



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA AUTOMATYKI I ROBOTYKI

Praca dyplomowa magisterska

Ewaluacja i integracja algorytmów wizyjnych stosowanych w pojazdach autonomicznych z użyciem symulatora jazdy samochodem

Evaluation and integration of vision algorithms used in autonomous vehicles with the use of a driving simulator

Autor: *Jarosław Borzęcki*
Kierunek studiów: *Automatyka i robotyka*
Opiekun pracy: *dr inż. Tomasz Kryjak*

Kraków, 2019

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpozna bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, videogram lub nadanie.”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Serdecznie dziękuję wszystkim.

Spis treści

1. Wstęp.....	7
1.1. Cele i założenia.....	7
1.2. Istniejące systemy.....	7
1.3. Zawartość pracy.....	7
2. Systemy wizyjne w pojazdach autonomicznych.....	9
2.1. Poziomy autonomiczności.....	9
2.1.1. Poziom zerowy - brak autonomiczności	9
2.1.2. Poziom pierwszy - asysty kierowcy	10
2.1.3. Poziom drugi - częściowa automatyzacja jazdy samochodem	10
2.1.4. Poziom trzeci.....	10
2.1.5. Poziom czwarty	11
2.1.6. Poziom piąty - pełna autonomia.....	11
2.1.7. Autonomiczne ciężarówki.....	11
2.2. Wykrywanie pasa ruchu.....	12
2.2.1. Detekcja zakrętów	15
2.3. Detekcja znaków drogowych.....	16
2.3.1. Radial Symmetry Transform.....	16
2.4. Detekcja świateł drogowych.....	19
2.4.1. Detekcja świateł drogowych z użyciem informacji o kolorze i krawędziach	19
2.4.2. Wnioski i rezultaty	21
2.5. Detekcja samochodu poprzedzającego.....	22
2.5.1. Przebieg algorytmu bazującego na operatorze symetrii.....	23
2.6. Opis wybranych zagadnień i algorytmów przetwarzania obrazu	25
2.6.1. Transformata Hougha dla okręgów	25
2.6.2. Przestrzenie barw	27
2.6.3. Filtr Canny'ego	27
3. Realizacja projektu	29

3.1.	Euro Truck Simulator 2	29
3.2.	Architektura systemu.....	30
3.2.1.	Przechwytywanie obrazu	31
3.2.2.	Przetwarzanie i analiza obrazu.....	31
3.2.3.	Symulacja kontrolera	31
3.2.4.	Wyświetlanie rezultatów	32
3.3.	Opis implementacji algorytmów wizyjnych użytych w symulatorze.....	32
3.3.1.	Algorytm detekcji linii	32
3.3.2.	Sytuacje potencjalnie problematyczne.....	34
3.3.3.	Algorytm detekcji czerwonych świateł drogowych	34
3.3.4.	Sytuacje potencjalnie problematyczne.....	36
3.3.5.	Detekcja samochodu poprzedzającego	39
3.4.	Badania wydajności aplikacji	42
3.5.	Ewaluacja systemu	43
4.	Podsumowanie	45
A.	Instrukcja do ćwiczenia	47
A.1.	Obsługa kontrolera	49
A.2.	Obsługa konsoli	50

1. Wstęp

1.1. Cele i założenia

Niniejsza praca powstała w celu stworzenia aplikacji umożliwiającej testowanie algorytmów wizyjnych za pomocą symulatora ciężarówki, która mogłaby być wykorzystana w ramach zajęć z Systemów Wizyjnych w Pojazdach Autonomicznych prowadzonych w ramach kierunku Automatyka i Robotyka na Akademii Górnictwo-Hutniczej.

1.2. Istniejące systemy

Aplikacje służące do testowania algorytmów wizyjnych są używane w przemyśle. Obecnie produkowane samochody są wyposażone w pewne asysty, które bazując na widoku z kamery pomagają prowadzić samochód. Testowanie tych algorytmów jest niezwykle istotne, ponieważ niewykrycie błędu prowadzi do katastrofalnych skutków, w tym potencjalnej śmierci człowieka. Niewystarczające testowanie algorytmów w komercyjnym samochodzie Tesla 3 doprowadziło do śmiertelnego wypadku, ponieważ systemy jazdy autonomicznej nie wykryły poprawnie naczepy ciężarówki.

1.3. Zawartość pracy

Praca składa się z następujących części. Rozdział drugi rozpoczyna się opisem poziomów autonomiczności samochodów. Zawarte w nim jest porównanie w odniesieniu do stosowanych systemów wizyjnych i innych asystentów jazdy. Następnie opisane są wybrane algorytmy cyfrowego przetwarzania obrazów takie jak: detekcja pasa ruchu, detekcja świateł drogowych, wykrywanie znaków oraz detekcja samochodu poprzedzającego. Opis jest oparty na uznanym w świecie naukowym artykułach. Rozdział kończy krótki opis podstawowych operacji cyfrowego przetwarzania obrazów, których wyjaśnienie uznano za istotne z uwagi na częste stosowanie w systemach wizyjnych w pojazdach autonomicznych.

Następny rozdział zawiera opis realizacji projektu. Na początku dokonana jest ewaluacja dostępnych symulatorów jazdy wraz z uzasadnieniem wyboru Euro Truck Simulator 2. Kolejno jest opisana architektura zaimplementowanej aplikacji. Następna część mówi o przykładowych algorytmach zaimplementowanych w celu przetestowania i weryfikacji aplikacji stworzonej w ramach pracy. Końcowa część to sprawdzenie wydajności aplikacji w różnych sytuacjach.

Ostatni rozdział zawiera krótkie podsumowanie oraz ocenę przydatności aplikacji do prowadzenia zajęć dydaktycznych.

2. Systemy wizyjne w pojazdach autonomicznych

Współczesne pojazdy autonomiczne wykorzystują w szerokim zakresie systemy wizyjne do analizy otoczenia. Jednym z pierwszych zastosowań algorytmów cyfrowego przetwarzania obrazów była detekcja pasa ruchu, która nie służyła do sterowania samochodem, lecz miała za zadanie wspomagać kierowcę w sytuacjach zmęczenia lub utraty koncentracji. Wraz z rozwojem technologii pojazdów autonomicznych, zaawansowanie systemów wizyjnych rosło od wspomnianej kontroli pasa ruchu, poprzez rozpoznawanie znaków drogowych i świateł ulicznych, na detekcji pieszych kończąc. Ważnym elementem o którym należy wspomnieć, że istotne znaczenie, oprócz algorytmów detekcji, ma sposób interpretacji danych odczytyanych z otoczenia. W kolejnych podrozdziałach zostaną opisane algorytmy wizyjne, które mogłyby być zastosowane w systemach w pojazdach autonomicznych.

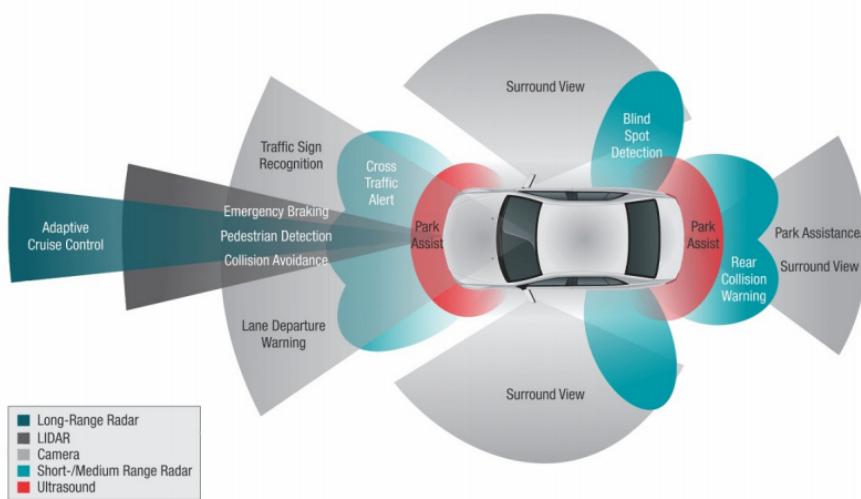
2.1. Poziomy autonomiczności

Całość systemów, które wspomagają kierowcę w trakcie jazdy nazwano ADAS (Advanced Driver Assistance Systems). Aby usystematyzować i podzielić poziom wpływu systemów na jazdę wprowadzono poziomy autonomiczności jazdy.

Na rysunku 2.1 daje się zauważyć znaczna liczba kamer i radarów wspomagająca kierowcę. Powszechną techniką jest stosowanie redundancji w krytycznych systemach, co jest regulowane poprzez normy ADAS (Advanced Driver Assistance Systems)

2.1.1. Poziom zerowy - brak autonomiczności

Obecnie większość samochodów na drodze zawiera się w tym poziomie. Człowiek wpływa na jazdę, chociaż mogą pojawiać się proste systemy, które mogą pomóc kierowcy np. system awaryjnego hamowania. Dopóki nie wpływa on na tor jazdy, nie jest to system autonomiczny. Innym przykładem jest system ABS, który również nie jest systemem, który zapewniałby autonomiczność pojazdowi. Służy on poprawie bezpieczeństwa i jego działanie opiera się tylko na odczytce danych z czujników niezależnie od aktualnej sytuacji na drodze i wokół pojazdu.



Rys. 2.1. Ogólny schemat kamer i radarów we współczesnych pojazdach autonomicznych[S1]

2.1.2. Poziom pierwszy - asysty kierowcy

Jest to najniższy poziom autonomiczności. Auto posiada pojedynczy system wspomagania kierowcy taki jak sterowanie lub przyspieszanie (tempomat). Adaptacyjny tempomat, czyli system, który zachowuje bezpieczny dystans od poprzedzającego pojazdu kwalifikuje się jako poziom pierwszy, ponieważ człowiek kontroluje pozostałe aspekty jazdy samochodem takie jak kierowanie i hamowanie.

2.1.3. Poziom drugi - częściowa automatyzacja jazdy samochodem

Oznacza zaawansowane asysty kierowcy. Samochód może sam kontrolować zarówno sterowanie i przyspieszanie oraz hamowanie. Jest w stanie jechać samodzielnie, lecz wymaga ciągłej obecności kierowcy za kierownicą, który może w każdej chwili przejąć kontrolę nad samochodem. Obecnie stosowane systemy jazdy autonomicznej takie jak np. Tesla Autopilot kwalifikują się jako poziom drugi.

2.1.4. Poziom trzeci

Różnica w stosunku do poziomu drugiego jest subtelna. Samochód posiada możliwość detekcji otaczającego go środowiska i na podstawie zgromadzonych informacji samodzielnie podejmować decyzje. System jest jednak wciąż zależny od człowieka, który musi pozostać czujny i być w stanie zareagować, gdy system nie będzie w stanie podjąć decyzji. W 2019r. Audi wprowadziło model A8, który zapowiadano jako pierwszy samochód poziomu trzeciego. System Traffic Jam Pilot bazując na danych z lidaru oraz kamer zapewniał autonomiczną jazdę w korkach. Problemem okazało się nieprzygotowanie prawne. W Stanach Zjednoczonych stosowanie tego systemu było zabronione, więc samochód w określonej wersji sprzedawano jako pojazd autonomiczny drugiego poziomu. W Europie samochód pojawił się z zaimplementowaną funkcjonalnością asystenta jazdy w korku.



Rys. 2.2. Tesla Model S - pierwszy samochód z drugim poziomem zaawansowania systemów wspomagania kierowcy (*źródło: Wikipedia*)

2.1.5. Poziom czwarty

Poziom czwarty zapewnia to czego brakowało w niższych poziomach, a więc może interweniować jeśli coś pójdzie nie tak (np. nagły wypadek, pęknięcie opony). Te samochody nie wymagają akcji w człowieka w znakomitej większości sytuacji, jednak człowiek zawsze jest w stanie przejąć kontrolę. Podobnie jak w poziomie trzecim problemem okazały się ograniczenia prawne. Samochody te mogą z reguły poruszać się na ograniczonym obszarze. Obecnie w fazie testów są samochody takie jak Waymo lub NAVYA

2.1.6. Poziom piąty - pełna autonomia

Samochody nie wymagają uwagi człowieka. Najprawdopodobniej nie będą wyposażone w kierownicę ani pedały. Będą w stanie jechać gdziekolwiek i robić to co jest w stanie zrobić doświadczony kierowca. Samochody te są obecnie w stanie wczesnych testów, jednak można się spodziewać, że w ciągu najbliższych lat pierwsze w pełni autonomiczne samochody będą pojawiać się na drogach.

2.1.7. Autonomiczne ciężarówki

Warto zaznaczyć, że ważnym polem do zastosowania systemów autonomicznej jazdy jest transport. Systemy wspomagające kierowcę pojawiają się analogicznie do samochodów osobowych. W najbliższych latach planuje się rozwijanie koncepcji autonomicznych ciężarówek poprzez tworzenie konwojów ciężarówek, w których kierowca znajduje się tylko w pierwszym samochodzie. Kolejnym krokiem będzie autonomiczna jazda na dystansie trasy ze wsparciem kierowcy na załadunku i rozładunku. Ostatnim



Rys. 2.3. Przykładowy obraz wejściowy algorytmu detekcji pasa ruchu

krokiem, podobnie jak w przypadku samochodów osobowych będzie w pełni autonomiczna jazda bez udziału człowieka.

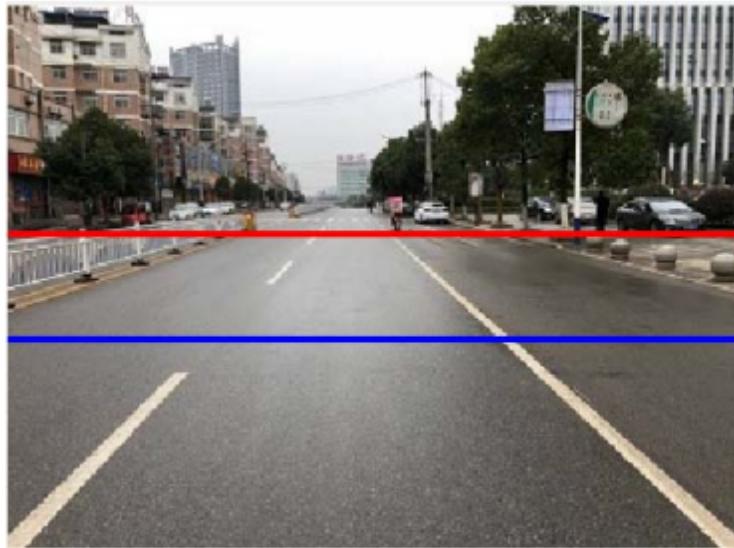
Systemy wizyjne stosowane w ciężarówkach nie odbiegają sposobem pracy od tych używanych w samochodach osobowych.

2.2. Wykrywanie pasa ruchu

W poniższej i kolejnych sekcjach zostaną opisane algorytmy wizyjne, których zadaniem jest detekcja i analiza otoczenia samochodu. W ostatnich latach dynamicznie rozwijającą się dziedziną w systemach wizyjnych i motoryzacji są sieci neuronowe, które są obecnie najczęściej wybierane do implementacji w samochodach. Ze względu na edukacyjny charakter pracy opisane zostaną algorytmy bazujące na klasycznych metodach przetwarzania obrazach cyfrowych. W niektórych metodach zostaną użyte proste techniki uczenia maszynowego.

Wykrywanie pasa ruchu było jednym z pierwszych badanych algorytmów. W jeździe kluczowym elementem jest utrzymanie samochodu w pasie jezdni niezależnie od stanu nawierzchni, pogody i predkości.

W najczęstszym przypadku obrazem wejściowym do algorytmu jest obraz z kamery umieszczonej pod pewnym kątem do nawierzchni zamontowanej w okolicy przedniego zderzaka lub atrapy chłodnicy. W niniejszej pracy, większość przetwarzanych obrazów pochodzi z symulatora Euro Truck Simulator 2, gry komputerowej, która posiada zaawansowaną grafikę, która przypomina otaczającą rzeczywistość,



Rys. 2.4. Linie obrazujące granice obszaru poddawanego analizie[T3]

a także udostępnia możliwości programistyczne, które ułatwiają stworzenie aplikacji do testowania algorytmów wizyjnych co jest założeniem niniejszej pracy. W niniejszej sekcji zostanie opisany algorytm z artykułu [T3] Autor korzysta w nim z podstawowych operacji przetwarzania obrazów.

Pierwszym krokiem jest wyodrębnienie z obrazu ROI¹. Ma to na celu ograniczenie ilości danych poddawanych analizie, a co bardziej istotne odrzuca te fragmenty obrazu, na których na pewno nie będzie jezdni (powyżej czerwonej linii), a także te gdzie obraz linii mógłby być zakłócony. Warto zauważyć, że pomiędzy liniami - czerwoną i niebieską znajduje się prawie dwa razy więcej długości drogi niż w obszarze pod niebieską linią.

Kolejnym etapem jest ekstrakcja pasów ruchu. W większości krajów mają one kolor biały, rzadziej żółty. Są to kolory, które na drodze można rzadko zauważać w innej funkcji niż malowanie znaków poziomych, w tym linii. Mając na wejściu obraz RGB² można zauważyc, że składowe czerwona i zielona mają większe wartości tam gdzie na obrazie są linie w porównaniu do standardowej nawierzchni jezdni. W artykule, który opisuje ta sekcja zaproponowano następującą metodę segmentacji linii:

$$IM(i, j) = \begin{cases} 255, & R(i, j) \geq (0.2R_{min} + 0.8R_{max}) \\ & G(i, j) \geq (0.2G_{min} + 0.8G_{max}) \\ 0, & wpp^3. \end{cases} \quad (2.1)$$

$$G(i, j) = \begin{cases} 255, & R(i, j) \geq G(i, j) \geq B(i, j) \\ & IM(i, j) > 0 \\ 0, & wpp. \end{cases} \quad (2.2)$$

$$Gray(i, j) = R(i, j) + G(i, j) - 2B(i, j) + 0.3 * 8|R(i, j) + G(i, j)| \quad (2.3)$$

¹ROI (ang.) - Region of interest - obszar obrazu poddawany analizie i dalszemu przetwarzaniu

²RGB - obraz o trzech składowych barwnych czerwonej(R), zielonej(G) i niebieskiej(B)



Rys. 2.5. Obraz GM – wysegmentowane linie[T3]

$$GM(i, j) = \begin{cases} 128, & Gray(i, j) \geq 0.8 * Gray_m \\ & 2 * Gray_{avg} \leq Gray(i, j) \leq Gray_m \\ 255, & \text{albo} \\ & G(i, j) = 255 \\ 0, & wpp. \end{cases} \quad (2.4)$$

W równaniu (2.1) $IM(i, j)$ oznacza tymczasową macierz cech koloru. $R(i, j)$ oznacza jasność składowej czerwonej, a $G(i, j)$ składową zieloną. R_{max} , R_{min} , G_{max} , G_{min} reprezentują maksymalną i minimalną wartość składowej czerwonej, a także maksymalną i minimalną wartość składowej zielonej. W równaniu (2.3) $Gray(i, j)$ wskazuje na obraz wejściowy w skali szarości, na wartości którego ma wpływ każda ze składowych barwnych. W ostatnim równaniu (2.4) $GM(i, j)$ oznacza obraz wynikowy, na którym zaznaczone jest wysegmentowane poziome oznaczenie jezdni. $Gray_{avg}$ to średnia wartość obrazu w skali szarości w danym wierszu, natomiast $Gray_m(i, j)$ oznacza wartość maksymalną dla danego wiersza.

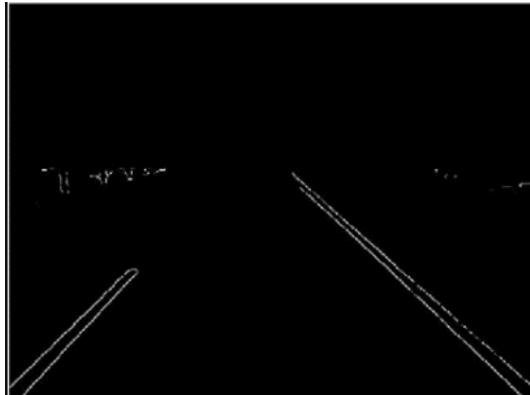
Kolejnym etapem opisywanego algorytmu jest wykrywanie krawędzi, tu zrealizowane za pomocą filtru Canny'ego. Jest to filtr nielinowy z histerezą. Dobrze sprawdza się do wykrywania krawędzi na obrazach niejednorodnych i rozmytych. Następnie podejmowana jest ekstrakcja cech linii. Na potrzeby wyjaśnienia działania podanego fragmentu algorytmu przyjęto:

- IME – Obraz z równania (2.4) po filtracji filtrem Canny'ego
- IMC – Obraz GM z równania (2.4)

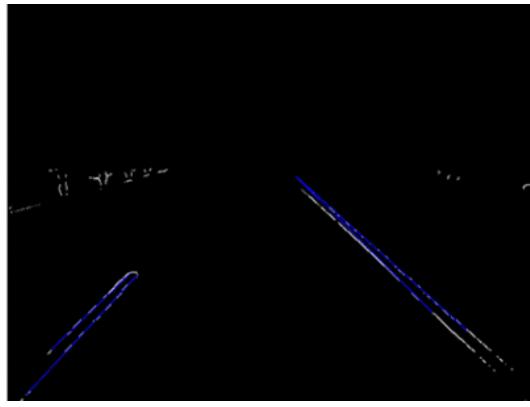
Dla każdego piksela w obrazie IME który został oznaczony jako krawędź zostaje sprawdzana następująca zależność:

$$\begin{aligned} & IMC(i, j + 1) + IMC(i, j) + IMC(i, j - 1) + IMC(i - 1, j + 1) \\ & + IMC(i - 1, j) + IMC(i - 1, j - 1) + IMC(i + 1, j + 1) + IMC(i + 1, j - 1) > 0 \end{aligned} \quad (2.5)$$

Jeśli jest ona spełniona, wybrany punkt na obrazie IME zostaje uznany jako krawędź pasa jezdni.



Rys. 2.6. Obraz po zastosowaniu filtra Canny'ego oraz ekstrakcji cech linii [T3]



Rys. 2.7. Wynik algorytmu detekcji linii [T3]

Do ostatecznego wykrycia linii prostych należy użyć zmodyfikowanej transformaty Hougha (*constraint Hough transform*). Główna różnica pomiędzy nią, a klasyczną transformatą Hougha polega na tym, że wartości ρ i θ są skwantowane i pogrupowane. Dzięki takiej modyfikacji linie które leżą blisko siebie lub w przypadku nieidalnie prostej linii redukuje się prawdopodobieństwo wielokrotnej detekcji prostej.

2.2.1. Detekcja zakrętów

Podobnym zadaniem jest detekcja pasa ruchu w zakręcie. Kształt zakrzywionej linii może być opisany za pomocą paraboli z równania ???. Zauważono, że fragment drogi bliżej samochodu z reguły jest prosty, zakrzywienie widać powyżej pewnej odległości, a na obrazie powyżej pewnej wysokości. Równanie ?? pokazuje model pasa ruchu, gdzie y_m to współrzędna punktu, w którym prosta przechodzi w krzywą.

$$x = a + by + cy^2 \quad (2.6)$$

$$x = \begin{cases} a + by, & y \leq y_m \\ c + dy + ey^2, & y > y_m \end{cases} \quad (2.7)$$

gdzie ponadto:

- a, b – parametry linii
- c, d, e – parametry krzywej

Linie prosta i krzywa powinny spotykać się w jednym punkcie, którego współrzędna wysokościowa jest równa y_m . Po pewnych przekształceniach otrzymano równanie ?? i ??.

$$a + by_m = c + dy_m + ey_m^2 \quad (2.8)$$

$$b = d + 2ey_m \quad (2.9)$$

Ostatecznie uzyskano zależność ?? pomiędzy parametrami krzywych.

$$\begin{cases} c = a + \frac{y_m}{2}(b - d) \\ e = \frac{1}{2y_m}(b - d) \end{cases} \quad (2.10)$$

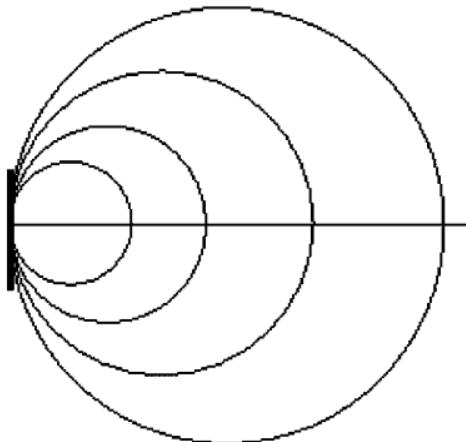
2.3. Detekcja znaków drogowych

Innym w swojej naturze zadaniem stawianym przed systemami wizyjnymi w pojazdach autonomicznych jest detekcja znaków drogowych. Systemy tego typu pojawiały się dość wcześnie w samochodach pełniąc jedynie funkcję ostrzegawczo-informacyjną. Coraz bardziej dynamicznie rozwijające się systemy wspomagające kierowcę wymagają od subsystemu odpowiedzialnego za detekcję znaków drogowych dużej skuteczności, ponieważ na ich detekcjach opierają decyzje o sterowaniu pojazdem. Istotnym problemem jest brak ujednolionego zestawu znaków dla całego świata. Obecnie każdy kraj posiada swój zestaw znaków, które w ogólności są podobne, jednak różnice w szczegółach sprawiają problemy systemom wizyjnym. Proponowanym rozwiązaniem jest zbudowanie odpowiednio dużej bazy wzorców znaków i stosowanie określonego zestawu w zależności od lokalizacji.

Rozwiążanie proponowane w [T2] składa się z dwóch etapów. Pierwszy to detekcja znaku, a drugi to jego klasyfikacja. Do wykrycia znaku drogowego na obrazie pochodzący z kamery umieszczonej w samochodzie użyto *Radial Symmetry Transform* do detekcji znaków ograniczenia prędkości. Obecnie stowarzyszone metody detekcji bazują na segmentacji koloru lub kształtu.

2.3.1. Radial Symmetry Transform

Klasyczne detektory kształtu wymagają często zamkniętych konturów. Odporne techniki takie jak transformata Hougha dla kół wymaga dużych nakładów obliczeniowych dla dużych obrazów. *Fast radial symmetry detector* - (ang. Szybki radialny detektor symetrii) może być używany jako detektor w czasie rzeczywistym. Znakomita większość znaków z ograniczeniem prędkości to koło z czerwonym brzegiem i wartością ograniczenia w środku na białym tle. Opisywana metoda detekcji jest kompatybilna ze wszystkimi głównymi metodami klasyfikacji takimi jak np. SVM (ang. Support Vector Machine



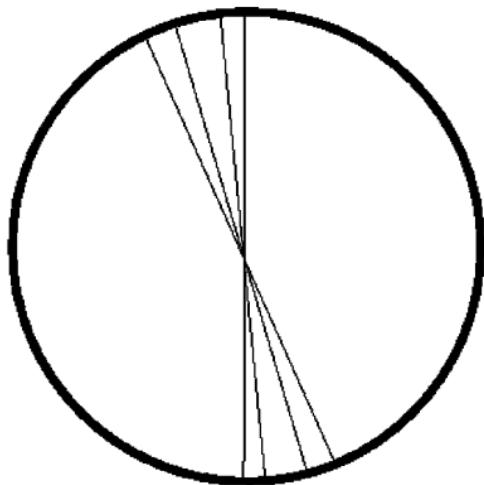
Rys. 2.8. Algorytm głosowania - dla danego punktu krawędzi środki rodzin krawędzi leżą na linii prostopadłej do krawędzi [T2]

- maszyna wektorów wspierających). Główną zaletą użycia opisywanej metody jest fakt, że wraz z informacją o wykrytym znaku podawana jest skala znaku, co znaczowo ułatwia klasyfikację, ponieważ niepotrzebne staje się używanie szablonów o wielu rozmiarach dla wielu różnych rozdzielcości.

Szybki radialny detektor symetrii jest wariantem transformaty Hougha dla wyszukiwania okręgów. Jest on wykonywany w porządku kp , gdzie k oznacza liczbę promieni, które są szukane, a p liczbę pikseli. Stanowi to różnicę w stosunku do klasycznej transformaty Hougha wykonywanej w porządku kbp , gdzie każdy piksel na obrazie krawędzi „głosuje” dla każdego koła z dyskretnego zestawu promieni. b oznacza dyskretyzację zestawu promieni okręgów, które mogą przechodzić przez aktualnie analizowany punkt.

FRSD eliminuje czynnik b poprzez pozyskanie informacji o kierunku gradientu z detektora krawędzi Sobela. Zamiast sprawdzania każdego możliwego kierunku promienia, sprawdzany jest jedynie kierunek prostopadły do kierunku gradientu co jest widoczne na rysunku 2.8. Powoduje to, że przestrzeń rozwiązań z trójwymiarowej staje się dwuwymiarowa, co pozwala na używanie algorytmu w czasie rzeczywistym. Operacja radialnej detekcji symetrii może być prosto zrozumiana jako rozważenie wszystkich możliwych okręgów, których dany piksel może być częścią, jeżeli znany jest kierunek krawędzi, to okręgi są szukane tylko na linii prostopadłej do krawędzi (rys. 2.9).

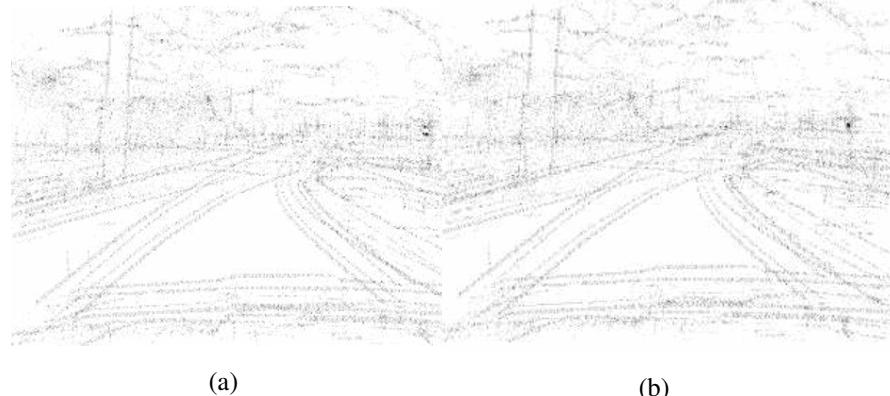
Praktyczna realizacja jest wykonywana na obrazie cyfrowym, więc promień jest podzielony na kilka zakresów długości. Istotne jest poprawne dobranie ograniczeń na długość promienia. Na rysunku 2.10 po prawej stronie jest widoczny znak ograniczenia prędkości, który powinien zostać wykryty. Daje się zauważyć, że można dobrać odpowiednie zakresy promienia dla wykrywanych znaków, a także wyodrębnić obszar na którym znaki na pewno nie będą się pojawiać, a także taki, na którym należy wykonać detekcję.



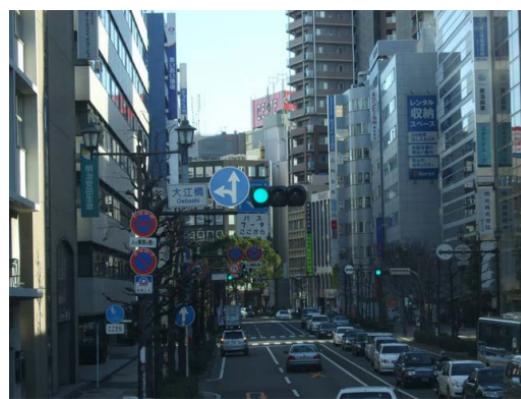
Rys. 2.9. Zachowanie FRSD dla okręgu, zauważalne przecięcie średnic skutkuje istnieniem maksimum lokalnego w środku okręgu.[T2]



Rys. 2.10. Obraz wejściowy algorytmu detekcji znaków drogowych[T2]



Rys. 2.11. Wyniki detekcji, gdy zakres promieni jest zbyt mały (a) i zbyt duży (b). [T2]



Rys. 2.12. Obraz wejściowy algorytmu detekcji świateł drogowych [T4]

2.4. Detekcja świateł drogowych

Kolejnym istotnym elementem systemów umieszczanych w pojazdach autonomicznych jest detekcja świateł drogowych. Informują one o możliwości przejazdu przez skrzyżowanie lub rondo i możliwości znalezienia się na trasie kolizyjnej w stosunku do innych użytkowników drogi. Światła drogowe spotykane są głównie w miastach, lecz wraz z rozwojem infrastruktury widywane są w mniejszych miejscowościach. Słupy z zamontowanymi światłami mogą być widoczne na prawej lub lewej krawędzi jezdni, a także nad nią.

2.4.1. Detekcja świateł drogowych z użyciem informacji o kolorze i krawędziach

Światła drogowe na świecie są ustandaryzowane. Istnieją trzy kolory: czerwony, żółty i zielony. Każdy kolor niesie za sobą informację: czerwony - stój, żółty - przygotuj się do zmiany z zielonego na czerwony lub odwrotnie i zielony, który oznacza jedź. Jedyną znaną odchyłką jest odpowiednik światła żółtego w Stanach Zjednoczonych, który jest pomarańczowy.

Masako Omachi w artykule [T4] proponuje algorytm, który bazuje na informacji o kolorze i krawędziach znajdujących się na obrazie. Światła drogowe mają z góry ustalony kształt - są okrągłe. Istnieją warianty



Rys. 2.13. Efekt normalizacji przestrzeni barw[T4]

ze strzałkami, lecz opisywany poniżej algorytm służy do detekcji świateł, które w [Kodeks] mają kształt pełnego koła.

Rysunek 2.12 pokazuje przykład sceny zawierającej światła drogowe. W opisywanej metodzie przestrzeń barw jest konwertowana do znormalizowanej przestrzeni RGB. Normalizacja przestrzeni RGB polega na zmapowaniu wartości pikseli do przedziału [0, 255]. A także „rozsunięciu” wartości pikseli na obrazie tak, by znajdowały się w całym możliwym zakresie wartości:

$$R = \begin{cases} 0, & s = 0 \\ \frac{r}{s} & wp.p. \end{cases} \quad (2.11)$$

$$G = \begin{cases} 0, & s = 0 \\ \frac{g}{s} & wp.p. \end{cases} \quad (2.12)$$

$$B = \begin{cases} 0, & s = 0 \\ \frac{b}{s} & wp.p. \end{cases} \quad (2.13)$$

gdzie:

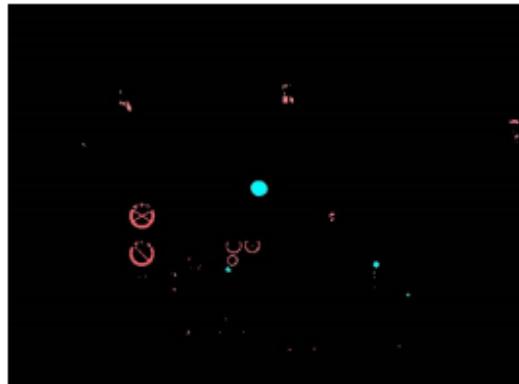
- r, g, b – składowe czerwona, zielona i niebieska nieznormalizowanego obrazu,
- $s = r + g + b$
- R, G, B – składowe czerwona, zielona i niebieska znormalizowanego obrazu.

Efekt przeniesienia przestrzeni barw do znormalizowanej przestrzeni RGB jest widoczny na rysunku 2.13. Następnie poprzez progowanie każdej ze składowych uzyskuje się kandydatów do detekcji świateł drogowych. Decyzja o tym czy piksel należy lub nie do światła drogowego jest podejmowana na podstawie poniższych warunków:

$$R > 200 \wedge G < 150 \wedge B < 150 \quad (2.14)$$

lub

$$R > 200 \wedge G > 150 \wedge B < 150 \quad (2.15)$$



Rys. 2.14. Wynik progowania w celu wykrycia obszarów będących kandydatami do bycia światłami drogowymi [T4]

Tabela 2.1. Porównanie klasycznego algorytmu detekcji świateł i opisanego w tej pracy [T4]

	Algorytm klasyczny	Algorytm opisany w tym rozdziale
Dokładność	20/30	26/30
Czas przetwarzania [s]	0.561	0.347

lub

$$R < 150 \wedge G > 240 \wedge B > 220 \quad (2.16)$$

Rezultat progowania jest widoczny na rysunku 2.14. Następnym krokiem jest wykrycie krawędzi na obrazie z kandydatami do detekcji świateł. Jedną z opcji jest użycie filtru Sobela. Ostatnim etapem, jako, że światła drogowe mają jasno określony kształt jest użycie transformaty Hougha dla okręgów, by wykryć właściwe światła drogowe. W artykule stosowana jest zmodyfikowana transformata Hougha do wyszukiwania okręgów, która polega na ustaleniu stałej długości promienia. Klasyczna transformata Hougha jest opisana w sekcji 2.6.

2.4.2. Wnioski i rezultaty

Opisany algorytm detekcji świateł drogowych daje lepsze wyniki niż użycie standardowej przestrzeni barw i klasycznej transformaty Hougha dla okręgów. Porównanie jest zamieszczone w tabeli 2.1. Główną zaletą opisywanej metody detekcji świateł drogowych jest fakt, że poprawnie odrzuca ona obrazy znaków drogowych, które również mają okrągłe kształty i jednolite kolory na krawędziach. Algorytm bierze dodatkowo pod uwagę czy kolor na krawędziach jest taki sam jak wewnętrz kształtu, co pozwala skutecznie odrzucić np. znaki zakazu. Problemem, który napotyka algorytm są tylne światła innych pojazdów, które mają jednolity kolor (z reguły czerwony), a także kształt zbliżony do czerwonego. Przykładowy błąd detekcji jest ukazany na rysunku 2.15.



Rys. 2.15. Przykład błędnej detekcji tylnego światła traktora[T4]

2.5. Detekcja samochodu poprzedzającego

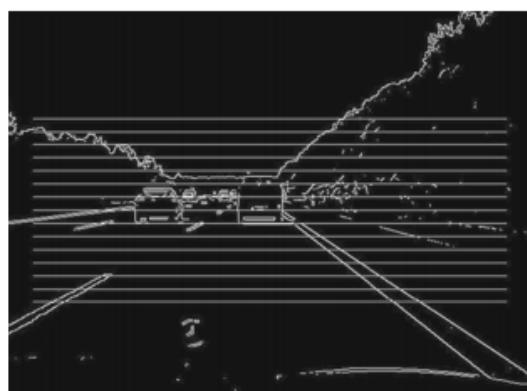
Istotnym zadaniem stawianym przed algorytmami wizyjnymi stosowanymi w pojazdach autonomicznych jest detekcja samochodów w najbliższym otoczeniu pojazdu. Detekcja może być wspierana odczytami z radaru lub lidaru, jednak w poniższej sekcji zostanie opisany algorytm bazujący jedynie na obrazie z kamer. Większość współczesnych samochodów widzianych od tyłu zachowuje symetrię względem pionowej osi przechodzącej przez środek pojazdu.

Pierwszym wstępnym krokiem jest zbadanie możliwych pozycji samochodów na obrazie i oznaczenie ich jako ROI. Dla systemu z fuzją danych wizyjnych i radarowych, może to być zrobione poprzez poprzez analizę odległości i prędkości względnej, czyli danych uzyskanych z radaru. Dla systemu, który posiada jedną kamerę pozycja samochodów musi być wyznaczona tylko na podstawie ruchu samochodów na obrazie w czasie.

Opisywany algorytm korzysta z detektora symetrii, który działa w następujący sposób. Dla każdego piksela wyznaczana jest liczba punktów, która jest wartością bezwzględną z różnicy wartości pikseli, które są równoodległe od ustalonej osi symetrii. Jest to zwykle robione z użyciem pewnego okna o z góry ustalonym rozmiarze dobieranym tak, aby pasować do rozmiaru samochodów, które mogą znajdować się na obrazie. Będąc świadomym faktu, że samochód im jest dalej od kamery, tym jest mniejszy zastosowano kilka predefiniowanych rozmiarów okien. Wartości wskaźnika symetrii mogą być obliczne dla każdego punktu na obrazie. Piksele z dużą jego wartością są dobrymi kandydatami do należenia do osi symetrii. Wyliczanie wskaźnika symetrii dla każdego punktu na obrazie jest bardzo czasochłonne, więc zdecydowano się na jego wyznaczanie tylko na wcześniej określonych poziomych liniach, które z grubsza pokrywają obszar, na którym mogą znajdować się samochody. Do obliczania wskaźnika symetrii może być użytych kilka cech obrazu takich jak: wartości pikseli w skali szarości, obraz krawędzi, składowa S w przestrzeni barw HSV. Szukanie obrazu samochodu na obrazie w skali szarości jest szybsze, jednak wrażliwe na zmiany oświetlenia (noc, deszcz). Dobra sprawdza się badanie nasycenia w przestrzeni barw HSV, ponieważ uniezależnia to obraz samochodu od ogólnej jasności otoczenia i częściowo od pogody.



Rys. 2.16. Obraz z samochodami po filtracji filtrem Canny'ego[T1]



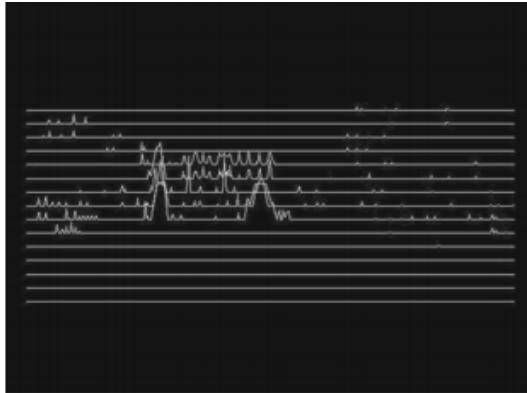
Rys. 2.17. Obraz z samochodami po filtracji filtrem Canny'ego i zaznaczonymi liniami skanu[T1]

2.5.1. Przebieg algorytmu bazującego na operatorze symetrii

Pierwszym krokiem jest wygenerowanie obrazu krawędzi na podstawie obrazu w skali szarości lub składowej S przestrzeni barw HSV. W opisanym algorytmie zaproponowano detektor Canny'ego. Rysunek 2.16 pokazuje rezultat wykrywania krawędzi dla typowego obrazu zawierającego samochód poprzedzający. Opierając się na znanej pozycji kamery i jej pochyleniu względem nawierzchni drogi można określić obszar na obrazie, na którym będą szukane samochody. Poziome ograniczenia znajdują się pomiędzy horyzontem i początkiem widocznej drogi na dole obrazu. Pionowe ograniczenia są ustawione tak, by odpowiadać lewemu i prawemu ograniczeniu jezdni.

Jak wspomniano w sekcji 2.5, w celu zredukowania czasu obliczeń nie każdy piksel w wybranym obszarze jest analizowany. Obliczenia dotyczące symetrii są przeprowadzane tylko dla 15 równoodległych linii skanu (rys. 2.17). Wejściowa rozdzielcość obrazu nie ma znaczenia dla obliczeń. Dzieje się tak ponieważ algorytm wykrywa tylko maksima wzdłuż linii skanu.

Kolejnym krokiem jest detekcja symetrii. Jest ona przeprowadzana dla każdego punktu leżącego na linii skanu. Wartość operatora symetrii dla piksela wyraża się wzorem:



Rys. 2.18. Wartości wskaźnika symetrii wyznaczone dla linii skanu[T1]

$$SymVal(x, y) = \sum_{x'=1}^{W/2} \sum_{y'=y-H/2}^{y+H/2} S(x, x', y') \quad (2.17)$$

gdzie:

–

$$S(x, x', y') = \begin{cases} 2 & \text{gdy } I(x - x', y') = I(x + x', y') = 1 \\ -1 & \text{gdy } I(x - x', y') \neq I(x + x', y') \\ 0 & \text{w p.p.} \end{cases} \quad (2.18)$$

- W – szerokość okna
- H – wysokość okna
- $I(x, y)$ – wartość piksela o współrzędnych x, y

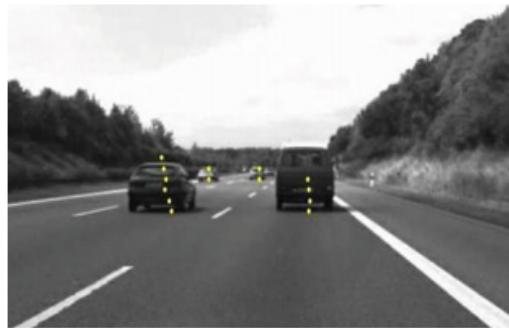
Szerokość okna powinna być właściwie ustawiona, aby poprawnie wykrywać symetryczne obiekty o różnych rozmiarach. W trakcie eksperymentów wykazano, że optymalne wartości W mieścią się w przedziale [8, 12].

Na rysunku 2.18 widać, że w niektórych punktach istnieją maksima, które wskazują, że dany punkt może należeć do osi symetrii. Wybiera się maksima i stosuje progowanie, to znaczy wartości maksimów lokalnych poniżej pewnej wartości są odrzucane. Zwykle wartości poniżej określonego progu wskazują na małe, symetryczne elementy tła. Progowanie dokonuje się według następującej formuły:

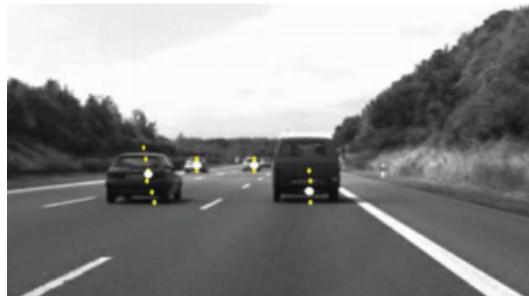
$$SymPts(x, y) = \begin{cases} 1 & \text{gdy } SymVal(x, y) > T \\ 0 & \text{w p.p.} \end{cases} \quad (2.19)$$

gdzie

- T - ustalony próg odrzucenia maksimum lokalnego



Rys. 2.19. Wykryte osie symetrii na liniach skanu[T1]



Rys. 2.20. Wynik algorytmu detekcji samochodów poprzedzających[T1]

Ostatecznie znalezione maksima oznaczają wykryte osie symetrii względnie dużych obiektów, w tym przypadku samochodów. Widać to na rysunku 2.19. Dla każdej linii skanu wykryta oś symetrii jest przesunięta o kilka pikseli, dlatego ostatnim etapem detekcji samochodu jest klasteryzacja.

Uzyskane punkty osi symetrii są klasteryzowane metodą k-średnich. Liczba samochodów na obrazie jest nieznana, więc klasteryzację robi się iteracyjnie, co iterację licząc wariancję, która przy poprawnej liczbie klastrów w stosunku do samochodów na obrazie będzie mniejsza niż określony próg. Końcowy wynik z zaznaczonymi środkami samochodów jest widoczny na rysunku 2.20

2.6. Opis wybranych zagadnień i algorytmów przetwarzania obrazu

W tej sekcji zostaną opisane podstawowe algorytmy i zagadnienia dotyczące cyfrowego przetwarzania obrazów, które są używane w zaawansowanych algorytmach wizyjnych w pojazdach autonomicznych.

2.6.1. Transformata Hougha dla okręgów

Systemy wizyjne w pojazdach autonomicznych często mają za zadanie wykrycie obiektów o kształcie koła. Algorymem do tego przeznaczonym jest transformata Hougha. Istnieje ona w wersji do detekcji prostych i okręgów. Podjęciem uogólniona transformata Hougha kryje się algorytm służący do detekcji dowolnego zadанego konturu.

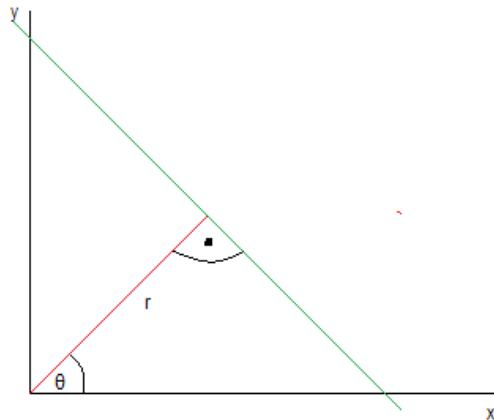
Okrąg można sparametryzować za pomocą następującego wzoru:

$$(x - x_0)^2 + (y - y_0)^2 = r^2 \quad (2.20)$$

gdzie

– x_0, y_0 – współrzędne środka okręgu

– r – promień okręgu



Rys. 2.21. Linia w układzie współrzędnych określona za pomocą parametrów (r, θ)

Jeżeli promień będzie ustalony, to okrąg zostanie sparametryzowany za pomocą dwóch liczb. Gdy szukamy okręgów o nieznanych promieniach, rośnie złożoność obliczeniowa, ponieważ wymiar przestrzeni parametrów zwiększa się o jeden.

Możliwa jest również parametryzacja okręgu w biegunowym układzie współrzędnych:

$$x = x_0 + r\cos(\theta) \quad (2.21)$$

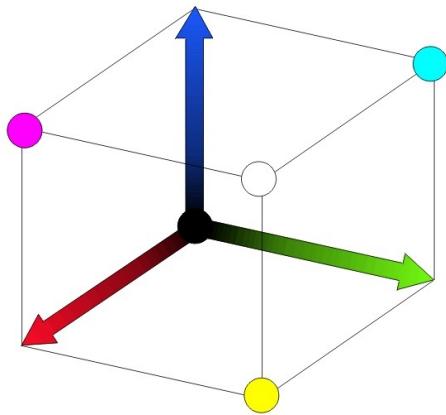
$$y = y_0 + r\sin(\theta) \quad (2.22)$$

po prostym przekształceniu otrzymano:

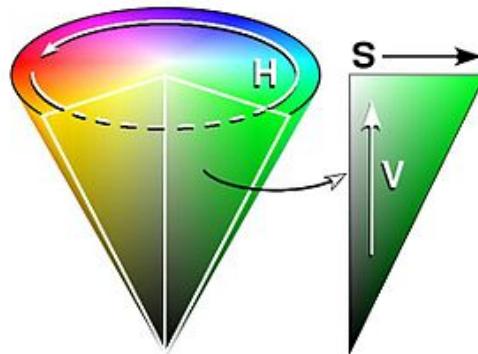
$$x_0 = x - r\cos(\theta) \quad (2.23)$$

$$y_0 = y - r\sin(\theta) \quad (2.24)$$

Następnie wyznaczana jest przestrzeń Hougha, w której dla każdego piksela obrazu wyznacza się liczbę możliwych okręgów do których mógłby należeć. Wartości maksymalne w przestrzeni Hougha oznaczają wykryty okrąg.



Rys. 2.22. Sześciągniektostyczne przedstawiające przestrzeń barw RGB[W4]



Rys. 2.23. Stożek przedstawiający przestrzeń barw HSV(*źródło: Wikipedia*)

2.6.2. Przestrzenie barw

Podstawową przestrzenią barw jest RGB. Przedstawiona na rysunku 2.22 za pomocą sześciągniektostyczne. Każda składowa jest odpowiedzialna za informację o zawartości danego koloru. Jej zaletą jest prostota opisu, natomiast wadą jest fakt, że po niewielkiej zmianie wartości składowych otrzymuje się zupełnie inną barwę. Dodatkowo, co jest ważne w przypadku systemów wizyjnych, niewielka zmiana poziomu jasności powoduje duże wahania składowych R, G, B.

Drugą, ważną przestrzenią barw używaną w cyfrowym przetwarzaniu obrazów jest przestrzeń HSV. Przedstawiona na rysunku 2.23 za pomocą stożka. Główną zaletą jest to, że przy niewielkich zmianach jasności są bardzo niewielkie zmiany w składowej S. Pozwala to na uniezależnienie się w pewnym stopniu od czynników takich jak pora dnia lub pogoda.

2.6.3. Filtr Canny'ego

Podstawowym, dobrze sprawdzającym się detektorem krawędzi jest filtr Canny'ego. Cechuje się właściwościami:

- niska liczba fałszywych detekcji krawędzi,
- poprawne wskazywanie pozycji krawędzi. Pozycja krawędzi wskazywana przez detektor powinna odpowiadać jej prawdziwemu położeniu
- jedna wykryta krawędź przypadająca na rzeczywistą krawędź

Detektor krawędzi Canny'ego jest algorytmem wieloetapowym:

1. Usunięcie z obrazu jakichkolwiek szumów. Używany jest filtr Gaussa. Przykładowa macierz filtru pokazana jest na rysunku 2.24

$$\frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Rys. 2.24. Maska filtru Gaussa stosowana w wykrywaniu krawędzi

2. Wyszukiwanie krawędzi z użyciem filtru Sobela o poziomej i pionowej orientacji
3. Określenie wartości gradientu i jego kierunku:

$$G = \sqrt{G_x^2 + G_y^2} \quad (2.25)$$

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \quad (2.26)$$

Kierunek jest zaokrąglany do jednego z czterech możliwych kierunków: $0^\circ, 45^\circ, 90^\circ, 135^\circ$

4. Wartości gradientu o niemaksymalnych wartościach są usuwane. Ma to na celu usunięcie pikseli, które prawdopodobnie nie są elementem krawędzi. W wyniku tej operacji pozostają tylko cienkie linie jako krawędzie.
5. Filtr Canny'ego jako argumenty otrzymuje dwa proggi - górny i dolny:

- jeżeli wartość gradientu przekracza górny próg, piksel jest zawsze uznawany jako krawędź
- jeżeli wartość gradientu nie przekracza dolnego progu, piksel nie jest uznawany za krawędź
- jeżeli wartość gradientu jest pomiędzy dwoma progami, jest krawędzią, tylko wtedy gdy jest połączony z pikselem, który został sklasyfikowany jako krawędź

Twórca filtru rekomenduje stosunek progów filtru pomiędzy 2:1 i 3:1

3. Realizacja projektu

W tym rozdziale zostanie opisana implementacja systemu służącego do testowania algorytmów wizyjnych z użyciem symulatora Euro Truck Simulator 2. Do stworzenia systemu użyto następujących elementów:

- Python 3.7 - język programowania wysokiego poziomu ogólnego przeznaczenia,
- gra Euro Truck Simulator 2,
- SDK (ang. Software Development Kit) udostępnione przez twórców gry,

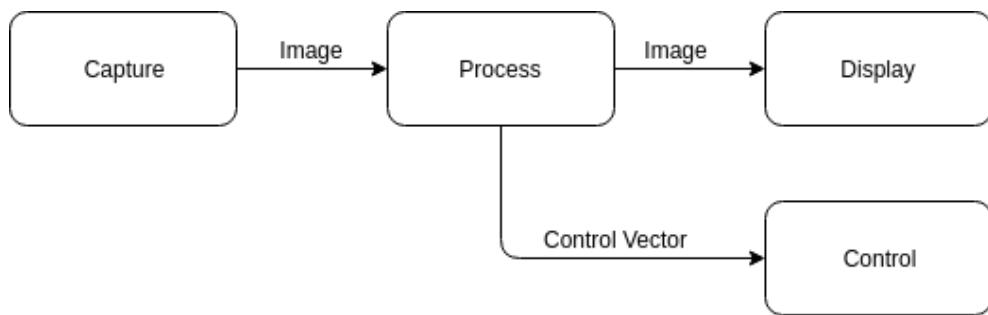
a także bibliotek dołączonych do Pythona:

- OpenCV 3.1.3 - biblioteka zawierająca funkcje do cyfrowego przetwarzania obrazów,
- threading - biblioteka wspierająca programowanie wielowątkowe,
- libWnck - biblioteka zapewniająca komunikację ze środowiskiem graficznym (Window Navigator Construction Kit),
- uinput - biblioteka pozwalająca symulować kontroler do sterowania grą.

System jest opracowany do działania na systemie Ubuntu 18.04 LTS wraz ze środowiskiem graficznym GNOME.

3.1. Euro Truck Simulator 2

Euro Truck Simulator 2 jest to symulator jazdy ciężarówką opracowany przez czeskie studio SCS Software. Pozwala on na jazdę wybranym modelem ciężarówki przez wiele dróg Europy oraz na realizację zleceń na przewóz towarów. Z uwagi na charakter pracy, interesującymi elementami gry jest wysoka jakość grafiki, wiernie odwzorowująca otaczający świat, w tym zestaw znaków i linii drogowych charakterystycznych dla poszczególnych krajów. Kolejnym elementem który opowiada za wyborem tego symulatora jest otwartość na wszelkie modyfikacje. Twórcy gry udostępnili API (ang. application programming interface), a także konsolę, za pomocą której można na bieżąco modyfikować parametry gry takie jak czas, prędkość gry lub pogodę. Alternatywnym symulatorem, który również był brany pod



Rys. 3.1. Schemat architektury aplikacji

uwagę jest American Truck Simulator 2, jednak jest on kalką symulatora opisywanego w tej sekcji ze zmienionym obszarem gry z równie dobrą grafiką i wsparciem dla programistów. Kolejną alternatywą są gry z serii Grand Theft Auto, jednak na ich niekorzyść wskazywało gorsze wsparcie dla programistów, a także niewspółmierne zużycie zasobów do uzyskiwanej jakości obrazu (GTA V) lub niskie zużycie zasobów przy niskiej jakości obrazu (GTA: San Andreas). Fakt, że jest to symulator ciężarówki, a nie samochodu osobowego w żaden sposób nie wpływa na podejście do problemu systemów wizyjnych, ponieważ po pierwsze, istnieją aplikacje wspomagające kierowcę samochodu ciężarowego, po drugie jeden z widoków w grze jest umieszczony w miejscu, które znajduje się na wysokości lusterka samochodowego.

Wspomniana konsola dostępna w grze pozwala na błyskawiczną zmianę warunków w symulatorze. Używając następujących komend można zmienić czas, pogodę oraz prędkość gry:

- `g_set_weather x` – komenda zmieniająca pogodę. Gdy x jest równy 1, pogoda jest deszczowa, natomiast, gdy jest równy 0 pogoda jest słoneczna
- `g_set_time x` – komenda ustalająca godzinę w grze. W miejsce x należy podać godzinę w formacie hh,
- `warp x` – komenda zmieniająca prędkość gry. W miejsce x należy wpisać współczynnik. Liczba mniejsza od 1 zwolni grę, a większa przyspieszy.

3.2. Architektura systemu

Z uwagi na dydaktyczną wartość opisywanego systemu ważnym elementem jest łatwość implementacji systemu analizy obrazu dla przyszłych użytkowników. Z tego powodu zdecydowano się na implementację wieloprocesową, której schemat jest widoczny na rysunku 3.1, w której jeden z procesów jest odpowiedzialny za analizę obrazu i wypracowanie sterowania, a pozostałe za poprawne przechwycenie obrazu z gry i "wstrzygnięcie" wypracowanego sterowania do gry.

Poszczególne procesy są połączone kolejkami, za pomocą których są transportowane dane pomiędzy nimi. W przypadku tej opisywanej aplikacji w kolejkach umieszczane są obrazy wejściowe i wyjściowe

z procesu *Process*, a także wektor sterowań do procesu *Control*. Szybkość działania aplikacji jest zależna od wielu czynników. Głównym elemetem warunkującym jest czas przetwarzania obrazu w bloku *Process*. W zależności od liczby obrazów czekających na przetworzenie w kolejce zmieniany jest interwał pomiędzy poszczególnymi przechwytciami obrazu w bloku *Capture*. Uniezależnienie przechwytywania obrazu od zapełnienia kolejki spowodowałoby szybkie jej przepełnienie i zabicie aplikacji przez system. Końcowe procesy tj. *Display* i *Control* nie wymagają wyzwalania w zależności od czasu przetwarzania obrazu, ponieważ wykonują się bardzo szybko (poniżej 10ms).

Używana biblioteka *multiprocessing*[S2] pozwala na tworzenie podprocesów. Korzyść uzyskana z tego tytułu względem korzystania z wątków polega na tym, że program napisany z użyciem tej biblioteki jest w pełni w stanie korzystać z wielu procesorów lub rdzeni komputera.

3.2.1. Przechwytywanie obrazu

Optymalnym rozwiązaniem byłoby przechwytywanie obrazu wprost z pamięci symulatora, lecz niestety póki co nie istnieją metody pozwalające na takie rozwiązanie. W procesie *Capture* realizowane są dwa podzadania. Pierwsze polega na znalezieniu i aktywowaniu okna symulatora. Ma to na celu zapobiegnięcie przechwytcia obrazu, gdy okno gry jest przysłonięte przez inną aplikację lub zminimalizowane. Następnie, za pomocą biblioteki libwnck odczytanie współrzędnych okna gry (współrzędne górnego rogu, szerokość i wysokość). Dopiero, gdy jest zapewniony poprawny obraz do przechwytcia jest to robione, a obraz w postaci macierzy o wymiarach 1024x768x3 jest wkładany w kolejkę i wysyłany do procesu przetwarzania i analizy obrazu. Jednorazowe przechwycenie okna zajmuje średnio (około 10ms).

3.2.2. Przetwarzanie i analiza obrazu

Jeśli kolejka wejściowa do procesu nie jest pusta, to obraz jest z niej pobierany, a następnie w zależności od wybranego algorytmu (detekcja linii, detekcja znaków) jest przetwarzany. Poszczególne opisy zaimplementowanych algorytmów znajdują się w sekcjach TODO: numery sekcji z opisem algorytmów. Istotne jest, aby po skończeniu obróbki obrazu i wypracowaniu sterowania włożyć obraz z zaznaczonym wynikiem przetwarzania i wektor sterowań do kolejki.

3.2.3. Symulacja kontrolera

Euro Truck Simulator 2 pozwala użytkownikowi na sterowanie ciężarówką za pomocą licznych kontrolerów. Może to być kawiatura, klawiatura w zestawie z myszką komputerową lub kierownica lub gampad. W systemach UNIX-owych każde urządzenie wejściowe podpięte do komputera jest widoczne jako plik w lokalizacji `\dev\input\`. Urządzenia generują zdarzenia (ang. eventy), które są przesyłane do korzystających z nich aplikacji. Biblioteka *winput* pozwala na zasymulowanie dowolnego kontrolera i sterowanie nim programowo. Na potrzeby opisywanej aplikacji stworzono zasymulowanego kontrolera,

która posiada trzy osie analogowe: kierownica, gej gazu i gej hamulca. Dodatkowo w celu umożliwienia programowej kontroli nad konsolą, kontroler posiada pełen zestaw klawiszy z układu QWERTY. Podczas inicjalizacji aplikacji jest inicjalizowany kontroler, tak, aby przed rozpoczęciem właściwej rozgrywki i przetwarzania obrazu gra wykryła poprawnie urządzenie do sterowania. Klasa *Control* posiada metodę *emit*, która odpowiada za wygenerowanie zdarzenia z odpowiednimi wartościami sterowania. Podczas wykonywania tej metody jest sprawdzane czy okno z grą jest aktywne, ponieważ w przypadku wyemitowania sterowania przy nieaktywnym oknie, sterowanie nie będzie poprawnie zinterpretowane przez symulator.

3.2.4. Wyświetlanie rezultatów

Wyświetlanie rezultatów jest opcjonalne. Jest realizowane w osobnym procesie ze względu na ustaloną architekturę systemu, która zakłada, że w procesie odpowiedzialnym za przetwarzanie obrazu będzie realizowane tylko przetwarzanie. Wyświetlanie obrazu jest realizowane za pomocą funkcji biblioteki OpenCV *imshow()*.

3.3. Opis implementacji algorytmów wizyjnych użytych w symulatorze

W tej sekcji zostaną opisane algorytmy służące do przetestowania działania framework'u stworzonego w ramach pracy magisterskiej. Cele jakie są postawione przed zaimplementowanym frameworkiem to: przetwarzanie obrazu w czasie rzeczywistym lub do niego zbliżonym, dostęp do wirtualnego kontrolera symulowanego w ramach aplikacji.

3.3.1. Algorytm detekcji linii

W symulatorze Euro Truck Simulator znajdują typy dróg z każdego kraju Europy. Zaczynając od szerokich autostrad, poprzez drogi ekspresowe na wąskich i krętych drogach w Alpach kończąc. Zaimplementowany algorytm jest przeznaczony do detekcji linii na drogach, których krzywizna łuku nie jest duża. Przykładowy obraz wejściowy jest widoczny na rysunku 2.3. Pierwszym krokiem jest konwersja przestrzeni barw z RGB do HSV. Wybranie składowej S jako obrazu poddawanego analizie (rys. 3.2) pozwala uniezależnić się od pory dnia i pogody.

Założenie, że obszar jezdni może znajdować się tylko na pewnym obszarze obrazu pozwala zaoszczędzić czas obliczeń. Po wybraniu ROI orzymano obraz 3.4.

Kolejnym krokiem jest wykrycie krawędzi z użyciem filtru Canny'ego. Progi dobrane są eksperymentalnie. Należy zwrócić uwagę, że gra pozwala na ustalenie jakości grafiki. W związku z tym, każdorazowo po zmianie ustawień należy dobrać współczynniki filtru na nowo. Wraz ze wzrostem jakości grafiki poprawia się jej ostrość. Porównanie niskich i wysokich ustawień graficznych jest widoczne na



Rys. 3.2. Obraz wejściowy (składowa S)

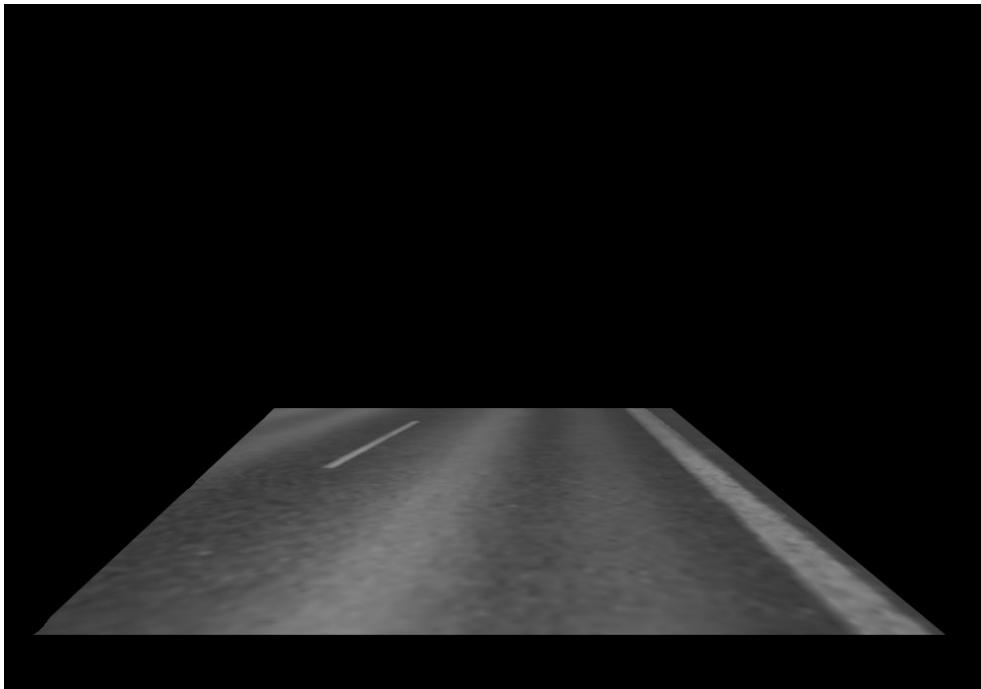


Rys. 3.3. Porównanie niskich (a) i wysokich (b) ustawień grafiki symulatora

rysunku 3.3. Kolejną rzeczą wartą uwagi przy wyborze ustawień grafiki jest to, że zarówno praca zaimplementowana w tej pracy jak i gra są uruchamiane na jednej maszynie. W związku z tym wybranie niskich ustawień grafiki pozwoli na przekazanie zasobów obliczeniowych do aplikacji.

Na obrazie krawędzi, za pomocą transformaty Hougha dokonuje się detekcji linii prostych. Linie fałszywe (linie nie będące liniami na jezdni) można odrzucić badając ich parametr θ . Właściwe linie będą pochycone pod odpowiadającym kątem. Dla opisywanego przypadku właściwym zakresem kąta nachylenia linii jest $\theta < 1.3\text{rad}$ dla linii po lewej i $\theta > 2\text{rad}$ dla krawędzi prawej.

W celu zaoszczędzenia czasu obliczeń i poprawienia sprawności algorytmu założono, że rezultatem powinna być linia dla każdej z krawędzi jezdni. Jeśli wykryto kilka linii, których parametry są zbliżone, dalszej analizie poddawana jest tylko jedna z nich (rys. 3.5).



Rys. 3.4. Wyznaczone ROI, na którym można spodziewać się linii

Aby wygenerować sterowanie bada się punkt przecięcia wykrytych linii (biały okrąg na rys. 3.5). Zakładając, że kamera jest ustawiona w osi ciężarówki można przyjąć, że każde odchylenie punktu przecięcia wykrytych linii powinno implikować ruch kierownicą.

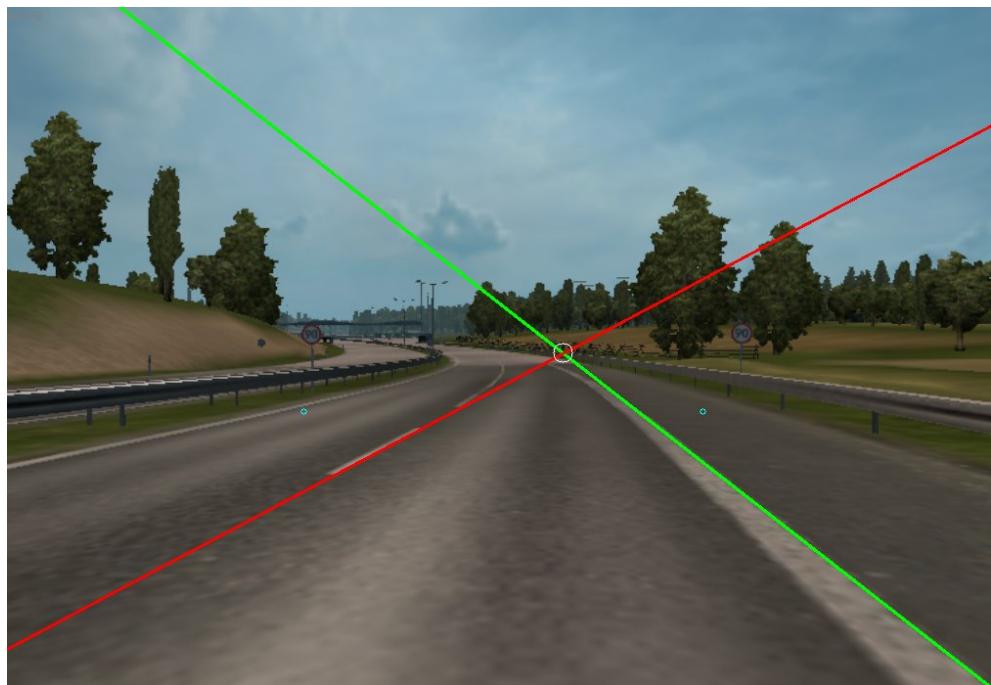
Średni czas obliczeń potrzebny do przetworzenia jednej klatki wyniósł 59ms, co daje możliwość przetworzenia 17 klatek na sekundę. Jest to wydajność wystarczająca, aby zapewnić stabilność całości algorytmu, ponieważ ciężarówka była w stanie utrzymać się przy stałej prędkości na pasie ruchu długim na ok. 500m.

3.3.2. Sytuacje potencjalnie problematyczne

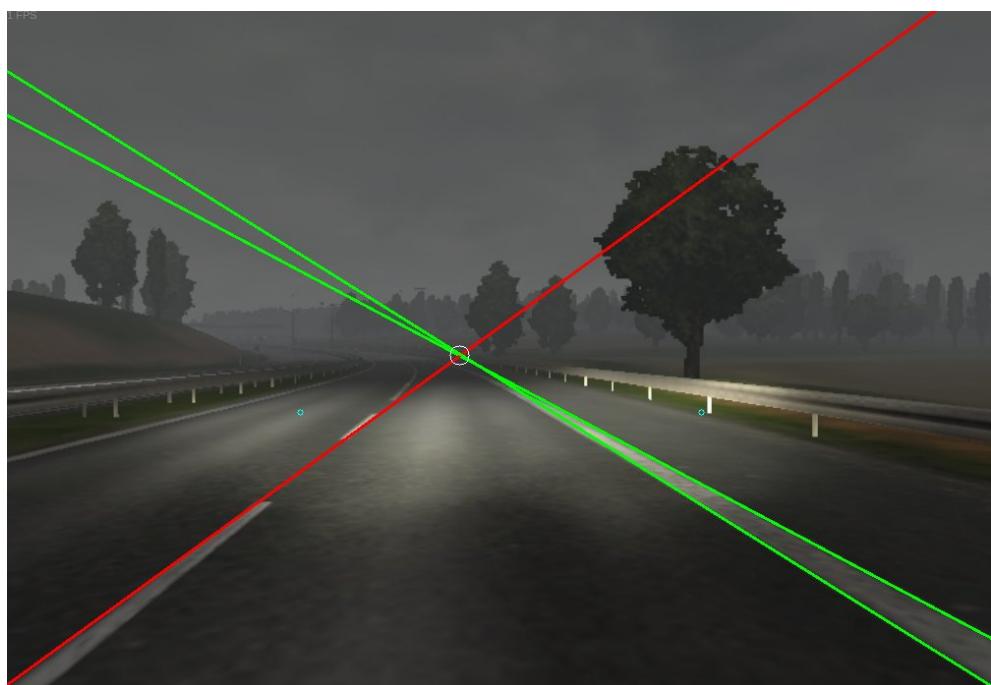
Jak wspomniano w sekcji 2.2 wymagane jest poprawne działanie algorytmu przy zmiennych warunkach oświetlenia i pogody. Symulator oferuje możliwość zmiany tych parametrów poprzez konsolę. Algorytm wykazał skuteczność zarówno przy zmianie godziny gry na późniejszą oraz dodaniu deszczu (rys. 3.6 i rys. 3.7)

3.3.3. Algorytm detekcji czerwonych świateł drogowych

Kolejnym algorymem testowym, który został zaimplementowany w ramach pracy jest uproszczona wersja algorytmu opisanego w sekcji 2.4. Przykładowy obraz wejściowy znajduje się na rysunku 3.8. Dają się tam zauważyć dwa sygnalizatory świetlne, a także kilka znaków drogowych. Sceneria została dobrana tak, aby jednocześnie sprawdzić odporność algorytmu na znaki zawierające czerwoną barwę.



Rys. 3.5. Rezultat działania algorytmu



Rys. 3.6. Działanie algorytmu podczas trudnych warunków oświetleniowych



Rys. 3.7. Działanie algorytmu podczas deszczu

Pierwszym krokiem opisywanej metody jest wyznaczenie kandydatów na światła drogowe. Dobierając ekperymentalnie współczynniki progowania ustalonono, że światło czerwone będzie spełniać poniższe warunki:

$$R > 180 \wedge 0 \geq R < 100 \wedge B < 120 \quad (3.1)$$

gdzie R, G, B oznaczają wartości składowych w przestrzeni RGB.

Aby zapewnić, że inne obiekty spełniające warunek 3.1 wykonano operację erozji i dylatacji zwaną również otwarciem morfologicznym. Otrzymany obraz indeksuje się. Iterując po każdym z oznaczonych elementów można sprawdzić jego powierzchnię i proporcje, które muszą spełniać pewne warunki, żeby zostać uznane za światło drogowe.

Aby zmodyfikować algorytm tak, by wykrywał światło zielone lub pomarańczowe należy zmienić wartości progów w warunku 3.1. Do indeksacji kandydatów wykorzystano funkcję biblioteki OpenCV *connectedComponentsWithStats()*, która zwraca listę obiektów wraz z ich statystykami.

3.3.4. Sytuacje potencjalnie problematyczne

Jak w przypadku każdego z omawianych algorytmów problemem jest zmiana warunków oświetleńowych oraz pogodowych. Nie można jak w przypadku algorytmu detekcji pasa ruchu uniezależnić się od zmian oświetlenia poprzez zastosowanie konwersji do przestrzeni HSV, ponieważ istotną rolę odgrywa tu barwa. Z uwagi na to, że światła drogowe emitują światło, w czasie warunków słabej widoczności i deszczu algorytm poprawnie wykrywa czerwone światło (rys. 3.11 i rys. 3.10)



Rys. 3.8. Przykładowy obraz wejściowy algorytmu



Rys. 3.9. Wykryte dwa palące się czerwone światła



Rys. 3.10. Wykryte światła drogowe w warunkach słabego oświetlenia



Rys. 3.11. Wykryta sygnalizacja świetlna podczas deszczu



Rys. 3.12. Przykładowy obraz wejściowy algorytmu detekcji samochodu poprzedzającego

3.3.5. Detekcja samochodu poprzedzającego

Ostatnim algorytmem, który został zaimplementowany w ramach tej pracy jest detekcja samochodów poprzedzających na drodze z użyciem detektora symetrii. Implementacja bazuje na algorytmie opisanym w sekcji 2.5.

Zaimplementowany algorytm operuje na obrazach w skali szarości. Przykładowy obraz wejściowy jest pokazany na rysunku 3.12. Daje się na nim zauważać ciężarówkę poprzedzającą samochód. Celem działania algorytmu jest poprawne wykrycie symetrii na tyle naczepy samochodu.

Pierwszym krokiem jest filtracja z użyciem filtra Canny'ego. Progi filtra są każdorazowo po zmianie ustawień graficznych symulatora dobierane eksperymentalnie. Efekt filtracji jest widoczny na rysunku 3.13.

Następnie wyznaczając linie skanu sprawdzana jest symetria obrazów w sposób opisany w sekcji 2.5. Na każdej z linii skanu wyznaczane są maksima, które wskazują na istnienie osi symetrii. Rezultaty detekcji są umieszczone na rysunkach 3.14 i 3.15. Daje się zauważać znaczna liczba detekcji fałszywych. Prawdopodobnie wynika to między innymi z faktu, że w symulatorze obiekty infrastruktury mają bardziej równomierny i symetryczny kształt.

Sprawdzono również zachowanie algorytmu w trudnych warunkach pogodowych i słabego oświetlenia. Samochód korzysta ze światła mijania, więc najbliższe otoczenie jest dobrze widoczne (rys. 3.16). W tym przypadku również pojawiają się fałszywe detekcje.



Rys. 3.13. Obraz wejściowy po detekcji krawędzi. Widoczna symetria w układzie krawędzi poprzedającej naczepy



Rys. 3.14. Rezultat detekcji samochodu osobowego. Widoczne liczne fałszywe detekcje.



Rys. 3.15. Poprawna detekcja naczepy samochodu ciężarowego



Rys. 3.16. Poprawna detekcja naczepy samochodu ciężarowego w trudnych warunkach oświetleniowych i pogodowych



Rys. 3.17. Rozmiar kolejki w trakcie działania programu

Obliczanie wskaźnika symetrii jest operacją bardzo czasochłonną. Średni czas przetworzenia jednej klatki to 5.45 sekundy. Oprócz dużej ilości obliczeń może to być spowodowane mało wydajną implementacją w Pythonie. Implementacja w języku C++ prawdopodobnie przyniosłaby lepsze efekty.

3.4. Badania wydajności aplikacji

Istotną rzeczą jest wydajność całej aplikacji. Powinna ona działać bez widocznych opóźnień, to znaczy, że po przechwyceniu obrazu z symulatora w możliwie najkrótszym czasie powinno zostać wygenerowane sterowanie. Jest to zapewnione poprzez mechanizm opisany w sekcji 3.2.1. Eksperymentalnie dowiedzono, że liczba elementów w kolejkach, zwłaszcza w kolejce pomiędzy modułem *Capture*, a *Process* nie powinna przekraczać 50. Powyżej tej wartości daje się zauważać widoczne opóźnienie reakcji systemu sterującego samochodem. Na wykresie przedstawiono rozmiar kolejki w kolejnych iteracjach programu podczas działania systemu wizyjnego odpowiedzialnego za detekcję czerwonego światła.

W trakcie implementacji kolejnych systemów wizyjnych używanych w pojazdach autonomicznych należy zwrócić uwagę na optymalizację kodu. Nie powinno się tworzyć wielu kopii obrazu w pamięci, a także z uwagi na asynchroniczny charakter aplikacji nie używać komend typu *wait()*. Ważnym aspektem, który znaczco poprawił wydajność całości systemu było zastosowanie wektoryzacji, czyli wbudowanej w język programowania metody operowania na macierzach. W przypadku, gdy algorytm ze względu na swoją złożoność wykonywałby się zbyt długo, należy użyć komendy *warpX* w konsoli symulatora udostępnionej przez twórców, która zmieni prędkość gry - *warpX*, gdzie *x* oznacza prędkość (1 - standardowa prędkość rozgrywki).

3.5. Ewaluacja systemu

Celem pracy było stworzenie aplikacji, która będzie w ramach przedmiotu Systemy wizyjne w pojazdach autonomicznych prowadzonego na II stopniu studiów stacjonarnych na kierunku Automatyka i Robotyka pozwalała w przyjazny użytkownikowi sposób na implementację i testowanie algorytmów wizyjnych stosowanych w pojazdach autonomicznych. Przykładowe algorytmy zaimplementowane w ramach tej pracy dowodzą, że jest to możliwe. Korzystanie z aplikacji w ramach zajęć jest zamienną formą w stosunku do korzystania ze zbiorów zdjęć np. KITTI. Nie jest możliwym utworzenie *groundtruth*, aczkolwiek inną formą sprawdzenia czy dany algorytm działa poprawnie jest weryfikacja poprzez zachowanie ciężarówki w symulowanym świecie. Opcjonalne jest generowanie sterowania na podstawie wyników z algorytmów wizyjnych. Końcowy użytkownik może tradycyjnie za pomocą klawiatury lub myszki sterować pojazdem i na bieżąco badać rezultaty osiągane przez algorytm.

Kolejnym etapem rozwoju projektu byłoby stworzenie prostego instalatora, który pobierze i zainstaluje wszystkie potrzebne biblioteki, a także skompiluje zmodyfikowane SDK udostępnione przez twórców. Możliwa jest także dalsza edycja SDK, poprzez umożliwienie odczytu pozostałych zmiennych gry oprócz aktualnej prędkości i stanu pauzy. Pełna lista parametrów jest dostępna w [S3]. Elementem mocno rozbudowującym aplikację byłoby także generowanie danych radarowych na podstawie otoczenia samochodu w symulatorze. Póki co twórcy gry nie udostępniają takiej możliwości przez SDK.

4. Podsumowanie

Zgodnie z założeniami zrealizowano cele pracy. Dokonano ewaluacji przykładowych algorytmów wizyjnych, które mogłyby znaleźć zastosowanie w pojazdach autonomicznych, a także przetestowano ich działanie z użyciem zaimplementowanego framework'a. Zdaniem autora przygotowany framework nadaje się do użycia w ramach zajęć do testowania algorytmów przetwarzania obrazu. Pozwala w łatwy sposób modyfikować parametry obrazu takie jak natężenie światła, pogodę oraz liczbę otaczających pojazdów. Znaczącą przewagą systemu nad analizą pojedynczych zdjęć jest możliwość przetestowania algorytmów w sytuacjach dynamicznych. Zaimplementowana możliwość sterowania samochodem w symulatorze z pewnością pozwoli na rozbudowanie powstających algorytmów i sprawdzanie ich pod kątem wielu nowych aspektów takich jak np. optymalizacja kodu, by przyspieszyć jego wykonanie.

A. Instrukcja do ćwiczenia

Aplikację należy pobrać z repozytorium dostępnego pod tym adresem. Do prawidłowego działania aplikacji wymagany jest komputer z zainstalowanymi:

- Python w wersji 3.6 lub wyższej
- openCV w wersji 3.1.3 lub wyższej
- system Linux (najlepiej Ubuntu) z systemem okien X11
- biblioteka multiprocessing języka Python
- biblioteka Wnck (zapewnia komunikację aplikacji z systemem okien). Instalacja polecienniem: APT-GET INSTALL PYTHON3-GI GIR1.2-WNCK-3.0
- biblioteka uinput służąca do symulacji kontrolera ([link](#))

Ćwiczenie należy zacząć od instalacji symulatora jazdy Euro Truck Simulator 2. Instalacja poprzez platformę Steam jest łatwa i intuicyjna. Dane kont są dostępne u prowadzącego. Do każdego konta jest przypisana jedna licencja na grę.

Po instalacji należy po raz pierwszy uruchomić grę i ustawić w ustawieniach grafiki wyświetlanie gry w oknie. Kolejnym krokiem jest odblokowanie konsoli i narzędzi deweloperskich. Aby to zrobić należy zmodyfikować plik *config.cfg*, który powinien znajdować się w lokalizacji: *lsriw/.local/share/Euro Truck Simulator 2/*. Dwie linijki w pliku:

- USET G_DEVELOPER "0"
- USET G_CONSOLE "0"

należy zmienić na:

- USET G_DEVELOPER "1"
- USET G_CONSOLE "1"

Aktytowanie tych opcji pozwoli na swobodny dostęp do konsoli w grze oraz używanie narzędzi deweloperskich udostępnionych przez twórców. W przypadku tej aplikacji jest to informacja o prędkości pojazdu oraz stanie gry (pauza/grą).



Rys. A.1. Okno informujące o poprawnym zainstalowaniu SDK

Następnym etapem jest komplikacja programu, który będzie na bieżąco w trakcie gry zapisywał wspomniane dane do pliku. W folderze z aplikacją znajduje się folder SDK. Tamże należy odnaleźć folder *telemetry* i będąc w nim wykonać polecenie MAKE. Plik wykonywalny *telemetry.so* skopiować do lokalizacji gry: /HOME/LSRIW/.STEAM/STEAM/STEAMAPPS/COMMON/EURO TRUCK SIMULATOR 2/BIN/LINUX_X64/PLUGINS. Ta operacja sprawi, że gra od tej pory przy każdym uruchomieniu będzie tworzyła plik *telemetry.log*, w którym na bieżąco będzie zapisywana prędkość pojazdu i informacja o aktywnej pauzie.

Uruchomić symulator, powinno pojawić się okno z informacją o uruchomieniu narzędzi deweloperskich (A.1).

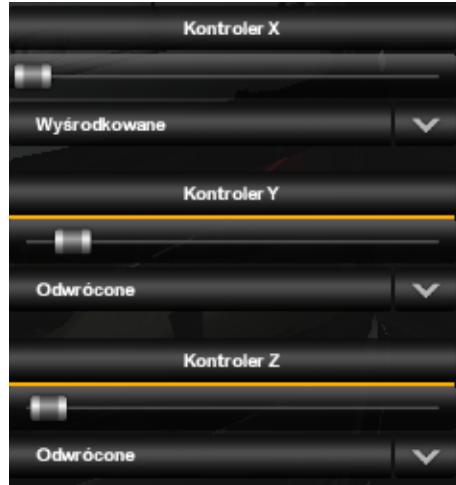
Przy okazji warto przetestować czy konsola również wyświetla się poprawnie. Konsolę aktywuje się klawiszem tyldy.

Ważna uwaga! Zawsze przed uruchomieniem aplikacji należy uruchomić symulator. Przesunięcie okna gry na inny ekran spowoduje niepoprawne działanie aplikacji do testowania algorytmów wizyjnych.

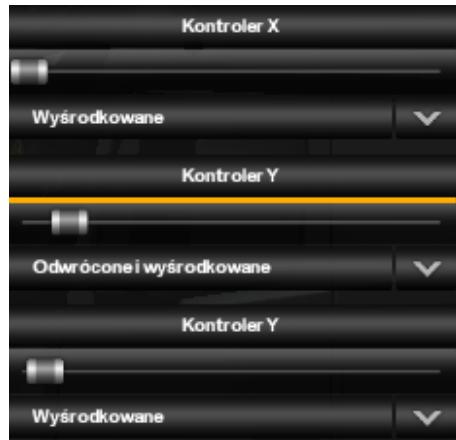
Jeśli wszystkie wymagane biblioteki są zainstalowane, testowo uruchomić aplikację polecienniem ./RUN.SH. Aplikacja wymaga dostępu do konta *roota*, ponieważ symulacja kontrolera tego wymaga.

Zaleca się korzystanie w grze z kamery numer 6 (klawisz 6). W bazowej wersji aplikacji, w której nie są zaimplementowane żadne algorytmy cyfrowego przetwarzania obrazów, obraz przechwycony jest przekazywany do procesu wyświetlającego bez żadnej modyfikacji. W pliku *Process.py* znajduje się funkcja PROCESS_IMAGE(SELF, IMAGE). Jako argument przyjmuje obraz przechwycony z symulatora. Funkcja zwraca listę złożoną z obrazu przetworzonego oraz wektora sterowania: RETURN [IMAGE, [NONE, 0, 0]]. W tej funkcji należy implementować algorytmy służące do przetwarzania obrazu. Początkowo sugeruje się w ramach testów nie generować sterowania. Dopiero po zapewnieniu pewnej stabilności algorytmu zalecane jest najpierwłączenie kontroli kierownicy samochodu, a następnie programowej kontroli prędkości za pomocą symulowanego pedału gazu i hamulca.

Jak wspomniano, początkowo może być trudne generowanie poprawnego (i sensownego) sterowania. Aby nie programowo nie wpływać na kontroler należy zwracać wektor sterowań w postaci: [NONE, 0, 0].



Rys. A.2. Poprawnie zidentyfikowany kontroler



Rys. A.3. Źle zidentyfikowana oś sterowania hamulcem

A.1. Obsługa kontrolera

Po zainicjalizowaniu aplikacji gra automatycznie wykryje symulowany kontroler. Posiada on trzy osie analogowe służące do sterowania kierownicą oraz pedałami gazu i hamulca. Analogowa oś kierownicy posiada zakres $[-32767, 32767]$, gdzie wartości skrajne oznaczają odpowiednio maksymalny skręt w lewo i w prawo. W razie nieprawidłowego działania kontrolera należy sprawdzić w ustawieniach czy okno ustawień kontrolera wygląda tak jak na rysunku A.2.

Czasami, prawdopodobnie w wyniku błędu gry, oś odpowiedzialna za kontrolę hamulca nie jest poprawnie wykrywana (A.3. Wtedy należy w ustawieniach kontrolera kliknąć na nazwę osi odpowiedzialną za hamulec i przy uruchomionej aplikacji wpisać:

```
IMPORT TIME  
TIME.SLEEP(10)  
DISPLAY2CONTROL.PUT([1, NONE, 100])
```

PASS

Zawsze w trakcie pracy aplikacji jest możliwość sterowania ciężarówką za pomocą klawiszy **W,S,A,D**, co jest przydatne na początku testowania algorytmów wizyjnych.

A.2. Obsługa konsoli

Konsola deweloperska zapewnia możliwość zmiany czasu gry, pogody, prędkości gry oraz lokalizacji. Użyteczne komendy:

- *g_set_time hh* – komenda zmieniająca czas gry, w miejsce *hh* należy wpisać pożdaną godzinę
- *g_set_weather x* – komenda zmieniająca pogodę. W miejsce *x* można wpisać 1 lub 0 (słonecznie lub deszcz)
- *goto CITY* – zmienia lokalizację. Później należy jeszcze przeteleportować ciężarówkę klawiszem F9.
- *warp x* – zmiana prędkości gry. *X* to współczynnik prędkości. Wartości mniejsze od 1 zwalniają symulator, a wartości większe przyspieszają.