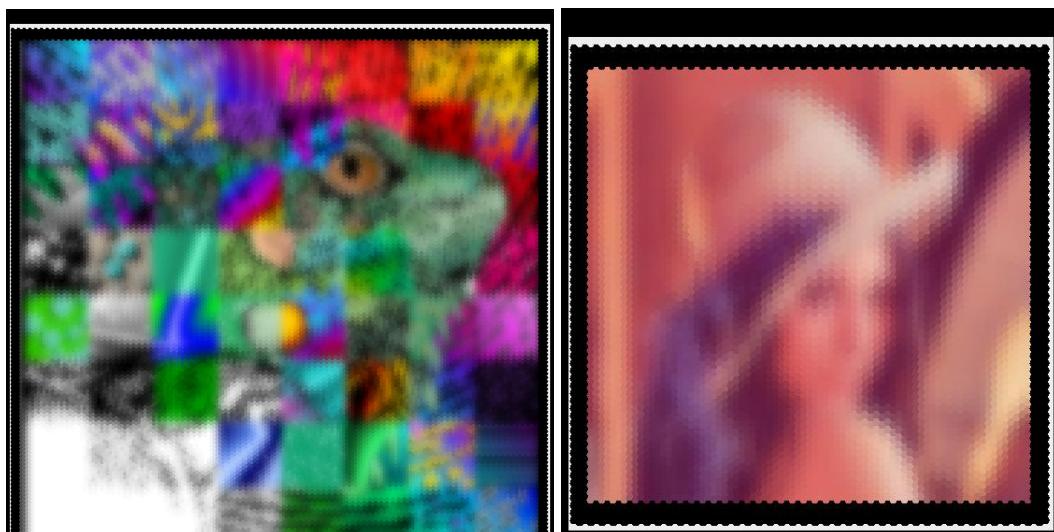
Gauss maszk esetében 3, 5 és 7 egységnyi maszkméret szerint lesznek elhelyezve a képek egymás alá. Itt is megfigyelhető, hogy a maszkméret növekedésével a képek is egyre homályosabbá válnak, azonban az átlag szűréssel ellentétben kisebb az elmosódás mértéke azokon a képeken, ahol a maszk mérete megegyező volt. A zajszűrési képesség a gyík képünk bal sarkában ismét jól megfigyelhető.



8.6. ábra. Hárrom egységnyi gauss-szűrő használata gyík és Lena képen



8.7. ábra. Öt egységnyi gauss-szűrő használata gyík és Lena képen



8.8. ábra. Hét egységnyi gauss-szűrő használata gyík és Lena képen

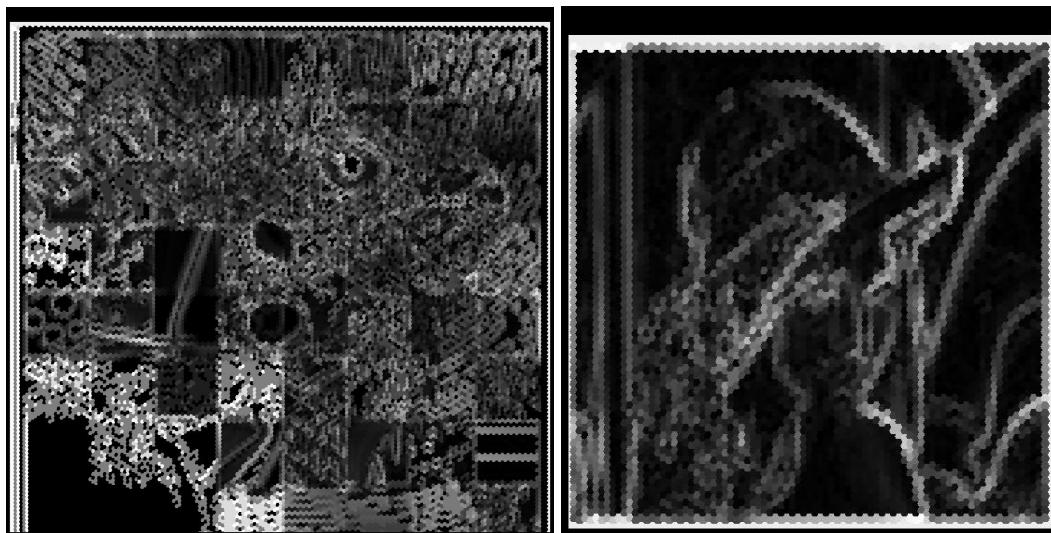
## 8.4. Unsharp szűrő

Unsharp szűrő esetében megfigyelhető, hogy a képek eredetileg is világos pontjai most még élénkebb értékeket vettek fel, ez által élesebb kontúrokat adva a képen szereplő alakoknak, ezek mellett mind a két kép esetében elmondható, hogy a színek intenzitása is megnőtt.

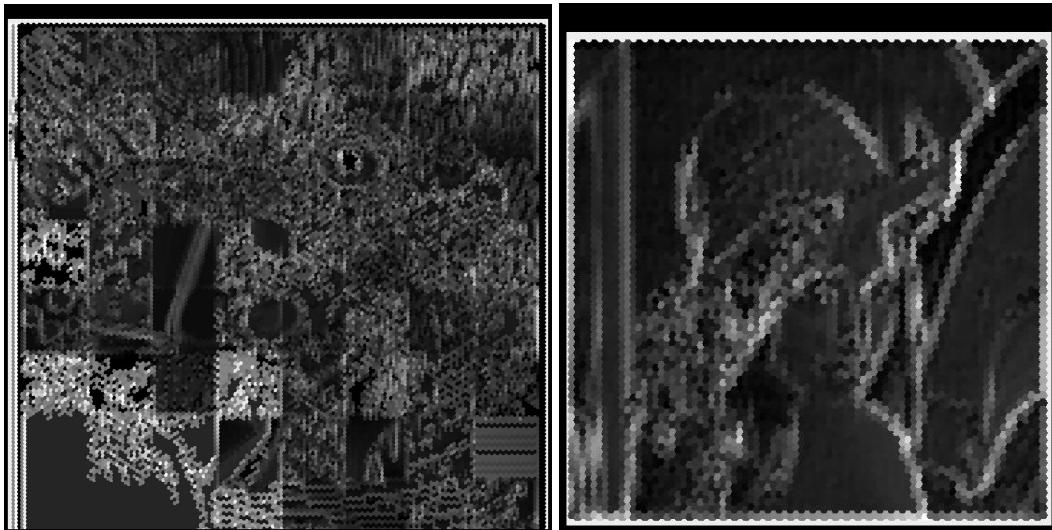


8.9. ábra. Unsharp szűrő használata gyík és Lena képen

## 8.5. Sobel és Fei-Chen operátor



8.10. ábra. Sobel operátor használata gyík és Lena képen



8.11. ábra. Frei-Chen operátor használata gyík és Lena képen

Sajnos a gyík esetében egy viszonylag zajos képet kaptunk, ennek oka az apró minta lehet ami a háttérben található, de a gyík szeme és fejének körvonala így is kivehető. Léna esetében egy sokkal tisztább képet látunk határozott vonalakkal. Habár mind a két operátor edeménye nagyon hasonló, megállapítható, hogy Frei-Chen operátor esetében egy finomabb, tónusoltabb képet kaphatunk eredményül.

## 8.6. Felhasználói felület

A felhasználói felület.fxml-ben készült. A program indítása után a felhasználónak meg kell adnia a használni kívánt kép elérési útvonalát, valamint meg kell adnia a mentési útvonalat is. Ezek megadására van lehetsége a fájlrendszerben való tallázás során is. Amennyiben a felhasználó ezek megadása nélkül szeretné elindítani a programot, vagy hibás elérési útvonalat ad meg, a program hibaüzenettel jelzi azt.

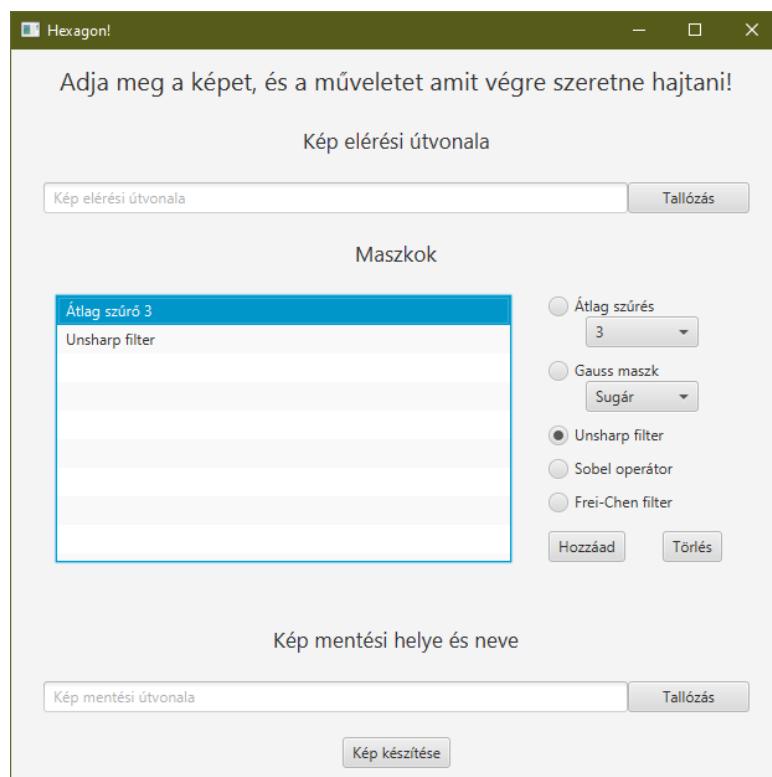
A különböző szűrőket, operátorokat a felhasználó rádiógombok segítségével tudja kiválasztani, valamint lenyíló listából választhatja ki a művelethez használt sugár értékét is, majd ezeket egy listához tudja adni, esetlegesen törlni a már korábban hozzá adott elemeket. Listának köszönhetően az egyes szűrők sorrendjét és darabszámát a felhasználó állíthatja be.

## 9. fejezet

# Telepítési és felhasználói útmutató

A projekt elérhető a <https://gitlab.com/czinkoczitomi/szakdolgozat> oldalon. Letöltése után a *Szakdolgozat.zip* file tartalmazza a program minden forráskódját, ezek szabadon megtekinthetőek. A *Szakdolgozat.jar* egy tömörített Java állomány, mely futtatása esetén a program telepítésre nincs szükség, de amennyiben a számítógép nem rendelkezik legalább Java 9 JDK-val, ennek telepítésére szükséges. Ez letölthető az Oracle hivatalos oldaláról: <https://www.oracle.com/java/technologies/downloads/#java19>.

A program indítása után az alábbi grafikus felület fogadja a felhasználót:



9.1. ábra. Felhasználói felület

## Képfeldolgozó szűrők alkalmazása hatszögmozaikon

---

A felületen minden beviteli mező rendelkezik feliratokkal, és input ellenőrzéssel is, amik segítségével a felhasználó minden hibáról egyértelmű értesítsek, hibaüzeneteket kap, hogy ezeket könnyen lehessen oldani.

A maszkok kiválasztására rádiógombok és gombok segítségével van lehetőség, ezek ekkor egy listába kerülnek, ahonnan a felhasználók törlni is tudják a kiválasztott elemeket.

# **10. fejezet**

## **Összefoglalás**

A kódom írásakor egy mindenki számára könnyen használható, érdekes és képfeldolgozási szempontból helyes megoldásokra törekedtem. Ezekhez sikerült tartanom is magam, és egy látványos eredményekkel rendelkező programot sikerült készítenem.

Az egyes algoritmusok átalakítása során az eddigi tudásom elmélyítésére volt szükség, különösen a hatszögrácsra való átalakítások során, melyhez több nemzetközi kutatást is elolvastam és felhasználtam.

Az eredmény képek színei megtartották eredeti színeiket, a rajtuk található tárgyak alakjait is érdemben megtartjáket, és összességében látványos eredmények kapunk.

# Nyilatkozat

Alulírott Czinkóczi Tamás, Programtervező informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Képfeldolgozás és Számítógépes Grafika Tanszékén készítettem, Programtervező informatikus BSC diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2022. december 8.

.....

aláírás

# **Köszönnetnyilvánítás**

Ezúton szeretnék köszönetet mondani témavezetőmnek Dr. Németh Gábor segítségéért és koordinálásáért, Dr. Palágyi Kálmánnak a témám ötletéért és képfeldolgozás területén szerzett tudásomért, valamint az egyetem minden dolgozójának és tanárjának az eddigi segítségéért és munkájáért.

# Irodalomjegyzék

- [1] Kardos Péter: Trianguláris vékonyítás  
<https://www.inf.u-szeged.hu/~pkardos/vekonyitas/topology.html>  
(Utolsó megtekintés: 2022.11.26.)
- [2] Hexagonal Grids  
<https://www.redblobgames.com/grids/hexagons/>  
(Utolsó megtekintés: 2022.11.16.)
- [3] P. Kardos, K. Palágyi: On topology preservation of mixed operators in triangular, square, and hexagonal grid, 2015
- [4] Kató Zoltán: Szűrés képtérben  
<http://www.inf.u-szeged.hu/~kato/teaching/DigitalisKepfeldolgozasTG/03-SpatialFiltering.pdf>  
(Utolsó megtekintés: 2022.11.21.)
- [5] R. Fisher, S. Perkins, A. Walker and E. Wolfart: Unsharp Filter, 2003  
<https://homepages.inf.ed.ac.uk/rbf/HIPR2/unsharp.htm>  
(Utolsó megtekintés: 2022.11.27.)
- [6] R. Fisher, S. Perkins, A. Walker and E. Wolfart: Sobel Edge Detector, 2003  
<https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm>  
(Utolsó megtekintés: 2022.11.26.)
- [7] Gradient-based edge detection on a hexagonal structure  
<https://ir.iba.edu.pk/cgi/viewcontent.cgi?article=1137&context=businessreview>  
(Utolsó megtekintés: 2022.11.25.)
- [8] Frei-Chen edge detector  
<https://www.rastergrid.com/blog/2011/01/frei-chen-edge-detector/>  
(Utolsó megtekintés: 2022.11.23.)
- [9] Dicky Apdilah, Muhammad Yasin Simargolang, Robbi Rahim: A Study of Frei-Chen Approach for Edge Detection, 2011  
[https://www.academia.edu/31220261/A\\_Study\\_of\\_Frei-Chen\\_Approach\\_for\\_Edge\\_Detection](https://www.academia.edu/31220261/A_Study_of_Frei-Chen_Approach_for_Edge_Detection)  
(Utolsó megtekintés: 2022.11.22.)

## Képfeldolgozó szűrők alkalmazása hatszögmozaikon

---

- [10] Jonathan M.Blackledge: Digital Image Processing, 2005  
<https://www.sciencedirect.com/topics/engineering/gaussian-blur>  
(Utolsó megtekintés: 2022.11.24.)
- [11] Aryaman Sharda: Image Filters: Gaussian Blur, 2021  
<https://aryamansharda.medium.com/image-filters-gaussian-blur-eb36db6781b1>  
(Utolsó megtekintés: 2022.11.26.)