Алгоритам за детекција на коловозни ленти во автономни возила

Изработиле: Илија Бунчески 221094 ; Теодор Дурацоски 223131

19.05.2024

Вовед

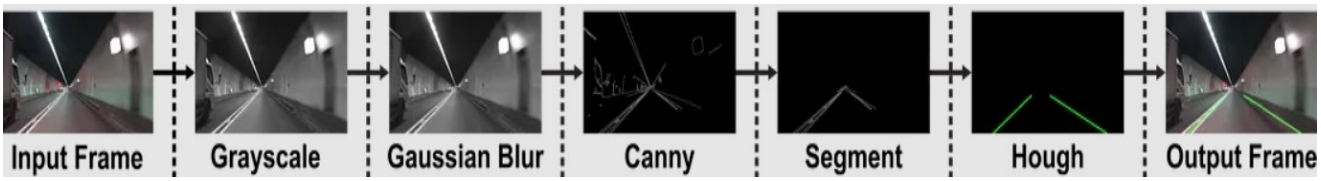
Во пребрзорастечкиот свет на автономните возила, еден од основните предизвици претставува обезбедување на сигурна и ефикасна навигација на патиштата. Детекцијата на коловозните ленти како критична компонента на системите за автономни возила, игра клучна улога во ова прашање. Додека возилата стануваат сè повеќе автономни, способноста да се разбере и интепретира околниот сообраќај, претставува од суштинско значење. Земајќи го сето ова во предвид, можеме да кажеме дека системите за детекција на ленти служат како очи на автономните возила, помагајќи им да го разберат и да навигираат низ комплексниот мрежен систем на коловозни ленти и патишта.

Во овој проект ќе претставиме алгоритам кој обработува видео во реално време и ги детектира и означува коловозните ленти во истото, користејќи ја моќта на OpenCV и Hough Transform алгоритмот.

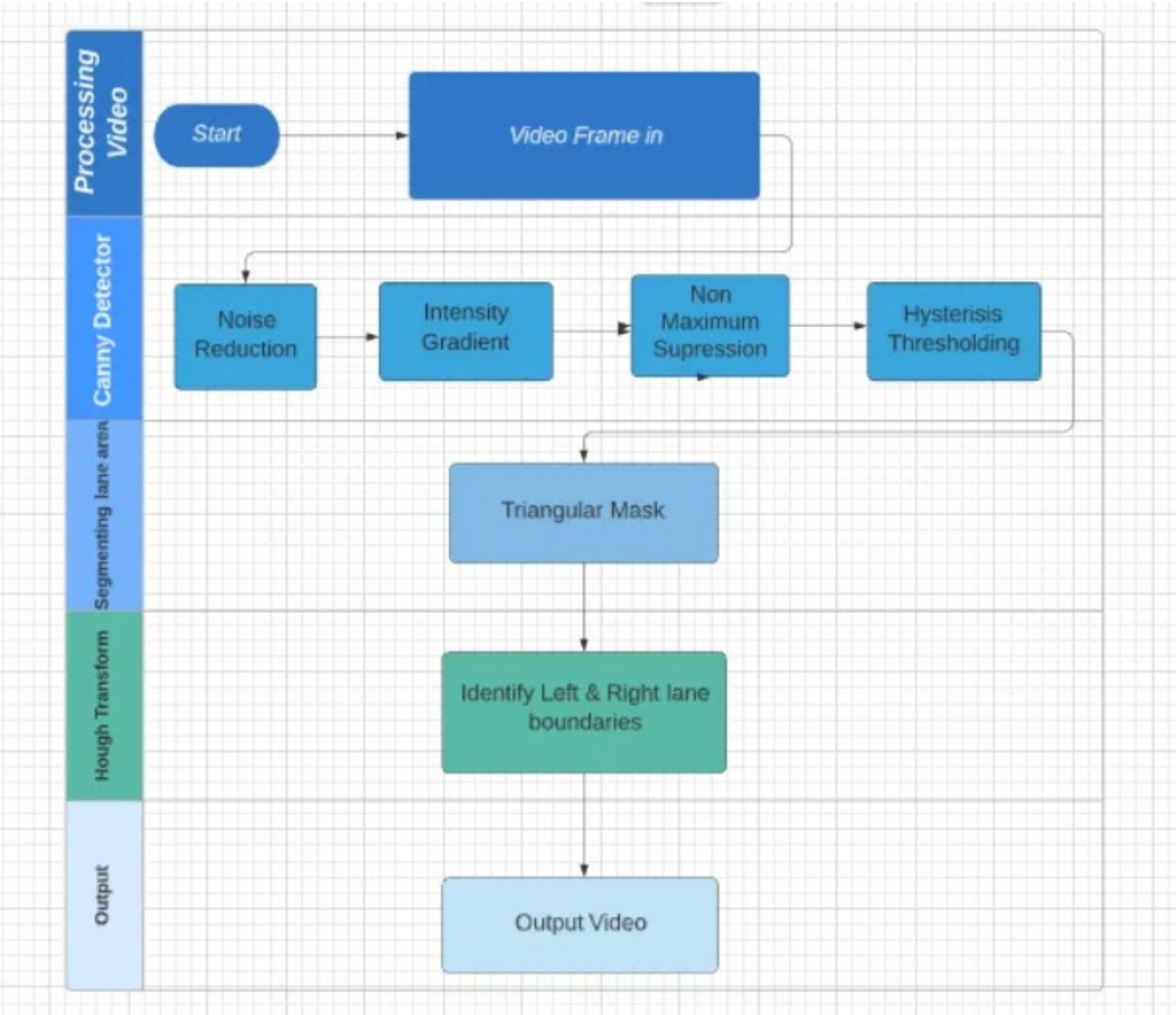
Општ преглед на алгоритмот

Како што е наведено во воведот погоре, ние имплементиравме алгоритам за детекција на коловозни ленти во реално време. Со цел да можеме да демонстрираме, во овој проект користиме веќе снимено видео, но теоретски овој алгоритам би работел и со видео во реално време. Начинот на кој што работи нашиот алгоритам е таков што ја зема и обработува секоја рамка(frame) посебно.

За детекција на коловозните ленти користиме canny detector - детекцијата на линии базирана на трансформацијата на Hough. Тоа е претставено со следните дијаграми:



Слика 1. Преглед на Canny-Hough системот за детекција



Слика 2. Текот на процесите на Canny-Hough детекторот

Пристап

Процесот се состои од следните чекори:

- Grayscale

- Гаусово заматување

- Детектирање на рабови со Canny

- Селекција на региони на интерест(ROI)

- Детектирање на коловозните линии со Hough трансформација

- Цртање на линиите врз оригиналната рамка

- Процесирање на видеото

Grayscale

Greyscale претставува техника на конверзија на слики за елиминирање на секоја форма на информација за бојата, оставајќи само различни нијанси на сиво, каде најсветлата нијанса е белата, додека пак најтемната е црната боја. Средните нијанси обично имаат еднакво ниво на осветленост за основните бои (црвена, сина и зелена). Секој пиксел е приказ на светлосниот интензитет на сликата.

Сликите добиени со оваа техника потоа се прошируваат со цел коловозните линии да дојдат до израз.

Горенаведеното е имплементирано со следниот код:

```
def grayscale(image):
    gray_img = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    gray_img = cv2.dilate(gray_img, kernel=np.ones((3, 3), np.uint8))

    return gray_img
```

Во продолжение се прикажани оригиналната слика и сликата која се добива после користење на оваа техника:

Слика 3. Оригинална слика

Слика 4. Сликата добиена после користење на Greyscale техниката

Гаусово заматување

Во светот на процесирање на слики, Гаусовото заматување (Gaussian blur) е резултат на заматување на слики со помош на Гаусовата функција.

Математички, Гаусовата функција е претставена на следниот начин:

- *Во еднодимензионален простор:*

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

- *Во дводимензионален простор:*

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

Ова е широко користен ефект најчесто за намалување на шумот и деталите на сликата.

Овој чекор нам ни е потребен поради тоа што градиентите во Canny се сензитивни на шум, па поради тоа наша цел е да го елиминираме колку што е можно повеќе.

Во нашиот алгоритам гаусовото заматување го имплементираме на следниот начин:

```
def gaussian_blur(gray_scaled_image):
    blur = cv2.GaussianBlur(gray_scaled_image, (5, 5), 0)
    return blur
```

Доколку ваквата имплементација ја искористиме на веќе обработената слика со Greyscale, тогаш го добиваме следниот резултат:

Слика 5. Сликата добиена после Гаусовото заматување

Детектирање на рабови со Canny

Детекторот за рабови Canny е оператор за откривање на рабови кој користи повеќестепен алгоритам за откривање на широк опсег на рабови на сликите.

Откривањето на рабови со Canny е техника за извлекување корисни структурни информации од различни видливи објекти и драматично намалување на количината на податоци што треба да се обработат. Општите критериуми за откривање на рабовите вклучуваат:

1. Откривање на раб со мала стапка на грешка, што значи дека откривањето треба прецизно да опфати што е можно повеќе рабови прикажани на сликата.
2. Врвната точка откриена од операторот треба точно да се локализира на центарот на работ.
3. Даден раб на сликата треба да се означи само еднаш, и каде што е можно, шумот на сликата не треба да создава лажни рабови.

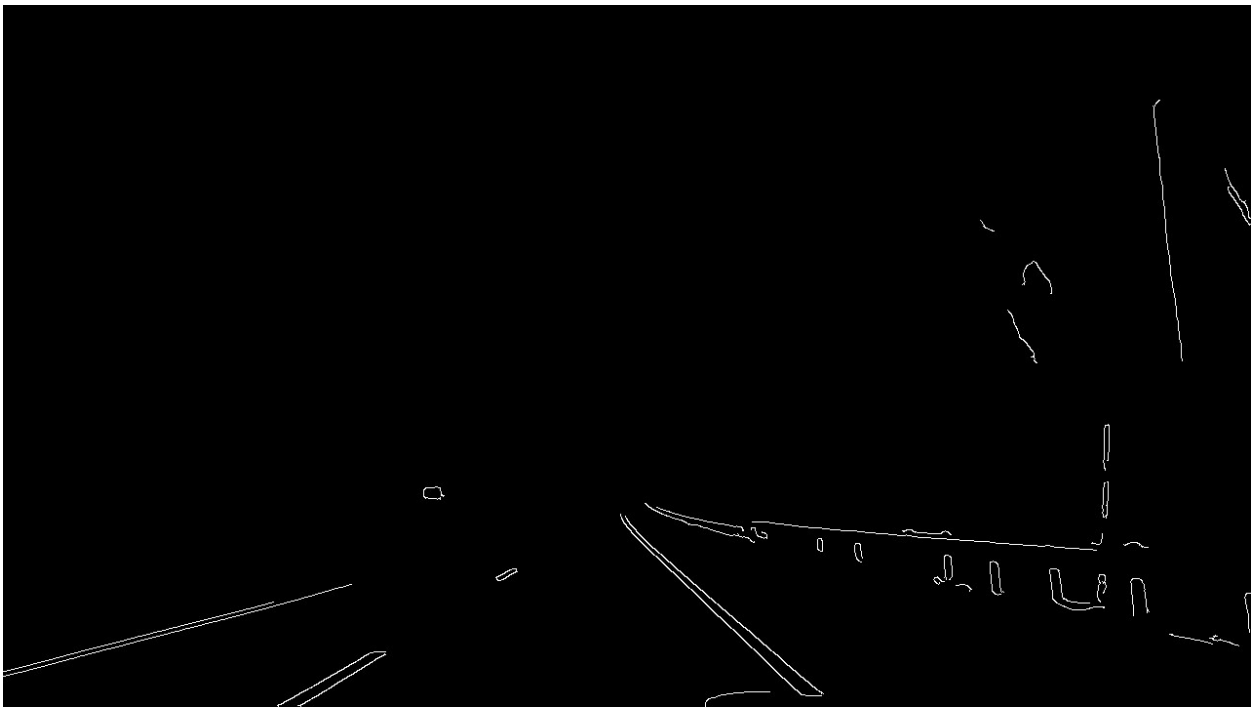
Процесот ги вклучува следните параметри:

- Параметарот *img* ја дефинира сликата на која ќе ги откриеме рабовите.
- Параметарот *treshold-1* ги филтрира сите градиенти пониски од овој број (тие не се сметаат за рабови).
- Параметарот *treshold -2* ја одредува вредноста за која раб треба да се смета за валиден.
- Секој градиент помеѓу двата *tresholds* ќе се зема во предвид ако е прикачен на друг градиент кој е над *treshold-2*.

Горенаведеното го имплементиравме на следниот начин:

```
def canny_img(blured_image):
    canny = cv2.Canny(blured_image, 130, 220)
    return canny
```

Доколку ваквата имплементација ја искористиме на веќе обработената слика со Гаусово заматување, тогаш го добиваме следниот резултат:



Слика 6. Сликата добиена со детектирање на рабови со Canny

Од сликата можеме да видиме дека ги содржи сите рабови од нашата слика и следен чекор е да ги изолираме само рабовите што ни требаат, односно тоа се коловозните ленти.

Селекција на региони на интерес(ROI)

Како што веќе спомнавме погоре, до сега ги имаме пронајдено сите рабови. Во овој чекор треба да ги изолираме рабовите кои кореспондираат на коловозните ленти.(Regions of interest) Тоа го правиме на следниот начин:

```
def roi(image, vertices):
    mask = np.zeros_like(image)
    mask_color = 255
    cv2.fillPoly(mask, vertices, mask_color)
    cropped_img = cv2.bitwise_and(image, mask)
    return cropped_img
```

Со горенаведениот код го извршуваме следното:

1. *Креирање маска:* маската во суштина е бинарна слика со иста големина како и оригиналната слика, иницијализирана со сите пиксели поставени на нула (обично црни). Внатре во полигонот дефиниран со темињата на ROI, пикселите на маската се поставени на вредност што не е нула (обично бели). Ова создава бинарна маска каде што регионот на интерес е „маскиран“ како бело, а остатокот од сликата е црна.
2. *Примена на маската:* маската потоа се комбинира со оригиналната слика со помош на операција на битови. Со примена на оваа маска, ги задржуваме само информациите во рамките на дефинираниот ROI, ефективно „маскирајќи ги“ ирелевантните делови од сликата.
3. *Дефинирање на регионот на интерес (ROI):* Дефинираме полигонална форма што ја опишува областа на сликата на која сакаме да се фокусираме. Овој многуаголник се одредува со обезбедување на координатите на неговите темиња, кои ги формираат границите на регионот кој не интересира. На пример, при откривање лента, може да дефинираме трапезоиден ROI што го покрива патот пред возилото.

Во пракса тоа би изгледало вака:

Слика 7. Сликата добиена после селектирање на региони на интерес

Детектирање на коловозните линии со Hough трансформација

Hough трансформацијата е моќна техника за обработка на слики што се користи за откривање прави линии дури и кога тие линии се скршени, нецелосни или заматени од шум.

Оваа техника функционира на следниот начин:

1. Вообичаено, првиот чекор во откривањето на Hough Line Detection е да се изврши откривање на рабовите на влезната слика користејќи техники како што е детекторот за рабови Canny. Ова ги идентификува точките на сликата каде што се случуваат значителни промени во интензитетот, кои често одговараат на рабовите на предметите или линиите.
2. Претставување на просторот на параметрите: Клучниот концепт зад Hough Transform е да се прикаже секоја откриена рабна точка во параметарскиот простор. За откривање на линии, параметарскиот простор често се дефинира со два параметри: аголот (θ) на правата и растојанието (ρ) од потеклото до најблиската точка на линијата.
3. Акумулаторна низа: дводимензионална податочна структура наречена „акумулаторна низа“ се користи за следење на параметрите (θ, ρ) кои одговараат на откриените рабови. Оваа низа обично се иницијализира со сите вредности поставени на нула.
4. Гласање: за секоја рабна точка на сликата откриена со раб, алгоритмот Hough Transform врши процес на гласање. За секоја рабна точка, тој разгледува опсег на можни вредности за θ и ρ и постепено акумулира гласови во низата на акумулаторот на соодветните (θ, ρ) позиции. Секоја рабна точка придонесува со гласови за комбинациите на параметри кои потенцијално би можеле да претставуваат линија што минува низ таа точка.
5. Наоѓање врвови: По обработката на сите рабни точки, низата на акумулаторот содржи врвови или локални максими. Овие врвови ги претставуваат параметрите (θ, ρ) кои одговараат на откриените линии на сликата. Локацијата и вредноста на овие врвови ја означуваат ориентацијата и локацијата на линиите.
6. Екстракција на линии: Откако ќе се идентификуваат врвовите во низата на акумулаторот, Hough Transform може да се користи за извлекување линии од сликата. Овие линии се претставени со нивните параметри (θ, ρ).

Следната линија код претставува срцето на нашиот алгоритам.

```python
def hough_lines(image):
    lines = cv2.HoughLinesP(image, 1, np.pi / 180, threshold=10, minLineLength=15, maxLineGap=2)

    return lines
```

Ги содржи следните параметри:

Параметар 1 - Изолираните градиенти (сликата добиена во претходниот чекор)

Параметар 2 - Овој параметар е резолуцијата на акумулаторот во пиксели. Во Hough Transform, тој ја претставува резолуцијата на растојанието на акумулаторот во истата единица како и сликата (обично пиксели). Во овој случај, таа е поставена на 1, што значи точност од еден пиксел при откривање линии.

Параметар 3: np.pi / 180 - Овој параметар ја претставува аголната резолуција на акумулаторот во радијани. Ја одредува аголната точност за откривање линии. np.pi / 180 ги претвора степените во радијани и е поставен на мал раст на аголот, што покажува дека ќе се откријат линии со мали варијации во аглите.

Параметар 4: treshold=10 - Параметарот treshold е клучен. Го дефинира минималниот број гласови (или пресеци во просторот на параметарот Hough) потребен за една линија да се смета за валидна линија и да биде вклучена во излезот. Зголемувањето на оваа вредност може да го направи алгоритмот поселективен при изборот на линии. Пониските вредности може да резултираат со откривање на повеќе линии, вклучително потенцијално бучни или лажни линии.

Параметар 5: minLineLength=15 - Овој параметар ја одредува минималната должина на линијата во пиксели што ќе се смета за откриена линија. Линиите пократки од оваа вредност ќе бидат отфрлени. Прилагодувањето на овој параметар ви овозможува да ги филтрирате сегментите на кратки линии што можеби не се релевантни.

Параметар 6: maxLineGap=2 - Параметарот maxLineGap го одредува максималниот дозволен простор помеѓу сегментите за да ги третира како една линија. Со други зборови, ако два линиски сегменти се доволно блиску и просторот меѓу нив е помал или еднаков на оваа вредност, тие ќе се поврзат за да формираат подолга линија. Ова помага при поврзување на скршени или раздвоени сегменти на линија.

Цртање на линиите врз оригиналната рамка

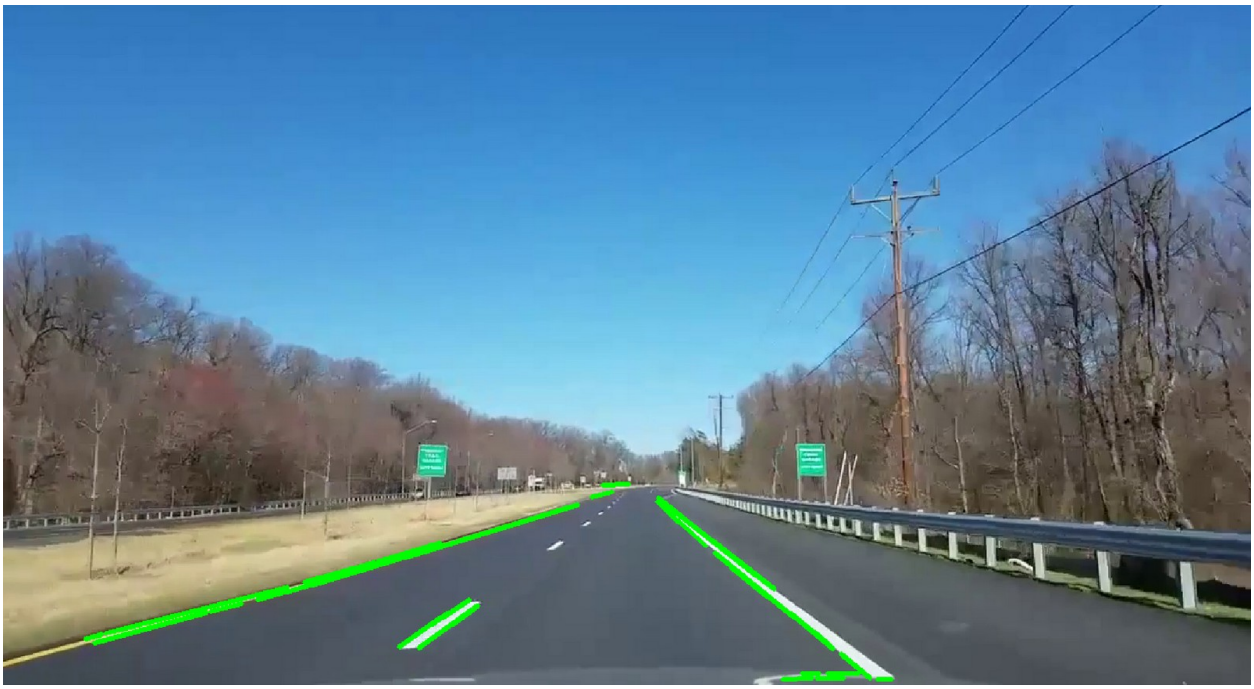
```python
def draw_lines(image, hough_lines):
    for line in hough_lines:
        x1, y1, x2, y2 = line[0]
        cv2.line(image, (x1, y1), (x2, y2), (0, 255, 0), 4)

    return image
```

Оваа функција ни служи за цртање на веќе детектираните линии врз оригиналната рамка. Функцијата прима 2 параметри:

1. Првиот параметар ни претставува оригиналната слика на која ќе ги цртаме линиите пронајдени со Hough трансформацијата.
2. Листа на линиите добиени од Hough трансформацијата. Тие се претставени преку четири вредности:
    - x_1 и y_1 кои ги претставуваат координатите на почетната точка на линијата.
    - x_2 и y_2 кои ги претставуваат координатите на крајната точка на линијата.

Функцијата ги изминува сите линии и со помош на cv2.line се исцртува секоја линија. Оваа функција како параметри ги прима позициите на точките на линијата, бојата со која треба да ја исцртаме линијата во RGB формат, како и нејзината дебелина.

После цртањето на линиите врз една рамка, го добиваме следниот резултат:



Слика 8. Сликата добиена со цртање на линиите

Процесирање на видеото

```
def process(img):
    height = img.shape[0]
    width = img.shape[1]
    roi_vertices = [
        (0, 650),
        (2*width/4.2, 2*height/3),
        (width, 1000)
    ]

    gray_img = grayscale(img)

    blur = gaussian_blur(gray_img)
    canny = canny_img(blur)

    roi_img = roi(canny, np.array([roi_vertices], np.int32))

    lines = hough_lines(roi_img)

    final_img = draw_lines(img, lines)

    return final_img
```

Оваа функција ни служи за извршување на целиот алгоритам врз една рамка од видеото. Со неа се извршуваат сите потребни чекори кои ги наведовме претходно.

```
cap = cv2.VideoCapture("./lane_vid2.mp4")

frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
fourcc = cv2.VideoWriter_fourcc(*"XVID")
saved_frame = cv2.VideoWriter("lane_detection.avi", fourcc, 30.0, (frame_width, frame_height))

while cap.isOpened():
    ret, frame = cap.read()

    try:
        frame = process(frame)

        saved_frame.write(frame)
        cv2.imshow("frame", frame)

        if cv2.waitKey(1) & 0xFF == 27:
            break

    except Exception:
        break

cap.release()
saved_frame.release()
cv2.destroyAllWindows()
```

Последното парче код служи за читање на изворното видео врз кое што сакаме да ја извршиме детекцијата на коловозните ленти. Потоа се дефинираат некои параметри како големина на прозорецот каде што ќе се прикаже крајното видео. И на крај со помош на циклус се изминуваат сите рамки од изворното видео. Секоја таква рамка се процесира со помош на process функцијата и таквата испроцесирана рамка се прикажува на екран.

Бидејќи ваквото процесирање се извршува во реално време, она што е прикажано на екранот претставува форма на видео.

Заклучок

Алгоритмот за детекција на коловозни ленти заснован на обработка на слики обезбедува робусно и во реално време решение за идентификување на коловозни ленти преку слики од патишта или видео преноси. Тој игра клучна улога во овозможувањето на автономните возила безбедно да се движат и да останат во нивните ленти. Понатамошните истражувања во оваа област се фокусираат на подобрување на перформансите на алгоритмот при предизвикувачки услови, како што се неповолните временски услови и различните услови на осветлување, за да се зголеми доверливоста на системите за автономно возење.

Користена литература

[1] https://www.isahit.com/blog/why-to-use-grayscale-conversion-during-image-processing

[2] https://medium.com/swlh/computer-vision-lane-finding-through-image-processing-516797e59714

[3] https://www.educative.io/answers/what-is-gaussian-blur-in-image-processing

[4] https://en.wikipedia.org/wiki/Canny_edge_detector

[5] https://www.mathworks.com/help/images/roi-based-processing.html

[6] https://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm

[7] https://en.wikipedia.org/wiki/Hough_transform