Matematikai és Informatikai Intézet

Szkenner megvalósítása egér szenzorral

Készítette
Bodnár Máté
Programtervező informatikus BSc

Témavezető
Dr. Geda Gábor
Egyetemi docens

Eger, 2024

Tartalomjegyzék

Bevezetés

A digitalizálás egyre nagyobb szerepet játszik az életünkben, különösen a dokumentumok kezelésében és tárolásában, hiszen a sok ideig tárolt papír dokumentumok elveszhetnek vagy könnyen sérülhetnek.

Manapság a családi fotókat vagy régebbi képeket is célszerű digitalizálni, hogy tovább megmaradjon. A digitalizálással lehetőségünk nyílik arra is, hogy rendszerezzünk hivatalos iratokat, így könnyebben tudunk majd hozzájuk férni és akár keresni is köztük.

Bár számos szkenner van a piacon, viszont ezek drágák és nagy helyigényűek. Ezért egy saját készítésű szkenner jobban illeszkedik majd a mi igényeinkhez, valamint sokkal olcsóbb, mint egy boltban vásárolt.

Ebben a projektben megmutatom, hogy lehet egy egérszenzort felhasználni hivatalos iratok vagy egyéb papír alapú dokumentumok szkennelésére. A szenzor két sínnel vízszintesen és függőlegesen fog mozogni, és soronként készíti el a képkockákat. Ezután egy C# alkalmazás feldolgozza azokat és kialakít belőlük egy nagy dokumentumot.

1. fejezet

Bevezető

1.1. Motiváció

Az ötletem mögött több tényező is áll. Elsősorban szerettem volna egy olyan eszközt létrehozni, amely megfizethető alternatívát nyújt a drága szkennerekkel szemben. Az egérszenzorok könnyen beszerezhetők és viszonylag olcsók, ezért jó választásnak tűntek egy saját fejlesztésű szkennerhez. Ez különösen hasznos lehet olyan helyeken, ahol a költségek csökkentése nagyon fontos, például iskolákban vagy kisebb cégeknél.

Mindig is érdekelt, hogyan lehet egy meglévő egyszerű technológiát új és kreatív módon felhasználni. Az egérszenzorokat alapvetően mozgásérzékelésre tervezték, de ebben a projektben meg szeretném mutatni, hogy dokumentumok szkennelésére is alkalmasak.

Emellett fontos számomra, hogy egy olyan eszközt alkossak, amelyet egy egyszerű felhasználó is használhat, otthon vagy akár a munkahelyén anélkül, hogy drága berendezésekre kellene költenie.

Valamint kihívást látok ebben a projektben, hogy hogyan is tudom ezt megvalósítani. Izgalmas feladat az, hogy ötvözzem az informatikát az elektronikával. Ez nem csak a szakmai tudásomat fejleszti, hanem egy olyan eredményt ad, amelyre büszke lehetek.

1.2. Célkitűzés

A szakdolgozatom[1] célja egy olyan szkenner létrehozása, amely egy egyszerű egérszenzort használ a dokumentumok fekete-fehér beolvasására. Az eszköz működésének alapja, hogy a szenzor monokróm felvételeket készít a dokumentumról, majd ezeket egy számítógépes programmal feldolgozom és összeállítom egy nagy dokumentummá. Mivel a szenzor csak egy 30 x 30-as képet rögzít így azt interpolációval nagyítom fel. Ezek az algoritmusok lehetővé teszik, hogy a képet megnöveljük kevés minőségvesztéssel.

A dolgozat eredményeként egy egyszerű és költséghatékony szkennert szeretnék létrehozni, amely hasznos és megkönnyíti az emberek életét.

2. fejezet

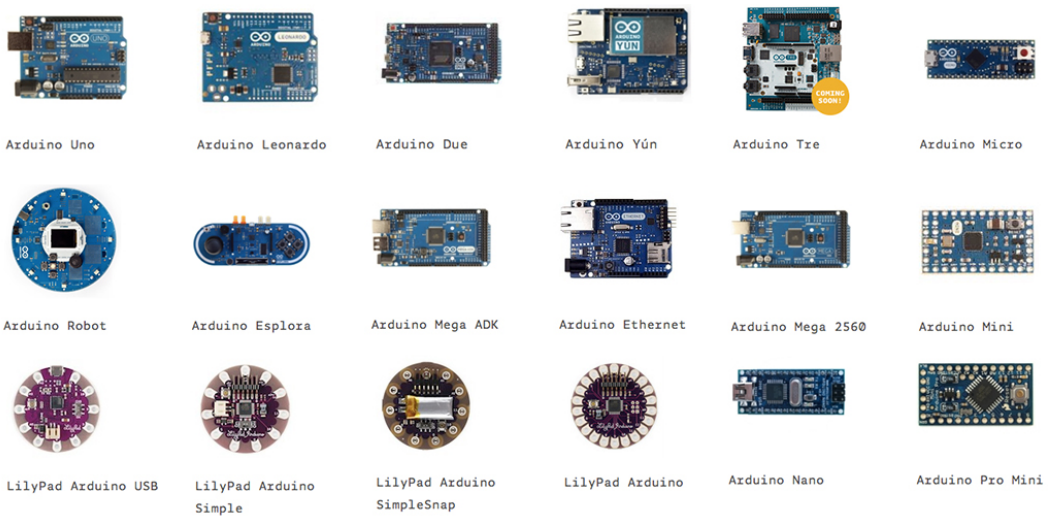
Felhasznált technológiák

Ebben a fejezetben a szakdolgozatomban használt technológiákról és azok előnyeiről, fogok beszámolni.

2.1. Arduino

2.1.1. Arduino platform bemutatása

Az Arduino[2][3] egy nyílt forráskódú platform, amiket az elektronikai projektekhez találtak ki, majd bekerült az oktatásba is, oktatási céllal. Sokan használják egyszerűbb feladatok automatizálására vagy akár okos otthon rendszerek kialakítására. Ezek mellett manapság már az ipari alkalmazásuk sem ritka. A működéséhez szükség van egy mikrokontrollerre, valamint egy fejlesztő környezetre az Arduino IDE-re, amivel általában USB kábelen keresztül tudjuk átküldeni a programot a fizikai eszközre.

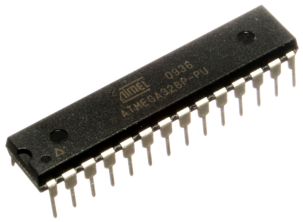


2.1. ábra. Arduino család néhány fajtája

2.1.2. Arduino Nano

Elsősorban UNO-val[4] terveztem elkészíteni ezt a projektet, de helyszűkében kénytelen voltam egy kisebb mikrokontroller után kutatni. Ekkor esett a választásom az Arduino Nano-ra[5]. Az Arduino Nano az Arduino UNO kompaktabb változata, mely ugyanazt az ATmega328 mikrovezérlőt használja, így teljesítményében és képességeiben nagyon hasonló az UNO-hoz. A mikrovezérlő tartalmaz:

- Processzort
- Memóriát
- Perifériákat:
  - Időzítő áramkörök
  - Analóg és digitális be- és kimenetek
  - Kommunikációs interfészek (UART, SPI, I2C) és egyéb funkciók

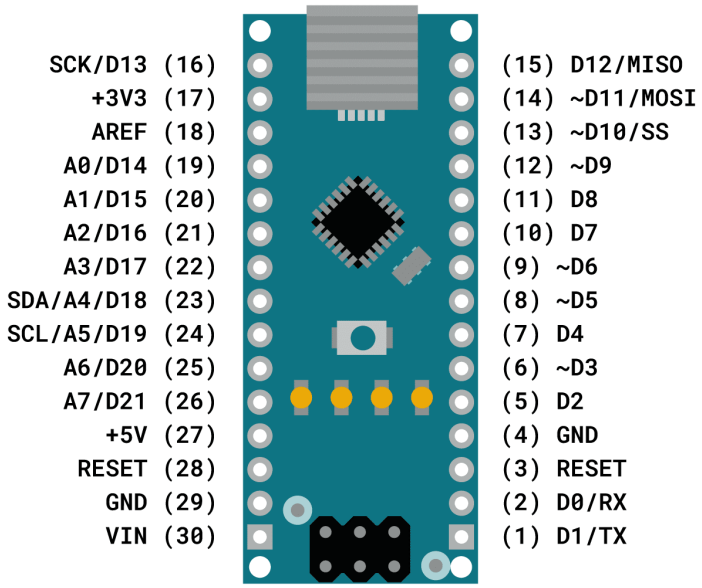


2.2. ábra. ATmega328P mikrovezérlő

Ezek segítségével tudunk szenzorjeleket mérni, nyomógombok vagy más beviteli eszközök állapotát beolvasni. A Nano áramköri lap szerepe az, hogy a mikrovezérlő lábait kivezesse. Így kényelmesebben és egyszerűbben rá tudjuk kötni a különböző eszközöket, amiket vezérelni szeretnénk, vagy értékeket beolvasni róluk.

A mikrokontrollereken általában nem fut operációs rendszer, ezért minden erőforrást a feladatra összpontosít és egy garantált maximális idő alatt képes végre hajtani a feladatokat.

Ahhoz, hogy külső eszközöket és áramköröket rátudjunk csatlakoztatni, ismernünk kell a Nano lábkiosztását, amikre a kódból tudunk hivatkozni és vezérelni őket, áramköri lapon fel van tüntetve, hogy melyik lábat melyik számmal érjük el. A Nano-n 14 digitális ki- és bemenet található (D0–D13), valamint 8 analóg bemenet (A0–A7), melyek közül az A0–A5 lábak digitális bemenetként és kimenetként is használhatók. A Nano-n található továbbá fix 3.3,V és 5,V-os kimeneti feszültségű láb is. Ez a lábkiosztás látható a 2.3. számű ábrán.



2.3. ábra. Arduino Nano lábkiosztása

2.1.3. Miért választottam az Arduino Nano-t?

Szakdolgozatom során az Arduino Nano-t választottam, mivel több olyan előnye van, amelyek különösen fontosak a projektem szempontjából:

- **1. Kompakt méret.** A Nano mérete kisebb, mint az UNO-é, így ideális választás volt, mivel helyszűkében voltam.
- **2. Egyszerű használat és támogatottság.** Az Arduino Nano széles körben elterjedt mikrokontroller, amelyhez könnyen elérhetőek könyvtárak és példakódok, ezzel egyszerűsítve a fejlesztést.
- **3. Megfelelő teljesítmény és elegendő számú I/O port.** A Nano szintén ATmega328 mikrovezérlőt használ, amely tökéletesen megfelel az ADNS-9800 szenzor kezelésére, a szenzor mozgatására, valamint az adatok továbbítására.
- **4. Megbízhatóság és stabilitás.** A stabil működés létfontosságú, mivel a projektem dokumentumok folyamatos szkennelését, adatgyűjtést és adatátvitelt igényel hosszabb ideig.
- **5. Költséghatékonyság.** Az Arduino Nano az egyik legkedvezőbb árú fejlesztőeszköz, amely tökéletesen kielégíti a projekt követelményeit.
- **6. Egyszerű programozhatóság és csatlakoztatás.** Az Arduino Nano könnyen programozható az Arduino IDE használatával, programokat egyszerűen USB-n keresztül tölthetünk fel rá.

2.1.4. Az Arduino alkalmazási területei

- **Kezdő projektekhez.** Az Arduino Board-ok tökéletesek a kezdők számára akik az elektronikát és az informatikát szeretnék ötvözni. A fejlesztő környezetet egyszerű kezelni, valamint a könyvtárak és a példa kódok is nagyon sokat segítenek azoknak az embereknek akik elkezdenek érdeklődni az ilyen dolgok iránt.
- **Oktatási platform.** Könnyen kezelhetősége miatt szokták alkalmazni, hogy ezekkel az eszközökkel tanítsák meg az elektronika és az informatika működését.
- **Robotikában.** A nagy vállalatok vezetői is felfedezték ezt a technológiát, és gyakran Arduino Board-okat használnak a robotok megvalósításához és vezérléséhez.
- **Zene és művészet.** Az Arduino Board-okat szokták egyszerű hangszerek létrehozására is alkalmazni, vagy már meglévő hangszerekbe beépíteni elektronikus alkatrészként.
- **IoT - Internet of Things.** Legtöbb esetben okos otthon rendszerekbe szokták beépíteni, mert sok szenzort rá tudunk csatlakoztatni amikkel könnyen vezérelhetjük a saját otthonunkat.
- **Viselhető eszközök. (Wearables)** Kompakt méretük miatt könnyen beépíthetőek ruhákba, akár ékszerekbe vagy más hordozható eszközökbe. Amelyekkel mozgásokra reagálhatunk mérhetünk testhőmérsékletet és még sok mást.

2.2. Visual Studio

A projekt szoftveres részét Visual Studio[6] fejlesztői környezettel készítettem el, mert ez áll hozzám a legközelebb. Itt valósítottam meg az interpolációt a kép minőségvesztés nélküli növelését. Valamint a program irányító felületét is itt programoztam le.

2.3. Github

A verziókövetéshez a Githubot[7] használtam azon belül is a Github Desktop-ot, mellyel könnyen tudtam több platformon is dolgozni a projekten, valamint ha valamit elrontottam könnyen vissza tudtam állítani a projektet egy korábbi verzióra.

3. fejezet

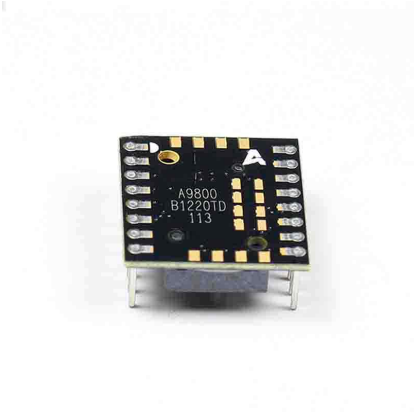
Hardveres megvalósítás

3.1. ADNS-9800 szenzor

Az általam készített szkennernek a lelke a kicsi és olcsó ADNS-9800[8] lézeres optikai érzékelő. Az eredeti felhasználási módja ennek az eszköznek az volt, hogy egerek mozgását kövessék. Ennek a szenzornak az a különlegessége, és azért volt megfelelő számomra, mivel úgy követi a mozgást, hogy folyamatosan képeket készít. Ezáltal fel tudtam használni a projektembe, mint egy kis kamerát, valamint gyors képfeldolgozási képességgel rendelkezik, ezért alkalmas dokumentumok beolvasására.

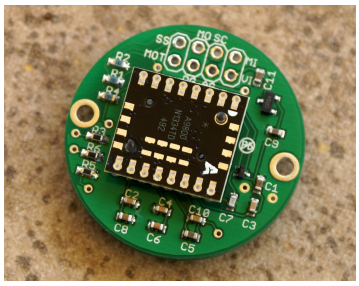
3.1.1. Működése

A szenzor működésének alapja a LaserStream technológia, amely során egy beépített lézer folyamatosan megvilágítja a felületet. A lencse ezt követően készít egy képet és a beépített digitális jelfeldolgozó (DSP) valós időben matematikai úton kiszámítja a felület elmozdulásának irányát és mértékét. Ez a megoldás lehetővé teszi a nagyon pontos mozgásérzékelést, hiszen a szenzor akár 8200 cpi (counts per inch) felbontással is képes mérni a mozgást.



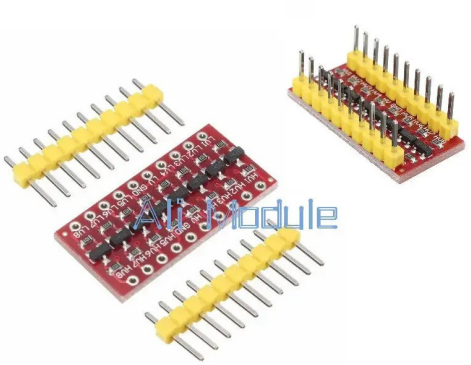
3.1. ábra. ADNS-9800 szenzor

Az általam választott szenzor már nyáklappal volt ellátva, ami nagymértékben egyszerűsítette a beépítési folyamatot.



3.2. ábra. ADNS-9800 szenzor nyák lapra szerelve

Csak egy problémám volt, hogy a nyáklappal szerelt verzió 3.3 V-os logikai szinttel működött viszont az Arduino Nano 5 V-os jeleket használ. Így be kellett szereznem egy két irányú logikai szintillesztő[12] modult ami átalakította a 3,3 V-os jelet 5 V-ossá. Ehhez a projekthez egy 8 csatornás logikai szintillesztő modult használtam.



3.3. ábra. Két irányú logikai szintillesztő

3.1.2. Adatok beolvasása

Az adatok beolvasása közben a szenzor egy sorokból és egy oszlopokból álló képkockát készít, amely egy 30 x 30-as pixel rács. Ezáltal egy dokumentum szkennelése lehet akár több ezer képkocka is. A működése a következő lépésekből áll:

1. **Feltöltjük a firmware-t a szenzorra.** Erre azért van szükség, mert enélkül a szenzor csak egy hardveres érzékelő, amely a lézerrel megvilágított felületről képeket készít. Ahhoz, hogy ezt az információt feldolgozza és továbbítsa szüksége van egy vezérlőprogramra, azaz firmware-re. Ezt a firmware-t minden bekapcsolás után fel kell tölteni, mert a szenzornak nincs állandó memóriája. A regiszterek

megfelelő beállítása után a firmwaret soronként továbbítja az SPI buszon keresztül a szenzornak. Ebben a metódusban fontos az időzítés, hogy a szenzor helyesen értelmezze az adatokat.

```
1   void adns_upload_firmware(){
2     adns_write_reg(REG_Configuration_IV, 0x02);
3     adns_write_reg(REG_SROM_Enable, 0xd);
4     delay(10);
5     adns_write_reg(REG_SROM_Enable, 0x8);
6     adns_com_begin();
7     SPI.transfer(REG_SROM_Load_Burst | 0x80);
8     delayMicroseconds(15);
9     unsigned char c;
10    for(int i = 0; i < firmware_length; i++){
11      c = (unsigned char)pgm_read_byte(firmware_data + i);
12      SPI.transfer(c);
13      delayMicroseconds(15);
14    }
15    adns_com_end();
16  }
```

2. Majd a **sendFrame()** metódust meghívva készítünk egy képet a szenzorral. Amit egy nagy sebességű, full–duplex kommunikációs protokollal (Serial Peripheral Interface – SPI) kiolvasunk és eltároljuk, ahol a kommunikáció egy órajel vezérelt mechanizmussal működik.

```
1   void sendFrame() {
2     adns_write_reg(REG_Power_Up_Reset, 0x5A);
3     delay(50);
4     adns_write_reg(REG_LASER_CTRL0, 0x00);
5     adns_write_reg(REG_Frame_Capture,0x93);
6     delayMicroseconds(120);
7     adns_write_reg(REG_Frame_Capture,0xc5);
8     delayMicroseconds(120);
9     delay(1);
10    adns_com_begin();
11    delayMicroseconds(100);
12    byte readys = 0;
13    while(readys == 0){
14      SPI.transfer(REG_Motion & 0x7f);
15      delayMicroseconds(100);
16      readys = SPI.transfer(0);
17      readys = readys & 1;
18      delayMicroseconds(20);
19    }
20    SPI.transfer(REG_Pixel_Burst & 0x7f);
21    delayMicroseconds(100);
```

```
22        Serial.print("FRAME:");
23        for (int i = 0;i<900;i++){
24          byte pixelValue = SPI.transfer(0);
25          Serial.print(pixelValue);
26          Serial.print("␣");
27        }
28        delayMicroseconds(15);
29        adns_com_end();
30        delayMicroseconds(5);
31        Serial.println();
32      }
```

A metódus működése:

1. **Szenzornak kiadjuk, hogy készítsen egy képet.** Ekkor bele írjuk a megfelelő értéket a REG_Frame_Capture regiszterbe. A szenzor ekkor egy 30 x 30 pixeles képkockát olvas be.
2. **Várunk amíg a szenzor elkészíti a képet.** Az Arduino egy ciklusban folyamatosan ellenőrzi a szenzor állapotát ezért amíg a szenzor nem jelez, nem olvassuk ki az értéket.
3. **A szenzor továbbítja a képkockát.** A Pixel Burst regiszteren keresztül történik a képkocka eljuttatása a szenzortól az Arduino-ig SPI kapcsolattal. Az SPI kapcsolat négy vezetéken keresztül történik:
   - **MOSI** (Master Out, Slave In): Az adatok a mikrokontroller felől a szenzor felé mennek.
   - **MISO** (Master In, Slave Out): Az adatok a szenzor felől mennek az Arduino felé.
   - **SCLK** (Serial Clock): Az órajel, amelyet az Arduino generál.
   - **CS** (Chip Select): Az adott szenzort választja ki a kommunikációhoz.
4. **Az Arduino továbbítja az adatokat a számítógépre.** Az adatok soros kommunikáción keresztül kerülnek a számítógépre ahol az feldolgozza ezeket. Minden képkocka ”FRAME:” előtaggal kezdődik, amit a szenzor által olvasott fényintenzitás értékek követnek. Egy példa frame sor:

   `FRAME: 124 128 130 122 120 119 118 117 116 ...`

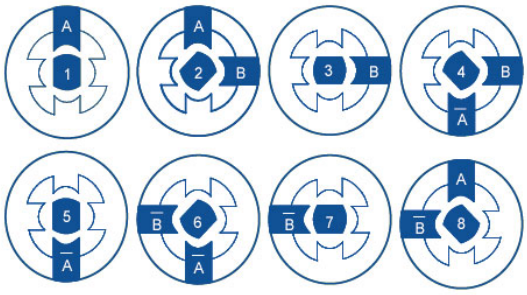
3.1.3. Szenzor mozgása

A szenzort két motor mozgatja az egyik vízszintesen a másik pedig függőlegesen. A projekthez egy bipoláris valamint egy unipoláris léptetőmotort[14] használtam. Mivel egy meglévő szkennerbe építettem bele az egész elektronikát, ezért a függőleges tengely

mozgatására szolgáló bipoláris léptetőmotor már adott volt. Viszont a szenzor nagyon kis képet készít ezért finom mozgásokra volt szükségem, ebben segítséget nyújtottak a már előre beépített fogaskerekek, amik áttétként funkcionálnak. Ezáltal a motor egy lépése jelentősen kisebb elmozdulás volt fizikailag, így jól tudtam pozicionálni. A motort egy L298N[15] motorvezérlőn keresztül irányítottam.

3.4. ábra. MITSUMI M42SP-4NP bipoláris motor a függőleges mozgáshoz

Mindkét motornál half-step vezérlési módot használtam, hogy még finomabb és pontosabb mozgást tudjak végezni velük. Ennek az a lényege, hogy a motor kétszer annyi lépésben teszi meg ugyanazt a fordulatot. Ez úgy lehetséges, hogy a négy tekercs közül először egy tekercs kap áramot, majd az első és a második egyszerre, majd csak a második, így a lépés szám a duplájára nő a szögelfordulás pedig a felére csökken.

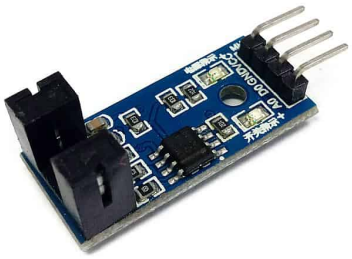


3.5. ábra. Half Step vezérlés működése

A vízszintes mozgáshoz beépítettem egy unipoláris léptetőmotort, amely egy ULN2003 motorvezérlő modullal működik, amit nagyon gyakran használnak ilyen kis apró finom mozgásokhoz, amire nekem is szükségem volt. Mivel ez a motor úgy van felépítve, hogy alapból áttétek vannak benne ezért azzal itt sem kellett foglalkoznom. A motorok egy egy bordás szíjas sínen mozognak, ezáltal a mozgás stabil lesz és csúszásmentes. Mindkettő végén van egy végállás kapcsoló, ha a vízszintes sor a végére ér ezt aktiválja és lentebb megy egy sort, valamint vissza megy a sor elejére a szenzor. Ekkor küld az Arduino egy ”NEW_ROW” kulcsszót a Serial porton a számítógép felé, ezzel jelezve a feldolgozó szoftvernek, hogy új sor következik az olvasásban. Ha pedig a függőleges végálláskapcsoló válik aktívvá akkor pedig az ”END” kulcsszót küldi el a számítógép felé, jelezve, hogy vége a dokumentumnak és vissza tér a kezdeti állásba.

3.6. ábra. Unipoláris léptetőmotor a vízszintes mozgáshoz

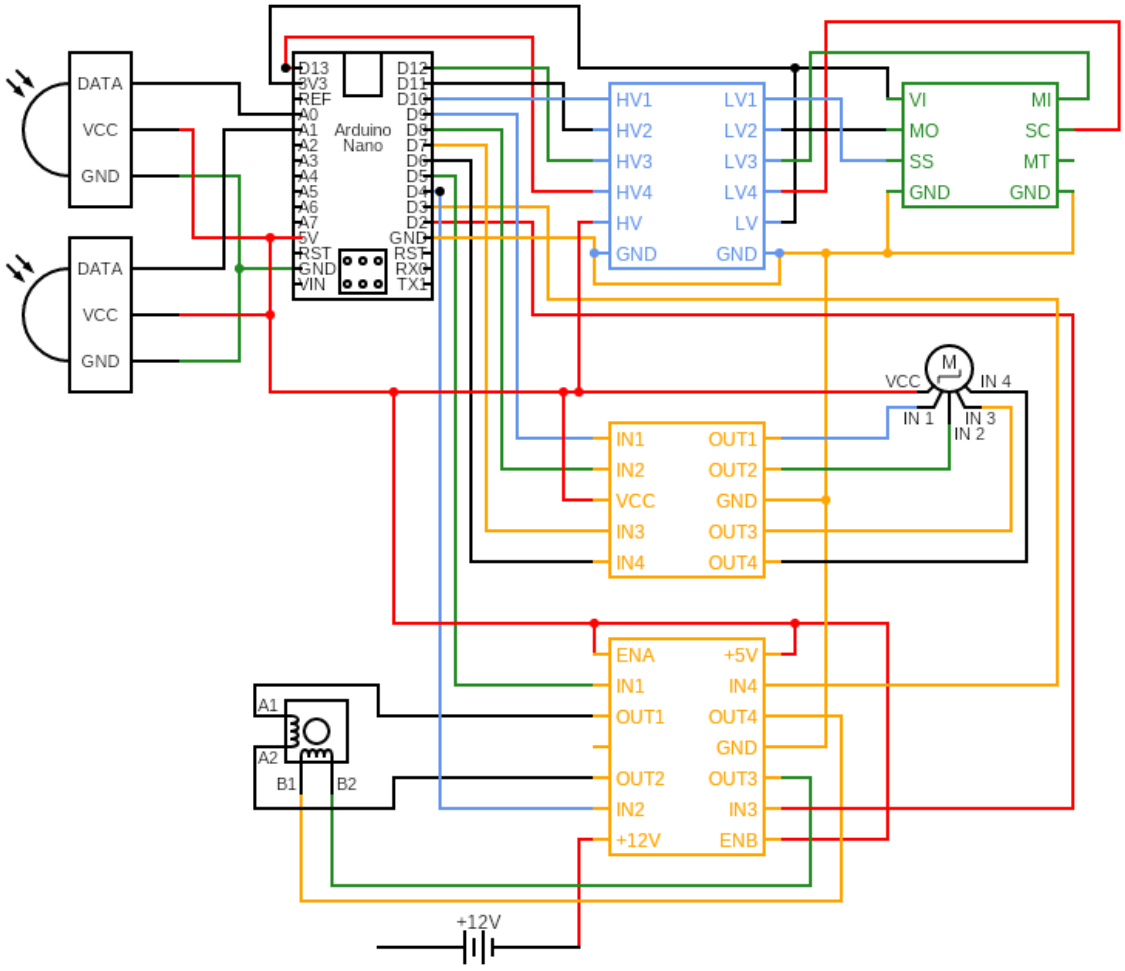
A szenzor mozgatásához még szükségem volt két darab Infravörös szenzorra amik a segítségével a szenzor kezdőállását tudtam meghatározni. Ez egy egyszerű kis szenzor ami infravörös fényt bocsát ki az egyik oldalt és a másik oldalt van egy infravörös vevő egység és ha megszakad a jel, például valamit közé helyezünk, akkor ad magas jelet a szenzor.



3.7. ábra. Infravörös kapu

3.2. Hardveres bekötés

A bekötési rajzon narancssárgával színezett elemek a motorvezérlők, pontosan az L298N és az ULN2003AN. A kék színű elem a szintillesztő és a zöld pedig az ADNS-9800 szenzor. Valamint van még két motor, két infravörös érzékelő és az Arduino Nano.



3.8. ábra. Hardveres bekötés

4. fejezet

Szoftveres megvalósítás

4.1. Az alap változók

A feldolgozó szoftver Serial porton kapja meg az adatokat az Arduino-tól. Van két darab konstans értékem az egyik a beérkező képkocka szélessége a másik pedig a magassága, mivel a beérkező képkocka 30 x 30 pixel ezért ez a két érték 30. Van egy buffer-em ahol a Serial porton beérkező adatokat tárolom, erre azért van szükség mert nem biztos, hogy egy sor át tud jönni egyszerre. Valamint van egy Bicubic osztály példányom amivel a Bikubikus interpolációt végzem. Végül van egy *row* és egy *column* változóm ami annak segítségére szolgál, hogy összerakjam a képkockákat egy nagy dokumentummá. Ezt a nagy dokumentumot tartalmazza a *listGrid* nevű változó, ami egy lista amiben listák vannak amiben egész típusú tömbök vannak.

```
1  private SerialPort serialPort;
2  private const int FrameWidth = 30;
3  private const int FrameHeight = 30;
4  private StringBuilder dataBuffer = new StringBuilder();
5  private const int scale = 2;
6  private Bicubic resizer = new Bicubic(FrameWidth, FrameHeight, scale);
7
8  int row = 0;
9  int column = 0;
10
11 private List<List<int[]>> listGrid = new List<List<int[]>>();
```

4.2. Inicializáló metódusok

Van egy soros portot beállító metódus. Aminek segítségével megnyitom a soros portot és beállítom a megfelelő BaudRate-t. Jelen esetben 115200 bit/másodperc. Hozzárendelem a serialPort változó DataReceived eseményéhez a saját metódusomat, ami azt

jelenti, hogy amikor érkezik adat a soros porton akkor meghívja azt a metódust. Ezután megnyitom a soros portot, hogy tudjam olvasni az adatokat róla, és írok egy ”reset” kulcsszót az Arduino felé ezen a porton keresztül, hogy alaphelyzetbe tegyem a szenzort.

```
1  private void InitializeSerialPort()
2  {
3    serialPort = new SerialPort("COM3", 115200);
4    serialPort.DataReceived += DataReceived;
5    serialPort.Open();
6    serialPort.Write("reset");
7  }
```

4.3. Bejövő adatot feldolgozó metódus

A bejövő adatokat egy külön metódus kezeli. A metódus elején megnézem, hogy a beérkezett data üres-e, ha igen akkor jelzem a hibát és kilépek. Viszont ha van adat akkor azt hozzáadom a dataBufferhez ami tárolja az eddig beérkezett adatokat. Átalakítom a jelenlegi dataBuffert String-é, hogy könnyebb legyen vele dolgozni. Végül deklarálok egy változót amiben az aktuális sor végén lévő új sor karakter pozícióját tárolom.

```
1  private void ProcessSerialData(string data)
2  {
3    if (string.IsNullOrEmpty(data))
4    {
5      MessageBox.Show("A beérkezett adat üres", "Hiba",
           MessageBoxButtons.OK, MessageBoxIcon.Error);
6      return;
7    }
8    dataBuffer.Append(data);
9    string bufferString = dataBuffer.ToString();
10   int newlineIndex;
```

Az adatok feldolgozását egy while ciklus végzi ami addig megy amíg a bufferstring tartalmaz új sor jelet, eközben a ciklus minden iterációjában kimentem a jelenlegi indexét az új sor jelnek. Ezen belül kinyerem az első teljes sort a bufferből és levágom a felesleges szóközöket. Eltávolítom ezt a teljes sort a dataBufferből. Frissítem a bufferstring változót is, hogy csak a még fel nem dolgozott rész maradjon. Mivel WinForms-al dolgozok ezért szükségem van az InvokeRequired feltételre, amivel megvizsgálom, hogy az adott hívás másik szálról történik-e, ha a hívás a főszálon történik akkor csak egyszerűen hozzáadom a listboxhoz a beérkezett adatot, ha nem akkor az Invoke metódussal hozzá adom a főszálon.

```
while ((newlineIndex=bufferString.IndexOf("\n"))!= -1)
{
  string fullLine = bufferString.Substring(0, newlineIndex).Trim();
  dataBuffer.Remove(0, newlineIndex + 1);
  bufferString = bufferString.Substring(newlineIndex + 1);
  if (this.InvokeRequired)
  {
    this.Invoke(new Action(() => listBox1.Items.Add("Beérkező sor:
        "+fullLine)));
  }
  else
  {
    listBox1.Items.Add("Beérkező sor: " + fullLine);
  }
```

Ha a jelenlegi beérkezett sor úgy kezdődik, hogy ”FRAME” akkor levágom az elejéről a ”FRAME” szót és a maradék adatot kimentem egy szöveg típusú tömbbe. Leellenőrzöm, hogy ennek a tömbnek megegyezik-e a hossza az általam előírttal. Tehát az alap *FrameWidth* és *FrameHeight* szorzatával ami 30 x 30, ha nem akkor hibát dobok. Az interpolációs osztály példányával meghívom erre a tömbre az interpoláló metódust, közben az összes elemet átalakítom egész típusú számmá és kimentem egy *frameData* nevű egész típusú tömbbe. Ezután megjelenítem a már interpolált képet az alkalmazásban egy pictureBox segítségével. Valamint megvizsgálom, hogy a *row* változónak az értéke nagyobb-e, mint a *listGrid* hossza, amivel itt a sor kezelést oldom meg, ha igen akkor azt jelenti, hogy már egy új sorban vagyunk már ezért kell hozzá adnunk egy új tömbökből álló listát. Végül hozzáadjuk a jelenlegi sorhoz a tömböt.

```
  if (fullLine.StartsWith("FRAME:"))
  {
    string[] pixels = fullLine.Substring(6).Split(new[] { ' ' });
    if (pixels.Length == FrameWidth * FrameHeight)
    {
      int[] frameData = resizer.BicubicResize(Array.ConvertAll(pixels,
          int.Parse));
      DisplayFrame(frameData, FrameHeight * scale, FrameWidth * scale);
      if (row >= listGrid.Count)
      {
        listGrid.Add(new List<int[]>());
      }
      listGrid[row].Add(frameData);
    }
    else
    {
      if (this.InvokeRequired)
      {
```

```
18            this.Invoke(new Action(() => listBox1.Items.Add("Hibás FRAME
                  sor hossz: "+pixels.Length)));
19          }
20          else
21          {
22            listBox1.Items.Add("Hibás FRAME sor hossz: "+pixels.Length);
23          }
24        }
25      }
```

Ha a ”NEW_ROW” kulcsszóval kezdődik a jelenlegi adat akkor csak simán növelem a sor indexet. Mivel ekkor jelet kapok az Arduino felől, hogy a szkennelés a következő sorban folytatódik tovább. Ha az ”END” kulcsszó van a jelenlegi adatban akkor pedig befejeződött a szkennelés tehát leállítom a szenzort valamint elmentem a dokumentumot.

```
1     if (fullLine.StartsWith("NEW_ROW"))
2     {
3       row++;
4     }
5     if (fullLine.StartsWith("END"))
6     {
7       serialPort.WriteLine("stop");
8       SaveDoc();
9     }
10  }
11 }
```

4.4. Kiegészítő metódusok

Az egyik legfontosabb kiegészítő metódus a **matrixToBitmap** amiben egy képkockákból álló két dimenziós listát alakítok át egy nagy szürkeárnyalatos Bitmap[25] képpé. A célja a metódusnak, hogy a szenzor által beolvasott és interpolált képkockákat egy nagy képpé összeállítsam. A listGrid szerkezet egy két dimenziós lista amely úgy értelmezhető, hogy az első dimenzió (külső lista) a sorokat jelenti, a második dimenzió (belső lista) pedig az oszlopokat jelképezi. Végül minden int[] elem egy képkockát tartalmaz ahol a fényintenzitások sorban vannak felsorolva.

Példa: listGrid[1][2] jelenti a második sor harmadik képkockáját.

Visszatérve a metódusra az egész metódus kezdődik egy ellenőrzéssel ahol megnézzük, hogy a *listGrid* üres-e, ha igen akkor hibát dobunk.

```
1     private Bitmap matrixToBitmap()
2     {
```

```
3     if (listGrid.Count == 0 || listGrid[0].Count == 0 || listGrid[0][0]
          == null)
4     {
5       MessageBox.Show("A mátrix üres, nem lehet képet generálni.",
            "Hiba", MessageBoxButtons.OK, MessageBoxIcon.Error);
6       return null;
7     }
```

Meghatározzuk a képkockák szélességét és magasságát úgy, hogy megszorozzuk a nagyítási változóval az eredeti kép méreteit. Ezután meghatározzuk a teljes összeállított képek méretét.

```
1     int frameWidth = FrameWidth * scale;
2     int frameHeight = FrameHeight * scale;
3     int width = listGrid[0].Count * frameWidth;
4     int height = listGrid.Count * frameHeight;
```

Az általános **SetPixel** metódus helyett én inkább a **LockBit** megoldást választottam, igaz, hogy az nem olyan egyszerű de a nagy képeknél viszont sokkal gyorsabb teljesítményt tudok elérni vele. Ennek az oka, hogy a memóriába írok közvetlenül a GDI hívások helyett.

Létrehozok egy Bitmap példányt 24 bites RGB formátumban, ami azt jelenti, hogy minden képpont 3 bájtot foglal el. A Rectangle osztállyal meghatározom a Bitmap teljes tartományát. A Bitmap **LockBits** metódusának átadom az előzőleg meghatározott rectangle-t, az ImageLockMode-ot amivel úgy zárom le a Bitmap-et, hogy csak írni tudjam,olvasni ne. Valamint átadom neki a Bitmap formátumát is, és vissza kapok egy BitmapData értéket ami tartalmazza ennek a zárolásnak az információit. Ebből kinyerve megkapom a stride-ot ami egy fontos információ, mivel ez mutatja meg, hogy a Bitmap memóriájában egy-egy sor hány bite hosszú. Ez nem feltétlenül egyezik meg a képpontok számának és a fényintenzitásnak megfelelő mérettel, mert a rendszer minden sort 4 bite-ra igazít a gyorsabb memóriakezelés miatt. Tehát ezzel fogom kiszámolni, hogy egy adott képpont hol helyezkedik el.

```
1     Bitmap bmp = new Bitmap(width, height, PixelFormat.Format24bppRgb);
2     Rectangle rect = new Rectangle(0, 0, width, height);
3     BitmapData bmpData = bmp.LockBits(rect, ImageLockMode.WriteOnly,
          bmp.PixelFormat);
4     int stride = bmpData.Stride;
```

Végig megyek a *listGrid*-en ami ugye a teljes nagy kép és ezen belül végig megyek a képkockákon is és minden egyes pixelt közvetlenül átmásolok egy byte tömbbe. Itt alkalmaztam egy kis kontraszt javítást is mert a szenzor képe nem a legszebb de így sokkal jobban néz ki. A stride segítségével kiszámolom, hogy az adott pozíció hányadik byte-nál kezdődik a memóriában.

```
    byte[] bytes = new byte[stride * height];
    for (int row = 0; row < listGrid.Count; row++)
    {
      for (int col = 0; col < listGrid[row].Count; col++)
      {
        int[] frameData = listGrid[row][col];

        int startX = col * frameWidth;
        int startY = row * frameHeight;

        for (int y = 0; y < frameHeight; y++)
        {
          for (int x = 0; x < frameWidth; x++)
          {
            int pixelValue = (int)(frameData[y * frameWidth + x] * 1.15);

            int globalX = startX + x;
            int globalY = startY + y;

            int offset = globalY * stride + globalX * 3;

            bytes[offset] = (byte)pixelValue;
            bytes[offset + 1] = (byte)pixelValue;
            bytes[offset + 2] = (byte)pixelValue;
          }
        }
      }
    }
```

Miután végig mentem a teljes képen és megtelt a tömb, visszamásolom a tömb adatait a Bitmap memóriába. Ezt egyetlen sorral meg tudtam tenni a **Marshal.Copy**[30] metódussal, át adtam neki a a byte tömböt, a másolás kezdőindexét, a célmemóriát amit a bitmapData-ból tudok lekérdezni a Scan0 metódussal, valamint a másolandó bitek számát.

```
    Marshal.Copy(bytes, 0, bmpData.Scan0, stride * height);
    bmp.UnlockBits(bmpData);
    return bmp;
   }
```

A következő kiegészítő metódus amit használtam az a **DisplayFrame** metódus aminek a segítségével megtudtam jeleníteni egy PictureBoxba, hogy mit is lát a szenzor. Ennek annyi a lényege, hogy megkapja paraméterként a fényintenzitásokat tartalmazó tömböt valamint a képmérethez szükséges adatokat.

A képkocka megjelenítését hasonlóan oldottam meg, mint a korábban bemutatott

matrixToBitmap metódusban. LockBits segítségével.

```
private void DisplayFrame(int[] frameData, int height,int width)
{
  Bitmap frameBitmap = new Bitmap(width, height,
      PixelFormat.Format24bppRgb);
  Rectangle rect = new Rectangle(0, 0, width, height);
  BitmapData bmpData = frameBitmap.LockBits(rect,
      ImageLockMode.WriteOnly, PixelFormat.Format24bppRgb);

  int stride = bmpData.Stride;
  byte[] bytes = new byte[stride * height];

  for (int y = 0; y < height; y++)
  {
    for (int x = 0; x < width; x++)
    {
      int pixelValue = (int)(frameData[y * width + x] * 1.15);
      int offset = y * stride + x * 3;

      bytes[offset] = (byte)pixelValue;
      bytes[offset + 1] = (byte)pixelValue;
      bytes[offset + 2] = (byte)pixelValue;
    }
  }

  Marshal.Copy(bytes, 0, bmpData.Scan0, bytes.Length);
  frameBitmap.UnlockBits(bmpData);

  if (this.InvokeRequired)
  {
    this.Invoke(new Action(() =>
    {
      pictureBox.Image = frameBitmap;
      pictureBox.Invalidate();
      pictureBox.Update();
    }));
  }
  else
  {
    pictureBox.Image = frameBitmap;
    pictureBox.Invalidate();
    pictureBox.Update();
  }
}
```

A dokumentum mentésére szolgáló metódus a **SaveDoc** nevezetű metódus, ami azt a célt szolgálja, hogy a dokumentumot képként vagy pdf formátumba lementi. A me-

tódus létrehoz egy külön mappát a dokumentumoknak SavedFrames néven. Ha a felhasználó bepipálta a *pdf format checkbox*-ot akkor a program pdf formátumba menti le a beszkennelt dokumentumot ellenkező esetben pedig png-ben. A pdf mentéshez az Aspose.Words[31] nuget csomagot használtam.

```
1  private void SaveDoc()
2  {
3    string folderPath = Path.Combine(Application.StartupPath,
       "SavedFrames");
4    if (!Directory.Exists(folderPath))
5    {
6      Directory.CreateDirectory(folderPath);
7    }
8    string filePath = Path.Combine(folderPath);
9    try
10   {
11     if (pdfCheckBox.Checked)
12     {
13       var doc = new Document();
14       var builder = new DocumentBuilder(doc);
15       builder.InsertImage(matrixToBitmap());
16       doc.Save(filePath+"\\Document.pdf");
17       if (this.InvokeRequired)
18       {
19         this.Invoke(new Action(() => MessageBox.Show("A pdf mentése
             sikeresen megtörtént.", "Siker", MessageBoxButtons.OK,
             MessageBoxIcon.Information)));
20       }
21       else
22       {
23         MessageBox.Show("A pdf mentése sikeresen megtörtént.",
             "Siker", MessageBoxButtons.OK, MessageBoxIcon.Information);
24       }
25     }
26     else
27     {
28       matrixToBitmap().Save(filePath + "\\frame.png",
           System.Drawing.Imaging.ImageFormat.Png);
29       if (this.InvokeRequired)
30       {
31         this.Invoke(new Action(() => MessageBox.Show("A kép mentése
             sikeresen megtörtént.", "Siker", MessageBoxButtons.OK,
             MessageBoxIcon.Information)));
32       }
33       else
34       {
35         MessageBox.Show("A kép mentése sikeresen megtörtént.",
```

```
                "Siker", MessageBoxButtons.OK, MessageBoxIcon.Information);
36          }
37
38        }
39      }
40      catch (Exception ex)
41      {
42        MessageBox.Show($"Hiba a képfájl mentésekor: {ex.Message}",
              "Hiba", MessageBoxButtons.OK, MessageBoxIcon.Error);
43      }
44    }
```

4.5. Bikubikus Interpoláció

A bikubikus interpoláció egy képfeldolgozási technika, amelyet elsősorban képek méretének változtatására használnak. A lényege, hogy az új pixelek színét vagy fényintenzitását nem a közvetlen szomszédjaiból számolja ki, hanem a tágabb környezetében található több pixelből. Ez a módszer egy 4 x 4-es pixelblokkot vesz figyelembe az új pixel színéhez, ez azért jobb, mert több értéket vesz figyelembe így pontosabb színt kapunk az egyes pixelekhez. A módszer alapja egy súlyfüggvény amely a közelebbi pixelekhez nagyobb súlyt rendel, míg a távolabbiak kevésbé lesznek figyelembe véve az új érték kiszámolásakor. Ez a súlyozás biztosítja, hogy a kép simább legyen és részletgazdagabb. Éppen ezért a bikubikus interpolációt sokan használják olyan rendszerekben ahol fontos a kép minőségének megőrzése, ezért volt tökéletes választás az én számomra is.

Visszatérve a **Bicubic** osztályhoz, van három alap mezőm ahol tárolom a kép szélességét, magasságát, valamint a nagyítási skálát. A konstruktor ezeket az értékeket kapja meg, és eltárolja őket a megfelelő mezőben.

```
1   private int width,height;
2   private int scale;
3
4   public Bicubic(int width, int height, int scale)
5   {
6     this.width = width;
7     this.height = height;
8     this.scale = scale;
9   }
```

A **GetWeight** metódus határozza meg, hogy az egyes környező pixelek milyen mértékben határozzák meg az új pixel értékét. Ez a legfőbb lényege ennek az interpolációnak, hogy nem egyszerűen átlagolom az értékeket hanem a súlyozott átlagukat veszem. Tehát minden környező pixel kap egy súly értéket, ami attól függ milyen messze van az

adott pixel az új még ki nem számított értékű pixeltől. A bemeneti paramétere egy távolság ami az új pixel helyzete és egy szomszédos pixel között van.

```
1
2   private float GetWeight(float x)
3   {
4     x = Math.Abs(x);
5     if (x <= 1) return (1.5f * x * x * x) - (2.5f * x * x) + 1;
6     if (1 < x && x < 2) return (-0.5f * x * x * x) + (2.5f * x * x) - (4
          * x) + 2;
7     return 0;
8   }
```

Matematikai képlettel leírva pedig így néz ki:

$$W(x) = \begin{cases} 1.5|x|^3 - 2.5|x|^2 + 1, & \text{ha } |x| \leq 1 \\ -0.5|x|^3 + 2.5|x|^2 - 4|x| + 2, & \text{ha } 1 < |x| < 2 \\ 0 & \text{ha } |x| \geq 2 \end{cases}$$

Egy kis rövid metódus aminek nagy szerepe van az interpolációban. Ahogy azt korábban említettem ehhez az interpolációhoz az új pixel körüli 16 pixelt veszem figyelembe, de mi van akkor ha a pixel a kép szélén van? Ebben segít a **Reflect** metódus ami megoldja ezt a problémát. A bemeneti paramétere egy kapott index, amit aszerint alakít, hogy az kisebb-e, mint 0 vagy már a képen kívül esne. Ha kisebb, mint 0 akkor vissza adja a –1-szeresét, ha a képen kívül esne az index akkor vissza felé türközi az értéket és azt veszi figyelembe.

```
1   private int Reflect(int i)
2   {
3     if (i < 0) return -i;
4     if (i >= 30) return 2 * 30 - i - 1;
5     return i;
6   }
```

Az egész osztály, valamint a szakdolgozatom lelke pedig a **BicubicInterpolation** metódus ahol a képnagyítás történik a megadott növelési arány szerint, miközben a hiányzó pixeleket kiszámolja bikubikus interpolációval. A metódus elején meghatározom az új mátrixnak a szélességét és magasságát, valamint létrehozom azt a tömböt ahol már az új adatokat tároltam.

```
1   public int[] BicubicInterpolation (int[] frameData)
2   {
3     int newWidth = width * scale;
4     int newHeight = height * scale;
5     int[] interpolatedData = new int[newWidth * newHeight];
```

A metódusnak egy kiegészítő része, hogy a beérkezett tömböt átalakítsam mátrixszá, mert így könnyebb volt meghatározni az adott pixelek körülötti 16 pixelt.

```
int[,] original = new int[width, height];
for (int y = 0; y < height; y++)
{
  for (int x = 0; x < width; x++)
  {
    original[y, x] = frameData[y * width + x];
  }
}
```

A metódusnak ezen részén végig iterálok a felnagyított képen és kiszámolom a kimaradt pixel értékeket. Kiszámolom először, hogy az adott koordináta melyik helyre esik az eredeti képen, ami nem biztos, hogy egész szám lesz. Ezután meghatározom az eredeti koordináták egész részét. Végül ebből és az eredeti koordináta tizedes részéből megkapom az elmozdulást, ami később a súlyozásnak lesz az alapja.

```
for (int y = 0; y < newHeight; y++)
{
  for (int x = 0; x < newWidth; x++)
  {
    float srcY = y / (float)scale;
    float srcX = x / (float)scale;
    int y0 = (int)Math.Floor(srcY);
    int x0 = (int)Math.Floor(srcX);
    float dy = srcY - y0;
    float dx = srcX - x0;
```

Miután meghatároztam, hogy az új pixel melyik pozíció köré esik, a következő dupla for ciklusban kiszámítom az interpolált értékét. Ehhez a pont körül lévő 4 x 4-es pixelblokk minden egyes elemét figyelembe veszem. Mivel a kép szélein előfordulhat, hogy a számított index a kép határain kívülre esik, ezért használom a **Reflect** metódust, így biztosítva azt, hogy az index a képen belülre essen. Minden szomszédos pixelhez kiszámítom a hozzátartozó súly értéket a *dx* és *dy* segítségével. A két irány súlyait összeszorozva kapom meg az adott szomszéd pixel teljes súlyát. A *value* változóba elmentem ezeket a súlyozott értékeket, ami az új pixel fényintenzitását adja meg. Végül elmentem a metódus elején létrehozott tömbbe az új pixel adatokat és azt a tömböt adom vissza a metódus végén.

```
int value = 0;
for (int j = -1; j < 3; j++)
{
  for (int i = -1; i < 3; i++)
  {
    int xIndex = Reflect(x0 + i);
```

```
7               int yIndex = Reflect(y0 + j);
8               float weight = GetWeight(dy - j) * GetWeight(dx - i);
9               value += (int)(original[yIndex, xIndex] * weight);
10            }
11          }
12          interpolatedData[y * newWidth + x] =
               Math.Min(Math.Max((int)value, 0), 255);
13        }
14      }
15      return interpolatedData;
16    }
```

5. fejezet

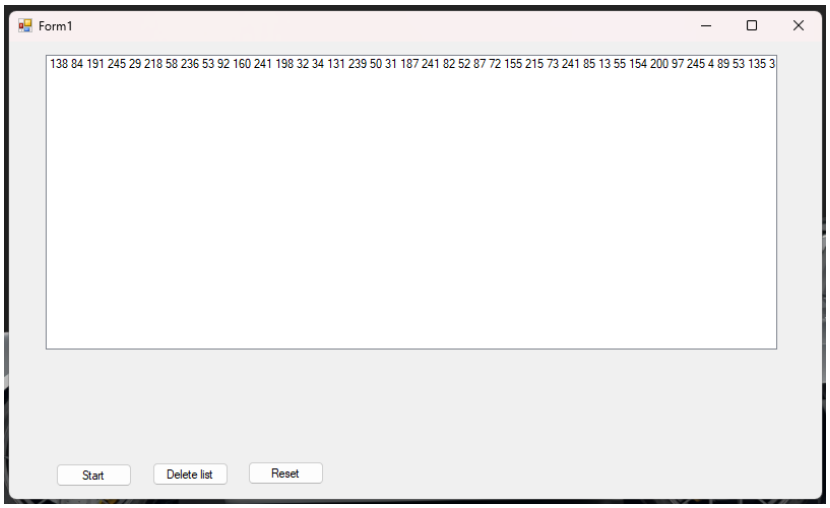
Tesztelés

A programomat az elejétől fogva folyamatosan teszteltem, először a szoftveres részt, majd miután megérkezett hozzám a szenzor elkezdtem azt tesztelni.

A szoftver teszteléséhez úgy fogtam hozzá, hogy egy alap mátrixot beadtam az általam írt algoritmusnak és megnéztem, hogy milyen pontosan számolja ki a pixelek közötti értékeket. Aztán megfogtam egy képet és megnéztem, hogy azzal, hogy viselkedik. Az eredmény az lett, hogy a duplájára növelte a kép méretét ami a célom is volt. Ezután összpontosítottam arra, hogy az adatot megkapjam az Arduino-tól, tehát elkezdtem a soros port kommunikációt tesztelni. Fogtam egy Arduino-t és írtam rá egy példa programot aminek csak annyi dolga volt, hogy elküldjön egy 900 elemű tömböt.

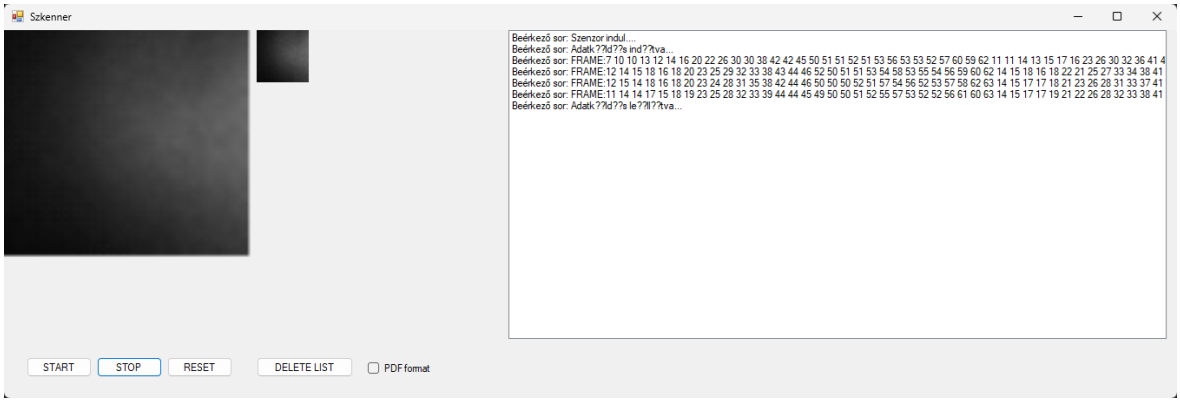
```
1  void setup() {
2    Serial.begin(9600);
3  }
4
5  void loop() {
6    for (int i=0;i<900;i++){
7      Serial.print(int(random(0,255))+"␣");
8    }
9    Serial.println();
10   delay(1000);
11 }
```

A szoftverben ehhez megírtam a szükséges metódust ami arra szolgál, hogy fogadja az adatokat. Ezt követően elkezdtem kialakítani a szoftver felületét, készítettem egy tesztelő Windows Forms alkalmazást ami állt egy listBox-ból valamint egy pár gombból. Egy **Start** gomb amivel elindítottam az adat küldést az Arduino felől, egy **Delete list** amivel törölni tudtam a lista eddigi adatait, valamint egy **Reset** gomb amivel a szenzort tudtam újraindítani.



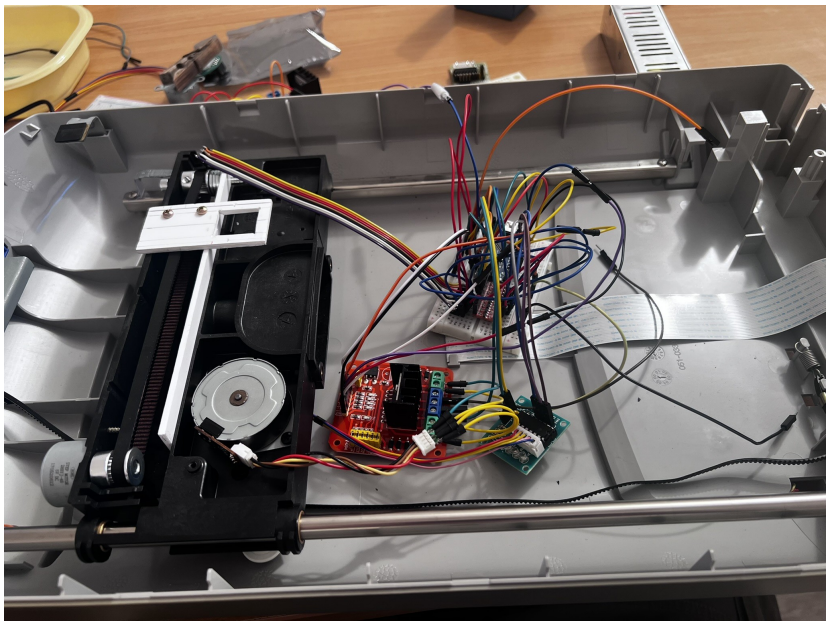
5.1. ábra. Teszt Windows Forms alkalmazás

Aztán következhetett a szenzor tesztelése. Először egy fehér lapon próbálkoztam amire kisbetűmérettel volt rá írva pár betű, viszont valamiért nem kaptam egy szép képet, ezért elkezdtem mozgatni a szenzort és rájöttem, hogy a fókusz távolság nem volt a megfelelő. Így azt kellett finom hangolnom, egyszerű kártya lapokat használtam, hogy megtaláljam a megfelelő távolságot. Miután sikerült beállítanom a kellő távolságot a papírtól, elkezdtem kinyerni a szenzorból a képeket. Megírtam hozzá a teszt alkalmazást amivel megjelenítem, hogy mit lát a szenzor, és nagyon nagy örömömre sikerült képet kinyernem a szenzorból.



5.2. ábra. Továbbfejlesztett teszt alkalmazás

Végezetül a motorok teszteléséhez half-step módot próbáltam alkalmazni, viszont a bipoláris léptető motorral nem működött, ezért ott a Stepper könyvtárat kellett használnom. Közben felmerült egy olyan probléma, hogy az általam használt motorvezérlő elromlott ezért egy másikat kellett beszerelnem. Ezután jöhetett a szkennerbe való beszerelés, amelyben Geda Gábor tanár úr segített nekem. Így született meg az 5.3. számú ábrán látható szkenner.



5.3. ábra. Összeépített szkenner

Ezután a helyére kerültek a fénykapuk is amikkel a kezdőpozícióját tudtam megállapítani a szenzornak. Két kis zászlót készítettünk amik a sínen valamint a szenzornál helyezkednek el, ezáltal ha a zászló be megy az infravörös érzékelők közé, jelez, hogy megérkezett a kezdőállapotba. Végezetül összeállt a végleges szkenner amelyet az 5.4. számú ábrán láthatunk.

5.4. ábra. Végleges szkenner

Összegzés

Összességében a szakdolgozatom elkészítése közben nagyon sok új elektronikai és programozási tapasztalatra tettem szert. A projekt alatt nem csak az Arduino környezetet értettem meg jobban, hanem sok hardveres eszköz működését is jobban megértettem, például a léptetőmotorok vagy maga a szenzor. A projekt során még azt is megtanultam, hogy hogyan lehet a szoftver és hardver működését összehangolni.

Ez a szkenner egy egyszerű fekete-fehér képolvasó, viszont jó alapot biztosít a további fejlesztésekhez, mint például:

- A jelenlegi színtelen szkennert 3 darab LED-el tovább lehetne fejleszteni egy színes szkennerré.
- A mostani változat sajnos csak egy kis részt tud beolvasni, ezért az is egy továbbfejlesztési gondolat lehetne, hogy egy teljes A4-es lapot beolvasson.
- Esetleg az is szóba jöhetne, hogy Wi-Fi-n keresztül telefonra kapnánk meg a beolvasott dokumentumot.

Ábrajegyzék

Irodalomjegyzék

[1] Github link a szakdolgozathoz: https://github.com/Bmate2/SZAKDOGA

[2] Arduino: https://www.arduino.cc/en/Guide/Introduction

[3] What is an Arduino: https://learn.sparkfun.com/tutorials/what-is-an-arduino/all

[4] Arduino UNO: http://dx.doi.org/10.6084/m9.figshare.11971794.v1

[5] Arduino Nano: https://docs.arduino.cc/hardware/nano/

[6] Microsoft Visual Studio: https://visualstudio.microsoft.com/

[7] Github: https://github.com/

[8] ADNS-9800 adatlap:https://datasheet.octopart.com/ADNS-9800-Avago-datasheet-10666463.pdf

[9] ADNS-9800 vásárlás: https://www.tindie.com/products/citizenjoe/adns-9800-motion-sensor/

[10] ADNS-9800 tutorial get travel distance: https://www.instructables.com/Arduino-Tutorial-ADNS-9800-Laser-Mouse-Traveled-Di/

[11] ADNS-9800 5 V mód aktiválása: https://forum.arduino.cc/t/using-avago-adns9800/292172/2

[12] Logikai szintillesztő modul: https://www.elektrobot.hu/termek.php?filename=3607.html&i=3607

[13] Soros kommunikáció: https://docs.arduino.cc/language-reference/en/functions/communication/serial/

[14] Léptető motorok: https://docs.arduino.cc/learn/electronics/stepper-motors/

[15] L298N motorvezérlő: https://techfun.hu/termek/l298n-vezerlomodul-dual-h-bridge-v2-0-motorkerekparokhoz/?gad_source=1

[16] Arduino Stepper könyvtár: https://docs.arduino.cc/libraries/stepper/

[17] Bicubic Interpolation by Computerphile: https://www.youtube.com/watch?v=poY_nGzEEWM

[18] Bicubic Interpolation from wikipedia: https://en.wikipedia.org/wiki/Bicubic_interpolation

[19] Bicubic Interpolation: https://cloudinary.com/glossary/bicubic-interpolation

[20] Catmull-Rom Spline: https://en.wikipedia.org/wiki/Centripetal_Catmull%E2%80%93Rom_spline

[21] Kép nagyítás Bikubik interpolációval: https://medium.com/@amanrao032/image-upscaling-using-bicubic-interpolation-ddb37295df0

[22] Windows Forms: https://learn.microsoft.com/hu-hu/dotnet/visual-basic/developing-apps/windows-forms/

[23] Listbox osztály: https://learn.microsoft.com/en-us/dotnet/api/system.windows.forms.listbox?view=windowsdesktop-9.0

[24] LaTeX Arduino stílus fájl: https://github.com/trihedral/ArduinoLatexListing

[25] Bitmap osztály: https://learn.microsoft.com/en-us/dotnet/api/system.drawing.bitmap?view=windowsdesktop-9.0

[26] Mátrix átalakítása Bitmap képpé: https://swharden.com/csdv/system.drawing/array-to-image/

[27] Bitmap.LockBits használata: URLhttps://learn.microsoft.com/en-us/dotnet/api/system.drawing.bitmap.lockbits?view=windowsdesktop-9.0

[28] Lockbit vs SetPixel: https://www.codeproject.com/Articles/617613/Fast-Pixel-Operations-in-NET-With-and-Without-unsa

[29] Rectangle osztály: URLhttps://learn.microsoft.com/en-us/dotnet/api/system.drawing.rectangle?view=net-9.0&viewFallbackFrom=windowsdesktop-9.0

[30] Marshal Copy metódus: https://learn.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal.copy?view=net-9.0

[31] Aspose.Words nuget csomag a dokumentum mentéshez: https://www.nuget.org/packages/aspose.words

Nyilatkozat

Alulírott *Bodnár Máté*, büntetőjogi felelősségem tudatában kijelentem, hogy az általam benyújtott, *Szkenner megvalósítása egér szenzorral* című szakdolgozat önálló szellemi termékem. Amennyiben mások munkáját felhasználtam, azokra megfelelően hivatkozom, beleértve a nyomtatott és az internetes forrásokat is.

Aláírásommal igazolom, hogy az elektronikusan feltöltött és a papíralapú szakdolgozatom formai és tartalmi szempontból mindenben megegyezik.

Eger, 2025. március 3.

aláírás