Forgalomszámláló (OpenCV) – Projekt-dokumentáció

Bevezetés

A képfeldolgozás tárgy keretében egy egyszerű, de jól bővíthető forgalomszámláló rendszert fogok készíteni, amely egy rögzített kamera videójából járműveket detektál és egy kijelölt vonal átlépése alapján számolja őket. A cél a klasszikus képfeldolgozási eszközök (ROI, szürkeárnyalat, GaussianBlur, MOG2 háttérkivonás, morfológiai műveletek, kontúr-analízis, egyszerű követés) lépésről lépésre történő felépítése és megértése, és ezek segítségével egy stabil, zajtól megtisztított előtér maszk előállítása, amelyből megbízhatóan számolhatók a középpontok és az átlépési események.

Fő részfeladatok:

1. Videó beolvasása képkockánként
2. Útfelület körülhatárolása maszkkal, hogy kizárjuk a nem releváns zavaró tényezőket.
3. Szürkeárnyalatra konvertálás, mert a mozgás detektálására elegendő a fényességkülönbség, a szín csak zaj.
4. Gauss szűréssel csökkentjük a zajt.
5. Elválasztjuk az előteret a háttértől ahol a háttér 0, az előtér 255 az árnyékok pedig 127 értéket vesznek fel, majd eldobjuk az árnyékokat.
6. Morfológiai műveletekkel (erózió, dilatáció) eltűntetjük az előtér apró zajait, és az objektumok szakadozottságát.
7. A tisztított bináris képen kontúrokat keresünk, majd minimális terület és oldalarány alapján kiszórjuk a nem megfelelő jármű jelölteket.
8. Minden jelöltnek számítunk centroidot. A képkockák között a centroidokat a korábbi képkockához egy legközelebbi szomszéd társítással párosítjuk.
9. Definiálunk egy virtuális számláló vonalat.
10. Ha egy jármű jelölt utolsó két pontja a vonal ellentétes oldalára kerül, akkor számoljuk.
11. A számlálást vizualizáljuk.

Kimenetként a program élőben mutatja:

- az eredeti képet,
- a ROI-ra (útszakaszra) korlátozott képet,
- a szűrt, elmosott szürkeárnyalatos képet,
- az előtér maszkot (árnyékmentesítve és morfológiával tisztítva),
- a kontúrokból képzett detekciókat (bbox + centroid)
- valamint az irányonkénti és összesített számlálókat.

Elméleti háttér

Ebben a fejezetben a projekt során használt képfeldolgozó eszközök elméleti alapjait tekintjük át: a ROI-maszkolást, a szürkeárnyalatot, a Gaussian Blur-t, a MOG2 háttérkivonást, árnyékkezelést, a morfológiai műveleteket, valamint a kontúrokból számolt momentumokat/centroidot, a vonalátlépés előjel alapú számítását és a képkockák közti legközelebbi szomszéd párosítás elvét.

Képkocka feldolgozás

A nyers kamerakép tele van olyan részletekkel, amik a feladat szempontjából nem érdekesek (ég, fák, járda), és olyan zajokkal, amik megzavarhatják a detektálást (apró fényességingadozások, textúrák). A célunk az, hogy a képet egyszerűbbé és megbízhatóbbá tegyük, csak azt nézzük, ahol történik valami (ROI), csak azt a jellegét nézzük, ami számít (fényesség), és mindezt stabil formában továbbítsuk a pipeline következő lépéseihez (előtér/háttér szétválasztás, kontúrok).

ROI (Region of Interest)

A ROI a képnek az a tartománya, amelyre a feldolgozást korlátozzuk. A gyakorlatban ezt egy bináris maszk reprezentálja, a vizsgált régió pixelei 1/255, a többi 0. A maszkolás célja, hogy csak a releváns részek maradjanak meg.

A maszkolás bitműveletekkel történik. A maszkot és az eredeti képet logikai AND viszonyba hozzuk, így a maszkban 0-val jelölt területek lenullázódnak (háttér), a 255-ös részek változatlanul megmaradnak (ROI).

Előnye, hogy csökkenti a számítási terhet, és mérsékli a téves detektálást. [1]

Szürkeárnyalat

Speciális feladatoknál (pl. színalapú szegmentáció) a szín fontos, mozgásdetektálásnál általában nem, ezért a szürkeárnyalatra váltás a mi esetünkben nem információ- hanem dimenzió csökkentés. Ennek a műveletnek az előnye, hogy a későbbiekben a küszöbölés, háttérkivonás, morfológia, élkeresés egy csatornán gyorsabb és stabilabb lesz, mint három csatornán.

Nem minden szín számít egyformán a szemünknek. Az ember zöldre érzékenyebb, kékre kevésbé. Ha egyszerűen átlagolnánk a csatornákat (R+G+B)/3, hamisan becsült fényességet kapnánk. Ezért használunk súlyozott keveréket:

$$\text{RGB[A] to Gray:} \quad Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

RGB to Gray OpenCV: Color conversions 2025.10.19.

GaussianBlur

A Gaussian-blur (Gauss-elmosás) az egyik leggyakoribb átlagszűrő, amelyet a képfeldolgozásban a zajcsökkentésre és a kép simítására alkalmaznak.
Lényege, hogy a képpont értékét a környezetében lévő pixelek súlyozott átlagaként számítja ki, ahol a súlyokat a Gauss-eloszlás határozza meg.
Ezáltal a közeli pixelek nagyobb, a távoliak pedig kisebb hatással vannak az eredményre.

A kétdimenziós Gauss-függvény a következőképpen írható fel:

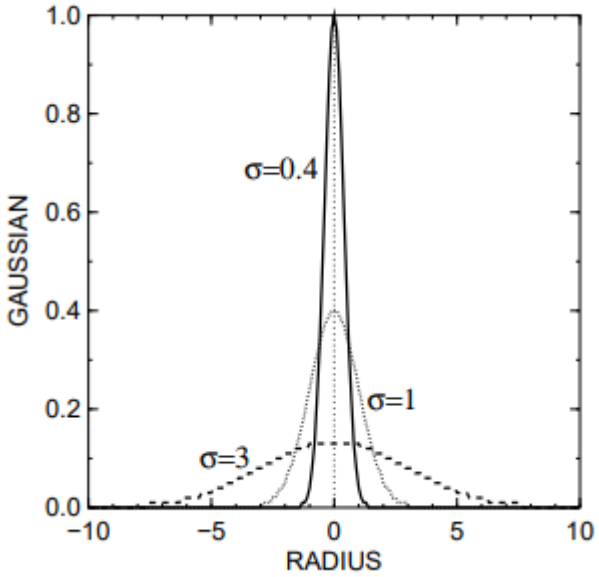
$$w_G(x,y) = \frac{1}{\sum_{(x,y)\in W} e^{-\frac{r^2(x,y)}{2\sigma^2}}} e^{-\frac{r^2(x,y)}{2\sigma^2}},$$

Gauss szűrő Csetverikov D. – Digitális képelemzés alapvető algoritmusai, ELTE, 2008. 2.3.1 fejezet

ahol

- (x,y) a pixel eltolása a középponttól, és $r^2(x, y) = x^2 + y^2$
- a **szórás** határozza meg az elmosás mértékét:
  minél nagyobb a szórás, annál erősebb a simítás, mivel a súlyok összege mindig 1.

[2]



A Gauss-szűrő alakja növekvő szórásra 2D-ben. Gauss szűrő Csetverikov D. – Digitális képelemzés alapvető algoritmusai, ELTE, 2008. 2.3.1 fejezet

Háttérkivonás (MOG2) és árnyék eldobása

Háttérkivonás

A háttérkivonás célja, hogy a videósorozat minden képkockáján elkülönítse a háttér (statikus) és az előtér (mozgó) pixeleket. A Mixture of Gauss módszere szerint minden pixelhez nem egyetlen, hanem több (K darab, jellemzően 3–5) Gauss-eloszlást rendelünk, amelyek a pixel lehetséges állapotait (pl. különböző színei, árnyalatai) írják le.

- Minden Gauss-komponensnek van átlaga (μ), szórása (σ) és súlya (w).
- A súly azt fejezi ki, hogy az adott színárnyalat mennyi ideig fordul elő a jelenetben, vagyis mennyire jellemző a háttérre.
- A háttérmodell a legnagyobb valószínűségű (legstabilabb) komponensekből áll össze.

Egy új képkocka feldolgozásakor az adott pixel értékét összevetjük a modell komponenseivel:

- Ha illeszkedik valamelyik Gauss-eloszláshoz (az eltérés nem haladja meg a kb. 2,5 szórást), a pixel háttér.
- Ha egyik komponenshez sem illeszkedik, új komponens jön létre (az új szín értékével, kis súllyal és nagy kezdeti szórással).

A modell folyamatosan frissül, a komponensek súlyai és átlagai az új adatok alapján módosulnak, így a rendszer alkalmazkodik a megvilágítás és környezet lassú változásaihoz.

A klasszikus MOG nem tudta megkülönböztetni a mozgó árnyékokat a tényleges mozgó tárgyaktól.
A továbbfejlesztett modell (MOG2) ezt egy szín–fényesség alapú megközelítéssel oldja meg.
Ha a pixel színe hasonló a háttérhez, de a fényereje kisebb, a rendszer árnyékként értelmezi, nem pedig mozgó objektumként. [3]

Árnyékok lenullázása

A háttérkivonás során az előtérmaszk nemcsak a mozgó objektumokat, hanem a világosságváltozásból származó árnyékokat is tartalmazhatja.
Ezek eltávolítása intenzitás-küszöböléssel (intensity thresholding) történik, amely a képelemeket intenzitásuk alapján osztályokba sorolja.

Az intenzitás-küszöbölés feltételezi, hogy a kép több, közel homogén régióból áll (például háttér, objektum, árnyék).
Egy vagy több küszöbérték megadásával az intenzitástartomány intervallumokra bontható, és minden pixel egy adott osztályhoz rendelhető.
A legegyszerűbb eset a kétszintű (bináris) küszöbölés, ahol az egyik osztály a háttér, a másik az előtér. [4]

$$g(x,y) = \begin{cases} 1 & \text{if } f(x,y) > T \\ 0 & \text{if } f(x,y) \leq T \end{cases}$$

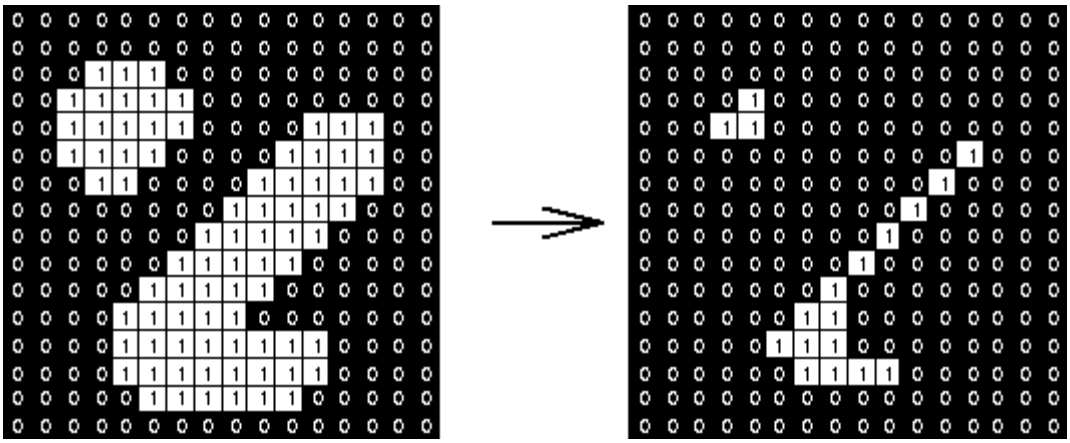
Bináris küszöbölés Image Thresholding | TheAILearner 2025.10.19.

Morfológia (erózió, dilatáció)

A morfológiai képfeldolgozás a képeket nem intenzitás-, hanem alakzat-információ alapján módosítja. Az alapműveletek az erózió és a dilatáció, amelyek egy struktúráló elemmel (általában kis méretű mátrixszal) vizsgálják a bináris kép geometriai tulajdonságait.

Erózió

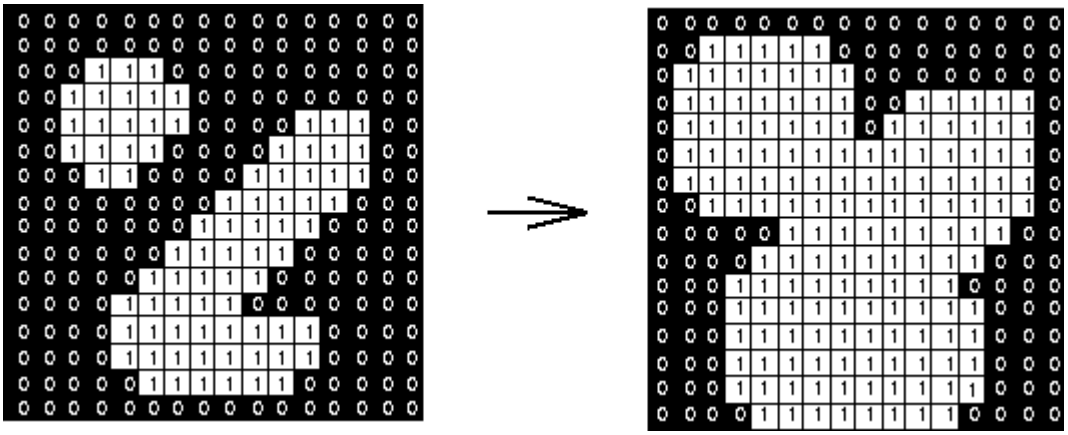
Az erózió a világos pixelek régióját kisebbíti. Egy pixel csak akkor marad fehér, ha a struktúráló elem minden pontja illeszkedik a világos régióba.
Ennek hatására a zajos, apró foltok eltűnnek, a vékony kapcsolatok megszakadnak.



3x3 erózió Morphological Image Processing 2025.10.19.

Dilatáció

A dilatáció az erózió ellentéte, a világos pixelek régióját kiterjeszti.
Egy pixel akkor lesz fehér, ha a struktúráló elem legalább egy pontja az eredeti fehér régióba esik.
Ezáltal a kis lyukak kitöltődnek, és a közeli objektumok összekapcsolódhatnak.



3x3 dilatáció Morphological Image Processing 2025.10.19.

Kombinált műveletek

- **Nyitás (Opening):** Erózió majd dilatáció. Eltávolítja a kis zajfoltokat, de megőrzi a nagyobb objektumok alakját.
- **Zárás (Closing):** Dilatáció majd erózió. Kitölti az objektumokon belüli kis réseket, és összekapcsolja a közeli elemeket.

[5]

Kontúrok detektálása, szűrése és centroidjaik számítása

Kontúrok detektálása

A kontúrdetektálás azért fontos, mert segítségével meghatározhatók a mozgó járművek pontos körvonalai, így megbízhatóan számíthatók az objektumok geometriai jellemzői (pl. méret, arány, centroid), amelyek alapját adják a járműszámlálásnak.

A kontúrok meghatározásához a képnek binárisnak kell lennie, azaz 0-val jelölt háttér és nem-nulla előtér értékekkel, ezért a kontúr detektálás előfeltétele a küszöbölés.

A kontúrok a képen található objektumok határvonalait jelentik, vagyis azon előtérbeli pixelek sorozatát, amelyeknek van háttérbeli szomszédja. [6]

Kontúrok szűrése

A detektált kontúrok közül csak azok kerülnek feldolgozásra, amelyek mérete és oldalaránya megfelel a járművekre jellemző tartománynak. A túl kicsi (pl. zajból keletkező) vagy extrém arányú (pl. árnyékcsík, vonal) objektumok kiszűrésre kerülnek.
Ez a szűrés nem igényel klasszikus képfeldolgozási algoritmusokat, csupán logikai feltételek alkalmazását az objektum területe és oldalaránya alapján.

Centroid számítás

A centroid számítás egy jól definiált és egyszerű eljárás a kontúr által határolt objektumok középpontjainak meghatározására. Ebben fontos szerepet játszanak a momentumok, amelyek az alakzat pontjainak x és y koordinátáiból képezett összegek, súlyozással (pl. intenzitás alapján). A képen N objektumpont esetén:

$$m_{pq} = \sum_{i=1}^{N} x_i^p \cdot y_i^q \cdot f_i$$

p+y-ad rendű momentum Momentumok 2025.10.20.

ahol f_i az adott képpont intenzitása (bináris képnél gyakran $f_i=1$).

A számunkra fontos, centorid számításhoz szükséges momentumok:

- m_{00} = az alakzat „mérete” (pontok száma vagy intenzitás-összeg)
- m_{10} = az x koordináták súlyozott összege
- m_{01} = az y koordináták súlyozott összege

A súlypont (centroid) koordinátái ezek alapján számíthatók:

- m_{10}/ m_{00} = a centroid x koordinátája
- m_{01}/ m_{00} = a centroid y koordinátája

[7]

Átlépés detektálása

Az átlépésdetektálás célja annak meghatározása, hogy egy mozgó objektum (például jármű vagy gyalogos) átlépett-e egy előre kijelölt virtuális vonalat vagy zónát.

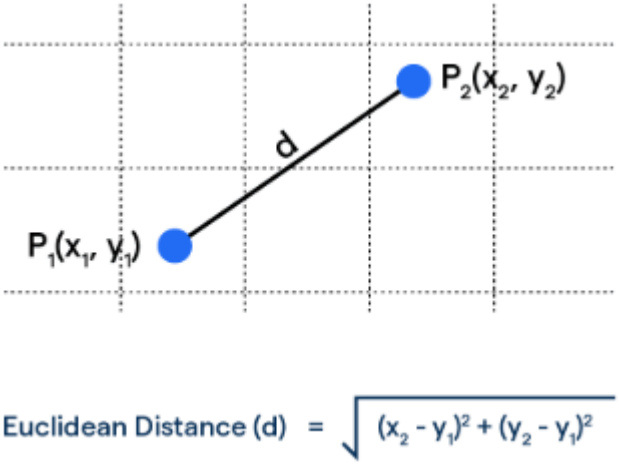
Az átlépésdetektálás a következő lépéseken fog alapulni:

1. A vonal matematikai definiálása a képsíkon két pont alapján. A vonal két végpontjának koordinátáinak megadásával fogjuk definiálni a vonalat.
2. Az objektumok centroidjainak helyzetének vizsgálata a vonalhoz képest (melyik oldalán van), a következő módszerrel:

$$(B_x - A_x)(P_y - A_y) - (B_y - A_y)(P_x - A_x)$$

Pont helyzetének meghatározása a vonalhoz képest linear algebra - Calculate if a point lies above or below (or right to left) of a line - Mathematics Stack Exchange 2025.10.20.

   - Ha az eredmény negatív, akkor a pont a vonal alatt van.
   - Ha negatív, akkor a vonal fölött van.
   - Ha 0, akkor a vonalon van.
3. Objektumkövetés a legközelebbi szomszéd (nearest neighbor) módszerrel. minden új centroidot ahhoz az előző frame-beli centroidhoz rendelünk, amely minimális euklideszi távolságra van tőle.



Euklideszi távolság Euclidean Distance: Advantages and Limitations 2025.10.20.

   Ha a távolság kisebb, mint egy előre megadott küszöbérték, akkor az objektum azonosnak tekinthető. Ezzel elkerülhető, hogy több objektum mozgása összekeveredjen a követés során.

4. Átlépés detektálása az előző és az aktuális centroid-helyzet szorzatának előjelváltozásával. Ha a centroid két egymás utáni vonalhoz viszonyított helyzetének szorzata negatív, vagy kétszer egymás után 0, akkor az átlépés megtörtént.

A megvalósítás terve és kivitelezése

A megvalósítást iteratív, verzióalapú megközelítéssel végzem. Minden verzió egy új pipeline-elemmel bővíti a rendszert, miközben a korábbi lépéseket változatlanul megtartom. Ez a módszer:

- átlátható (minden verzió külön futtatható és dokumentálható),
- könnyen hibakereshető (ha gond van, azonosítható, melyik lépés okozza),
- tanulhatóságot segíti (minden új elem mögött látszik az indoklás és a hatás).

A verziók sorrendje:

- **v1** - videó megnyitása és képkockák megjelenítése
- **v2** – Képkocka előkészítés: ROI + szürkeárnyalat + GaussianBlur
- **v3** – MOG2 háttérkivonás + árnyék eldobás + morfológia
- **v4** – kontúrok, bbox, centroid, alap szűrés
- **v5** – nearest-neighbor párosítás, vonalátlépés-detektálás és irány, számlálók

V1- videó megnyitása és képkockák megjelenítése

A rendszer első lépése a videóforrás megnyitása és egy ciklus kialakítása, amely képkockánként dolgozza fel a bejövő adatot. Ehhez a `cv2.VideoCapture()` függvényt használtam, amely lehetővé teszi videófájl vagy kameraeszköz kezelését. A függvény bemenete egy fájlútvonal vagy kameraindex, és egy VideoCapture objektummal tér vissza, amelyen keresztül a videó képkockái elérhetők. A művelet sikeressége az `isOpened()` metódussal ellenőrizhető, amely True értékkel tér vissza, ha sikerült.

```
def main(params):
    cap = cv2.VideoCapture(params[0])
    if not cap.isOpened():
        print("Nem sikerült megnyitni a videót!")
        exit()
    while True:
        ret, frame = cap.read()
        if not ret:
            break
        cv2.imshow("Eredeti", frame)
        if cv2.waitKey(params[1]) == ord('q'):
            break
    cap.release()
    cv2.destroyAllWindows()
params = [src := "C:/út/a/videóhoz.mp4",
        waitkey := 30]
main(params)
```

A képkockák beolvasásához a `cap.read()` metódust alkalmaztam, amely két értékkel tér vissza: egy logikai változóval, amely jelzi a sikerességet (True vagy False), és magával a beolvasott képpel. Ha a beolvasás sikertelen például a videó véget ér, a ciklus megszakad.

A képek megjelenítéséhez az `cv2.imshow()` függvényt használtam, ahol bemenetnek megadjuk az ablak nevét és a megjelenítendő képet, majd az ablakban megjeleníti a képet.A kép tényleges megjelenítéséhez szükség van még a `cv2.waitKey(delay)`

függvényre, amely a paraméterként megadott időt (ms) várakozással tölti, és ezalatt billentyűlenyomást is képes érzékelni. A q billentyűgomb megnyomásával a programból azonnal ki lehet lépni.

A videóforrás felszabadításáról és az ablakok bezárásáról a `release()` amely bezárja a videófájlt és a `cv2.destroyAllWindows()` gondoskodik amely bezár minden opencv ablakot.

A `main()` függvény paramétereként a videó fájlútvonalat és a két frame megjelenítése között eltelt időt adhatjuk meg.

[8] [9] [10]

V2– Képkocka előkészítés: ROI + szürkeárnyalat + GaussianBlur

A második fejlesztési mérföldkőben létrehoztam egy, a videó első képkockája alapján számított trapéz alakú maszkot, amely kizárja a kép nem releváns területeit, így jelentősen csökkenti a számítási igényt és a hamis detekciókat. A maszk létrehozásához a `cv2.fillPoly()` függvényt használtam, amelynek első paramétere a kép(esetünkben egy fekete pontokból álló tömb aminek mérete megegyezik a videó képkockáinak méretével) amin a műveletet végezzük, a második a sokszög tömbje ami a maszkot reprezentálja, a harmadik pedig a sokszög színe. A kimenet a maszk lesz. A

```
def create_mask(cap, BF, JF, JA, BA):
    # Trapéz alakú maszk készül az első frame alapján
    ret, first = cap.read()
    H, W = first.shape[:2]
    mask = np.zeros((H, W), np.uint8)
    points = np.array([
        (int(W* BF[0]), int(H* BF[1])),          # bal felső
        (int(W* JF[0]), int(H* JF[1])),          # jobb felső
        (int(W* JA[0]), int(H* JA[1])),          # jobb alsó
        (int(W* BA[0]), int(H* BA[1]))           # bal alsó
    ], np.int32)
    cv2.fillPoly(mask, [points], 255)
    cap.set(cv2.CAP_PROP_POS_FRAMES, 0)
    return mask

def frame_preparation(frame, mask, blur_kernel):
    frame_roi = cv2.bitwise_and(frame, frame, mask=mask)
    gray = cv2.cvtColor(frame_roi, cv2.COLOR_BGR2GRAY)
    gray_blur = cv2.GaussianBlur(gray, blur_kernel, 0)
    return frame_roi, gray_blur
```

`cap.set(cv2.CAP_PROP_POS_FRAMES, 0)` biztosítja, hogy az első képkocka feldogozása után visszatérjünk a legelső (0 indexű) képkockához, tehát ne veszítsük el a feldolgozás során.

A maszk alkalmazása a képkockákra bitművelettel történik: `cv2.bitwise_and(frame, frame, mask=mask)`. Ez a művelet azokat a pixeleket tartja meg, amelyeknél a maszk értéke 255 (fehér), míg a 0 (fekete) maszkterületeket lenullázza.

A következő lépésben a maszkolt képet szürkeárnyalatossá alakítom `cv2.cvtColor(frame_roi, cv2.COLOR_BGR2GRAY)` függvénnyel, amely átalakítja a bemeneti képet (első paraméter) egyik színtérből a másikba (esetünkben BGR-ből szürkébe a második paraméter alapján), majd Gauss-szűrőt alkalmazok `cv2.GaussianBlur(gray, (5,5), 0)`, ami elhomályosítja a képet (első paraméter) egy Gauss-szűrővel amelynek a kernel mérete a második paraméter és a szórása X irányban a harmadik paraméter. Mindezt azért teszem, hogy a háttérkivonás számára stabilabb, kevésbé zajos intenzitásképet kapjak.

Ebben a verzióban az előző mellett a maszkolt és szürkeárnyalatos és elhomályosított videókat is megjelenítem, hogy lássuk a műveletek hatását.

A `main()` függvény paramétereként a V1 paraméterein felül a maszk paramétereit és a gauss kernel méretét adhatjuk meg.

[8] [11] [12] [13]

V3 - Háttérkivonás (MOG2) és morfológiai tisztítás

A projekt harmadik iterációjának célja az volt, hogy a mozgó járműveket elkülönítsem az útburkolat állandó elemeitől, vagyis előállítsam az úgynevezett előtér-maszkot. Ez most már nem csupán egyszerű szűrésen és maszkoláson alapul, hanem egy időben adaptálódó háttérmodellen, amely minden képpont változását megfigyeli, és képes eldönteni, hogy egy pixel éppen az előtérhez tartozik-e, vagy csak a háttér apró fényviszonyváltozása miatt módosult a színe.

Ehhez a Mixture of Gaussians 2 (MOG2) algoritmust alkalmaztam, amely módszer minden pixel fényességértékét Gauss-eloszlás keverékével közelíti, és hosszabb távú megfigyelés alapján dönti el, hogy egy adott pixelt háttérnek tekinthetünk-e. A `cv2.createBackgroundSubtractorMOG2()` függvény létrehoz egy ilyen modellt, amely képes folyamatosan frissülni, és így dinamikusan alkalmazkodik például az enyhe fényváltozásokhoz vagy lassan mozgó árnyékokhoz is. A függvény paraméterei közül a `history` értéke határozza meg, hány korábbi képkockát vegyen figyelembe a modell, a `varThreshold` pedig azt befolyásolja, mennyire érzékenyen reagáljon a képpont-intenzitás változásokra. Ezt a két értéket úgy választottam meg, hogy megfelelően stabil hátteret kapjak, de a valódi mozgásról se maradjon le a rendszer. Ezen kívül van egy harmadik `detectShadows` paramétere is amely ha True, akkor az algoritmus észleli az árnyékokat és megjelöli azokat. Ez egy kicsit csökkenti a sebességet, ezért ha nincs szükség erre a funkcióra,akkor érdemes a paramétert False értékre állítani. Mivel a forgalom-számlálás kültéren történik, ezért az értékét True-ra állítom.

```
def foreground_preprocessing(background_history=300, varThreshold=25, ellipse_kernel=(5, 5), rect_kernel=(5, 5)):

    bg = cv2.createBackgroundSubtractorMOG2(background_history, varThreshold, detectShadows=True)

    kernel_open = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, ellipse_kernel)

    kernel_bridge = cv2.getStructuringElement(cv2.MORPH_RECT,   rect_kernel)

    return bg, kernel_open, kernel_bridge
```

A `cv2.createBackgroundSubtractorMOG2()` függvény által létrehozott modell az `apply()` metódus segítségével minden képkockából készít egy előtér-maszkot, amelyben az előtér pixelek 255-ös értéket, a háttér pixelek pedig 0-át kapnak. Ha az árnyékdetektálást bekapcsolva

hagyjuk, akkor az árnyékokat 127-es értékkel jelöli meg, amit rögtön kinullázok, hiszen a célom a járművek detektálása, és az árnyékok rendszerint csak rontják az előtér tisztaságát.

Ez az előtér-maszk már mutatja a járműveket, de még tele van kisebb hibákkal, apró elszórt zajokkal, szakadozott felületekkel, járművek szélei körüli hézagokkal, ezért morfológiai műveleteket alkalmazok a tisztításra. Először egy OPEN (erózió majd dilatáció) műveletet hajtok végre, amely előbb lekicsinyíti az objektumokat, majd visszatölti azokat, ezzel az elszórt, csupán néhány pixeles zajok eltűnnek. Ezután egy CLOSE (dilatáció majd erózió) művelettel zárom a nagyobb lyukakat az autók testén belül, hogy az objektum jól kontúrozható maradjon. Végül egy rövid dilatáció következik, amely enyhén megvastagítja a járműfoltot, ami később, a kontúrkeresés során megbízhatóbb eredményt biztosít.

A műveleteket különböző formájú strukturáló elemekkel hajtottam végre a `cv2.getStructuringElement()` segítségével melynek első paraméterével a struktúra formáját határozhatom meg a második paraméterrel pedig kernel méretét adhatom meg. Az OPEN lépéshez ellipszis alakút használtam, mert ez jobban illeszkedik a zajok általában kerekded formájához, a CLOSE művelethez pedig téglalap alakút választottam, amely hatékonyabban hidalja át a szétszakadt járműrészeket.

A strukturáló műveleteket illetve a háttérmodellt létrehozó MOG2-t egy külön előtér előkészítő függvényben helyeztem el, mivel ezeket csak egyszer kell létrehozni még a ciklus előtt. Az apply(), árnyék eldobása, OPEN, CLOSE és dilatáció műveleteket egy másik függvényben helyeztem el, mivel ezeket minden képkockára el kell végezni.

Ebben a verzióban az előzőek mellett az előtér-maszk és a morfológiával tisztított videókat is megjelenítem, hogy lássuk a műveletek hatását.

A `main()` függvény paramétereként a V2 paraméterein felül a A `cv2.createBackgroundSubtractorMOG2()` függvény `history` és `varThreshold` paramétereit, a `cv2.getStructuringElement()` függvény strukturáló elemeinek kernel méreteit és a dilatáció művelet iterációinak számát adhatjuk meg.

[14] [15] [16]

```
def moving_object_highlighting(gray_blur, bg, kernel_open, kernel_bridge, fg_clean_iter=1):

    fg = bg.apply(gray_blur)  # 0=háttér, 127=árnyék, 255=előtér

    fg[fg == 127] = 0 #árnyék eldobása

    fg_open = cv2.morphologyEx(fg, cv2.MORPH_OPEN, kernel_open, iterations=1) # Morfológia: OPEN
(erode -> dilate)

    fg_close = cv2.morphologyEx(fg_open, cv2.MORPH_CLOSE, kernel_bridge, iterations=20)

    fg_clean = cv2.dilate(fg_close, kernel_bridge, iterations=fg_clean_iter)

    return fg, fg_clean
```

v4 – kontúrok, bbox, centroid, alap szűrés

A negyedik iterációban a célom az volt, hogy a v3-ban előállított, megtisztított előtér-maszkból jármű jelölteket nyerjek ki, illetve elvégezzem az előkészületeket a majd a v5-ben megvalósuló átlépés detektáláshoz. Ennek során:

- először egy számlálóvonalat definiálok a képek, amelyhez képest majd a v5-ben eldől, hogy egy jármű átlépte-e.
- detektálom az előtér kontúrjait,
- kiszámolom a területüket, és befoglaló téglalapot határozok meg hozzájuk
- méret és oldalarány alapján szűröm őket,
- kiszámolom a középpontjaik koordinátáit és elhelyezem őket egy listában.
- Végül a középpontokat, befoglaló téglalapokat és a számlálóvonalat megjelenítem a ROI-val maszkolt eredeti képen.

A vonal meghatározását egyszerűen hajtom végre. A képmagasság egy arányát (`y_line`) veszem, és ehhez a tényleges H és W ismeretében két pontot hozok létre, a kép bal és jobb szélein. A gyakorlatban ez azt jelenti, hogy a vonal nem adott pixelekhez van kötve, hanem a felbontástól függetlenül, relatív helyen fut, ami megkönnyíti a különböző videók közötti átállást. A `line_position(y_line, H, W)` függvény ennek megfelelően visszaadja az `A=(0, y_line)` és `B=(W, y_line)` pontokat. Az OpenCV-ben ezek a pontok már közvetlenül rajzolhatók a képre egy egyszerű `cv2.line()` hívással. Ennek a folyamatnak a gördülékenysége érdekében módosítottam a v2 során létrehozott `create_mask()` függvényemen, hogy adja vissza a képkocka magasságát és szélességét is amelyek paraméteréül szolgálnak a `line_position()` függvénynek, így nem szükséges őket újra kiszámítani. Ezért a vonal létrehozására a ROI-maszk létrehozása után kerül sor a `main()` függvényben. Az `y_line` paraméter megadását is lehetővé teszem a `main()` függvény paramétereként. [17]

```
def line_position(y_line, H, W):
    y_line = int(H * y_line)
    A = (0, y_line)
    B = (W, y_line)
    return A, B
```

A jármű-jelöltek kinyeréshez a v3-ban előállított bináris, morfológiával megtisztított maszkot használom bemenetként. A `cv2.findContours()` függvény a fehér régiók határait adja vissza összefüggő kontúrok formájában, ezek azok az alakzatok, amelyekből jármű-jelölteket szeretnék képezni. A függvény első paramétere bemeneti kép. A második paramétere a kontúrvisszakeresés módja, ami a mi esetünkben `RETR_EXTERNAL`, ez azt jelenti, hogy csak a szélső külső kontúrokat keresi vissza. A harmadik paraméter pedig a kontúrközelítési módszer ami a mi esetünkben `CHAIN_APPROX_SIMPLE`, ami azt jelenti, hogy összenyomja a vízszintes, függőleges és átlós szegmenseket, és csak a végpontjukat hagyja meg. Például egy függőleges téglalap alakú kontúr 4 ponttal van kódolva. Visszatérési értéke az észlelt kontúrok és a hierarchia amely képtopológiára vonatkozó információkat tartalmaz, de erre nekünk nincsen szükségünk. [18]

A morfológiával tisztított képen előforduló kis zajok és a járművek letört darabjai miatt szűrést vezetek be a kontúrokra. Először kiszámítom a kontúr területét `cv2.contourArea()` függvénnyel (amelynek bemeneti paramétere a kontúr és visszatér a nem nulla pixelek számával) és egy minimális küszöb alatt egyszerűen eldobom az adott kontúrt. Ez a lépés rögtön kiszedi az apró zajokat. Ezután `cv2.boundingRect()` segítségével (amely visszaadja a ponthalmaz minimális up-right téglalapját) befoglaló téglalapokat képezek, és megnézem az oldalarányt (szélesség osztva magassággal), és a túl keskeny, extrém magas vagy lapos objektumok itt kiesnek. Ennek a szűrésnek a paramétereit (`MIN_AREA, ASPECT_MIN, ASPECT_MAX`) megadhatóvá teszem a `main()` függvény paraméterei között, hogy a videók közötti átállást megkönnyítsem az esetleges más kamerabeállítások miatt. [18]

Miután egy kontúr átjut a szűrőkön, a középpontját is kiszámolom. Erre a kép-momentumokat használom (cv2.moments melynek bemenete a kontúr és visszatérési értéke a momentumok) ahol a `cx = int(M["m10"] / M["m00"])` és a `cy = int(M["m01"] / M["m00"])` hányadosok adják a súlypont x és y koordinátáját. Itt figyelek kell arra is, ha `M["m00"]` (az alakzat mérete) nulla lenne (bár ez a `MIN_AREA` ellenőrzésekor többnyire kiderül), akkor ne végezzük el a fenti műveleteket. A működő jelöltekből egy középpont-listát (`centers`) készítek, ez lesz a v5 belépő adata a képkockák közötti párosításhoz. [18]

A rajzolást már itt, a v4-ben beépítettem. A frame_roi képen először megjelenítem a számlálóvonalat, majd minden elfogadott jelölthöz zöld befoglaló téglalapot rajzolok (`cv2.rectangle(detections, (x, y), (x+w, y+h), (0, 255, 0), 2),` és egy 4 pixeles piros ponttal jelölöm a középpontját (`cv2.circle(detections, (cx, cy), 4, (0, 0, 255), -1)`). A vizuális réteg nagyon sokat segít a paraméterhangolásban, ha a `MIN_AREA` túl kicsi, a kép teleszóródik apró, zöld téglalapokkal, ha az oldalarány-korlát túlságosan szűk, akkor a valós járművek is kieshetnek. [17] [18]

```
def drawings(fg_clean, frame_roi, A, B, MIN_AREA=500, ASPECT_MIN=0.3, ASPECT_MAX=4):
    contours, _ = cv2.findContours(fg_clean, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    detections = frame_roi.copy()
    cv2.line(detections, A, B, (255, 0, 0), 2)
    centers = []
    for c in contours:
        area = cv2.contourArea(c)
        if area < MIN_AREA:
            continue
        x, y, w, h = cv2.boundingRect(c)
        aspect = w / float(h) if h > 0 else 0.0
        if aspect < ASPECT_MIN or aspect > ASPECT_MAX:
            continue
        M = cv2.moments(c)
        if M["m00"] <= 0: # m00 = az alakzat mérete
            continue
        cx = int(M["m10"] / M["m00"]) # az alakzat súlypontjának x koordinátája
        cy = int(M["m01"] / M["m00"]) # az alakzat súlypontjának y koordinátája
        centers.append((cx, cy))
        cv2.circle(detections, (cx, cy), 4, (0, 0, 255), -1)
        cv2.rectangle(detections, (x, y), (x+w, y+h), (0, 255, 0), 2)
    return detections, centers
```

v5 – nearest-neighbor párosítás, vonalátlépés-detektálás és irány, számlálók

Az ötödik, és egyben végső verzió célja az, hogy a v4-ben kinyert jármű-jelölteket időben (képkockáról képkockára) követni tudjam és detektáljam, ha egy jármű áthalad a kijelölt számlálóvonalon. Ezzel elérem a projekt eredeti célkitűzését: a forgalom tényleges, irány szerinti számlálását.

Először létrehozok egy `vehicle_position()` nevű függvényt amely a számlálóvonal két végpontja (`A` és `B`) és a jármű centroidjának koordinátáiból (`P`) előjel vizsgálatot végez és megmondja, hogy egy adott járműközéppont a számlálóvonal fölött vagy alatt helyezkedik el. A pozíciót egyetlen szám jelöli:

- pozitív, ha a vonal alatt van,
- negatív, ha a vonal felett van,
- illetve akkor is negatív, ha a vonalon van. Így megelőzhetjük, hogy anélkül haladjanak át járművek a számlálóvonalon, hogy előző és jelenlegi pozíciójuk szorzata negatív legyen. Ez kritikus fontosságú, mivel ez lesz az átlépés detektálás kulcsa.

```
def vehicle_position(A, B, P):
    (Ax, Ay), (Bx, By) = A, B
    Px, Py = P
    side = (Px - Ax)*(By - Ay) - (Py - Ay)*(Bx - Ax)
    return 1 if side > 0 else -1
```

A képkockánként kiszámolt járműközéppontok egy adott frame-re érvényesek. A valóságban azonban egy jármű több ezer képkockán keresztül halad át a látómezőben, miközben folyamatosan mozog. Ahhoz, hogy egy járművet ne minden frame-ben új objektumként értékeljek, össze kell kapcsolni a középpontokat az egymást követő frame-ek között. Ezt nearest-neighbor párosítással valósítottam meg a `crossing_over()` függvény első részében. Ez azt jelenti, hogy minden aktuális középponthoz megkeresem az előző képkocka legközelebbi középpontját, és ha a távolságuk nem túl nagy (kisebb mint a `MAX_DIST` paraméter amelyet a `main()` függvény paramétereként adhatunk meg), akkor ugyanannak a járműnek tekintem őket.

Az átlépés detektálása arra a megfigyelésre alapul, ha egy jármű egyik frame-ről a másikra a vonal egyik oldaláról a másikra kerül (vagy esetünkben éppen a vonalra), akkor átlépte a vonalat. A `crossing_over()` függvény összeköti ezt a megfigyelést a nearest neighbor párosítással. Ha egy korábbi pont és az aktuális pont előjelének szorzata negatív, akkor a jármű áthaladt a vonalon. Ekkor a mozgás irányától függően növelem a megfelelő számlálót:

- ha a jármű pozíciója a `vehicle_position()` függvény alapján pozitív, akkor a `count_up` -ot növelem,
- ha negatív, akkor a `count_down` -t növelem.

A már párosított pontokat egy `matched_prev` halmaz tárolja, így egy jelölt nem kapcsolódik többször ugyanazzal az előző ponttal, tehát nincs duplán számlálás.

```
def crossing_over(centers, prev_pts, prev_pos, count_up, count_down, A, B, MAX_DIST):
    matched_prev = set()    # Már párosított pontok
    for (cx, cy) in centers:
        pos_now = vehicle_position(A, B, (cx, cy)) # Jármű melyik oldalon van most
        # egyszerű nearest-neighbor párosítás az előző frame-hez és átlépés logika
        nearest_neighbor, best_distance = -1, MAX_DIST + 1
        for i, (px, py) in enumerate(prev_pts):
            if i in matched_prev:
                continue
            d = math.hypot(cx - px, cy - py)
            if d < best_distance:
                best_distance, nearest_neighbor = d, i    # Legkisebb távolságú pont
        #átlépés detektálás
        if nearest_neighbor != -1 and best_distance <= MAX_DIST:
            pos_prev = prev_pos[nearest_neighbor] if nearest_neighbor < len(prev_pos) else 0
            if pos_prev * pos_now < 0:
                if pos_now > 0:
                    count_up += 1
                else:
                    count_down += 1
            matched_prev.add(nearest_neighbor)
    return count_up, count_down
```

Az előző állapotok frissítése (`prev_pts, prev_pos`) minden ciklus végén történik.

A számlálók értékét szintén a számlálóvonalat, bounding box-ot és centroidokat megjelenítő képkockákra írom ki, hogy a felhasználó azonnal láthassa az aktuális értékeket. Ehhez a `cv2.putText` függvényt használom, amely a detections képre a bal felső sarok közelébe (`(10, 30)` pixel) kiírja a be-, ki- és összesített átlépések számát. A szöveg a `FONT_HERSHEY_SIMPLEX` (normál méretű sans-serif betűtípus) betűtípussal, `0.8`-as méretarányban, `2` pixeles vastagsággal jelenik meg, sárgás BGR-színnel (`0, 255, 255`), mert ez jól elüt a legtöbb útfelvétel sötétebb hátterétől. A felirat minden képkockán frissül, így valós időben követhető a számlálás. [19]

```
cv2.putText(detections, f"UP: {count_up}  DOWN: {count_down}  TOTAL: {count_up+count_down}",
                (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0,255,255), 2)


        cv2.imshow("Eredeti", frame)
        cv2.imshow("ROI, csak az út", frame_roi)
        cv2.imshow("Gray + Blur", gray_blur)
        cv2.imshow("FG Mask (MOG2)", fg)
        cv2.imshow("FG (open + dilate)", fg_clean)
        cv2.imshow("Detections (contours)", detections)


        # előző állapot frissítése
        prev_pts = centers
        prev_pos = [vehicle_position(A, B, p) for p in centers]
```

Tesztelés

Irodalomjegyzék / hivatkozások

[1] Image Processing Part 5: Arithmetic, Bitwise, and Masking | DigitalOcean 2025.10.18.

[2] D. Csetverikov, *Digitális képelemzés alapvető algoritmusai 2.3.1 fejezet*. Eötvös Loránd Tudományegyetem, Budapest, 2008. [Online 2025.10.19]. https://www.inf.elte.hu/dstore/document/297/Csetverikov_jegyzet.pdf

[3] P. KaewTraKulPong and R. Bowden, *An Improved Adaptive Background Mixture Model for Real-time Tracking with Shadow Detection 2. fejezet*, 2025.10.19.

[4] D. Csetverikov, *Digitális képelemzés alapvető algoritmusai 6.1 fejezet*. Eötvös Loránd Tudományegyetem, Budapest, 2008. [Online 2025.10.19]. https://www.inf.elte.hu/dstore/document/297/Csetverikov_jegyzet.pdf

[5] D. Csetverikov, *Digitális képelemzés alapvető algoritmusai 9.1 fejezet*. Eötvös Loránd Tudományegyetem, Budapest, 2008. [Online 2025.10.19]. https://www.inf.elte.hu/dstore/document/297/Csetverikov_jegyzet.pdf

[6] Kontúrok detektálása 2025.10.20.

[7] Momentumok 2025.10.20.

[8] OpenCV: cv::VideoCapture Class Reference 2025.10.25.

[9] OpenCV: Getting Started with Videos 2025.10.25.

[10] OpenCV: High-level GUI 2025.10.25.

[11] OpenCV: Drawing Functions 2025.10.25.

[12] OpenCV: Operations on arrays 2025.10.25.

[13] OpenCV: Color Space Conversions 2025.10.25.

[14] OpenCV: Motion Analysis 2025.10.25.

[15] OpenCV: cv::BackgroundSubtractorMOG2 Class Reference 2025.10.25.

[16] OpenCV: Morphological Transformations 2025.10.25.

[17] OpenCV: Drawing Functions in OpenCV 2025.10.26.

[18] OpenCV: Structural Analysis and Shape Descriptors 2025.10.26.

[19] OpenCV: Drawing Functions 2025.10.26.