

**Projekt Indywidualny na kierunku Automatyka i
Robotyka Stosowana**

Politechnika Warszawska

Wydział Elektryczny

SYMULACJA SAMOCHODU W ŚRODOWISKU WEBOTS



Yaroslau Yakubouski, 01184625@pw.edu.pl

Semestr VI studiów I stopnia, Automatyka i Robotyka Stosowana

Data wykonania: **31.05.2025**

Prowadzący: dr inż. Witold Czajewski.

Streszczenie

Dany projekt powinien obejmować zamodelowanie w środowisku symulacyjnym Webots samochodu posługującego się systemem wizyjnym z kilku kamer oraz czujnikami ultradźwiękowymi dla realizacji algorytmu autonomicznego parkowania. Z wykonaniem projektu wiąże się pogłębienie wiedzy z lokalizowania robotów (w tym przypadku samochodu), ich nawigacji i percepji. Projekt więc obejmuje praktyczne wykorzystanie narzędzi dla sterowania samochodem autonomicznym, dla którego jest potrzebne zarówno zamodelowanie systemów czujnikowych, jak i kondycjonowanie i dalsze wykorzystanie danych pochodzących z nich. Głównym zastosowaniem systemu percepcyjnego samochodu w tym problemie jest wsparcie samodzielnego parkowania równoległego.

Na końcu opracowano prototyp automatu parkowania równoległego na podstawie systemu czujników ultradźwiękowych z wizualizacją przestrzeni „widoku z lotu ptaka”. W projekcie wykorzystano obszerną bibliotekę *OpenCV* dla zagadnień z wizji komputerowej.

Zastosowano rozwiązania zarówno oparte na czystej algorytmice (bez sztucznej inteligencji), jak i sieciach neuronowych. Projekt nie obejmował implementacji na rzeczywistym pojazdzie ani pełnej autonomii w ruchu drogowym, co stanowi naturalny kierunek dalszych prac.

Spis treści

1 WPROWADZENIE.....	3
2 MOŻLIWE ROZWIĄZANIA	4
2.1 WYBÓR ROZWIĄZANIA.....	6
2.2 WYBÓR NARZĘDZI.....	7
3 REALIZACJA PROJEKTU	9
3.1 RZECZYWISTY HARMONOGRAM I PODZIAŁ ZADAŃ	9
3.2 SZCZEGÓLOWY OPIS REALIZACJI POSZCZEGÓLNYCH ETAPÓW I ZADAŃ	10
3.3 OPIS WYKONANYCH ZADAŃ	10
3.4 SZCZEGÓŁY WYKONANIA ZADAŃ.....	11
3.4.1 OPIS ŚRODOWISKA SYMULACYJNEGO	11
3.4.2 CZUJNIKI ULTRADŹWIĘKOWE:	15
3.4.3 TWORZENIE WIDOKU „Z LOTU PTAKA”:	16
3.4.4 ESTYMACJA PARAMETRÓW ZEWNĘTRZNYCH KAMERY I TWORZENIE NA BIEŻĄCO BRYŁ WOKÓŁ OBIEKTÓW:.....	30
3.4.5 IMPLEMENTACJA STEREOWIZJI W FUZJI Z SIECIĄ NEURONOWĄ.....	32
3.4.6 AUTOMAT PARKUJĄCY	41
3.5 WYZWANIA NAPOTKANE PODCZAS WYKONANIA ZADAŃ.....	43
3.6 OPIS IMPLEMENTACYJNY	45
3.6.1 PLIK GŁÓWNY KONTROLERA PARKING_PARALLEL_NEW.PY.....	46
3.6.2 PLIK PARK_ALGO.PY	47
3.6.3 PLIK CAMERA_CALIBRATION.PY.....	49
3.6.4 PLIK VISUALISE.PY	51
3.6.5 PLIK STEREO_YOLO.PY	52
3.7 OPIS URUCHOMIENIOWY.....	54
4 PODSUMOWANIE I WNIOSKI.....	57
4.1 MOŻLIWOŚCI ROZBUDOWY.....	57
4.2 WYKORZYSTANIE SZTUCZNEJ INTELIGENCJI.....	58
BIBLIOGRAFIA	59
DODATEK	62
PLIK CAMERA_CALIBRATION.PY	63
PLIK VISUALISE.PY.....	70
PLIK PARK_ALGO.PY.....	78
PLIK STEREO_YOLO.PY.....	81
PLIK PARKING_PARALLEL_NEW.PY	84

1 Wprowadzenie

Powszechnie w samochodach autonomicznych stosuje się zestawy kamer, radarów, LiDAR-ów czy sonarów. Efektywne kondycjonowanie i przetwarzanie danych z tych sensorów stanowi złożone zagadnienie – problemy środowiskowe czy zakłóczenia nie zawsze da się wyeliminować jedynie sprzętowo, dlatego algorytmy muszą uwzględniać wiele scenariuszy i adaptować się do zmennego otoczenia. Badanie wszystkich możliwych sytuacji w rzeczywistym ruchu jest niepraktyczne, stąd korzystniejsze jest wykorzystanie symulatorów, takich jak CARLA, Gazebo czy Webots, które pozwalają w prosty sposób tworzyć scenariusze z wzorcami kalibracyjnymi, pojazdami i przeszkodami. W danym projekcie wykorzystano właśnie Webots – open-source'owe środowisko symulacyjne, posiadające obszerne możliwości dla modelowania robotów autonomicznych.

Webots udostępnia biblioteki dla tworzenia scenariuszy dla samochodów [14] z różnorodnymi czujnikami: ultradźwiękowe, kamery, radary, optyczne (LiDARy) itd.

W tym projekcie skonfigurowano tylko ultradźwiękowe czujniki (sonary) i system wizyjny 360°. W implementacji użyto 12-u sonarów oraz 8-u kamer z prototypu Tesla Hardware 3/4 [23], dostosowując ich parametry wewnętrzne i zewnętrzne, by uzyskać harmonijny widok 360° podobny do realnych instalacji Tesli Vision [23], [21].

Dla skutecznego projektowania tych systemów tych systemów dedykowanych specjalistycznemu użyciu zdefiniowano następujące cele:

- ogólny: realizacja parkowania równoległego,
- szczegółowe:
 - wykorzystanie fuzji czujników dla estymacji pozy samochodu w świecie,
 - badanie różnych aspektów wizji komputerowej dla samochodów,
 - segmentacja i lokalizacja przeszkód w globalnym układzie odniesienia,
 - planowanie ścieżki od bieżącego położenia samochodu do docelowego (w miejscu parkingowym).

Analizowano różne metody lokalizacji i nawigacji wykorzystywane w przemyśle – Tesla stosuje wyłącznie wizję komputerową [21], podczas gdy Waymo łączy kamery, radary i LiDARy [22]. Równocześnie przegląd literatury dotyczącej detekcji obiektów, segmentacji scen czy „sklejania” przekształconych przez homografię panoram dostarczył inspiracji do wyboru wiarygodnych rozwiązań [1], [10]. Projekt nie dąży jednak do pełnej autonomii w stylu produkcyjnym, lecz koncentruje się na sprawdzonym zestawie sensorów i sieci neuronowych do realizacji bezpiecznego i precyzyjnego parkowania.

2 Możliwe rozwiązania

Dla realizacji parkowania autonomicznego równoległego musimy rozważyć takie kroki:

- skuteczne przetwarzanie obrazów z kamer dla detekcji obiektów, segmentacji otoczenia i lokalizacji ewentualnych przeszkód wobec samochodu,
- wykorzystanie czujników ultradźwiękowych dla detekcji miejsca parkingowego,
- wykorzystanie tychże czujników dla reaktywnego planowania ścieżki do miejsca parkingowego.

Inaczej mówiąc, kamery lub czujniki powinny odnajdywać miejsce parkingowe, moduł odometryczny z połączeniem z IMU lub GPS musi śledzić położenie samochodu względem tego miejsca, a dla modułu sterowania pozostaje zadanie ustawienia parametrów ruchu samochodu: prędkości i skrętu kół. Wszystkie te operacje powinny się wykonywać w czasie rzeczywistym – oznacza to dodatkowe zadanie ewentualnej optymizacji programu.

Na początku należy przedstawić istniejące, zweryfikowane na rzeczywistych pojazdach metody estymacji pozycji miejsca parkingowego. Niektóre dotyczą estymacji pozycji miejsca parkingowego na podstawie przekształconego widoku „z lotu ptaka” na podstawie linii oznakowania poziomego, na przykład praca [8] wykorzystuje transformację na przestrzeń Radona dla odnalezienia tych linii. Jest to czysto algorytmiczny sposób, nie wykorzystujący sieci neuronowych, ale nie jest z natury uniwersalny – w projekcie danym podjęta była decyzja, aby nie opierać się na znacznikach zewnętrznych, a tylko na lokalizacji samych przeszkód. Opisywana metoda jest skuteczna na parkingach typu lotniskowego lub przy wielkopowierzchniowych centrach handlowych, jednak w środowisku miejskim, gdzie brakuje oznakowania poziomego, jej zastosowanie jest ograniczone.

Rozwiązanie [9] dla przykładu wykorzystuje sieć YOLO-v3 dla detekcji narożników miejsc parkingowych oraz własną architekturę dla określenia wolnych miejsc, nauczoną na zbiorze danych, składającym się z dużej ilości zdjęć widoku „z lotu ptaka”. Ten sposób też korzysta z oznakowania poziomego, co jest niezgodne z problematyką projektu.

Lokalizacja obiektów w przestrzeni może też być sporządzona na podstawie fuzji wyników inferencji sieci segmentacyjno-klasyfikacyjnych (typu YOLO) oraz transformerów do estymacji głębi [17]. Takie metody jednak są „czarną skrzynką”, nie możemy przewidzieć jednoznacznie wyników analizy obrazów przez sieci neuronowe. Również nie każda sieć może mieć tak szybką inferencję, jak opisują twórcy modeli.

Estymację głębi można też sporządzić za pomocą tzw. „*disparity estimation*” (dysparcja, lepiej to jest udokumentowane w literaturze anglojęzycznej). Krótko mówiąc, to zagadnienie pozwala na estymację położenia punktu w przestrzeni 3D na podstawie różnic w projekcji obrazów z dwóch kamer. Jest to tak zwane zagadnienie stereowizji. Szczegóły techniczne będą opisane w sekcji „Realizacja projektu”.

Przesuwając się do nie bazujących na wizji sposobach, to przykładem może posłużyć publikacja [6], która pokazuje całkowicie automatyczny pipeline jedynie z czujnikami ultradźwiękowymi (przejazd po parkingu i automatyczne poszukiwanie miejsca). Ten artykuł bazuje się na wspomnianych w tymże artykule projektach [12]. Te obie prace mogą posłużyć dużą pomocą przy projektowaniu algorytmu parkowania autonomicznego równoległego, osobiście w zakresie lokomocji.

Budowanie widoku „z lotu ptaka” jest osobnym zagadnieniem. Jest to wciąż korzystne narzędzie, nawet jeżeli i nie zawsze ono się sprawdza jako medium pomiarowe. Na widoku z lotu ptaka, znając odpowiednią homografię, można odzwierciedlić punkty w świecie, w globalnym układzie odniesienia. Na podstawie tych punktów można w bardziej dogodny sposób ocenić położenie różnych obiektów.

Widok „z lotu ptaka” można otrzymać z każdej kamery, znając odpowiednie przekształcenie między płaszczyznami – jest to w efekcie homografia. Można robić to na podstawie punktów charakterystycznych (detektory typu SIFT, ORB, SURF [11]) – takie rozwiązanie pomija potrzebę ciągłej obecności wzorców kalibracyjnych typu znaczników i szachownic w polu widzenia kamer. W bibliotece wizji komputerowej OpenCV możemy znaleźć klasę (API) „cv2.Stitcher” [19], służącą do automatycznego sklejania i mieszania obrazów w panoramę. Posiada ona również metody dla sporządzenia panoram, pod warunkiem, że obrazy już zostały przekształcone na jedną płaszczyznę [18]. Pozostaje do sprawdzenia, jak złożoność takich metod wpłynie na wydajność projektu w czasie rzeczywistym.

Jest również wiele sposobów na sporządzenie takiego widoku off-line – estymacja przekształceń na podstawie znanych wzorców i stosowanie tych przekształceń w czasie rzeczywistym. Implementacja takiego rozwiązania może być o wiele prostsza, a efektywność nawet w niektórych przypadkach może wzrosnąć.

Firma Google do tworzenia swoich znanych zdjęć Google Earth [16] stosuje algorytm przepływu optycznego („optical flow”) – precyzyjny sposób na śledzenie punktów charakterystycznych poprzez kilka płaszczyzn obrazowych, natomiast może być bardzo wymagające obliczeniowo i trudne w implementacji na systemach działających w czasie rzeczywistym.

2.1 Wybór rozwiązania

Podstawowymi kryteriami, jakimi kierowano się przy wyborze rozwiązania, były prostota implementacji, efektywność działania w czasie rzeczywistym oraz łatwa optymalizacja kodu. Pod uwagę wzięto kilka wariantów możliwych do realizacji, co umożliwiało elastyczną adaptację w przypadku, gdyby pierwsze podejście nie spełniło oczekiwania.

Po wstępnej analizie istniejących podejść, takich jak ręczne operowanie homografią versus automatyczne narzędzia do sklejania obrazów oraz metody głębi bazujące na sieciach neuronowych versus stereowizja, dokonano wyboru rozwiązania obejmującego następujące działania:

- 1) sporządzenie odpowiedniego środowiska (mapy) w Webots, umożliwiającego wygodne testowanie różnych warunków i scenariuszy,
- 2) automatyczna kalibracja kamery na podstawie znanych wzorców i przekształcenie obrazów z kamer za pomocą homografii:
 - działania te są podyktowane możliwością łatwego i powtarzalnego testowania różnych scenariuszy bez konieczności kosztownych prób sprzętowych. Technika ta jest szeroko stosowana i dobrze udokumentowana ([OpenCV Camera Calibration](#)),
- 3) generowanie widoku „z lotu ptaka”:
 - rzutowanie obrazów na wspólną płaszczyznę: uniwersalna i teoretycznie najdokładniejsza metoda,
 - łączenie obrazów poprzez wzajemne przekształcenia homograficzne: elastyczna metoda, pozwalająca na łatwe odtwarzanie i ponowne sporządzenie widoku,
 - próba użycia klasy „cv2.Stitcher”: potencjalnie prosta w użyciu, choć mniej elastyczna opcja, wybrana jako alternatywa do szybkich testów, z możliwością rezygnacji, jeśli nie spełni wymogów jakościowych lub wydajnościowych,
- 4) użycie czujników ultradźwiękowych do skanowania przestrzeni wokół pojazdu
 - rozwiązanie sprawdzone w licznych implementacjach ([2], [6], [8]) i cechujące się niską złożonością obliczeniową,
- 5) detekcja i lokalizacja obiektów za pomocą sieci neuronowej YOLO i rysowanie brył na ich miejscu:
 - metoda ta zapewnia bardzo dobrą równowagę pomiędzy szybkością a dokładnością detekcji obiektów [13] – najpowszechniej spotykany sposób wizualizacji obiektów w przestrzeni 3D dla samochodów autonomicznych,

- 6) estymacja głębi metodą stereowizji:
 - jest to podejście klasyczne i dobrze opisane w literaturze [20]. Zaletą jest wysoka dokładność przy stosunkowo prostej implementacji, choć w rzeczywistych warunkach wymagane jest precyzyjne skalibrowanie kamer,
- 7) próba estymacja głębi za pomocą sieci neuronowych [7],
- 8) implementacja wielowątkowości:
 - realizacja systemu autonomicznego wymaga jednoczesnej obsługi wielu sensorów i równoległych obliczeń, stąd decyzja o wielowątkowości; zapewnia to realistyczną simulację zachowania pojazdu autonomicznego,
- 9) optymalizacja obliczeń graficznych z wykorzystaniem CUDA:
 - dzięki użyciu biblioteki OpenCV z CUDA, intensywne operacje, takie jak korekcja perspektywy i obróbka dużych obrazów, będą wykonywane na karcie graficznej, co znaczco zwiększy efektywność całego systemu ([OpenCV CUDA documentation](#)).

Ostatnie dwa działania są realizowane równocześnie z innymi, i zapewni to stopniowe podnoszenie efektywności i realizmu działania systemu. Alternatywne rozwiązania, jak manualne podejścia bez wsparcia GPU lub wielowątkowości czy zastosowanie innych bibliotek sieci neuronowych, pozostają otwarte na wypadek, gdyby przepisanie większości kodu wizyjnego na funkcje dopasowane pod CUDA byłoby problematyczne.

2.2 Wybór narzędzi

Dla skutecznej realizacji wszystkich postawionych w projekcie zadań należało wykorzystać różne narzędzia zarówno sprzętowe, jak i programistyczne, ułatwiające programowanie symulacji i testowanie różnych scenariuszy.

Poniżej znajduje się spis owych narzędzi, ich opis, powody wyboru i wykorzystanie w projekcie:

Narzędzia sprzętowe:

- karta graficzna NVIDIA RTX 2060 Mobile:
 - przeznaczenie i funkcjonalność: akceleracja obliczeń równoległych na dużych obrazach wysokiej rozdzielczości. Dzięki architekturze CUDA (Compute Capability 7.5) możliwe było znaczne przyspieszenie intensywnych obliczeniowo operacji, takich jak korekcja perspektywy (homografia).
 - powód zastosowania: dostępność karty z wydajną architekturą CUDA oraz wystarczającą ilością pamięci (6 GB DDR6), umożliwiającą komfortową pracę z obrazami o dużej rozdzielczości.

Narzędzia programistyczne:

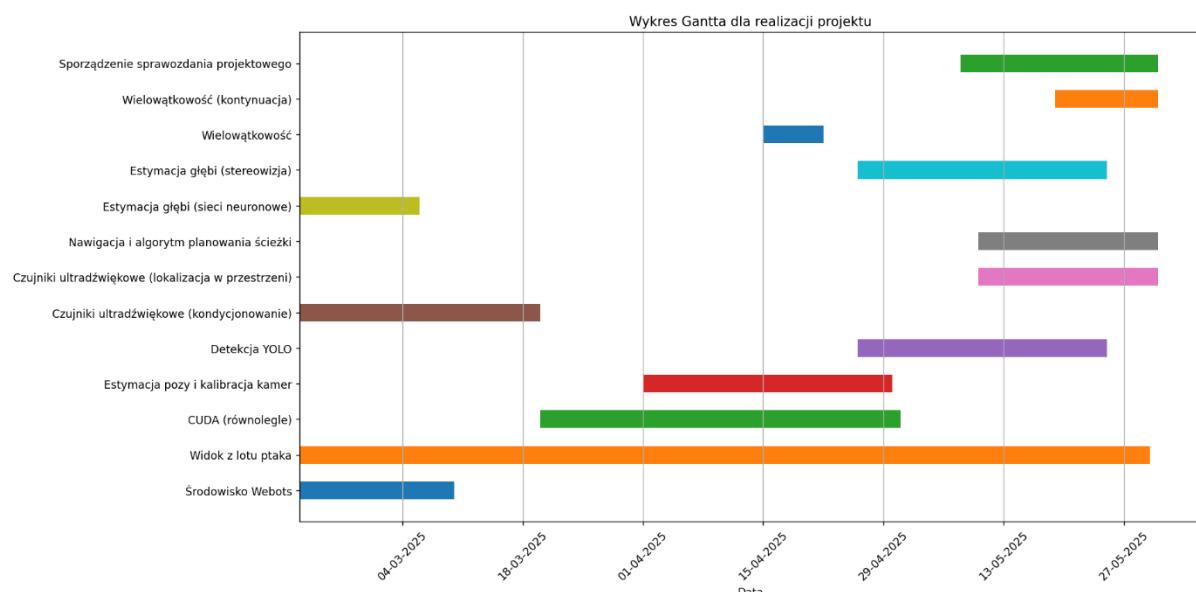
- Python:
 - przeznaczenie i funkcjonalność: zasadniczy język programowania, użyty do napisania kontrolera symulacji i pozostałych skryptów w środowisku Webots. Umożliwia łatwą implementację skryptów do przetwarzania obrazów i integrację z sieciami neuronowymi.
 - powód zastosowania: bogata baza bibliotek (między innymi OpenCV, PyTorch), dobra dokumentacja oraz łatwość integracji różnych narzędzi.
- OpenCV z CUDA:
 - przeznaczenie i funkcjonalność: biblioteka przetwarzania obrazów, umożliwiająca wykonywanie operacji bitowych, transformacji geometrycznych (np. korekcja perspektywy przez homografię) oraz efektywną obsługę dużych zbiorów danych (obrazów o wysokiej rozdzielcości) na karcie graficznej NVIDIA.
 - powód zastosowania: wsparcie dla jąder CUDA, przyspieszające obliczenia na karcie graficznej, będącej w wyposażeniu laptopa osobistego,
 - uwagi instalacyjne: konfiguracja biblioteki na innych językach, niż Python (np. C++), wymagałaby budowania OpenCV ze źródła, co jest procesem nietrywialnym i zależnym od specyficznych wersji sterowników NVIDIA, CUDA Toolkit i pozostałych pakietów. Ogólna procedura dla Python jest opisana w [opisie uruchomieniowym](#). Szczegółowa instrukcja, która pomoże uniknąć problemów, dostępna jest [tutaj](#).

3 Realizacja projektu

W danej sekcji będą opisane szczegóły wykonania wspomnianych w poprzednich rozdziałach zadań programistycznych. Będą tutaj również przytoczone fragmenty z bazy kodowej, wspomagające zrozumienie działania aplikacji: detekcja i analiza wzorców kalibracyjnych, przekształcenie homografii, algorytmy sklejania i mieszania ROI („region of interest”) obrazów, estymacja pozy kamery, detekcja obiektów i lokalizacja w przestrzeni; skrypty dla wizualizacji widoku „z lotu ptaka” i strefy detekcji czujników ultradźwiękowych; algorytm poszukiwania miejsca parkingowego i nawigacji samochodu po ścieżce do znalezionej miejsca. Podane tutaj zostaną oprócz zaimplementowanych i działających metod również eksperymentalne i niektóre nawet ukończone niepowodzeniem. Opisane zostaną sposoby

3.1 Rzeczywisty harmonogram i podział zadań

Poniżej umieszczony jest wykres Gantta, opisujący rzeczywisty harmonogram wykonywania zadań w projekcie:



Rys. Wykres Gantta, podsumowujący chronologię realizacji projektu i wspomagający ocenę skuteczności działań.

Większość czasu zajęło zagadnienie, które się okazało nie tak trywialne – sporządzenie widoku „z lotu ptaka”. Jak będzie to opisane w dalszym toku sprawozdania, istnieje wiele różnych sposobów na to, a dla każdego należy wypracować pewną intuicję i metodykę dla szybkiego wykonania.

Fenomen skupienia działań bliżej daty oddania pracy jest w dużej mierze często spotykany w sferze akademickiej, istnieje on również i tu.

W ostatnich fazach projektu pewien wysiłek był oddany opracowaniu metod detekcji obiektów za pomocą systemu wizyjnego (sieci neuronowe i stereowizja) oraz poszukiwania miejsca parkingowego wraz z planowaniem ścieżki.

Należy jasno zaznaczyć, że algorytm parkowania na razie nie został zaimplementowany do końca, natomiast trwały aktywne prace rozwojowe. Szczegóły są w sekcji „*Automat parkujący*”.

3.2 Szczegółowy opis realizacji poszczególnych etapów i zadań

W dalszych rozdziałach w głębszych szczegółach będą opisane wszystkie etapy projektu, napotkane wyzwania i rozwiązania dla nich. Większość czasu była spędzona za dorabianiem i wyrafinowaniem systemów wspomagających parkowanie, zaś same autonomiczne parkowanie nie było zrobione do końca.

3.3 Opis wykonanych zadań

W całokształcie dotychczas zostały wykonane następujące czynności:

- 1) kondycjonowanie sygnałów z czujników ultradźwiękowych, przetworzenie je na metry oraz wizualizacja dynamiczna ich zakresu detekcji,
- 2) zbudowanie widoku z lotu ptaka za pomocą 6 wyżej wymienionych kamer w następujące sposoby:
 - a) zszywanie wyprostowanych na płaszczyznę ziemi obrazów z kamer za pomocą szachownic umieszczonych we wspólnych dla kolejnych par kamer obszarach łańcuchowo;
 - b) zszywanie wyprostowanych obrazów w podobny sposób, jak wyżej, ale w innej kolejności – klejanie przednich obrazów, tylnych obrazów oraz układanie tych połówek za pomocą dwóch szachownic z każdej strony,
 - c) nałożenie wyprostowanych obrazów zgodnie z położeniem narożników szachownic na metrycznie wyskalowanej kanwie,
- 3) kalibracja każdej kamery – znalezienie ich parametrów wewnętrznych oraz położenia w globalnym układzie odniesienia, który bazowany jest na środku samochodu, a jego osie są skierowane zgodnie z modelem Ackermann^[1],
- 4) projekcja trójwymiarowych prostopadłościanów w miejscu odnalezionych samochodów na podstawie obrazu z kamery przedniej oraz ich estymowanej lokalizacji w układzie globalnym (środek samochodu) za pomocą sieci neuronowej YOLO8,
- 5) zagadnienie stereowizji i polepszenie poprzedniego zadania o sieć segmentacyjną YOLO11,
- 6) przygotowanie automatu parkowania i prototypu odczytu miejsca parkingowego.

¹ [P. Corke, Robotics, Vision & Control, Chapter 4.](#)

3.4 Szczegóły wykonania zadań

W poniższych sekcjach zostaną opisane wykonane zadania z większym uwzględnieniem szczegółów implementacji.

3.4.1 Opis środowiska symulacyjnego

Pierwszym etapem było zamodelowanie samego samochodu oraz mapy w środowisku symulacyjnym Webots. Jako wzorzec dla mapy wzięto przykładową ulicę importowaną za pomocą narzędzia OpenStreetMapImporter w symulatorze^[2]. Na poniższym rysunku (Rys. 1) pokazano jej ogólny widok:

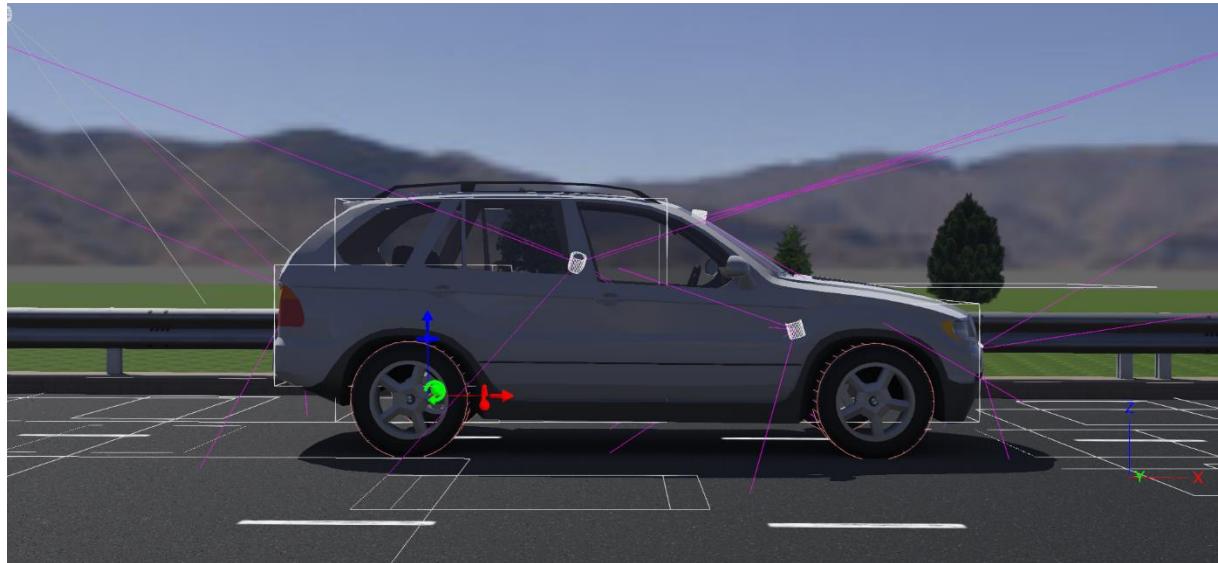


Rys. 1. Widok mapy dla testowania autonomicznego parkowania w symulatorze.

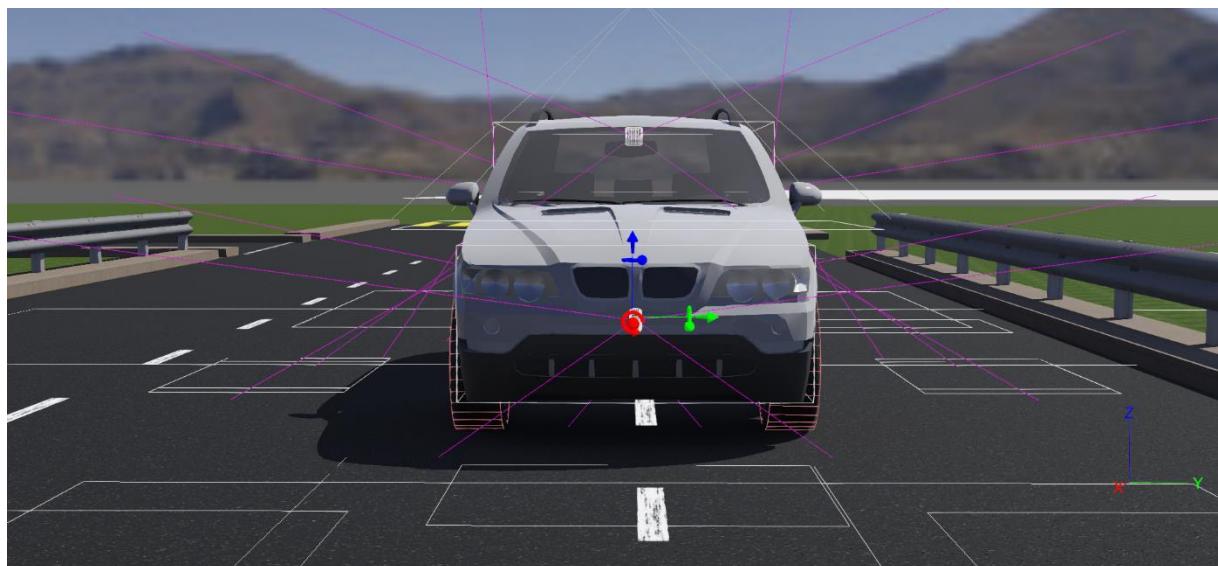
Na mapie są ulice na których można umieścić samochody i które w przybliżeniu odpowiadają takim, które zwykle można zobaczyć w europejskich miastach – nie oznakowane liniami poziomymi (tylko na początku i końcu), ale oznaczone jako obszary zezwalające na postój. W tym projekcie jednak nie zakłada się rozpoznania linii parkingowych za pomocą sieci neuronowych, tak więc wykonano tylko uproszczoną wersję mapy.

² <https://cyberbotics.com/doc/automobile/openstreetmap-importer>

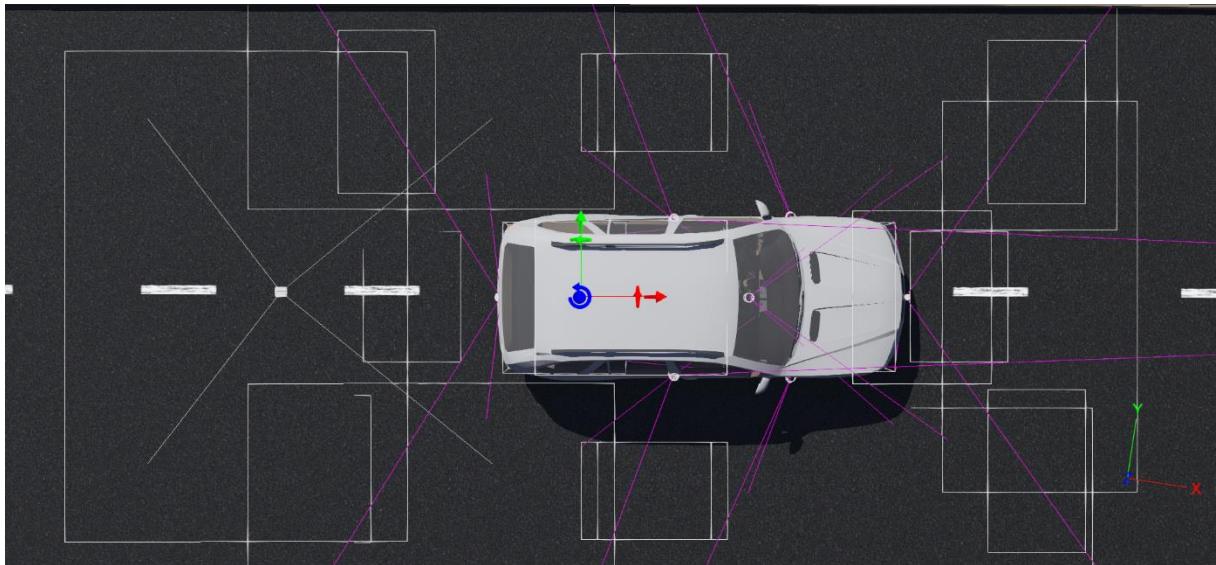
Dalej na Rys. 2, Rys. 3, Rys. 4, Rys. 5 są przedstawione widoki z góry, z boku oraz z przodu samochodu wykorzystanego w symulacji. Dodatkowo na Rys. 6 pokazano przybliżone pola widzenia kamer z góry:



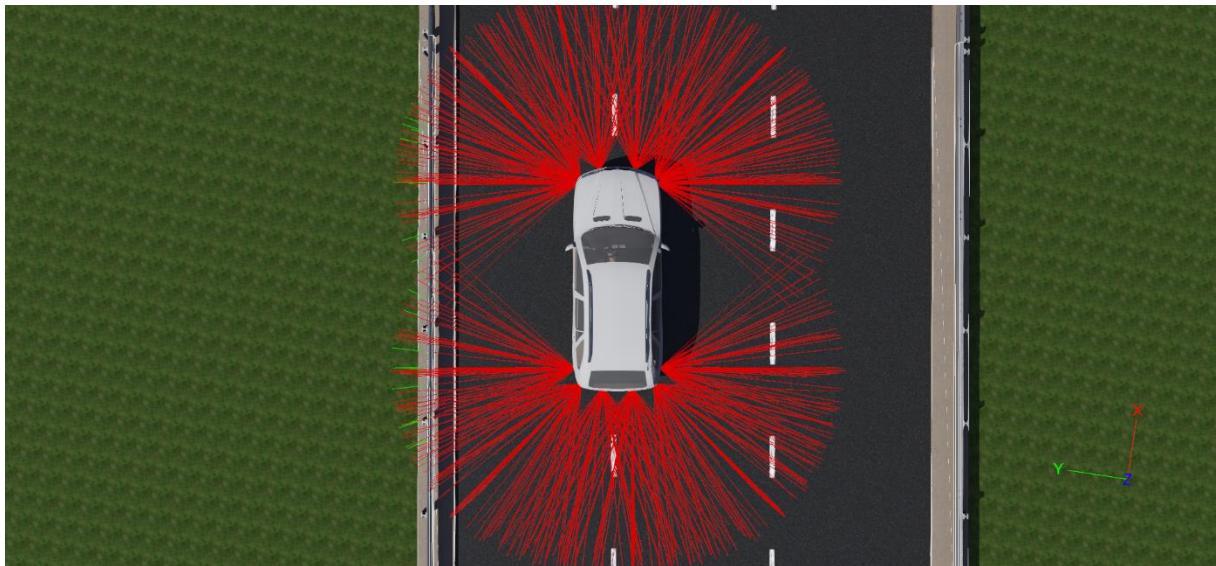
Rys. 2. Widok z boku samochodu. Cylindryczne obiekty i fioletowe linie – symboliczna reprezentacja kamer i projekcji ich pola widzenia. Białe linie – gabaryty szachownic, o których mowa dalej w sprawozdaniu.



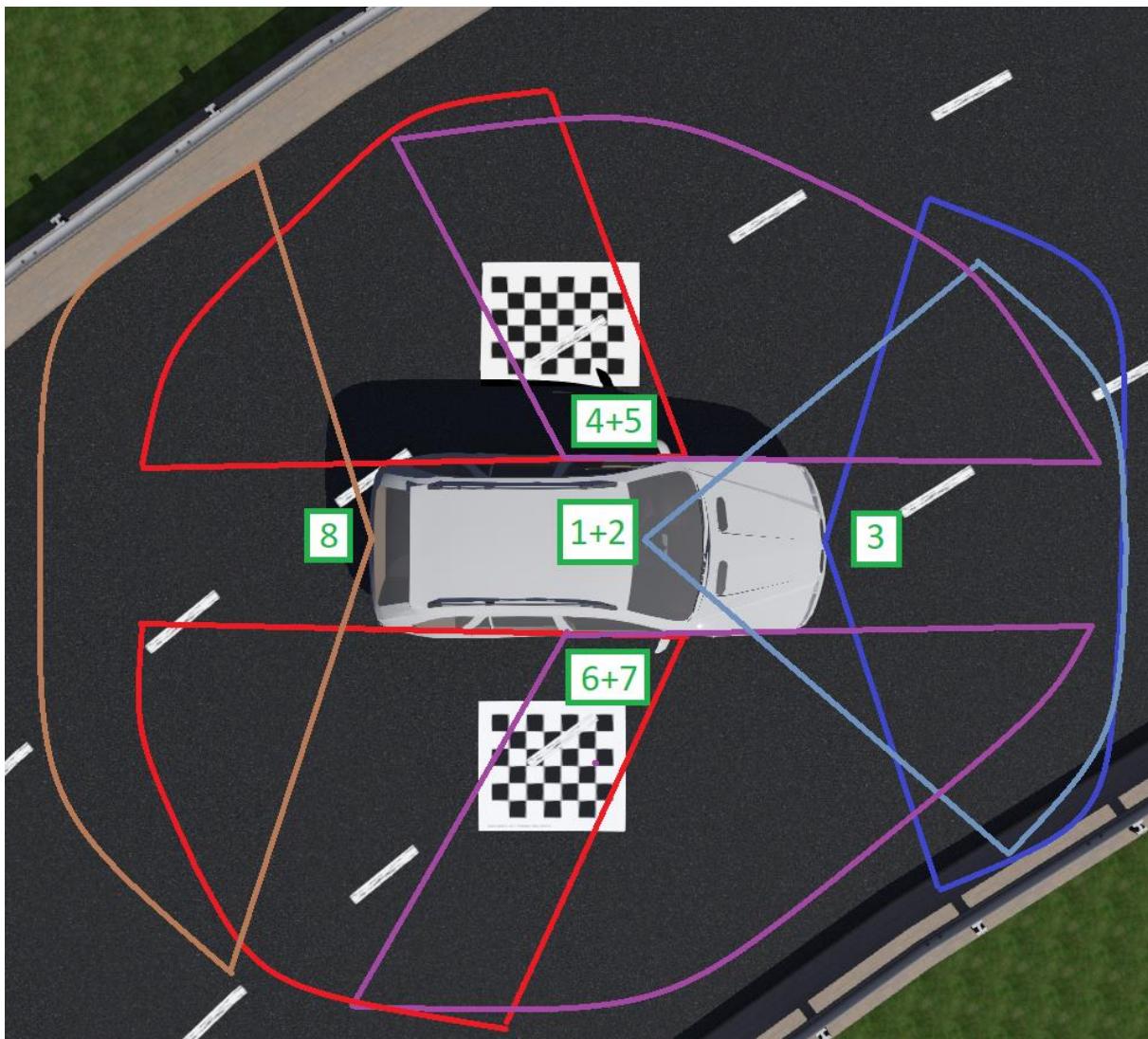
Rys. 3. Widok z przodu samochodu.



Rys. 4. Widok z góry samochodu. Dodatkowa kamera z tyłu patrząca w dół – wykorzystywana do kalibracji, mowa o niej dalej w sprawozdaniu.



Rys. 5 .Widok z góry samochodu. Czerwone „wąsy” – symboliczna reprezentacja zakresu czujników ultradźwiękowych.



Rys. 6. Widok z góry samochodu, gdzie są zaznaczone użyteczne pola widzenia poszczególnych kamer.

Podsumowanie systemów czujnikowych:

- **8 kamer widoku 360°**, 6 z których pozwala na zbudowanie widoku z lotu ptaka:
 - 1) kamera o polu widzenia 90° za szybą czołową najlepiej nadająca się do rozpoznawania rozpoznawania obiektów w odległości ok. 10 m od samochodu (nr 1),
 - 2) taka sama kamera, jak wyżej, dla stereowizji: pole widzenia 90°, przesunięta za szybą czołową o 3 cm wlewo (nr 2),
 - 3) kamera szerokokątna o polu widzenia 135° na przednim zderzaku (nr 3),
 - 4) 2 kamery szerokokątne o polu widzenia 150° usytuowane na kolumnach drzwiowych i patrzące do przodu (nr 4 i 6),
 - 5) 2 kamery o polu widzenia 75° położone na błotnikach prawym i lewym (za przednimi kołami) i patrzące do tyłu (nr 5 i 7),
 - 6) kamera tylna szerokokątna o polu widzenia 135° położona nad tablicą rejestracyjną (nr 8),

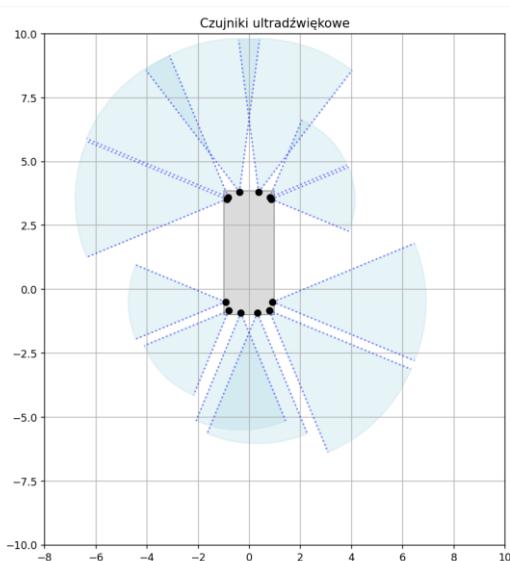
- 7) rozdzielcość robocza 1080p (1920x1080), kalibracja i zbudowanie widoku „z lotu ptaka” wykonane przy rozdzielcości 4K (3840x2160).
- **12 czujników ultradźwiękowych**, 6 z przodu, 6 z tyłu, 2 przednich bocznych czujnika są w stanie precyjnie odnaleźć takie niskie obiekty, jak np. krawężnik.

3.4.2 Czujniki ultradźwiękowe:

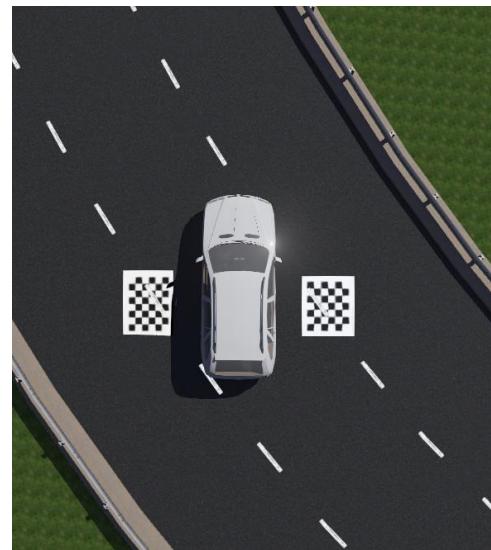
Zainstalowanie czujników na samochodzie w Webots jest rzeczą trywialną, a środowisko symulacyjne pozwala na łatwe odzyskanie ich współrzędnych w układzie samochodu.

Jak było omówiono wcześniej, na samochodzie usytuowano 12 czujników, które pozwalają na lokalizację przeszkód, które są postawione maksymalnie pod kątem 45° od normalnej do nich. Można zaobserwować, że niektóre samochody mają 10 lub 12 – dla większej rozdzielcości i czasami dla systemu autonomicznego parkowania. Te dodatkowe 2 (lub 4) znajdują się właśnie prawie przy kołach, prostopadle z boku. Pewne publikacje wspominają, że dodatkowe czujniki prostopadłe do samochodu wspomagają w dużej mierze parkowanie prostopadłe [3].

Sporządzono więc wizualizację strefy detekcji tych czujników dla wspomagania procesu decyzyjnego w sprawie parkowania autonomicznego. Dalej na rys. Rys. 7 i rys. Rys. 8 jest pokazany przykładowy widok tej wizualizacji i samochodu wobec przeszkód:



Rys. 7. Wizualizacja stref detekcji czujników ultradźwiękowych.



Rys. 8. Widok samochodu z góry na drodze – odnośnik do wizualizacji czujników ultradźwiękowych.

W wizualizacji posługiwano się zadanimi parametrami modeli czujników: każdy jest modelowany jako stożek prążków (im więcej, tym lepsza rozdzielcość, ale też zapotrzebowanie na zasoby). Kierunek stożków definiuje ich kąt zamontowania, położenie da się znaleźć w symulatorze, a maksymalne pole tych stożków jest definiowane przez maksymalną odległość przetwarzaną (tworząca stożku) oraz kąt otwarcia („aperture”). Jest

podyktowana przez tzw. „lookup table”^[3], sygnały i odległości są przetwarzane według tej tabeli.

Niektóre przeszkody nie są wskazywane przez czujniki. W symulatorze każdy obiekt może mieć tzw. „bounding object” – definiuje to kolizję i modelowanie fizyki w ogóle. Nie wszystkie jednak obiekty da się rozpoznać – możliwe też, że obiekty niewidoczne są dla czujników, ponieważ są położone tak, że większość promieni stożka „pola widzenia” czujnika pada na obiekt tak, że odbity promień nie powraca do czujnika i jego wartość jest wskazywana jako maksymalna przez „lookup table”. Jak wspomniano na początku tego rozdziału, zgodnie z [dokumentacją Webots](#) czujniki ultradźwiękowe właśnie mogą nie odnaleźć płaszczyzny, jeżeli jej normalna jest położona dalej niż $22,5^\circ$ od promieni czujnika. To zjawisko jest pokazano na rysunku dostępnym w powyższym odnośniku.

3.4.3 Tworzenie widoku „z lotu ptaka”:

Widok „z lotu ptaka” jest tworzony za pomocą odpowiednich przekształceń obrazów z kamer, w wyniku których powstaje obraz podobny do takiego z wirtualnej kamery. Obrazy są ogólnie rzutowane poprzez homografię na płaszczyznę drogi. W projekcie nie jest wykorzystywany ten widok jako narzędzie pomiarowe wskutek pewnych swoich niedoskonałości, natomiast pozostaje dobrą wizualizacją otoczenia.

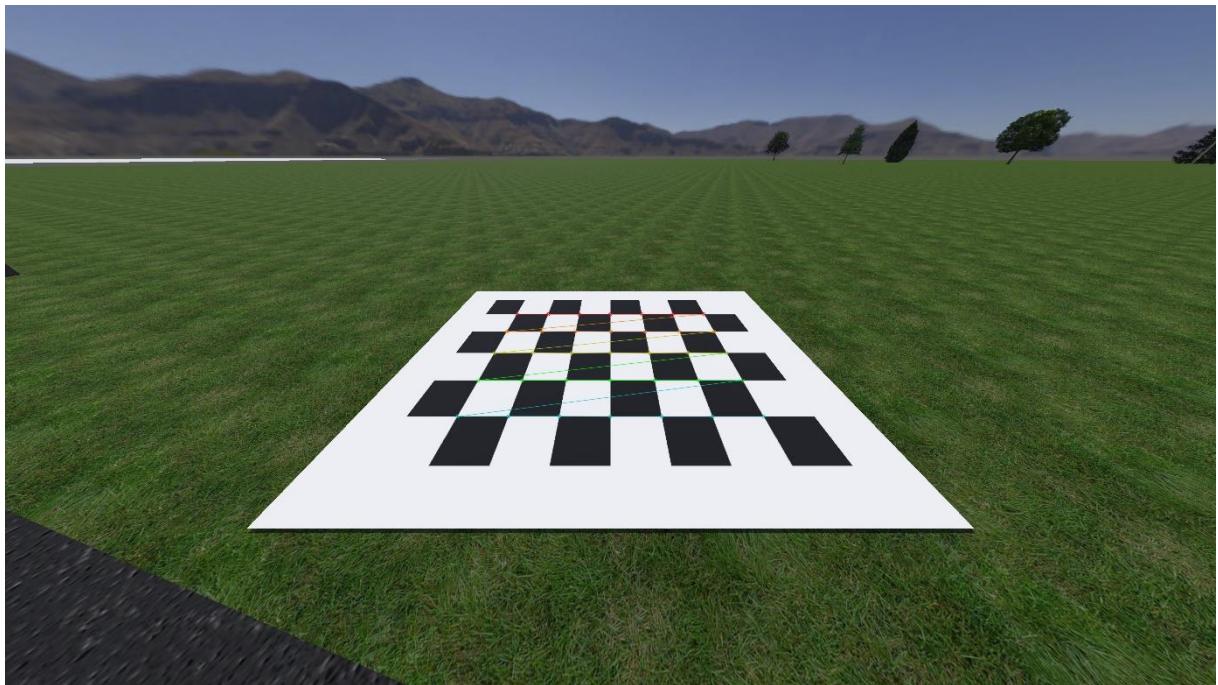
Można wyodrębnić sposoby, na które został wykonany ten widok:

- 1) zszywanie obrazów za pomocą szachownic znajdujących się we wspólnych obszarach dla kamer i łączenie je łańcuchowo – jedna po jednej po kolej i aż do zamknięcia pętli,
- 2) zszywanie obrazów w częściach – część kamer, patrzących do tyłu i część kamer, patrzących do przodu są łączone osobno (względem odpowiednio przedniej i tylnej); później połówki są łączone łańcuchowo z obu stron za pomocą szachownic we wspólnych obszarach dla kamer,
- 3) nakładanie wyprostowanych obrazów z kamer na kanwę o określonym rozmiarze: dla każdej kamery jest określony zestaw punktów na płaszczyźnie drogi, i rzutowanie tych obrazów na kanwę odbywa się za pomocą homografii, łączącej punkty na wyprostowanym obrazie z odpowiadającymi im na kanwie.

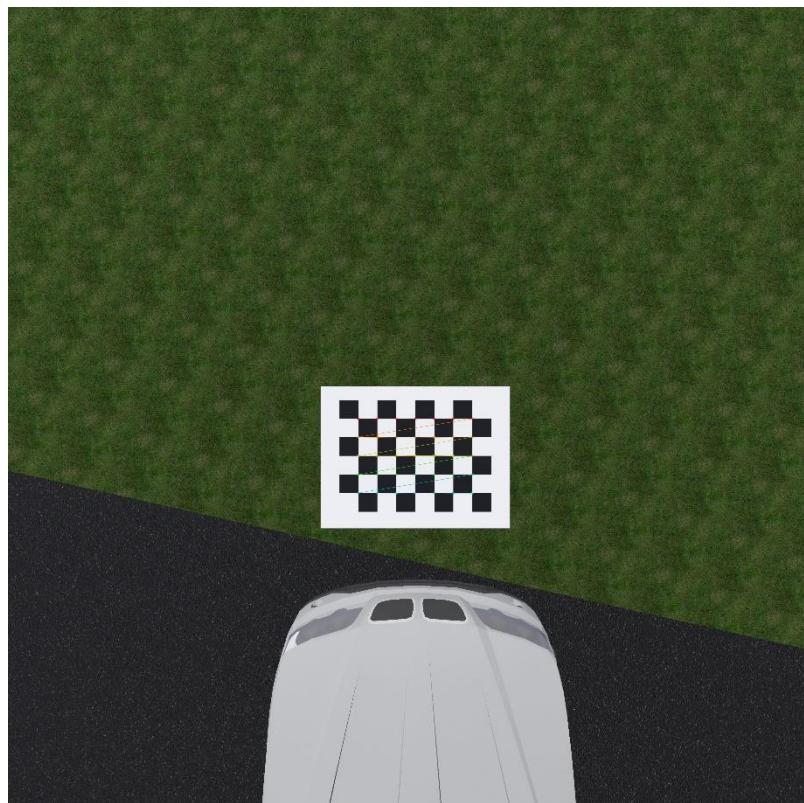
Procedura wyprostowania (nie mylić z pojęciem „rectify”) obrazów z poszczególnych kamer była następująca: na kamerze badanej są odnajdywane narożniki szachownicy, a na kamerze z góry o rozdzielczości obrazu 3600×3600 i poziomym polu widzenia 90° widoczna jest ta sama szachownica. Kamera druga „widzi” płaszczyznę drogi, w ten sposób jest określone przekształcenie między tą płaszczyzną a płaszczyzną obrazową badanej kamery.

³ <https://cyberbotics.com/doc/reference/distancesensor#lookup-table>

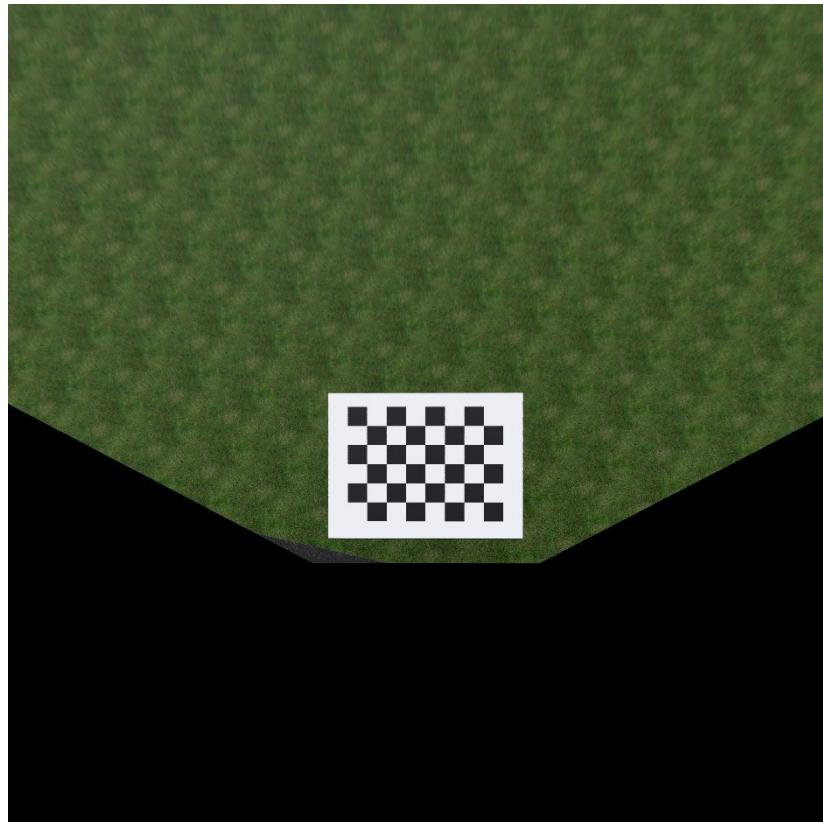
Przykładowy sposób działania tej metody w symulatorze jest zilustrowany na Rys. 9, Rys. 10 oraz Rys. 11:



Rys. 9. Obraz z kamery przedniej na zderzaku, z której widać szachownicę 7x5. Widoczne również narysowane narożniki przez metodę cv.drawChessboardCorners().



Rys. 10. Obraz z kamery u góry. Również są widoczne narysowane narożniki tej samej szachownicy.



Rys. 11. „Wyprostowany” obraz z przedniej kamery. Wszystkie narożniki szachownicy na obrazie tej kamery zostały umieszczone dokładnie w miejscach, gdzie były widoczne narożniki z drugiego obrazu – wynik nałożenia homografii.

Na monitorze FullHD (1920x1080) punkty i linie z obrazu 4K (3840x2160) są cieńsze i mniejsze, niż byłyby na przykład z 720p. Takie wyprostowanie daje średni błąd reprojekcji dla obrazów 4K i 3600x3600 równy 0.294-0.297 px. To jest wystarczająco dobry wynik, aby móc na nim oprzeć dalsze postępowania (mniej niż 0.5 px dla rozdzielczości 4K traktuje się jako bardzo dobry wynik). Na obrazach o mniejszej rozdzielczości ewentualne błędy będą się powielaty i będą bardziej widoczne. Błąd reprojekcji liczy się jako odległość euklidesowa (między wektorami znormalizowanymi) punktów na obrazie otrzymanych po zastosowaniu homografii od punktów źródłowych ponownie przekształconych homografią. Inaczej mówiąc, to bardziej określa błędy obliczeniowe po drodze, m. in. w metodach `cv.warpPerspective()` i `cv.findHomography()`.

Czasami kolejność narożników szachownicy do analizy przez `cv.findHomography()` trzeba odwracać, żeby nie policzyło się przewrócone przekształcenie.

Ponadto każdy obraz wyprostowany został poddany odpowiednim translacjom takim, żeby pozbyć się niepotrzebnych czarnych obszarów oraz widocznych na bocznych kamerach drzwi i błotników samochodu. Takie przekształcenie uzyskiwane jest poprzez lewostronne przemnożenie macierzy homografii przez macierz translacji:

$$H_{nowa} = T * H = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} * H$$

Gdzie t_x i t_y – przesunięcie odpowiednio w kierunkach osi X i Y. Na obrazie przyjęto, że początek leży w górnym lewym rogu obrazu, i oś Y skierowana dodatnio w dół, a oś X – dodatnio w prawo.

Metoda ta jest wygodniejsza niż ręczne określanie narożników punkt po punkcie w układzie metrycznym, ponieważ pozwala używać wzorca szachownicy o dowolnej wielkości i w każdej chwili zmienić jego rozmiar, wystarczy dostosować liczbę pól w kodzie. W warunkach rzeczywistych montaż konstrukcji ruchomej kamery do pomiaru homografii wymagałby znacznego nakładów fizycznych i finansowych, podczas gdy zastosowanie wzorca i kalibracja na podstawie obrazu referencyjnego dają porównywalną dokładność przy minimalnym koszcie. Ponadto to właśnie ta procedura łączy metodę prostowania i nakładania widoków (patrz rozdział dotyczący [trzeciego sposobu](#)) z uzyskiwaniem jednorodnego dopasowanego obrazu „z lotu ptaka” bez konieczności stosowania jedynie metrycznych pomiarów.

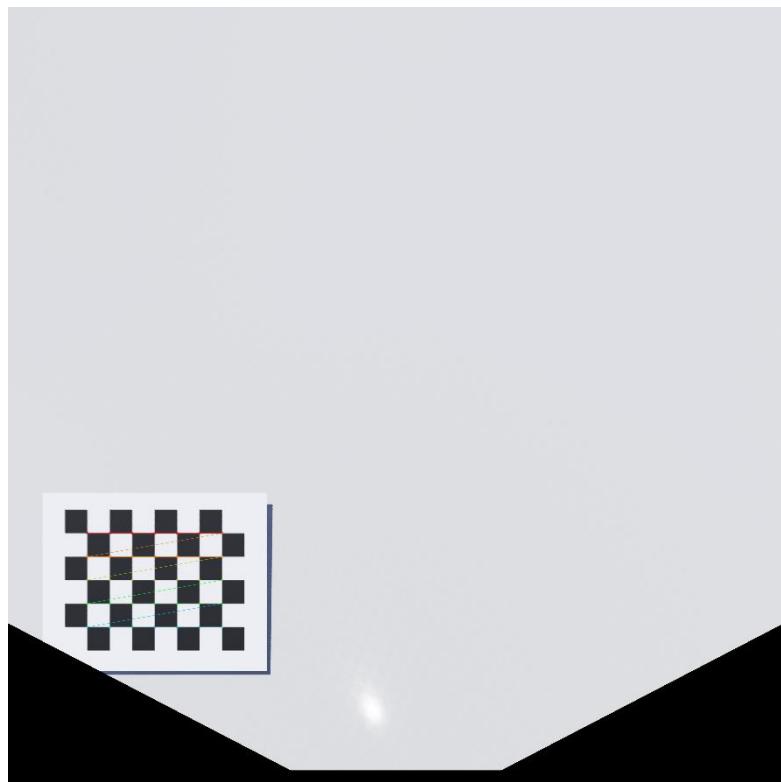
3.4.3.1 Pierwszy sposób łączenia widoku „z lotu ptaka”:

Ten sposób okazał się jednym z najłatwiejszych do powtarzania – można go szybko wykonać w dowolnym momencie i bez dużego wysiłku. Niemniej jednak, szybkie jego wykonanie wymaga intuicji co do umieszczenia wzorców (szachownic) i ich rozmiarów. Każda szachownica powinna bezwzględnie znaleźć się we wspólnym obszarze pary kamer w tym momencie łączonych. Liczba takich szachownic więc optymalnie wynosi 6, większa ilość jest zbędna.

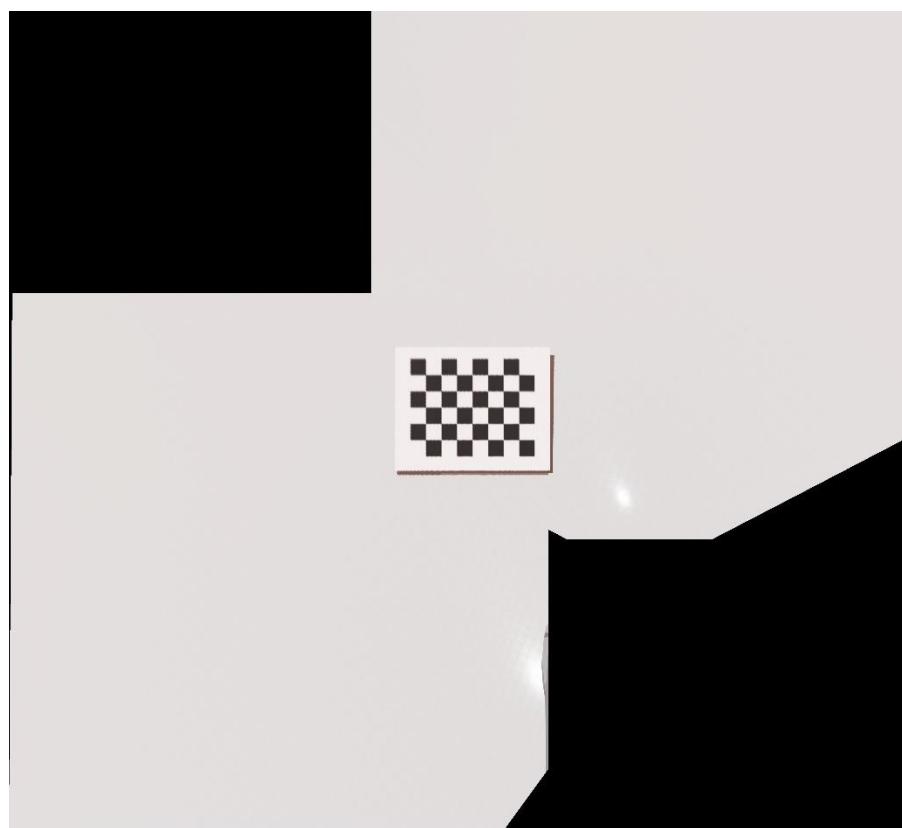
Proces polega na wyznaczeniu homografii między pierwszym a drugim obrazem, nałożeniu pierwszego obrazu w tym miejscu, gdzie teraz już są przekształcone (idealnie) takie same narożniki szachownicy, a rozmiar nowego obrazu jest definiowany dynamicznie przez skrajne piksele pierwszego i drugiego obrazu. Przykład wyznaczenia takiej homografii jest pokazany na Rys. 12, Rys. 13 oraz Rys. 14 dla kamery lewej na kolumnie drzwiowej oraz przedniej na zderzaku:



Rys. 12. Znalezione narożniki szachownicy na wyprostowanym obrazie z kamery lewej na kolumnie drzwiowej.



Rys. 13. Znalezione narożniki szachownicy na kamerze przedniej.

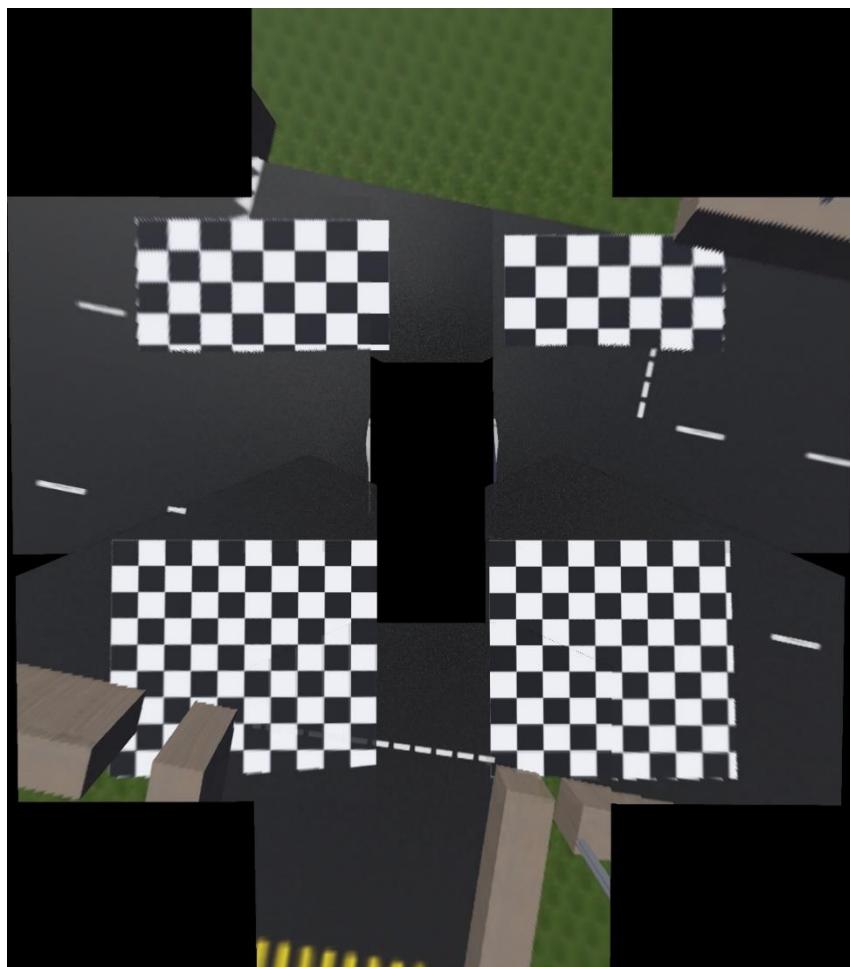


Rys. 14. Połączzone dwa obrazy w jeden łańcuchowo.

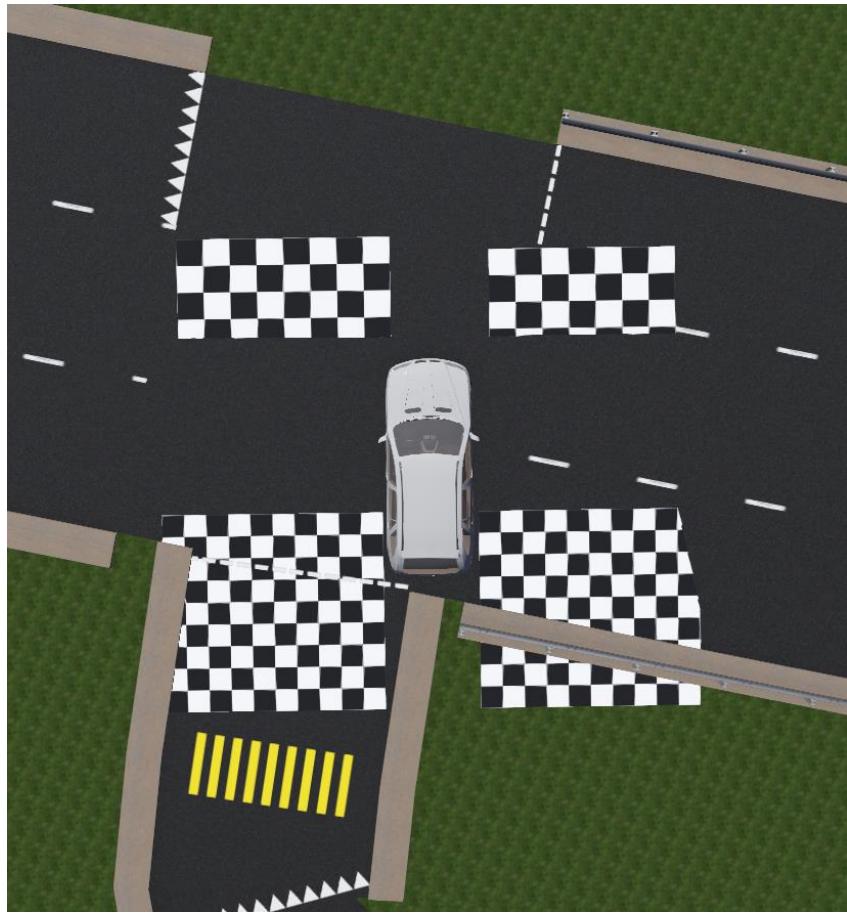
Jak widać, obraz lewej kamery, który przekształcono na perspektywę przedniej, lekko został zakrzywiony – na razie nie da się powiedzieć, czy to jest dobrze czy źle. Tak więc program będzie uruchamiany dla każdej z par kamer, gdzie obraz kolejny będzie łączony z już sumarycznym obrazem poprzednio nałożonych kamer. Tak się staje, że w wyniku tych przekształceń obraz z przedniej kamery zostaje niezmieniony (lewa kolumna -> przednia -> prawa kolumna -> prawy błotnik -> tylna -> lewy błotnik -> lewa kolumna). Nie jest to więc całkowicie łańcuchowo połączony obraz, który by powstał w razie, gdy połączenie kamery z lewej kolumny i przedniej było przekształcone na widok prawej kolumny, a nie na odwrót.

Ostatnim przekształceniem w łańcuchu jest przekształcenie obrazu z kamery na lewym błotniku zarówno na widok kamery z lewej kolumny, jak i tylnej. Wykonane to zostało pośrednictwem zmapowania jednocześnie dwóch szachownic we wspólnych obszarach kamery tylnej, na lewym błotniku i na lewej kolumnie.

W wyniku powstaje obraz, przedstawiony na Rys. 15, natomiast na Rys. 16 jest pokazany rzeczywisty widok z góry w symulatorze:



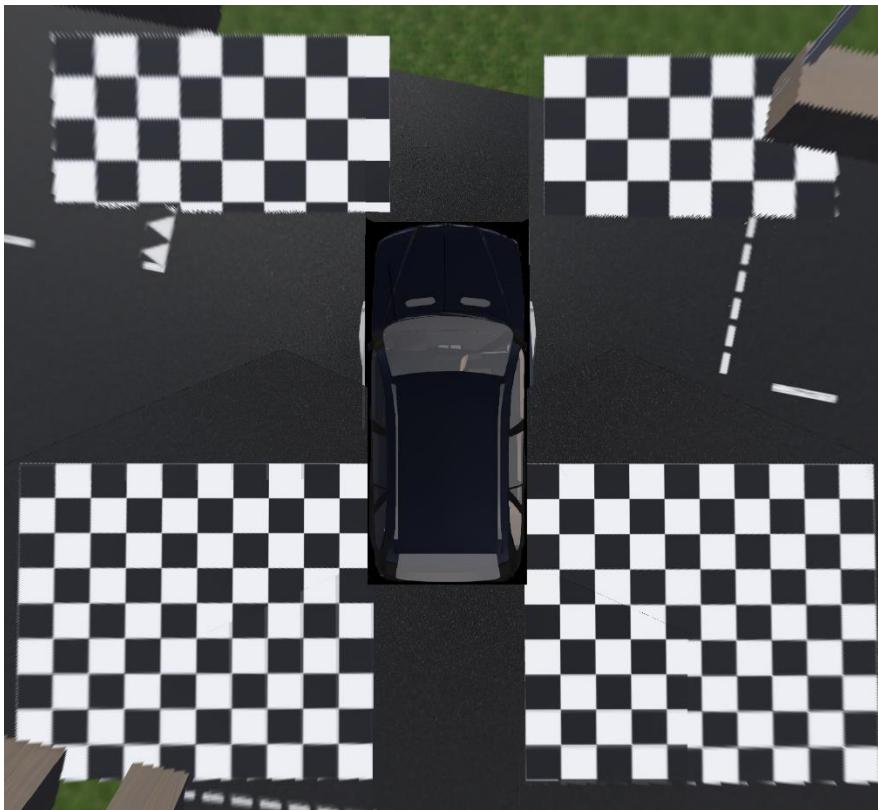
Rys. 15. Widok „z lotu ptaka” wykonany za pomocą 1. sposobu.



Rys. 16. Widok z góry samochodu dla porównania z widokiem „z lotu ptaka”, wykonanym według 1. sposobu.

Widzimy, że każdy obraz został przekształcony w mniej lub bardziej symetryczny sposób, i głównie obrazy są zawężone, ponieważ podczas poszczególnego „prostowania” punkty położone najdalej od kamery są niedokładnie odwzorowane na płaszczyźnie. Dzieje się tak, dlatego że homografia zostało wykonana za pomocą szachownic, wzorców, położonych bliżej kamery. Takie działanie było podyktowane mocnymi zniekształceniami perspektywistycznymi kamer szerokokątnych, na których nawet w wysokiej rozdzielcości dalsze punkty były słabo rozpoznawane.

Dla bardziej estetycznego widoku należy pozbyć się czarnych obszarów w miejscach, gdzie zostały zszyte obrazy z kamer. Po ucięciu i przeskalowaniu obrazu otrzymujemy taki wynik (Rys. 17):



Rys. 17. Ucięty i proporcjonalny widok „z lotu ptaka”, wykonany według 1. sposobu.

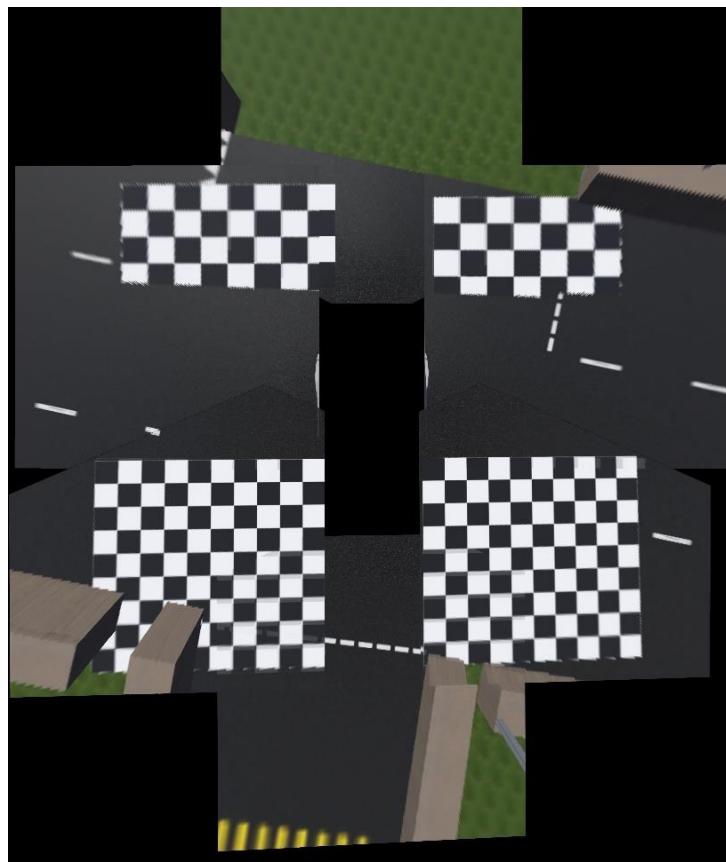
3.4.3.2 Drugi sposób sporządzenia widoku „z lotu ptaka”:

Drugi sposób jest modyfikacją pierwszego i polega na zmianie kierunku zszywania: zamiast zszywać każdy obraz jeden po jednym po kolej, zszywane są połówki przednia i tylna. Te połówki są tworzone poprzez nałożenie obrazów z kamer odpowiednio prawej i lewej na kolumnach na przednią oraz prawej i lewej na błotnikach na tylną kamerę.

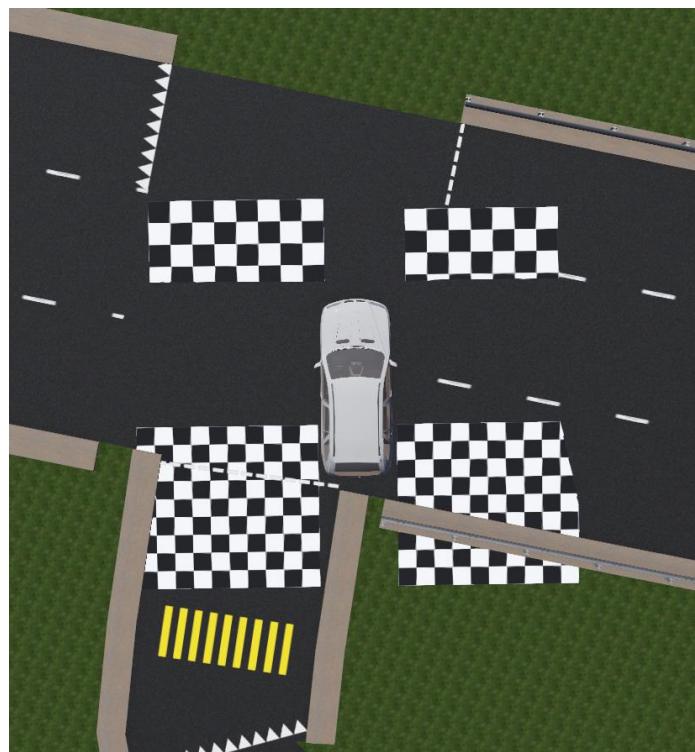
Połówki są łączone jednocześnie za pomocą dwóch wzorców-szachownic po obu stronach. Kamery na kolumnach i błotnikach mają małe zniekształcenia perspektywistyczne bliżej obiektywów, ale to byłoby pomocne i słuszne, jeżeliby nie były one nakładane na perspektywę przedniej kamery, która ma zniekształcenia po bokach obrazu. Można więc nakładać albo przednią połowę na tylną, albo na odwrót, a błędy będą podobne.

Do łączenia za pomocą dwóch szachownic potrzebne było wykonanie skryptu oddzielnego, który zamaskowałby już znalezioną szachownicę na czarno, i wtedy algorytm będzie mógł bez zakłóceń rozpocząć poszukiwanie drugiego wzorca (funkcja *chess_homography_multiple_boards* w dodatku).

Przykładowy wynik takiego sklejania jest pokazany na Rys. 18, a oryginalny widok z góry – na Rys. 19:



Rys. 18. Widok „z lotu ptaka”, wykonany według 2. sposobu.

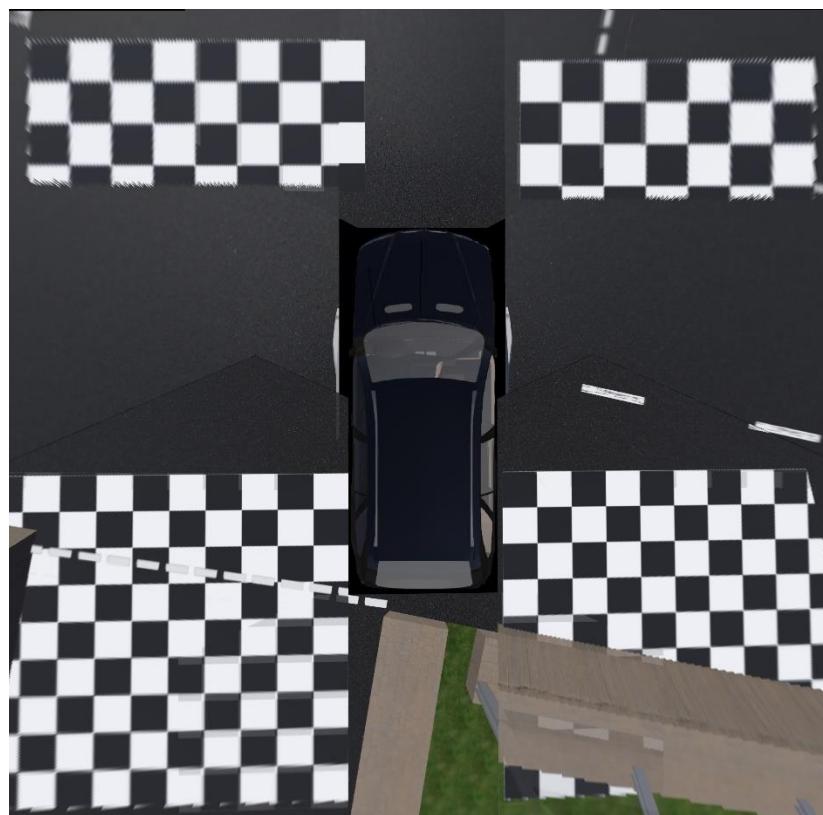


Rys. 19. Widok z góry na samochód – obraz-odnośnik do widoku „z lotu ptaka” z poprzedniego rys. Rys. 18.

Jak widać, jest zakrzywiona tylna część obrazu, przez to że została ona nałożona na perspektywę przedniej połowy widoku. Na pewno będzie potrzebna korekcja takiego widoku co najmniej z tylnej części. Przez to taki widok będzie słabo się nadawał do pomiarów, ponieważ nie jest jednorodna długość i szerokość na takiej płaszczyźnie, bliżej rogów poszczególnych obrazów widoczne będą zniekształcenia perspektywy. Można to zauważać po niedopasowaniu oznakowania poziomego na drodze. Nie znajdowały się tam wzorce, które mogłyby pomóc w bardziej dokładnej korekcji perspektywy.

Widać mocne zniekształcenia wzorców. Wszystkie błędy zostały skumulowane dla każdej kamery i nie zredukowane, dlatego obraz jest mocno zakrzywiony. Takie połączenie zawsze jednak uwzględnia wzajemne położenie kamer wokół samochodu, ze względu na zszywanie sąsiadujących obrazów z kamer. Większych zniekształceń doświadczyła dolna połowa widoku, ponieważ właśnie ona została przekształcona na płaszczyznę przedniej połowy (kamery na lewej kolumnie, prawej kolumnie i przedniej kamerze).

Aby pozbyć się widocznych czarnych obszarów bez żadnych punktów, przeskalowano odpowiednio widok „z lotu ptaka, jak i w poprzednim rozdziale. Przeprowadza się to metodą prób i błędów, ponieważ ciężko na zniekształconym obrazie sporządzić dokładną jednolitą skalę. Na Rys. 20 jest pokazany wynik takiego ucięcia i przeskalowania:



Rys. 20. Widok „z lotu ptaka” ucięty z zachowaniem proporcji i ze wstawionym obrazem samochodu.

Podobno jak w 1. sposobie, pozwala takie podejście na stosowanie dowolnych wzorców z jednym ograniczeniem: muszą one się znaleźć we wspólnych obszarach dla danej

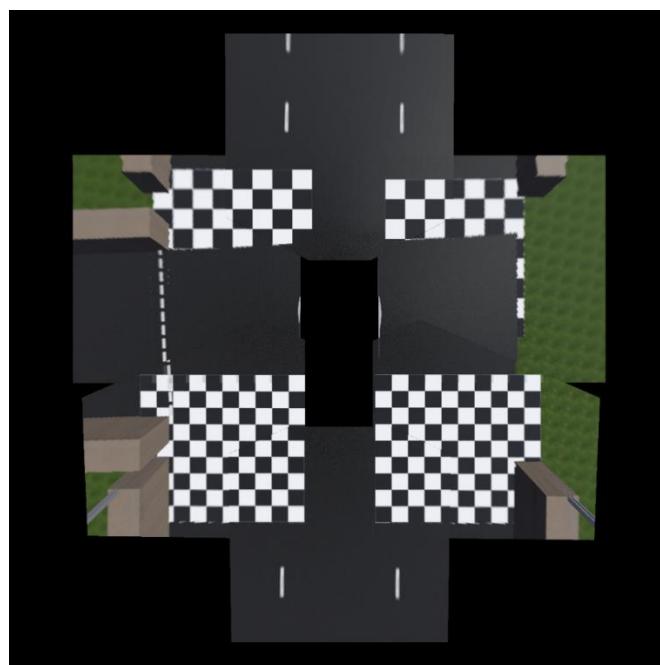
przetwarzanej (łączonej) pary kamer. Oczywiście wnioskiem jest to, że inaczej narożniki szachownicy na jednym z obrazów nie będą znalezione. Tak więc szachownice należy umieścić we wspólnych obszarach dla kamer oraz z pewnym odstępem od krawędzi wyprostowanych obrazów.

3.4.3.3 Trzeci sposób sporządzenia widoku „z lotu ptaka”:

Widok taki w tym podejściu jest robiony następująco: dla każdej kamery istnieje jedna szachownica (dowolnie duża, ale musi ona zmieścić się na obrazie wyprostowanym, i algorytm `cv.findChessboardCorners()` nie będzie miał trudności z jej znalezieniem), jej wymiary są zmierzone w metrach i wpisane do list jako 4 punkty skrajnych narożników, a wówczas na obrazie wyprostowanym taka szachownica jest zdefiniowana jako znalezione subpiksele narożników. Później te narożniki w pikselach są przekształcane na obraz w kanwie według takiego wzorca: narożniki są poddawane przekształceniu homografii na piksele na kanwie, odpowiadające metrom w globalnym układzie odniesienia (uprzednio zmierzone w metrach szachownice). Inaczej mówiąc, jest kanwa, dla której istnieje homografia metry na piksele, oraz istnieje szachownica na wyprostowanym obrazie z kamery, dla której się liczy homografię z pikseli kamery na piksele kanwy.

Obrazy z kamer muszą być koniecznie wyprostowane przed nakładaniem na kanwę, ponieważ na surowych obrazach z kamer nie da się znaleźć dużych szachownic, są poddawane one zniekształceniom perspektywistycznym i stąd stają się niewidoczne dla algorytmu. Dodatkowo, jak było omówiono wcześniej, każdy obraz musiałby być poddany przesunięciom ROI (region of interest), aby pozbyć się m. in. widocznych z boku drzwi i błotników pojazdu.

Taki widok jest pokazany na Rys. 21:



Rys. 21. Widok „z lotu ptaka” wykonany według 3. sposobu.

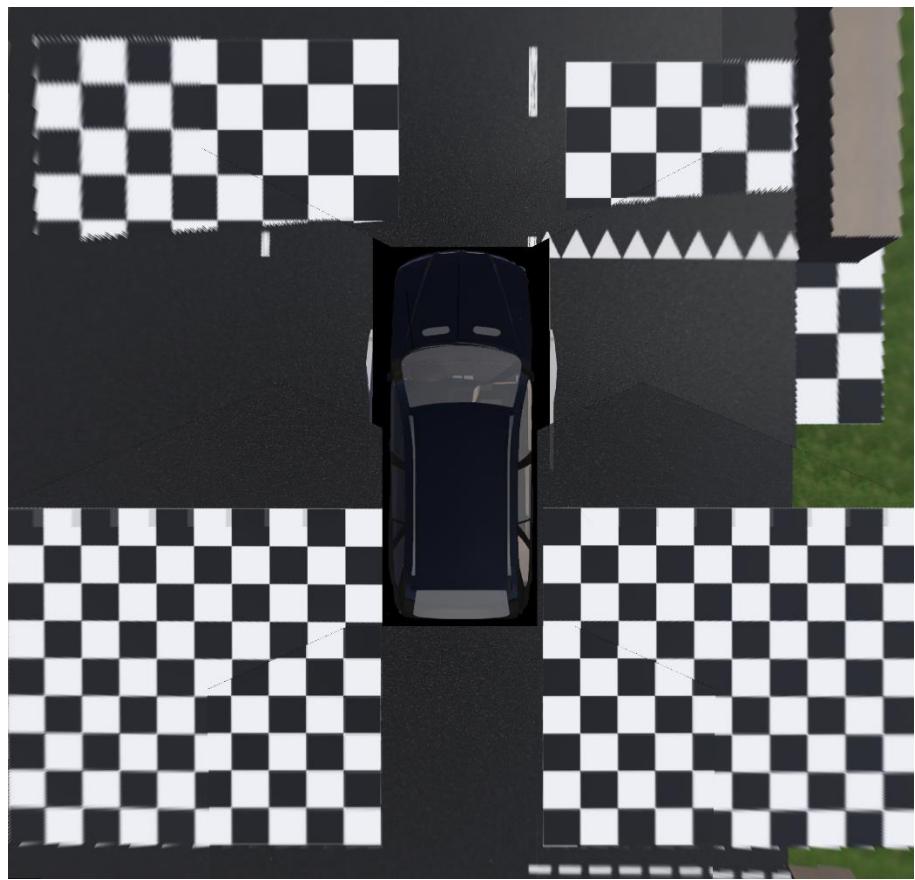
Widok cechuje się zdecydowanie lepszą jednorodnością wymiarów płaszczyzny i również pozwala na skuteczniejszą i wygodniejszą poprawę samego widoku: można każdą kamerę oddzielnie według wzorców „prostować” dalej, aż do maksymalnego zmniejszenia błędu reprojekcji.

Złą stroną takiego sposobu jest o wiele cięższa implementacja: potrzeba większego wysiłku, aby szachownice zmierzyć i wpisać parametry. Nawet po napisaniu skryptu automatycznie przeliczającego wymiary szachownicy, przy zmianie ich położenia należy to zmieniać również w kodzie.

Wstępnie jeszcze jedna niedogodność – przez to, że szachownicę trzeba wygodnie zmierzyć, wzorzec powinien pozbyć się białej „otoczki” wokół pól, co utrudnia detekcję. Bardziej szczegółowo to jest opisane w dalszych sekcjach raportu.

Jednoznacznym plusem tego sposobu jest jednak istniejąca z założenia skala metry-piksele na kanwie. Dla 1. i 2. sposobów musielibyśmy tą skalę opracowywać ręcznie, poprzez uprzednią korekcję jednorodności widoku, pomiar wzorcowej szachownicy i przeliczenie homografii piksele-metry. Te wszystkie kroki są pominięte w 3. sposobie, ponieważ już została zdefiniowana macierz piksele-metry z poziomu sporządzenia samego widoku. To ułatwiłoby rzutowanie podstaw brył otaczających samochody lub inne obiekty i wyliczonych za pomocą sieci neuronowej YOLO i połączonej z zewnętrznymi parametrami kamer. Sposób na wyznaczenie takiej lokalizacji jest w opisie następnego wykonanego zadania.

Również został przeskalowany ten widok, natomiast tym razem możemy dokładnie posłużyć się skalą piksele-metry: mamy pełną kontrolę nad widoczną przestrzenią (Rys. 22).



Rys. 22. Widok „z lotu ptaka” przeskalowany do długości 11.6 m (5.65 m do przodu od środka i 5.95 do tyłu od środka) oraz szerokości 12 m (6 m w lewo i w prawo od środka samochodu).

3.4.4 Estymacja parametrów zewnętrznych kamery i tworzenie na bieżąco brył wokół obiektów:

Parametry zewnętrzne najczęściej estymuje się przy rozwiązyaniu pozycji z odpowiedników 3D i 2D punktów `cv2.solvePnp()`⁴. Ta funkcja zwraca wektory translacji i rotacji pozycji obiektu w układzie kamery zdefiniowanego w 3 wymiarach. To jednak nie pozwala na definicję globalnego położenia kamery, a tylko względem przykładowej szachownicy. Wobec tego musimy policzyć pozycję obiektu (szachownicy) względem środka samochodu – taki przyjęliśmy wcześniej przy tworzeniu widoku „z lotu ptaka” układ (3. sposób), – i stąd policzenie pozycji kamery względem środka samochodu jest następujące:

$$T_K^C = T_S^C T_K^S$$

Gdzie T_K^C – pozycja kamery w układzie środka samochodu,

T_S^C – pozycja szachownicy w układzie środka samochodu,

T_K^S – pozycja kamery w układzie szachownicy.

Od `cv2.solvePnP()` dostajemy pozycję obiektu w układzie kamery T_S^K , więc musimy odwrócić macierz:

$$T_K^S = (T_S^K)^{-1}$$

Macierze wszystkie są jednorodne, to oznacza, że punkty przekształcane mają 4 współrzędną.

Do narysowania brył są wykorzystane autorskie funkcje, pominięto metody typu `cv.projectPoints()`. Ułatwia to „debugging” oraz wspomaga merytoryczną analizę działania algorytmów. Każdy punkt w przestrzeni (bardziej płaszczyźnie według której zostały skalibrowane parametry zewnętrzne kamery) jest rzutowany na 2D obraz kamery według jej macierzy. Symulator pomija zniekształcenia radialne i tangencjalne obiektywów i asymetryczność przetwornika obrazu. Ogniskowa w pikselach jest więc równa w obu kierunkach i liczona jest następująco:

$$f = \frac{w}{2} \tan\left(\frac{\alpha}{2}\right)$$

Gdzie w – szerokość obrazu,

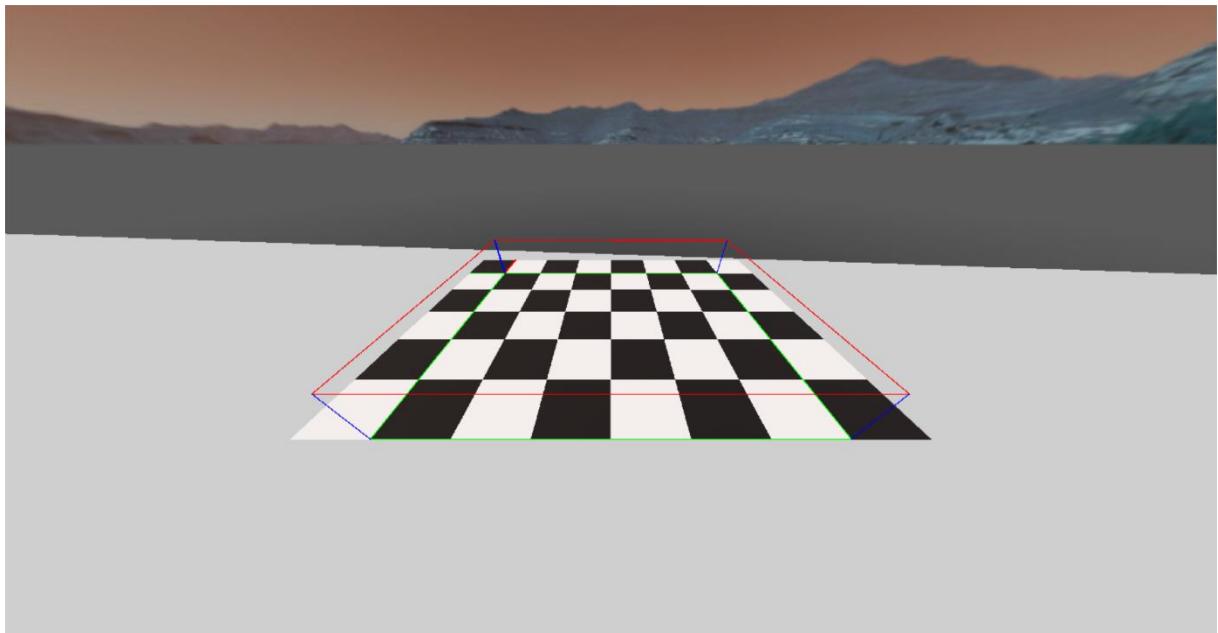
α – kąt poziomy pola widzenia kamery, określony przez użytkownika w symulatorze.

Współrzędne środka obrazu są dokładnie w środku geometrycznym, czyli połowa szerokości i długości.

Na Rys. 23 jest pokazane przykładowe działanie algorytmu z narysowaniem prostopadłościanu, zdefiniowanym metrycznie względem środka samochodu. Kierunki osi XYZ

⁴ https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html

są zgodne z modelem Ackermannia, który jest wykorzystywany w symulatorze do modelowania samochodu – oś X do przodu, Y w lewo, Z w górę:



Rys. 23. Działanie metody odnalezienia pozy kamery w układzie samochodu.

Jak widać, bryła zostaną przeliczona dla przedniej kamery i narysowana prawidłowo (nie jest to ręczne rysowanie między wybranymi pikselami).

W tym zadaniu użyte zostały wszystkie kamery do estymacji pozy, łącznie z przednią kamerą za szybą czołową. Będzie ona służyła właśnie do detekcji obiektów wokół samochodu ze względu na swoje wygodne położenie. Na Rys. 24 widoczne są właśnie te prostopadłościany. Ich wymiary są wybrane arbitralnie, ponieważ sieć YOLO rysuje wyłącznie ramki 2D bez estymacji długości i szerokości samochodu.



Rys. 24. Narysowane bryły w miejscach samochodów za pomocą sieci YOLO8.

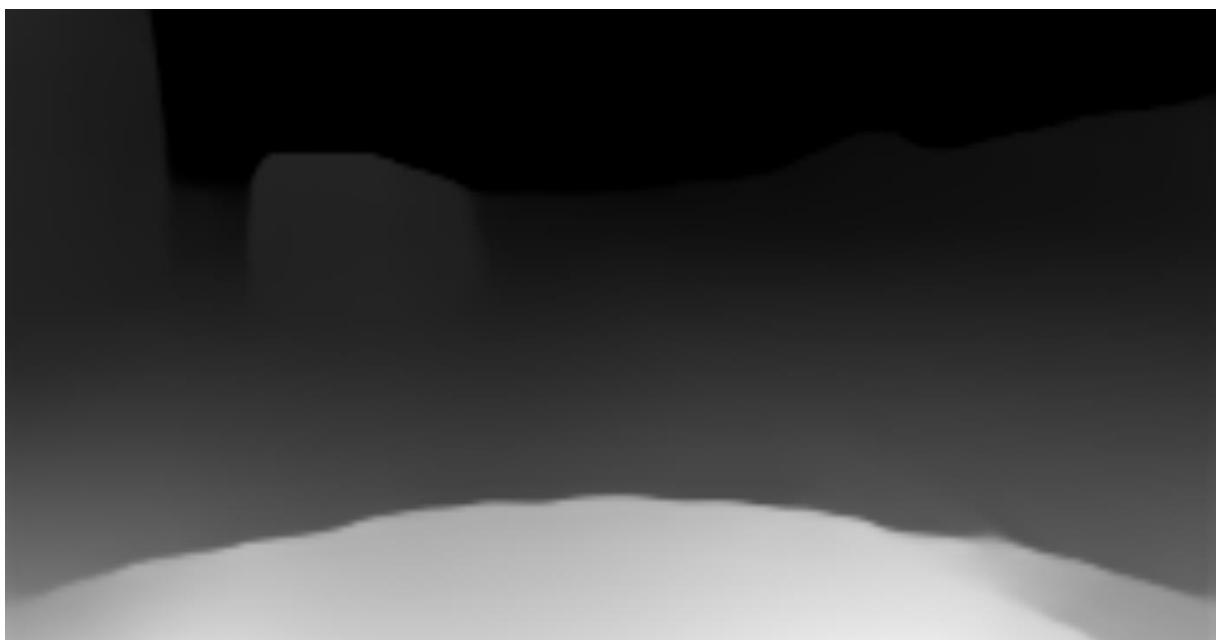
Jak widać, nie jest to doskonała metoda – przy tym że parametry zewnętrzne kamery zostały określone poprawnie, projekcja brył o sztywnej długości, szerokości i wysokości ($4,5 \times 1,8 \times 1,8$ m) jest co najmniej niedokładna, uwzględnia ona tylko orientację kamery i działa najlepiej, jeżeli samochód jest ustawiony równolegle do odnalezionych przeszkód. Wtedy nie ma problemów z określeniem punktu zaczepienia, co widać po bryle w dalszym miejscu, która jest zaczepiona w lewym dolnym punkcie, gdyż algorytm obliczył jej odległość w Y jako dodatnią (zgodnie z przyjętym układem odniesienia).

Nie wszystkie samochody też da się odnaleźć, możliwe że w pewnym stopniu różnią się od obiektów, na których została nauczona sieć w modułu *Ultraalytics*.

Ponadto ta implementacja nie pozwoliła na estymację rzeczywistych rozmiarów trójwymiarowych obiektów, ponieważ z natury kamera ma płaszczyznę obrazową 2D – estymacja głębi może być albo z sieci neuronowej, albo z dysparycji. O tym w następnej sekcji.

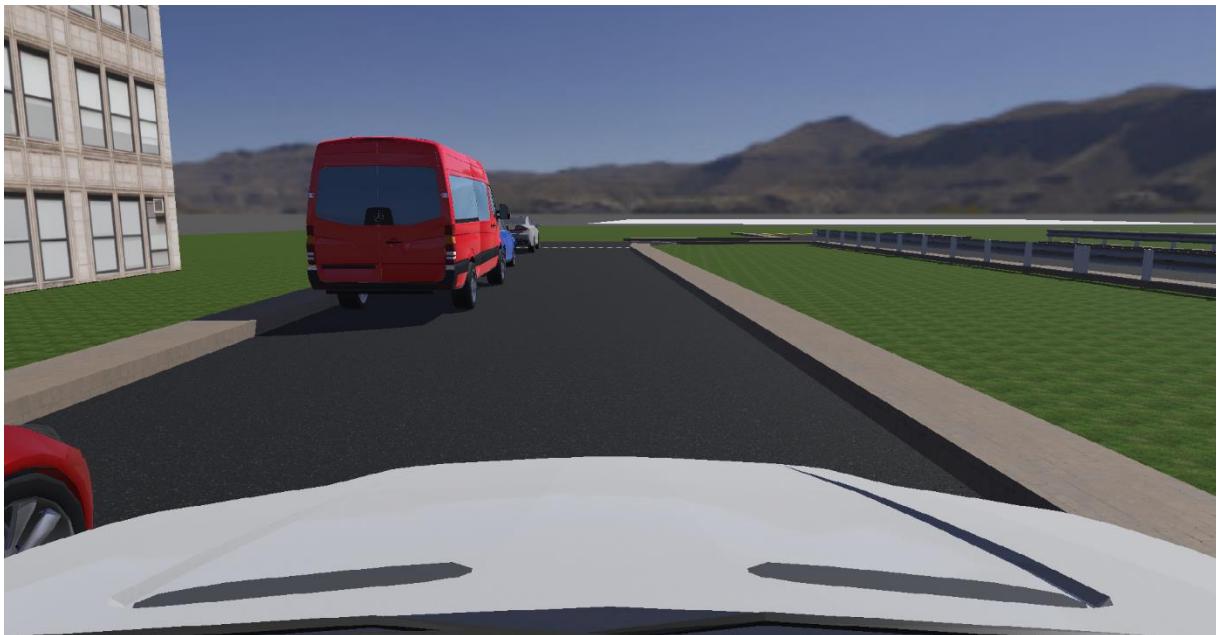
3.4.5 Implementacja stereowizji w fuzji z siecią neuronową

Dany rozdział będzie poświęcony jednemu ze sposobów lokalizacji obiektów, który będzie się opierał na stereowizji (estymacja głębi) oraz segmentacji obiektów przez sieć neuronową YOLO. Taki pomysł pojawił się w związku z problemem sztywnej wizualizacji brył otaczających obiekty: w poprzednim zadaniu opisano, jakie w rysowaniu brył o podanych rozmiarach od wybranego punktu zaczepienia pojawiają się niedokładności. Estymacja głębi za pomocą sieci neuronowych (takich jak MiDAS lub DepthPro [15]) potrafi być czasami niedokładne, a czasami bardzo wymagające obliczeniowo. Poniżej można zobaczyć zdjęcie z jednej z frontalnych kamer, przetworzonych na mapę głębi (Rys. 25) :



Rys. 25. Obraz z kamery frontalnej, pokazujący mapę głębi, oszacowaną przez sieć neuronową MiDAS.

Obraz jest rozmyty, i z niego nie da się precyzyjnie ocenić głębi – tym bardziej, że niektóre obiekty sieć pominęła w wynikowym obrazie. Ocenić to można w odniesieniu do oryginalnego obrazu na Rys. 26:



Rys. 26. Oryginalny obraz-odnośnik dla oceny estymacji głębi przez sieć neuronową MiDAS.

Obiekty pominięte: dalsze samochody, krawężnik, ogrodzenia na drodze, zderzak stojącego po lewej samochodu. Ponadto, rozdzielcość przetwarzanego obrazu była zmniejszona aż do 640x480 przez długi czas inferencji. Nawet jeżeli dałoby się znaleźć sieć neuronową o większej dokładności i lepszej architekturze, nie byłaby taka monokularna estymacja głębi szybsza, niż podejście ze stereowizją.

Dla oceny wyników takich operacji umieszczono niżej (Rys. 27, Rys. 28) dwa obrazy oryginalnych z kamer i wychodzący z algorytmu sterowizyjnego `cv2.StereoSGBM_create()`, dodatkowo odfiltrowanego dla większej dokładności (rys. Rys. 29):



Rys. 27. Obraz oryginalny z kamery po prawej stronie dla pokazu estymacji głębi stereowizją.



Rys. 28. Obraz oryginalny z kamery po lewej stronie dla pokazu estymacji głębi stereowizją. Prawie nie ma różnic – jedynie można zobaczyć po otworach wentylacyjnych na masce, zostały przesunięte.



Rys. 29. Obraz przetworzony przez algorytm stereowizyjny.

Jak widać, wyniki są o wiele lepsze i dokładniejsze, a ponadto są szybsze (szybkość symulacji z siecią neuronową i innymi procesami w tle kontrolera: 0.08x; dla stereowizji i takich samych warunków: 0.13x).

Jednym z najważniejszych warunków dla poprawnej poziomej (nie pionowej) stereowizji jest to, żeby dla kamer była zobrazowana ta sama płaszczyzna – powinny one być położone na tej samej prostej, tzw. „*baseline*”, linii bazowej; muszą mieć też bardzo zbliżone parametry wewnętrzne. Często wykorzystuje się metody typu `cv2.StereoRectify()` dla korekcji zniekształceń soczewkowych, natomiast w symulatorze kamery domyślnie są pozbawione tego. System stereowizyjny realizowany w projekcie posiada następujące cechy:

- za szybą czołową znajdują się dwie kamery, przesunięte względem siebie o 3 cm (0.03 m) w osi Y układu pojazdu (w lewo prostopadle do kierunku ruchu) o następujących parametrach:
 - rozdzielcość 1920x1080, pole widzenia poziome 75°, brak zniekształceń soczewkowych,
- dla tych dwóch kamer jest estymowana głębia za pomocą algorytmu `cv2.StereoSGBM_create` oraz doprecyzowującego filtra WLS (`cv2.ximgproc.DisparityWLSFilter`, „weighted least square”),
- segmentacja sceny za pomocą sieci *YOLO11* w modułu od firmy *Ultralytics*,
- nałożenie maski segmentacji *YOLO* na mapę stereowizyjną i odnalezienie skrajnych punktów (lewego i prawego) maski,
- obliczenie współrzędnych światowych punktów pośrednictwem następujących wzorów (położenie punktów względem kamery):

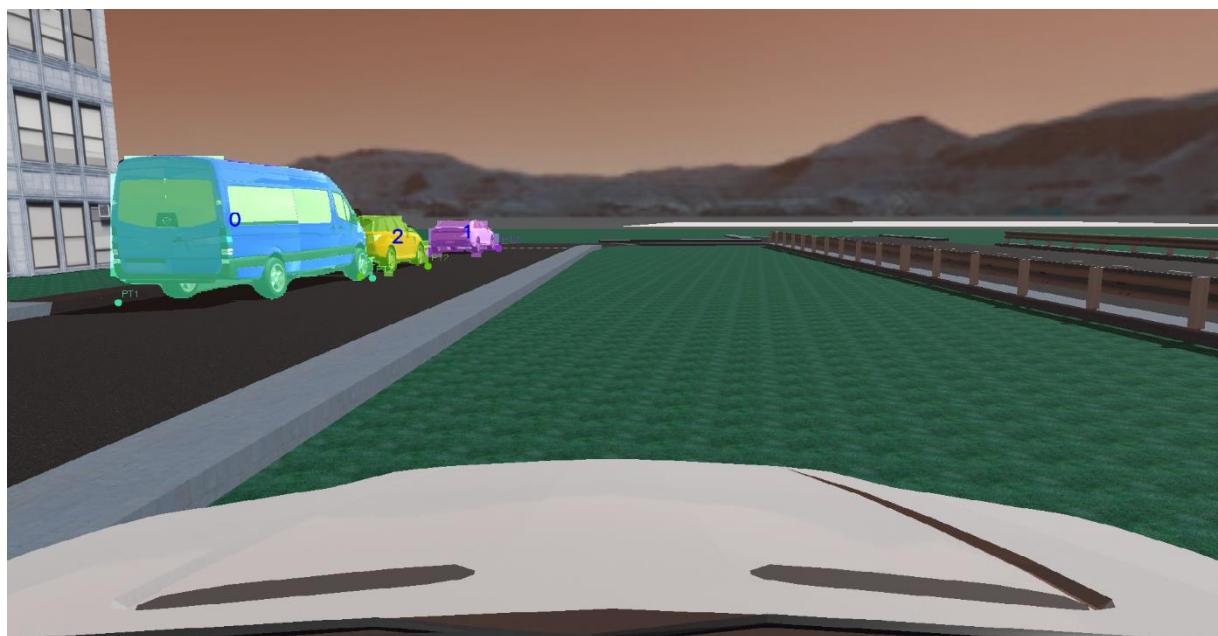
$$Z = f \frac{B}{D}$$

$$X = (u - c_x) \frac{Z}{f}$$

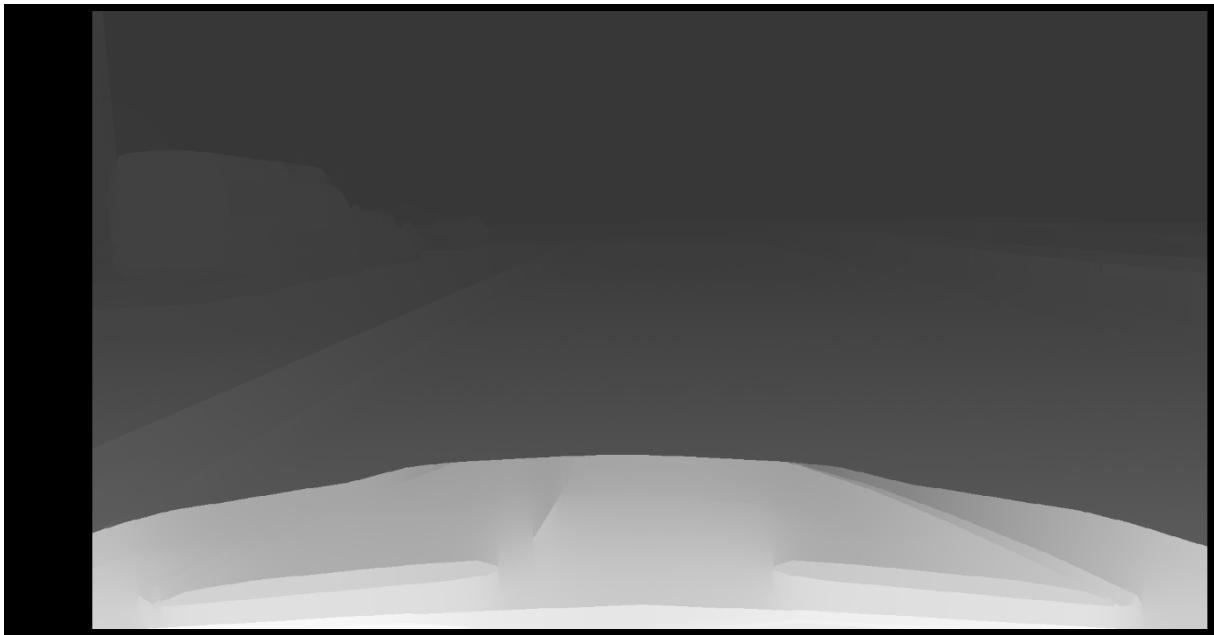
$$Y = (v - c_y) \frac{Z}{f}$$

- rzutowanie odnalezionej punktu na płaszczyznę drogi ($X = 0$, skoro X w układzie kamery jest skierowane prostopadle do dołu),
- rysowanie oszacowanych punktów na obrazie z kamery przy pomocy znanego położenia kamery w układzie samochodu (kamera może być dowolna z dwóch, ale w tym momencie była obliczona poza tylko jednej, prawej),
- łączenie punktów dla wizualizacji prostokąta otaczającego na płaszczyźnie drogi segmentowany obiekt.

Dla oceny tego rozwiązania przytoczone są poniżej dwa obrazy: jeden z już zaznaczonymi punktami i maskami segmentacji (Rys. 30), a drugi – mapa głębi (Rys. 31).



Rys. 30. Obraz prawej kamery z maskami segmentacji, numerami obiektów i punktami skrajnymi masek, rzutowanymi na płaszczyznę drogi.



Rys. 31. Obraz głębi dla odniesienia do Rys. 30.

Charakterystyczną cechą tego widoku mapy głębi jest przesunięcie płaszczyzny obrazowej: dzieje się to tak dlatego, że bazą dla stereowizji służy lewa kamera, a dla prawej jest tworzony tzw. *matcher*. Dalej znajduje się Listing 1 z częściową implementacją tego algorytmu:

Listing 1. Częściowy kod algorytmu stereowizjnego z fuzją sieci neuronowej YOLO11.

```
# Obiekt stereo matcher
stereo_left = cv2.StereoSGBM_create(
    minDisparity=0,
    numDisparities=32,
    blockSize=20,
    disp12MaxDiff=1,
    uniquenessRatio=10,
    speckleWindowSize=100,
    speckleRange=8
)
stereo_right = cv2.ximgproc.createRightMatcher(stereo_left)

# obliczenie mask i disparity
results = model(img_right, half=True, device=0,
                classes=[2,5,7,10], conf=0.6,
                verbose=False, imgsz=(1280,960))
if results and results[0].masks is not None:
    disp_left = stereo_left.compute(grayL, grayR).astype(np.float32)/16.0
    disp_right = stereo_right.compute(grayR, grayL).astype(np.float32)/16.0
    # ustawienie parametrów filtra
    wls = cv2.ximgproc.createDisparityWLSFilter(matcher_left=stereo_left)
    wls.setLambda(8000); wls.setSigmaColor(1.9)
    filtered_disp = wls.filter(disp_left, grayL, None, disp_right)

    for mask in results[0].masks.data.cpu().numpy():
        # zmienić rozmiar maski, żeby się dopasowała do oryginalnego
        mask_u8 = cv2.resize(mask.astype(np.uint8),
                            (orig_w, orig_h),
                            interpolation=cv2.INTER_NEAREST)
```

```

# dodanie tylko poprawnej disparity
disp_masked = np.nan_to_num(filtered_disp) * mask_u8
valid = disp_masked[(mask_u8>0)&(disp_masked>0)]
if valid.size==0: continue
# przetworzenie skrajnych punktów na 3D
p1_3d, p2_3d = points_from_mask_to_3D(
    mask_u8, filtered_disp,
    K_right, 0.03, T_center_to_camera
)
if p1_3d is None or p2_3d is None: continue

# rzut 3D→2D
p1_px = project_points_world_to_image([p1_3d], T_center_to_camera, K)
p2_px = project_points_world_to_image([p2_3d], T_center_to_camera, K)
# ... dalsze operacje na obrazie ...

```

Użyteczność wyników algorytmu jest uwarunkowana dobrym oszacowaniem maski obiektu – najczęściej zachodzą te maski na inne punkty obrazu, które na mapie głębi są położone o wiele dalej niż właściwy obrys obiektu. W funkcji „*points_from_mask_to_3D*” jest kod pozwalający na oszacowanie głębi punktu jako mediany z pewnego obszaru. Sama ta funkcja (Listing 2) przelicza współrzędne obrazowe z mapy dysparcji na współrzędne trójwymiarowe w układzie kamery.

Listing 2. Kod w funkcji „*points_from_mask_to_3D*”, pozwalający na estymację głębi punktów średnich w obszarze.

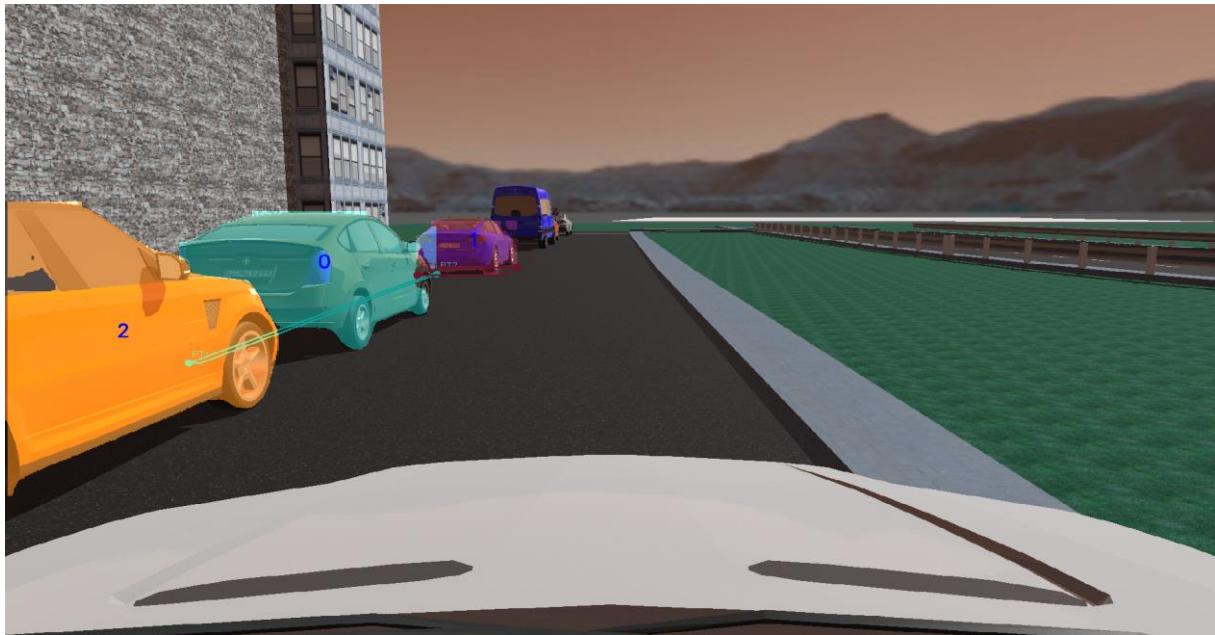
```

def get_valid_disparity(disp_map, x, y, window=30):
    h, w = disp_map.shape
    x0 = max(0, x - window)
    x1 = min(w, x + window + 1)
    y0 = max(0, y - window)
    y1 = min(h, y + window + 1)
    patch = disp_map[y0:y1, x0:x1]
    valid = patch[patch > 0]
    if valid.size == 0:
        return None
    return np.median(valid)

```

Funkcja „*get_valid_disparity*” pobiera mapę dysparcji oraz współrzędne piksela (x, y), wycina z tej mapy kwadratowe okno o boku $2 \cdot \text{window} + 1$ punktów (przycinając je do krawędzi obrazu), filtryuje z tego fragmentu wszystkie wartości większe od zera (uznane za poprawne pomiary) i zwraca ich medianę jako reprezentatywną wartość dysparcji w okolicy danego piksela. Jeśli w oknie nie ma żadnych dodatnich wartości, funkcja zwraca *None*.

Wskutek tego niepoprawnego wskazania skrajnych punktów pojawiają się różne artefakty w wizualizacji, świadczące jedynie o złym dopasowaniu maski segmentacji do rzeczywistego obrysu obiektu. Dzieje się to przez to, że położenie obiektu na obrazie jest obliczane z obrazu o mniejszej rozdzielczości (1280x960 px) i później skalowane. Na poniższym zdjęciu (Rys. 32) jest pokazany przykład takiego zachowania:



Rys. 32. Obraz wskazujący odnalezione obiekty i ich zrzutowane na ziemię i aproksymowane prostokątami obrysy.

Ciemnoczerwony samochód zaznaczono w miarę dobrze – taki wynik chcielibyśmy obserwować w większości przypadków. Niestety, najczęściej punkty są mierzone tak, jak na turkusowym samochodzie – część maski weszła na zupełnie inny obszar, przez to prostokąt jest rozciągnięty wzdłuż drogi. Można jednak zobaczyć, że skrajny lewy punkt jest dobrze ułożony – około tego punktu na drodze zaczyna się z lewej strony karoseria samochodu.

Też niektóre samochody (jak ten pomarańczowy), ustawione bliżej kamer, system nie potrafi odnaleźć – możliwe, że w czasie przetwarzania punktów jeden z nich został zwrócony przez jedną z funkcji jako *None*.

Dla niektórych przypadków prostokąty nie mogą być narysowane zgodnie ze zwrotem samochodu. To się dzieje dlatego, że algorytm w końcu nie wie, jaki jest zwrot obiektu po jednej tylko przekątnej – prostokąty, które mogą powstać z jednej prostej mogą być absolutnie dowolnej proporcji. Skoro łączono współrzędne tych punktów na sztywno, uwzględniana jest tylko perspektywa, i najlepiej wskazywane te prostokąty, jeżeli kamera jest równoległa do jezdni i samochodów obok. Na Rys. 33 jest pokazany przykład takiego zachowania:



Rys. 33. Obraz, pokazujący błędne narysowanie prostokątów na ziemi ze stereowizji i segmentacji.

Dużym ograniczeniem jest sieć YOLO, która nie jest nauczona na dużej liczbie klas obiektów, które mogą się znaleźć na drodze, jak to: śmieci, pudełka, inne losowe obiekty, które mogą stać na jakimkolwiek miejscu w drodze. Do tego trzeba albo nauczać sieć na większym własnym zbiorze danych, albo w inny sposób filtrować pozycję przeszkód względem samochodu.

3.4.6 Automat parkujący

Automat parkowania równoległego jest opisany w pliku skryptowym *park_algo.py*.

Plik *park_algo.py* zawiera klasę *Parking*, która na podstawie danych czujników przełącza maszynę stanów. Na razie algorytm efektywnie nie jest zaimplementowany, i nie jest realizowane parkowanie, natomiast wyodrębniono ogólną strukturę:

- miejsce jest poszukiwane w fazie „*searching_start*”, kiedy wykryto gwałtowną zmianę wskazania wybranego czujnika prostopadłego do samochodów,
- idzie faza „*searching_progress*”, dopóki znowu nie wykryto gwałtowną zmianę odległości (z większą na mniejszą),
- faza „*waiting_for_park*” dodana dla ewentualnej komendy od kierowcy, że można zacząć parkowanie,
- jeżeli miejsce okaże się za małe (odometria wskaże długość mniejszą niż dopuszczalną), to maszyna powraca do stanu „*searching_start*”.

Dla poszukiwania miejsca na razie wykorzystany jest jeden czujnik z lewej lub prawej strony z przodu prostopadle na zderzaku. Ponadto zrobiono dla wizualizacji wskazań jednego czujnika (jest wystarczający, żeby odnajdywać samochody wzduż kierunku ruchu samochodu).

Pozycja samochodu jest odnajdywana poprzez całkowanie bieżącej prędkości, odzyskiwanej z klasy *Driver()*, a zwrot samochodu jest obliczany względem miejsca, gdzie został uruchomiony algorytm parkowania, za pomocą IMU. W symulatorze prędkość samochodu jest liczona z uwzględnieniem już promieni kół, tak więc to jest rzeczywisty wskaźnik prędkości, jak w samochodach. Prowadzone są eksperymenty z estymacją pozycji za pomocą GPS.

Wszystkie urządzenia w Webots łatwo aktywować i zamodelować – najczęściej można zostawić je jako mające nieskończoną rozdzielcość i pozbawione szumów. Na następnej stronie (Listing 3) dla wygody zamieszczono inicjalizację wszystkich urządzeń wykorzystywanych w pętli kontrolera.

Klasa *Parking* jest szczegółowo opisana w opisie implementacyjnym, natomiast ogólną ideą jest przełączenie między stanami w zależności od zmian odległości i komend użytkownika.

Przyjęto kilka uproszczeń dla takiego automatu:

- 1) miejsce parkingowe znajduje się równolegle do kierunku ruchu samochodu,
- 2) samochód porusza się tylko do tyłu,
- 3) pozycja odometrii i końcowa to pozycja czujnika odległości, na podstawie którego poprzednio wyznaczano położenie wolnego miejsca.

Listing 3. Kod w funkcji „main” z inicjalizacją urządzeń dla wykorzystania później w pętli kontrolera.

```
def main():
    camera_names = [
        "camera_front_bumper_wide", "camera_rear",
        "camera_left_fender", "camera_right_fender",
        "camera_left_pillar", "camera_right_pillar",
        "camera_front_top", "camera_front_top_add",
        "camera_helper"
    ]

    sensor_names = [
        "distance sensor left front side", "distance sensor front left",
        "distance sensor front lefter",
        "distance sensor front righter", "distance sensor front right",
        "distance sensor right front side",
        "distance sensor left side", "distance sensor left", "distance sensor
        lefter",
        "distance sensor righter", "distance sensor right", "distance sensor
        right side"
    ]
    names_dists = {}
    dists = []
    for name in sensor_names:
        sensor = driver.getDevice(name)
        if sensor:
            sensor.enable(TIME_STEP)
            distance_sensors.append(sensor)

            print(f"Found sensor: {name}")
        else:
            print(f"Sensor not found: {name}")

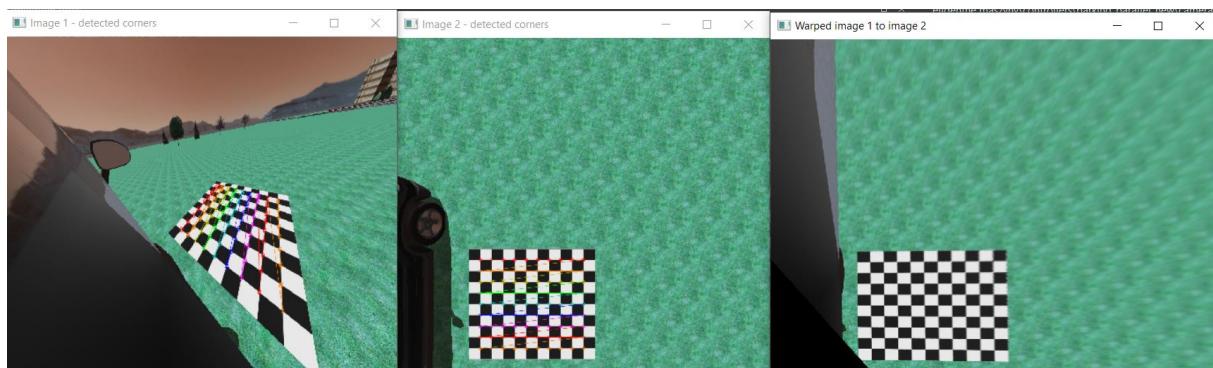
    # Inicjalizuj kamery
    for name in camera_names:
        cam = driver.getDevice(name)
        if cam:
            cam.enable(TIME_STEP)
            cameras.append(cam)
            width = cam.getWidth()
            height = cam.getHeight()
            fov_rad = cam.getFov()
            K = sy.calculate_intrinsic_matrix(width, height, fov_rad)
            cam_matrices[name] = K
            print(f"Odnaleziono kamerę: {name}")

    # GPS inicjalizacja
    gps = driver.getDevice("gps")
    if gps:
        gps.enable(TIME_STEP)
        print("GPS enabled")
    #Inicjalizacja IMU
    imu = driver.getDevice("inertial unit")
    if imu:
        imu.enable(TIME_STEP)
        print("IMU enabled")
```

3.5 Wyzwania napotkane podczas wykonania zadań

W długim czasie wykonywania wszystkich zagadnień projektu większość sukcesu osiągnięto metodą prób i błędów. Nigdy nie wiadomo bez właściwego doświadczenia, czy zadziała jedno czy drugie podejście. Poniżej znajduje się lista uwag i ewentualnych rozwiązań niektórych problemów:

- Podczas wstępnych prób ułożenia widoku „z lotu ptaka” zastosowano deskryptory i detektory cech takie jak SIFT, ORB i FAST, dostępne w bibliotece OpenCV. Metody te okazały się jednak nieefektywne zarówno pod względem szybkości, jak i stabilności punktów charakterystycznych, co skłoniło do przejścia na podejście wykorzystujące narożniki wzorców szachownicy – łatwiejsze do precyzyjnego wykrycia i bardziej odporne na zakłócenia otoczenia.
- Kolejnym podejściem było użycie klasy `cv2.Stitcher()` do jednoczesnego wykrywania cech i zszywania obrazów. Ze względu na brak wsparcia GPU metoda ta okazała się wolna i nieprzewidywalna: detektor cech na bieżąco uzyskiwał różne punkty, co skutkowało zmiennym rezultatem sklejania obrazów. Podobne ograniczenia wykazała funkcja `cv2.composePanorama()`. Rzetelniejsze wyniki dawało wykorzystanie wzorców szachownicy, jednak ich pole widzenia w symulatorze bywało częściowo zasłonięte przez obiekty. Ponadto metoda ta stanowi „czarną skrzynkę” – trudno jest wydobyć macierz transformacji odpowiadającą ostatecznemu złożeniu obrazu.
- W przypadku niższych rozdzielczości narożniki szachownicy były trudne do wykrycia, zwłaszcza w obszarach oddalonych od optyki, co prowadziło do znacznego błędu reprojekcji. Dlatego konieczne stało się przejście na obraz o rozdzielczości 4K dla każdej kamery (Rys. 34). Potwierdzono także z doświadczenia, że wzorzec szachownicy musi mieć wyraźną, białą „otoczkę” wokół pól, inaczej detekcja narożników jest niestabilna. Ze względu na ograniczenia Webots, jedynym sposobem wstawienia niezmodyfikowanego pikselowo wzorca okazało się ręczne przygotowanie maski 4x4 w zewnętrznym edytorze i jej skalowanie – rozwiązanie pomocne w większości scenariuszy, ale wciąż zawodnie działające przy zasłoniętych fragmentach.

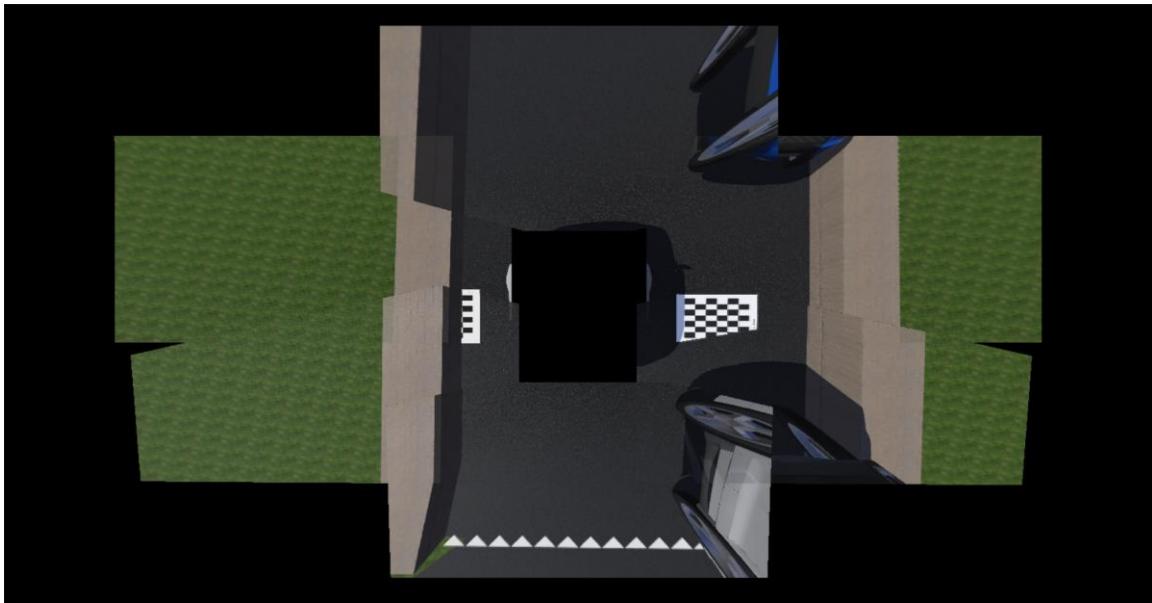


Rys. 34. Źle odnalezione narożniki szachownicy na zdjęciu o niższej rozdzielczości.

- Szachownice powinny mieć białą „otoczkę” wokół swoich pól, inaczej będzie ciągle powstawał problem z odnalezieniem narożników.

Przy sporządzeniu widoku „z lotu ptaka” według 3. sposobu szachownice nie mogły mieć białą otoczkę, przez to jedyny sposób w Webots na wstawienie niezmienionego pikselowo obrazu jest wstawienie wzorca 4x4 wyrobionego w programie Paint i jego skalowanie. Nie można było wstawić takich wzorców, ponieważ właśnie bez białej „otoczki” są łatwiejsze do zmierzenia w symulatorze wzory, da się łatwo z pikseli odwołać się do metrów. Z tego powodu na mapie w środowisku jest miejsce, gdzie samochód można ustawić na tle bliskim białemu kolorowi. Takie rozwiązanie w większości przypadków jest pomocne, jednak nie we wszystkich, czasami narożniki są wciąż trudne do znalezienia.

- Istotnym wyzwaniem była również wydajność obliczeń. Przetwarzanie strumieni 4 K w czasie rzeczywistym, w połączeniu z detekcją narożników, generowało znaczne obciążenie CPU i GPU. Dodatkowo sama generacja obrazu w symulatorze w trybie jałowym spowalniała działanie całego systemu. Konieczne okazało się dostosowanie pętli sterującej w Webots (blok while (driver.step() != -1)) tak, aby umożliwić płynne i bezopóźnieniowe zmiany parametrów modelu oraz reagowanie na komendy użytkownika.
- Nawet najlepszy widok „z lotu ptaka” wymagał po zszyciu dalszej korekcji perspektywy. Tylko w trzecim podejściu, w którym każdej kamerze przypisywano oddzielną homografię, dało się niezależnie korygować obrazy. W wariantach łańcuchowego stitchingu nie było możliwości wydzielenia pojedynczych obrazów ze złożonej panoramy, co wymuszało stosowanie globalnej korekcji, obarczonej ryzykiem utraty precyzji.
- Ostatecznie, mimo że dla niektórych zastosowań można zastosować sieci neuronowe rozpoznające obiekty zorientowane (Oriented Bounding Box), analiza wykazała, że dla widoków generowanych w projekcie identyfikacja obiektów pod różnymi kątami padania jest znacznie utrudniona z powodu zniekształceń perspektywicznych. Implementacja takiego rozwiązania wymagałaby dodatkowych mechanizmów kompensacyjnych i głębszego treningu na niestandardowych danych. Na Rys. 35 pokazano wspomniane zniekształcenia.



Rys. 35. Zniekształcony widok „z lotu ptaka” przy samochodach i krawężnikach.

3.6 Opis implementacyjny

Programowo projekt składa się z czterech plików skryptowych .py: *park_algo.py*, *camera_calibration.py*, *visualise.py*, *parking_parallel_new.py*.

W plikach znajdują się następujące grupy tematyczne funkcji:

- *park_algo.py*: klasy i funkcje wspomagające parkowanie za pomocą czujników ultradźwiękowych,
- *camera_calibration.py*: funkcje pomocnicze dla budowania homografii, kalibracji i estymacji pozy kamery,
- *visualise.py*: funkcje do wizualizacji widoku „z lotu ptaka” i przekształconych obrazów z kamer; strefy detekcji czujników ultradźwiękowych,
- *parking_parallel_new.py*: główny plik projektu, w którym znajduje się pętla kontrolera samochodu w symulatorze i w ogóle są funkcje przetwarzające surowe dane z czujników i kamer na inne uniwersalne typy danych.

Cały projekt jest napisany w języku Python ze względu na łatwość implementacji różnych algorytmów i powszechność dokumentacji wszelakich projektów. Również Python pozwala na łatwą rozbudowę o przetwarzanie obrazów przez sieci neuronowe (*PyTorch*, *HuggingFace Transformers* itd.).

Oprócz plików skryptowych znajdują się pliki .npy – gotowe macierze przekształceń (homografie i pozy kamer względem samochodu). Pozostałe pliki są związane albo z Webots, albo pomocnicze typu obrazy wzorców szachownic.

3.6.1 Plik główny kontrolera *parking_parallel_new.py*

Pętla główna pliku kontrolera samochodu *parking_parallel_new.py* jest podzielona na dwa podrozdziały:

- część deklaracji zmiennych, funkcji, stałych,
- program główny *main()* z dodatkowymi deklaracjami funkcji oraz pętlą *while driver.step() != -1*.

Program główny wykonuje się w zasadzie jeden raz, a wszystkie operacje przetwarzania czujników i obrazów z kamer oraz zadania komend ruchu powinny się znaleźć w pętli.

Wszystkie próby implementacji wielowątkowości skończyły się niepowodzeniem ze względu na działanie kontrolera: w jednej pętli w każdym przypadku są wykonywane po kolej instrukcje, a np. dla przetwarzania obrazów z siecią neuronową lub na widok „z lotu ptaka” musimy wziąć pod uwagę czas przenoszenia danych do karty graficznej. Ponadto Matplotlib wykorzystywany dla wizualizacji czujników ultradźwiękowych nie jest zoptymalizowany pod kątem wykreślania figur na żywo.

Wszystkie te czynniki powodują, że przy włączeniu jakiegokolwiek przetwarzania z taką samą częstotliwością, jak i *basicTimeStep* w symulatorze (ustawiony na 4 ms), spowoduje bardzo spowolnione działanie kontrolera. *basicTimeStep* jest zadany z poziomu „Scene Tree” i definiuje, jak często mają być wykonywane akcje modelowania sceny i fizyki. Domyślnie jest wartość 16 (ms), natomiast z samochodami w Webots istnieje problem, że przy większym czasie są niewłaściwe zachowania modelu.

Ze względu na powyższe przyczyny zastąpiono wielowątkowość rozstawieniem w czasie różnych działań kontrolera: co *SENSOR_INTERVAL* (lub *IMAGE_INTERVAL* i *KEYBOARD_INTERVAL*) są wykonywane w pętli głównej komendy do czujników, kamer albo klawiatury. Dzięki temu można uniknąć zacięcia się programu w jednej chwili czasowej i rozłożyć je w czasie. Poniżej jest Listing 4, ilustrujący działanie tej pętli.

Listing 4. Kod pętli głównej kontrolera Webots, obsługujący czujniki, kamery i klawiaturę do sterowania samochodem.

```
01         first_call = True
02     while driver.step() != -1:
03
04         now = driver.getTime()
05
06         if parking:
07             # jeżeli z klawiatury włączono parkowanie, to:
08
09             if now - last_sensor_time >= SENSOR_INTERVAL:
10                 # co SENSOR_INTERVAL (zadany na początku) wykonujemy:
11                 last_sensor_time = now
12
13                 if first_call:
14                     #plotter = palg.LivePlotter(ax_live)
15                     parker = palg.Parking(driver, "left", now)
16                     yaw_init = imu.getRollPitchYaw()[2]
17
18                     first_call = False
19                     # odczyt odległości
20                     dists = [process_distance_sensors(s) for s in distance_sensors]
21                     names = dict(zip(sensor_names, dists))
22
23                     # draw_cones na ax_cones
```

```

22         vis.draw_cones(ax_cones, fig, dists)
23
24     # live-plot na ax_live
25     #plotter.val = names["distance sensor left front side"]
26     #plotter.update(0)
27
28     # automaty parkowania
29     yaw = imu.getRollPitchYaw()[2] - yaw_init
30     parker.update_state(names, yaw)
31     if parker.state == "waiting_for_park":
32         odom, spot = parker.update_state(names, yaw)
33         parker.exec_path(odom, spot, names["distance sensor front left
34 side"])
34         #fig.canvas.draw_idle()
35         #fig.canvas.flush_events()
36
37     # co IMAGE_INTERVAL - przetwarzanie obrazów
38     if now - last_image_time >= IMAGE_INTERVAL:
39         last_image_time = now
40         images = [get_camera_image(c) for c in cameras]
41         names_images = dict(zip(camera_names, images))
42
43         viss = vis.alt_collect_homo(names_images, homographies, car, streams)
44         #cv2.imwrite("img3_vis.png",viss)
45         cv2.waitKey(1)
46
47     if now - last_key_time >= KEYBOARD_INTERVAL:
48         last_key_time = now
49         check_keyboard()

```

3.6.2 Plik *park_algo.py*

Listing 5 zawiera częściowy opis klasy *Parking* – maszyny stanów obsługującej parkowanie, sterowanie samochodem i odczyt czujników.

Listing 5. Opis klasy „Parking” do parkowania równoległego.

```

class Parking:
    # pomocnicze żeby skompensować opóźnienie czujników
    marg_start = 0.2
    marg_end = 0.4
    def __init__(self, driver, side, times,
                 min_width=CAR_WIDTH*1.1,
                 min_length=CAR_LENGTH*1.25,
                 threshold=1*CAR_WIDTH):
        # Inicjalizacja parametrów parkingu
        self.min_width = min_width          # minimalna szerokość miejsca
        self.min_length = min_length        # minimalna długość miejsca
        self.threshold = threshold          # próg zmiany odległości

        self.state      = "searching_start"  # aktualny stan maszyny stanów
        self.start_pose = None              # miejsce wykrycia początku miejsca
        self.spots      = []                # wykryte miejsca parkingowe
        self.driver     = driver             # interfejs do samochodu w Webots
        self.spot = None

        # Pomocnicze zmienne do detekcji zmian odległości
        self.prev_distance_front = 6.0
        self.prev_distance_rear = 6.0
        self.dist_start_far = 6.0
        self.dist_start_cl = 6.0
        self.dist_end_far = 6.0
        self.dist_end_cl = 6.0
        self.side = side                   # strona parkowania: "left" lub "right"
        self.x = 0.0                        # os x w układzie globalnym
        self.y = 0.0                        # os y w układzie globalnym
        self.yaw = 0.0                       # orientacja pojazdu
        self.last_time = times             # czas ostatniej aktualizacji

    # Parametry regulatora PID dla sterowania
    self.Kp = 1.2
    self.Kd = 0.001

```

```

        self.K1 = 0.5
        self.prev_yaw_err = 0.0

    def update_odometry(self, yaw):
        """
        Aktualizuje pozycję (x,y) na podstawie prędkości i kąta yaw.
        """
        # Pobierz prędkość w km/h i przelicz na m/s
        v = self.driver.getCurrentSpeed() / 3.6
        # Integracja prostokątna z krokiem czasowym 0.05s
        self.yaw = yaw
        dx = v * math.cos(self.yaw) * 0.06
        dy = v * math.sin(self.yaw) * 0.06
        self.x += dx
        self.y += dy
        return (self.x, self.y, self.yaw)

    def update_state(self, dists_names, yaw):
        """
        Maszyna stanów wykrywająca luki parkingowe.
        """
        # Wybór odpowiednich czujników w zależności od strony
        if self.side == "left":
            distance_front = dists_names["distance sensor left front side"]
            distance_rear = dists_names["distance sensor left side"]
        else: # self.side == "right"
            distance_front = dists_names["distance sensor right front side"]
            distance_rear = dists_names["distance sensor right side"]

        # Oblicz zmiany odległości
        delta_front = distance_front - self.prev_distance_front
        delta_rear = distance_rear - self.prev_distance_rear
        self.prev_distance_front = distance_front
        self.prev_distance_rear = distance_rear

        # Aktualizuj pozycję z odometrii
        odom_pose = self.update_odometry(yaw)
        x, y, yaw = odom_pose
        print(f"Odometria : {odom_pose}")
        # Stan: poszukiwanie początku luki
        if self.state == "searching_start":
            if delta_front > self.threshold:
                # Znaleziono gwałtowny wzrost odległości => początek luki
                self.start_pose = (x - self.marg_start, y, yaw)
                self.dist_start_far = distance_front
                self.dist_start_cl = self.prev_distance_front
                self.state = "searching_progress"
                print("Kandydat na miejsce znalezione.")

        # Stan: poszukiwanie końca luki
        elif self.state == "searching_progress":
            if -delta_front > self.threshold:
                # Znaleziono gwałtowny spadek odległości => koniec luki
                end_pose = (x + self.marg_end, y, yaw)
                self.dist_end_far = distance_front
                self.dist_end_cl = distance_front - delta_front

                # Utwórz obiekt miejsca parkingowego
                spot = self._make_spot(self.start_pose, end_pose)
                if spot:
                    self.state = "waiting_for_park"
                    self.spot = spot
                    print("Miejsce znalezione. Wciśnij Y, aby rozpocząć parkowanie.", spot)
                else:
                    print("Miejsce okazało się za małe.")
                    self.state = "searching_start"

        if self.spot is not None: return odom_pose, self.spot

```

Pozostałe funkcje są w realizacji i są przytoczone w dodatku. W komentarzach jest opisane działanie każdej funkcji.

3.6.3 Plik *camera_calibration.py*

W tym pliku, jak było wspomniano, znajdują się funkcje do różnego rodzaju operacji odnajdywania wzorców, tworzenia homografii, wyznaczenia pozy itd. Znajdą się tutaj funkcje, które bezpośrednio zostały użyte do projektu, a pomocnicze znajdą się w dodatku.

Funkcje

solve_camera_pose(image, pattern_size, K, camera_name, show=True)

Dla podanego obrazu z kamery i rozmiaru wzorca szachownicy (np. 5×7) wykonuje:

- wykrycie narożników szachownicy i poprawienie ich współrzędnych subpixselowo,
- wyliczenie pozycji kamery względem wzorca (wektor rotacji i translacji) za pomocą metody PnP,
- narysowanie osi układu współrzędnych na obrazie oraz zaznaczenie narożników.

Zwraca macierz obrotu (R) i wektor przesunięcia ($tvec$).

warp_images(img1, img2, homography)

Nakłada drugi obraz (img2) na pierwszy (img1), wykorzystując całą macierz homografii. Tworzy kanwę wystarczająco dużą, by zmieścić oba zdjęcia, a następnie łączy je, stosując mieszanie „feather”.

chess_homography(img1, img2, pattern_size, margin=200)

Wyznacza homografię między dwoma obrazami na podstawie wykrycia tego samego wzorca szachownicy w każdym. Kolejne kroki:

1. Wykrycie narożników szachownicy w oryginalnych img1 i img2.
2. Obliczenie obramowania (bounding box) wokół wykrytej szachownicy i rozszerzenie go o margin w pikselach.
3. Przycięcie obu zdjęć tylko do obszarów z szachownicą.
4. Ponowne wykrycie narożników w przyciętych fragmentach, co zmniejsza liczbę szumów.
5. Obliczenie homografii metodą RANSAC na podstawie dopasowanych narożników.
6. (Opcjonalnie) Wyświetlenie wykrytych narożników i efektu wyprostowania.
Zwraca macierz H oraz wyprostowaną wersję img1 („warped_img”).

chess_homography_multiple_boards(img1, img2, pattern_size, second_patt_size, margin=200)

Zaawansowana wersja, gdy na zdjęciu są dwa oddzielne wzorce szachownicy (np. w różnych miejscach sceny). Procedura:

- 1) znajduje pierwszy wzorzec o wymiarach *pattern_size*, tworzy maskę i zaciemnia ten obszar,

- 2) na pozostałym fragmencie obrazu wykrywa drugi wzorzec o wymiarach *second_patt_size*,
- 3) łączy narożniki obu wzorców w jeden zestaw punktów dopasowania,
- 4) oblicza homografię RANSAC-em na podstawie połączonych narożników,
- 5) wyświetla wyniki detekcji i efekt prostowania.
Umożliwia uzyskanie większej ilości punktów dopasowania, gdy standardowa detekcja jednego wzorca byłaby niewystarczająca.

```
solve_chess_size(image, name, pattern_size,
pattern_size2, second=False)
```

Pomocnicza funkcja do szybkiego określenia współrzędnych narożników wzorca (lub dwóch wzorców) na obrazie:

- wykrywa narożniki wzorca *pattern_size* w całym *image* i wyświetla je.
- jeśli *second=True*, maskuje pierwszy wzorzec i w tym samym obrazie szuka drugiego wzorca *pattern_size2*,
- łączy wykryte punkty w jedną listę *corners_4* lub *corners_combined*.
- zwraca tablicę współrzędnych narożników w pikselach.
Przydatna do szybkiego pomiaru wymiarów wzorca bez pełnego pipeline'u PnP czy homografii.

3.6.4 Plik *visualise.py*

W danym pliku są funkcje do nałożenia homografii na obrazy, tworzenia widoku „z lotu ptaka” oraz wizualizacji czujników ultradźwiękowych. Umieszczono tutaj opis funkcji wykorzystywanych bezpośrednio w finalnej wersji projektu, pozostałe testowe i przestarzałe można znaleźć na repozytorium GitHub (w dodatku również umieszczono tylko aktualne funkcje):

Funkcje

```
draw_cones(ax, fig, distances)
```

Wizualizacja stożków czujników ultradźwiękowych (interaktywne rysowanie w pętli), umieszczonych na samochodzie i zorientowanych jak na rys. Rys. 5,

```
collect_homo(names_images, homographies, car, streams)
```

Drugi sposób sporządzenia widoku z lotu ptaka, bierze „streams” do przetwarzania na GPU, obrazy z kamer surowe, homografie do rzutowania każdego obrazu na płaszczyznę drogi i skleja po dwóch częściach w cały widok.

```
alt_collect_homo(names_images, homographies, car, streams)
```

Pierwszy sposób tworzenia widoku „z lotu ptaka” – łączy łańcuchowo obrazy na kanwie również za pomocą GPU.

```
chain_collect_homo(names_images, homographies, car, streams)
```

Trzeci sposób na widok „z lotu ptaka” – rzutowanie obrazów na wspólnej kanwie (lepiej opisane w dodatku w komentarzach).

```
warp_with_cuda(image, H, name, h, w, stream, gpu=True,  
show=False, first_time=True)
```

Nałożenie korekcji perspektywy na obrazie z GPU (może być na wejściu albo *GpuMat*, albo macierz pikseli *numpy*).

```
warp_and_blend_gpu(img1, img2, H, canvas_size=None, alpha=0.8)
```

Nakładanie wzajemnej homografii i mieszanie obrazów, wszystkie operacje wykonywane na strumieniach na GPU.

```
blend_warp_GPUONLY(g1, g2, H, stream, canvas_size=None,  
alpha=0.85)
```

Nakładanie wzajemnej homografii i mieszanie obrazów, wszystkie operacje wykonywane na strumieniach na GPU, ale na wejściu i wyjściu są przyjmowane i zwracane tylko *GpuMat*.

3.6.5 Plik *stereo_yolo.py*

W tym pliku znajdują się wszystkie funkcje, z których korzystano w eksperymentach z sieciami neuronowymi, YOLO łącznie. Dodatkowo tutaj są funkcje do obsługi stereowizji. Funkcje muszą być użyte razem z innymi, a kod nie był zaprojektowany jako modułowy – każda funkcja rzadko jest używana pojedynczo, częściej co najmniej w parze z innymi.

Funkcje

calculate_intrinsic_matrix(width, height, fov_rad)

Buduje macierz wewnętrzną kamery (3×3) na podstawie rozdzielczości obrazu (width, height) i poziomego kąta widzenia (fov_rad). Zakłada brak dystorsji oraz kwadratowe piksele, więc ogniskowe w osiach X i Y są identyczne.

build_homogeneous_transform(R, t)

Tworzy jednorodną macierz $4 \times 4 T$, w której górny lewy blok to macierz rotacji R , a wektor przesunięcia t trafia do pierwszych trzech elementów czwartej kolumny. Pozwala w prosty sposób otrzymać transformację przestrzenną (kamery – scena).

build_pose_matrix(position, yaw_deg)

Generuje macierz 4×4 opisującą pozycję wzorca w przestrzeni, przyjmując wektor położenia $position = [x, y, z]$ oraz kąt yaw w stopniach yaw_deg . Obrót wokół osi Z konstruowany jest przez macierz 3×3 , a następnie łączony z translacją.

pixel_to_world(u, v, K, T_center_to_camera)

Przelicza współrzędne piksela (u, v) w obrazie kamery na punkt w płaszczyźnie drogi ($Z=0$) w układzie pojazdu:

- 1) tworzy promień w przestrzeni kamery $ray_camera = inv(K) \cdot [u, v, 1]$,
 - 2) rzutuje na świat przez część rotacyjną $T_center_to_camera$:
$$ray_world = R_cam_to_car \cdot ray_camera,$$
 - 3) oblicza parametr t , gdzie promień przecina płaszczyznę $Z=0$:
$$t = -camera_position_z / ray_world_z,$$
 - 4) punkt w świecie: $camera_position + t \cdot ray_world$.
- Zwraca $[X, Y, 0]$ lub *None*, gdy promień równoległy do ziemi.

project_points_world_to_image(points_world, T_world_to_camera, K)

Rzutuje listę punktów 3D w układzie świata (points_world) na piksele obrazu kamery:

- 1) odwraca $T_world_to_camera$ (by uzyskać $T_camera_to_world$),
- 2) dla każdego punktu $point_h = [X, Y, Z, 1]$ liczy $point_in_camera = T_camera_to_world \cdot point_h$,

- 3) ześli $Zc > 0$, rzutuje do obrazu:
 $p_image = K \cdot [Xc, Yc, Zc]$, a następnie $u = p_x/p_z, v = p_y/p_z$,
- 4) zwraca listę punktów na obrazie (u, v) .

```
create_3d_box(anchor_point,      side,      length=4.5,      width=1.8,
height=1.8)
```

Generuje osiem punktów (podstawa + wierzchołki) 3D prostopadłościanu (bryły) zadanego wymiaru, zaczepionego w punkcie $anchor_point = [x, y, z]$ (najczęściej na osi $Z=0$). Parametr $side$ (*left* lub *right*) określa kierunek poszerzenia szerokości ($width$) wzduż osi Y. Zwraca listę 8 punktów $[x, y, z]$.

```
classify_object_position_and_anchor(bbox,   K,   T_center_to_camera,
camera_name)
```

Dla danego prostokąta obwiedni obiektu ($bbox = (x, y, w, h)$) w obrazie:

- 1) znajduje dwa dolne narożniki *pixel_to_world*: lewy $(x, y+h)$ i prawy $(x+w, y+h)$.
- 2) przelicza je na współrzędne BEV (*pixel_to_world*).
- 3) jeżeli żaden nie zwraca *None*, na podstawie wartości w osi Y decyduje, która strona jest bliżej (jeśli $pt_right[1] < 0$ – *right*, w przeciwnym razie *left*).
- 4) modyfikuje *anchor* (punkt zaczepienia) oraz *side* uwzględniając dodatkowe reguły dla różnych nazw kamery (*camera_name*) – okazało się nieużyteczne ze względu na
- 5) zwraca punkt zaczepienia prostokąta YOLO *anchor* i stronę (*left/right*) lub (*None, None*), jeśli oba punkty wychodzą poza zasięg.

```
disp_to_cam3D(u, v, d, K, B)
```

Przelicza pojedynczą wartość dysparcji d (odległość w pikselach między stereo) oraz piksel (u, v) w obrazie na współrzędne $[X, Y, Z]$ w układzie kamery.

```
points_from_mask_to_3D(mask_resized, filtered_disp, K, baseline,
T_cam_to_car)
```

Przygotowuje dwa punkty 3D (ekstrema obiektu) w BEV na podstawie binarnej maski segmentacyjnej i mapy dysparcji:

- 1) w *mask_resized* (binarnej) szuka skrajnych pikseli (najmniejszego $x - left_x$, największego $x - right_x$),
- 2) dla każdego z tych x wybiera medianę y w danej kolumnie (*left_y, right_y*),
- 3) używa *get_valid_disparity* (medianą okienka 30×30 wokół punktu) do wyznaczenia stabilnej dysparcji *disp_left, disp_right*,
- 4) jeśli wartości dysparcji są dodatnie, liczy współrzędne kamery $[X1, Y1, Z1], [X2, Y2, Z2]$ w modelu otworkowym kamery (analogicznie jak w *disp_to_cam3D*),

- 5) buduje wektory homogeniczne $[X, Y, Z, 1]$ i mnoży przez $T_{cam_to_car}$ (transformuje do układu pojazdu),
- 6) zwraca $point_car1[:3]$, $point_car2[:3]$ lub $(None, None)$, jeżeli nie udało się uzyskać ważnych dysparcji.

Użyteczna przy wizualizacji wykrytych obiektów: wyznacza realną pozycję boków pojazdu (lub innego obiektu) w płaszczyźnie .

3.7 Opis uruchomieniowy

Projekt był wykonany na komputerze osobistym PC z systemem operacyjnym **Windows 10 Home wersji 22H2**. Nie ma gwarancji, że nie pojawią się nieprzewidywalne problemy przy instalacji na innych platformach, związane z losowymi, niezależnymi od ingerencji obu stron, szczegółami konfiguracji systemów.

Wersja Pythona wykorzystana w projekcie: **3.12.9**.

Aby uruchomić projekt, należy zainstalować symulator **Webots R2025a**. Można posłużyć się tutorialem oficjalnym dostępnym na stronie:

<https://cyberbotics.com/doc/guide/installation-procedure#installation-on-windows>

Po instalacji może się przydać dodanie zmiennej środowiskowej systemowej **WEBOTS_HOME** jako wybranej ścieżki instalacyjnej symulatora.

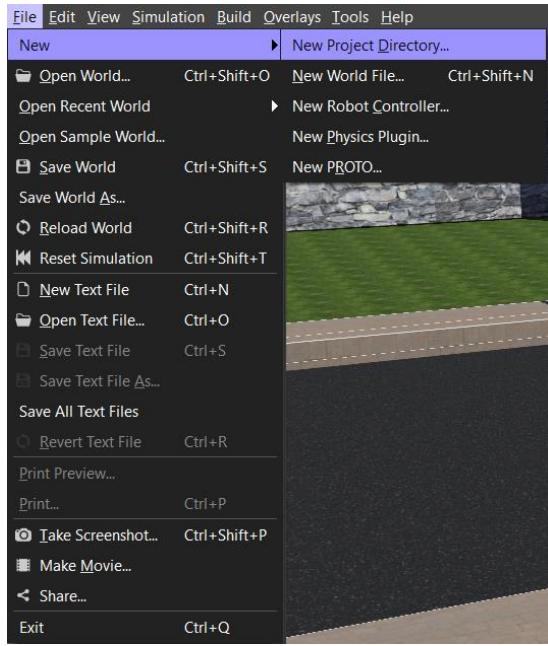
Należy postępować, jak jest opisane w następującej procedurze:

- 1) otworzyć dowolny folder w terminalu (kliknąć *prawym przyciskiem myszy* na wolnym miejscu w wybranym folderze i wcisnąć „*Otworzyć w Terminalu*”); jeżeli nie ma takiej funkcji, to należy otworzyć terminal i wprowadzić „*cd sciezka/do/folderu*”, gdzie „*sciezka/do/folderu*” jest wybrany folder;
- 2) następnie wprowadzić poniższą komendę do linii poleceń:

```
git clone https://github.com/fallenleaves04/webots_car.git
```

Ona skopiuje całe repozytorium do wybranego folderu.

- 3) wewnątrz symulatora należy utworzyć dowolną ścieżkę projektową wciskając ukazane na poniższym rys. Rys. 36 instrukcje:



Rys. 36. Procedura utworzenia własnej ścieżki projektowej w Webots.

- 4) nie wybierając niestandardowych opcji, użytkownik powinien otrzymać czystą ścieżkę projektową. Doń należy wstawić i zamienić na żądanie eksploratora plików wszystkie pliki z uprzednio sklonowanego repozytorium;
- 5) następnie, będąc w tym samym folderze, należy zastosować poniższą komendę do zainstalowania pozostałych wymaganych pakietów Python:

```
pip install -r requirements.txt
```

- 6) zainstalować pakiet „*opencv....whl*” znajdujący się w folderze sklonowanego repozytorium pośrednictwem następującej komendy (lepiej to instalować po *requirements.txt*, ponieważ moduł *ultralytics* może nadpisać OpenCV i nie zadziałają funkcje z GPU):

```
pip install opencv_contrib_python-4.11.0.86-cp37-abi3-win_amd64.whl
```

Ta wersja OpenCV została [zbudowana](#) na podstawie CUDA Toolkit 12.6, cuDNN 9.6.0, Nvidia Video Codek SDK 12.2 (przynajmniej ostatni komponent jest zbędny).

- 7) uruchomić plik *projekt.wbt*. Webots powinien odnaleźć się w tej ścieżce i pokazać w dostępnych kontrolerach obiektu „BMW X5 „SAMOCHOD TESTOWY”” (drugi z końca na drzewie sceny po lewej),
- 8) wybrać kontroler „*parking_parallel_new*”.

Po prawidłowym wykonaniu wszystkich kroków powinien się uruchomić domyślnie projekt ze wskazówkami w konsoli.

Instrukcje w sytuacjach, gdzie by zaistniały różne konflikty:

- w przypadku ewentualnych konfliktów można odwołać się do adresu mailowego: 01184625@pw.edu.pl;
- bardzo często pojawiają się problemy ze zmiennymi środowiskowymi, które mogą sprawić, że moduły nie są odczytywalne. Należy więc sprawdzić, czy wszystkie związane z Pythonem i Webots zmienne są w miejscu – i są poprawne. Ścieżkę do aplikacji Python można znaleźć w następujący sposób:

```
python -c "import sys; print(sys.executable)"
```

Wyświetloną ścieżkę trzeba dodać do **Path** (zmienne środowiskowe w ustawieniach (bez .exe),

- powyższe nie rozwiąże wszystkich ewentualnych problemów, a najczęściej konflikty konfiguracji spowodowane są nieczystą instalacją Pythona. Trzeba też sprawdzić, czy nie ma innych wersji Pythona.

4 Podsumowanie i wnioski

W ramach projektu udało się zasymulować samochód z systemem wizyjnym i czujnikami ultradźwiękowymi jako przygotowanie do systemu autonomicznego parkowania. Ze względu na różne czynniki nie udało się ukończyć podstawowego zadania, natomiast były sporządzane do tego wymagane systemy. Również podjęto próby implementacji algorytmu poszukiwania miejsca za pomocą maszyny stanów.

Jednym z najważniejszych wniosków jest to, że nie da się wykonać absolutnie precyzyjnych narzędzi do takich celów. Zawsze będą różnorodne błędy, pojawiające się tylko z eksperymentami i które da się wyjaśnić jedynie nabierając doświadczenie i intuicję. Na przykład, przekształcenie homograficzne wydaje się być trywialnym zagadnieniem na pierwszy rzut oka, ale w czasie sporządzenia widoku „z lotu ptaka” nawet w wyidealizowanych warunkach symulatora (kamery bez zniekształceń, dowolne kąty widzenia) pojawiały się problemy z precyzją. To daje wywnioskować, że przy wielu uproszczeniach też jednak możemy wiernie odzworzyć rzeczywiste warunki, co potwierdza korzyść symulatorów dla motoryzacji.

Trzeba też zaznaczyć, że wykorzystanie gotowych uniwersalnych rozwiązań zawsze wymaga dopracowania pod własne warunki. Na przykład sieć segmentacyjna YOLO w parze ze stereowizją nie jest najlepszym rozwiązaniem, i lepiej jest sporządzić własny zbiór danych do wyznaczenia miejsca parkingowego nawet z widoku „z lotu ptaka” lub z pojedynczej kamery. Rozważając przykład [Tesli](#), możemy przypuścić, że wysiłek wydany na stworzenie własnej [architektury](#) sztucznej inteligencji dla samochodu autonomicznego byłby korzystniejszy, niż jeżeli korzystano ze sztywnych rozwiązań przetwarzania obrazów.

Python jest językiem interpretowanym, nie ma potrzeby eksperymentowania z kompilatorem, a duże projekty są zdecydowanie łatwiejsze do napisania.

Przepisanie projektu na język C++ byłoby zalecane ze względu na efektywność systemu w czasie rzeczywistym, jednak niektóre cechy projektów mogą być utracone: wizualizację dla czujników trzeba byłoby oprzeć na aplikacjach *Qt*, a dla *OpenCV* potrzebny byłby zbudować we własnym zakresie pakiet ze wsparciem przetwarzania na jądrach *CUDA*. Owszem, jest możliwa taka konwersja, i do tego istnieją już narzędzia programistyczne, jednak przez początkowo выбрany kierunek rozwoju projektu oddano czas na taki, a nie inny, sposób napisania skryptów.

4.1 Możliwości rozbudowy

W ramach polepszenia projektu można rozważyć wykorzystanie bardziej zaawansowanych i niezawodnych metod detekcji miejsca parkingowego i optymalizacja bazy kodowej pod względem skuteczności operacji w czasie rzeczywistym.

Również rozważa się poszukiwanie innych, bardziej niezawodnych metod sporządzenia widoku „z lotu ptaka”, które mogłyby być robione automatycznie, bez sztucznych wzorców.

Samochody produkcyjne, takie jak Tesla, kalibrują kamery w ruchu, korzystając najprawdopodobniej ze znanych wzorców typu oznakowania poziomego.

4.2 Wykorzystanie Sztucznej Inteligencji

Jako wspomaganie przy pisaniu i porządkowaniu dokumentacji użyto ChatGPT, jak i przy generowaniu przykładowych fragmentów kodu (np. funkcji do przycinania i skalowania widoku „z lotu ptaka”, funkcje do poszukiwania szachownicy – istnieją w Internecie, ale nie są dopracowane). Sieć YOLO posłużyła do detekcji i segmentacji pojazdów na obrazach z kamer, a eksperymenty z sieciami do estymacji głębi typu MiDaS umożliwiły wstępne testy oszacowania głębi. Sztuczna inteligencja przyspiesza rozwój oprogramowania, choć wymaga ręcznej weryfikacji wyników i optymalizacji modeli, aby działały one stabilnie w symulatorze.

Bibliografia

W danej sekcji wszystkie przytoczone źródła informacji są dostępne w wersji internetowej, więc wypisano wszystko z linkami źródłowymi:

Publikacje drukowane i formalne

- [1] Hsu, C.-M.; Chen, J.-Y. *Around View Monitoring-Based Vacant Parking Space Detection and Analysis*. *Applied Sciences* 2019, **9**(16), 3403. <https://doi.org/10.3390/app9163403>, ostatni dostęp: 25.05.2025.
- [2] Hsu, T.-H.; Liu, J.-F.; Yu, P.-N.; Lee, W.-S.; Hsu, J.-S. *Development of an Automatic Parking System for Vehicle*. *IEEE Vehicular Technology Conference (VTC Spring)*, 2010, pp. 1–4. <https://ieeexplore.ieee.org/document/4677655>, ostatni dostęp: 26.05.2025.
- [3] Lee, Y.; Chang, S. *Development of a Verification Method on Ultrasonic-Based Perpendicular Parking Assist System*. *18th IEEE International Symposium on Consumer Electronics (ISCE 2014)*, Jeju, Korea (South), 2014, pp. 1–3. <https://ieeexplore.ieee.org/abstract/document/6884292>, ostatni dostęp: 28.05.2025.
- [4] Li, M.-H.; Tseng, P.-K. *Implementation of an Autonomous Driving System for Parallel and Perpendicular Parking*. *IEEE International Conference on Consumer Electronics–Asia (ICCE-Asia 2018)*, 2018, pp. 276–279. <https://ieeexplore.ieee.org/abstract/document/7843998>, ostatni dostęp: 25.05.2025.
- [5] Li, W.; Cao, L.; Yan, L.; Li, C.; Feng, X.; Zhao, P. *Vacant Parking Slot Detection in the Around View Image Based on Deep Learning*. *Sensors* 2020, **20**(7), 2138. <https://doi.org/10.3390/s20072138>, ostatni dostęp: 25.05.2025.
- [6] Park, W.-J.; Kim, B.-S.; Seo, D.-E.; Kim, D.-S.; Lee, K.-H. *Parking Space Detection Using Ultrasonic Sensor in Parking Assistance System*. *IEEE Transactions on Intelligent Transportation Systems* 2008, **9**(3), 400–407. <https://ieeexplore.ieee.org/abstract/document/4621296>, ostatni dostęp: 25.05.2025.
- [7] Ranftl, R.; Lasinger, K.; Hafner, D.; Schindler, K.; Koltun, V. *Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-shot Cross-dataset Transfer*. *arXiv* 2019, arXiv:1907.01341. <https://arxiv.org/abs/1907.01341>, ostatni dostęp: 26.05.2025.
- [8] Shao, Y.; Chen, P.; Cao, T. *A Grid Projection Method Based on Ultrasonic Sensor for Parking Space Detection*. *IEEE/CAA Journal of Automatica Sinica* 2020, **7**(7), 180–190. <https://ieeexplore.ieee.org/abstract/document/8519022>, ostatni dostęp: 25.05.2025.
- [9] Wang, C.; Zhang, H.; Yang, M.; Wang, X.; Ye, L.; Guo, C. *Automatic Parking Based on a Bird's Eye View Vision System*. *ISRN Automatica* 2014, Article ID 847406. <https://journals.sagepub.com/doi/full/10.1155/2014/847406>, ostatni dostęp: 25.05.2025.

[10] Wang, W.; Song, Y.; Zhan, J.; Deng, H. *Automatic Parking of Vehicles: A Review of Literatures*. *Symmetry* 2018, **10**(3), 64. <https://doi.org/10.3390/sym10030064>, ostatni dostęp: 25.05.2025.

[11] Watada, J.; Zhang, H.; Melo, H.; Wang, J.; Vasant, P. *SURF Algorithm-Based Panoramic Image Mosaic Application*. W: *Proceedings of the International Symposium on Neural Networks*, Vol. 10170, Springer, Cham, 2016, s. 455–462. https://link.springer.com/chapter/10.1007/978-3-319-63856-0_43, ostatni dostęp: 26.06.2025.

[12] Zhang, L.; Li, X.; Huang, J.; Shen, Y.; Wang, D. *Vision-Based Parking-Slot Detection: A Benchmark and a Learning-Based Approach*. *Symmetry* 2018, **10**(3), 64. <https://doi.org/10.3390/sym10030064>, ostatni dostęp: 25.05.2025.

[13] Zhou, X.; Chen, W.; Wei, X. *Improved Field Obstacle Detection Algorithm Based on YOLOv8*. *Agriculture* 2024, **14**(12), 2263. <https://doi.org/10.3390/agriculture14122263>, ostatni dostęp: 26.06.2025.

Materiały internetowe

[14] Cyberbotics, *Webots for Automobiles*. <https://cyberbotics.com/doc/automobile/index>, ostatni dostęp: 25.05.2025.

[15] GitHub, *ml-depth-pro (DepthPro od Apple)*. <https://github.com/apple/ml-depth-pro>, ostatni dostęp: 27.05.2025.

[16] Google Research Blog, *Seamless Google Street View Panoramas*. <https://research.google/blog/seamless-google-street-view-panoramas/>, ostatni dostęp: 27.05.2025.

[17] HuggingFace, *Spis modeli do estymacji głębi*. https://huggingface.co/models?pipeline_tag=depth-estimation&sort=trending, ostatni dostęp: 26.05.2025.

[18] OpenCV, *Opis dodatkowych funkcji klasy cv::Stitcher, cv::composePanorama*. https://docs.opencv.org/4.x/d2/d8d/classcv_1_1Stitcher.html#acc8409a6b2e548de1653f0dc5c2ccb02, ostatni dostęp: 26.06.2025.

[19] OpenCV, *Strona OpenCV z opisem klasy cv::Stitcher*. https://docs.opencv.org/4.x/d2/d8d/classcv_1_1Stitcher.html, ostatni dostęp: 26.06.2025.

[20] OpenCV, *Tutorial „Depth Map From Stereo Images”*. https://docs.opencv.org/4.x/dd/d53/tutorial_py_depthmap.html, ostatni dostęp: 26.05.2025.

[21] Strona Tesla, *sekcja Autopilot*. <https://www.tesla.com/autopilot>, ostatni dostęp: 25.05.2025.

[22] Waymo, sekcja Waymo Driver. <https://waymo.com/waymo-driver/>, ostatni dostęp: 25.05.2025.

[23] Wikipedia, Tesla Autopilot Hardware. https://en.wikipedia.org/wiki/Tesla_Autopilot_hardware#Hardware_4, ostatni dostęp: 25.05.2025.

Dodatek

W dodatku umieszczono kod źródłowy plików programu. W większości plików usunięto takie funkcje, które nie użyto w finalnej implementacji, oraz odkomentowano różne fragmenty kodu do np. wyznaczenia pozy kamery (takie, które korzystają z innych funkcji i zamieszczone na takie momenty w pętli głównej) albo testowe fragmenty do widoku „z lotu ptaka”. Cały kod natomiast znajduje się na repozytorium [GitHub](#).

Plik camera_calibration.py

W tym pliku są funkcje do estymacji pozy, wyznaczenia homografii, zapisu macierzy, odnalezienia wzorców.

```
import cv2 as cv
import numpy as np
import os
import sys

CAMERA_HEIGHT=2160
CAMERA_WIDTH=3840

# Vehicle parameters
TRACK_FRONT = 1.628
TRACK_REAR = 1.628
WHEELBASE = 2.995
MAX_WHEEL_ANGLE = 0.5 # rad
WHEEL_RADIUS = 0.374
import visualise as vis

def average_chessboard_size(corners, pattern_size):
    """
    Funkcja instrumentalna licząca wymiary kwadratów szachownicy.
    Wykorzystywana przez funkcję "solve_chess_size".
    Zlicza wszystkie narożniki i na podstawie tego wylicza średnią.
    """
    cols, rows = pattern_size
    corners = corners.reshape(-1, 2)

    horizontal_lengths = []
    vertical_lengths = []

    for row in range(rows):
        for col in range(cols - 1):
            idx1 = row * cols + col
            idx2 = idx1 + 1
            dist = np.linalg.norm(corners[idx1] - corners[idx2])
            horizontal_lengths.append(dist)

    for row in range(rows - 1):
        for col in range(cols):
            idx1 = row * cols + col
            idx2 = idx1 + cols
            dist = np.linalg.norm(corners[idx1] - corners[idx2])
            vertical_lengths.append(dist)

    width_lengths = []
    for row in range(rows):
        start_idx = row * cols
        end_idx = start_idx + cols - 1
        width = np.linalg.norm(corners[start_idx] - corners[end_idx])
        width_lengths.append(width)

    height_lengths = []
    for col in range(cols):
        start_idx = col
        end_idx = start_idx + (rows - 1) * cols
        height = np.linalg.norm(corners[start_idx] - corners[end_idx])
        height_lengths.append(height)

    avg_width = np.mean(horizontal_lengths)
    avg_height = np.mean(vertical_lengths)
    total_width = np.mean(width_lengths)
    total_height = np.mean(height_lengths)

    return avg_width, avg_height, total_width, total_height

def solve_camera_pose(image, pattern_size, K, camera_name, show=True):
    """
    Estimate camera pose using a chessboard pattern with solvePnP.

    :param image: input BGR image from Webots camera
    :param pattern_size: tuple of (cols, rows) in the chessboard (e.g., (5, 7))
    :param square_size: size of a square in meters (e.g., 0.03 for 3cm)
    :param K: intrinsic camera matrix
    :param camera_name: name for logging
    :param show: whether to display the detection
    :return: R (3x3), t (3x1), or None if not found
    """
    pass
```

```

wszystkie punkty wymiarowania szachownic są liczone
od lewego górnego odnalezioneego przez algorytm - w metrach
przeliczone

"""
# Define 3D object points (0,0,0), (1,0,0), ..., in chessboard frame
objp = np.zeros((4,2),dtype=np.float32)
if camera_name == "camera_front_bumper_wide":

    objp = np.array([[0.0,0.0],[0.0,-1.2],[-0.8,-1.2],[-0.8,0]]).astype(np.float32)
elif camera_name == "camera_rear":

    objp = np.array([[0,0],[0,1.2],[0.8,1.2],[0.8,0]]).astype(np.float32)
elif camera_name == "camera_left_fender":

    objp = np.array([[0,0],[0,-0.8],[-1.2,-0.8],[-1.2,0]]).astype(np.float32)
elif camera_name == "camera_right_fender":

    objp = np.array([[0,0],[0,-0.8],[-1.2,-0.8],[-1.2,0]]).astype(np.float32)
elif camera_name == "camera_left_pillar":

    objp = np.array([[0,0],[0,-0.8],[-1.2,-0.8],[-1.2,0]]).astype(np.float32)
elif camera_name == "camera_right_pillar":

    objp = np.array([[0,0],[0,0.8],[1.2,0.8],[1.2,0]]).astype(np.float32)
elif camera_name == "camera_front_top":
    objp = np.array([[0.0,0.0],[0.0,1.2],[0.8,1.2],[0.8,0]]).astype(np.float32)
objp_fixed = np.zeros((4, 3), dtype=np.float32)
objp_fixed[:, :2] = objp #dodajemy Z=0

gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
ret, corners = cv.findChessboardCorners(gray, pattern_size)

if not ret:
    print(f"[{camera_name}] chessboard not found.")
    return None, None

# Subpixel refinement
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 120, 0.00001)
corners_refined = cv.cornerSubPix(gray, corners, (21, 21), (-1, -1), criteria)

# Assume no distortion
distCoeffs = np.zeros((4,1))
cols, rows = pattern_size

top_left = corners_refined[0]
top_right = corners_refined[cols - 1]
bottom_left = corners_refined[(rows - 1) * cols]
bottom_right = corners_refined[rows * cols - 1]

chessboard_corners_4 = np.array([
    top_left,
    top_right,
    bottom_right,
    bottom_left
], dtype=np.float32)
# Solve PnP
success, rvec, tvec = cv.solvePnP(objp_fixed, chessboard_corners_4, K, distCoeffs)

if not success:
    print(f"[{camera_name}] solvePnP failed.")
    return None, None

R, _ = cv.Rodrigues(rvec)

#print(f"\n[{camera_name}] === Camera Pose ===")
#print("Rotation matrix:\n", R)
#print("Translation vector (in meters):\n", tvec.ravel())

if show:

    # os X (czerwona), Y (zielona), Z (niebieska)
    axis = np.float32([[0.2,0,0], [0,0.2,0], [0,0,0.2]]).reshape(-1,3) # 20 cm osie
    imgpts, _ = cv.projectPoints(axis, rvec, tvec, K, distCoeffs)

    origin = tuple(chessboard_corners_4[0].ravel().astype(int))
    vis = cv.line(image, origin, tuple(imgpts[0].ravel().astype(int)), (0,0,255), 3) # X
    vis = cv.line(image, origin, tuple(imgpts[1].ravel().astype(int)), (0,255,0), 3) # Y
    vis = cv.line(image, origin, tuple(imgpts[2].ravel().astype(int)), (255,0,0), 3) # Z
    vis = cv.drawChessboardCorners(image, pattern_size, corners_refined, ret)
    cv.namedWindow(f"Chessboard - {camera_name}",cv.WINDOW_NORMAL)
    cv.imshow(f"Chessboard - {camera_name}", vis)

```

```

    return R, tvec

#-----
def gaussian_blur_cuda(image, ksize, sigma):
    """
    Funkcja do nałożenia gaussowskiego rozmycia.
    Jak w poprzedniej funkcji, najlepiej skopiować do innej funkcji,
    aby można było cały proces robić na gpu i nie przenosić danych zbytnio.
    """
    if len(image.shape) == 2:
        src_type = cv.CV_8UC1
    elif image.shape[2] == 3:
        src_type = cv.CV_8UC3
    else:
        raise ValueError("Obraz musi być 1- lub 3-kanałowy")

    gpu_img = cv.cuda_GpuMat()
    gpu_img.upload(image)

    gaussian_filter = cv.cuda.createGaussianFilter(srcType=src_type,
                                                    dstType=src_type,
                                                    ksize=ksize,
                                                    sigma1=sigma)

    gpu_blurred = gaussian_filter.apply(gpu_img)
    blurred = gpu_blurred.download()
    return blurred

def expand_bbox_fixed(x, y, w, h, margin, image_shape):
    """
    Funkcja pomocnicza do rozszerzenia prostokąta o jakiś margin.
    Pomocne dla zdefiniowania maski i robienia homografii -
    jeżeli np. chcemy usunąć z ROI (region of interest) narożniki jednej szachownicy,
    a znaleźć zamiast tego drugą
    """
    x_new = max(x - margin, 0)
    y_new = max(y - margin, 0)
    x2 = min(x + w + margin, image_shape[1])
    y2 = min(y + h + margin, image_shape[0])
    return x_new, y_new, x2 - x_new, y2 - y_new

def apply_mask_to_image(img, bbox):
    """
    Nałożyć maskę aby tylko prostokąt był widoczny (bbox)
    """
    mask = np.zeros_like(cv.cvtColor(img, cv.COLOR_BGR2GRAY))
    x, y, w, h = bbox
    mask[y:y+h, x:x+w] = 255
    return mask

def mask_out_chessboard(img, bbox, margin=0):
    """
    Nakłada czarne piksele na szachownicę - albo dowolny bounding box.
    Zdefiniowane również opcjonalnie odstępem margin
    """
    x, y, w, h = bbox
    x, y, w, h = expand_bbox_fixed(x, y, w, h, margin, img.shape)
    out = img.copy()
    out[y:y+h, x:x+w] = 0
    return out

def compute_reprojection_error(H, src_points, dst_points):
    """
    Na podstawie homografii, docelowych i źródłowych punktów liczy
    błąd euklidesowy (średniokwadratowy) dla wyznaczonego przekształcenia.
    Pomocne przy wyłonieniu najlepiej dopasowanego wyprostowania.
    """
    # Project the source points using the homography
    projected_points = cv.perspectiveTransform(src_points, H)

    # Compute the L2 distance between projected and actual destination points
    errors = np.linalg.norm(projected_points - dst_points, axis=2)
    mean_error = np.mean(errors)

    return mean_error, errors

def chess_homography(img1, img2, pattern_size, margin=200):
    """
    Jedna z najważniejszych funkcji w tym pliku. Na podstawie znalezionej szachownicy na jednym obrazie
    """

```

oraz drugim pozwala wyznaczyć homografię. Wyposażona w zabezpieczenie przez znalezieniem innych szachownic po wykryciu, czyli

mocuje się na jednej już znalezionej.

Na wejście wchodzą dwa obrazy: źródłowy i docelowy; wymiary szachownicy; opcjonalnie odstęp dla zamaskowania obszaru wokół szachownicy.

Jest przeczyźnijeszym sposobem, ponieważ nie musimy znać wymiarów szachownicy w rzeczywistym układzie.

Ten sposób może być przydatny w przypadku symulatora, ponieważ łatwo jest zmienić rozdzielcość kamery, ale w rzeczywistym świecie taka zmiana spowodowałaby stratę precyzji.

"""

```
#gray1 = cv.cvtColor(img1, cv.COLOR_BGR2GRAY)
#gray2 = cv.cvtColor(img2, cv.COLOR_BGR2GRAY)

gray1 = cv.cvtColor(img1, cv.COLOR_RGB2GRAY)
gray2 = cv.cvtColor(img2, cv.COLOR_RGB2GRAY)
gray1 = gaussian_blur_cuda(gray1, (5, 5), 0)
gray2 = gaussian_blur_cuda(gray2, (5, 5), 0)

pattern_size_corrected = (pattern_size[0], pattern_size[1])

ret1, corners1 = cv.findChessboardCorners(gray1, pattern_size_corrected,
                                            cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_NORMALIZE_IMAGE)
ret2, corners2 = cv.findChessboardCorners(gray2, pattern_size_corrected,
                                            cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_NORMALIZE_IMAGE)

if not (ret1 and ret2):
    print("Nie udało się znaleźć narożników szachownicy na jednym z obrazów.")
    return None,None
# Oblicz bounding box i rozszerz
x1, y1, w1, h1 = cv.boundingRect(corners1)
x2, y2, w2, h2 = cv.boundingRect(corners2)

x1_crop, y1_crop, w1_crop, h1_crop = expand_bbox_fixed(x1, y1, w1, h1, margin, gray1.shape)
x2_crop, y2_crop, w2_crop, h2_crop = expand_bbox_fixed(x2, y2, w2, h2, margin, gray2.shape)

# Przytnij obrazy
gray1_crop = gray1[y1_crop:y1_crop+h1_crop, x1_crop:x1_crop+w1_crop]
gray2_crop = gray2[y2_crop:y2_crop+h2_crop, x2_crop:x2_crop+w2_crop]

# Znajdź narożniki na przyciętych
ret1, corners1 = cv.findChessboardCorners(gray1_crop, pattern_size)
ret2, corners2 = cv.findChessboardCorners(gray2_crop, pattern_size)

if not (ret1 and ret2):
    print("Nie znaleziono narożników po przycięciu")
    return None,None

# Dodaj przesunięcie do narożników (od przycięcia)
corners1 += np.array([[x1_crop, y1_crop]], dtype=np.float32)
corners2 += np.array([[x2_crop, y2_crop]], dtype=np.float32)

criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 100, 0.00001)
corners1 = cv.cornerSubPix(gray1, corners1, (21, 21), (-1, -1), criteria)
corners2 = cv.cornerSubPix(gray2, corners2, (21, 21), (-1, -1), criteria)

# Homografia
H, _ = cv.findHomography(corners1, corners2, cv.RANSAC, 2.0)

H = H.astype(np.float32)
h, w = img2.shape[:2]

warped_img = vis.warp_with_cuda(img1,H,"frrrr",h,w,stream=None,gpu=False)
print(H)

img1_draw = img1.copy()
img2_draw = img2.copy()
cv.drawChessboardCorners(img1_draw, pattern_size_corrected, corners1, ret1)
cv.drawChessboardCorners(img2_draw, pattern_size_corrected, corners2, ret2)

# Błąd reprojekcji
reproj_error, per_point_errors = compute_reprojection_error(H, corners1, corners2)
print(f"Sredni blad reprojekcji: {reproj_error:.3f}")
# Pokaz obrazów
cv.namedWindow("Image 1 - detected corners", cv.WINDOW_NORMAL)
cv.imshow("Image 1 - detected corners", img1_draw)
cv.imwrite("img111.png",img1_draw)
cv.namedWindow("Image 2 - detected corners", cv.WINDOW_NORMAL)
cv.imshow("Image 2 - detected corners", img2_draw)
cv.imwrite("img222.png",img2_draw)
cv.namedWindow("Warped image 1 to image 2", cv.WINDOW_NORMAL)
cv.imshow("Warped image 1 to image 2", cv.cvtColor(warped_img, cv.COLOR_BGR2RGB))
cv.imwrite("img333.png",cv.cvtColor(warped_img, cv.COLOR_BGR2RGB))
```

```

return H, warped_img

def chess_homography_multiple_boards(img1, img2, pattern_size, second_patt_size, margin=200):
    """
    Druga funkcja pozwalająca de-fakto na wyznaczenie homografii na podstawie dwóch szachownic.
    Jedną musi znaleźć, zamaskować ją, a później na zamaskowanym obrazie znaleźć drugą.
    Ilość punktów do dopasowania może się powiększyć nawet więcej niż 2 razy.
    Najlepiej wykorzystywać różne szachownice w różnych obszarach obu obrazów.

    """
    gray1 = cv.cvtColor(img1, cv.COLOR_BGR2GRAY)
    gray2 = cv.cvtColor(img2, cv.COLOR_BGR2GRAY)
    #gray1 = gaussian_blur_cuda(gray1, (5, 5), 0)
    #gray2 = gaussian_blur_cuda(gray2, (5, 5), 0)

    pattern_size_corrected = (pattern_size[0], pattern_size[1])
    first = True
    ret1, corners1 = cv.findChessboardCorners(gray1, pattern_size_corrected,
                                              cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_NORMALIZE_IMAGE)
    ret2, corners2 = cv.findChessboardCorners(gray2, pattern_size_corrected,
                                              cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_NORMALIZE_IMAGE)

    if not (ret1 and ret2):
        print("Nie udało się znaleźć narożników szachownicy na jednym z obrazów.")
        return None, None

    criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 100, 0.00001)
    corners1 = cv.cornerSubPix(gray1, corners1, (21, 21), (-1, -1), criteria)
    corners2 = cv.cornerSubPix(gray2, corners2, (21, 21), (-1, -1), criteria)
    # Przytnij obraz i zamaskuj
    x1, y1, w1, h1 = cv.boundingRect(corners1)
    x2, y2, w2, h2 = cv.boundingRect(corners2)

    x1_crop, y1_crop, w1_crop, h1_crop = expand_bbox_fixed(x1, y1, w1, h1, margin, gray1.shape)
    x2_crop, y2_crop, w2_crop, h2_crop = expand_bbox_fixed(x2, y2, w2, h2, margin, gray2.shape)

    img1_masked = mask_out_chessboard(gray1.copy(), (x1_crop, y1_crop, w1_crop, h1_crop), margin=100)
    img2_masked = mask_out_chessboard(gray2.copy(), (x2_crop, y2_crop, w2_crop, h2_crop), margin=100)
    pattern_size_corrected = second_patt_size

    ret1_, corners1_ = cv.findChessboardCorners(img1_masked, pattern_size_corrected,
                                                cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_NORMALIZE_IMAGE)
    ret2_, corners2_ = cv.findChessboardCorners(img2_masked, pattern_size_corrected,
                                                cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_NORMALIZE_IMAGE)

    corners1_combined = []
    corners2_combined = []
    # Jeśli druga szachownica się różni - połącz dane
    # po subpix i boundingRect/ maskowaniu...
    if ret1_ and ret2_ and not np.array_equal(corners1_, corners1_) and not np.array_equal(corners2_, corners2_):
        corners1_combined = np.vstack((corners1_, corners1_))
        corners2_combined = np.vstack((corners2_, corners2_))
        corners1_combined = cv.cornerSubPix(gray1, corners1_combined, (21, 21), (-1, -1), criteria)
        corners2_combined = cv.cornerSubPix(gray2, corners2_combined, (21, 21), (-1, -1), criteria)
    else:
        # jeśli nie ma drugiej - użyj tylko pierwszej
        print("Nie znaleziono drugiej szachownicy")
        corners1_combined = corners1_.copy()
        corners2_combined = corners2_.copy()

    # Homografia
    H, _ = cv.findHomography(
        corners1_combined,
        corners2_combined,
        method=cv.RANSAC,
        ransacReprojThreshold=6.0,
    )

    H = H.astype(np.float32)
    h, w = img2.shape[:2]

    warped_img = vis.warp_with_cuda(img1, H, "frrrr", h, w, stream=None, gpu=False)
    print(H)

    img1_draw = img1.copy()
    img2_draw = img2.copy()
    # Narysuj pierwszy zestaw
    cv.drawChessboardCorners(img1_draw, pattern_size, corners1[:pattern_size[0]*pattern_size[1]], ret1)
    cv.drawChessboardCorners(img2_draw, pattern_size, corners2[:pattern_size[0]*pattern_size[1]], ret2)

    # Narysuj drugi zestaw jeśli jest

```

```

if ret1_ and ret2_:
    cv.drawChessboardCorners(img1_draw, pattern_size_corrected, corners1_[...], ret1_)
    cv.drawChessboardCorners(img2_draw, pattern_size_corrected, corners2_[...], ret2_)

# Błąd reprojekcji
reproj_error, per_point_errors = compute_reprojection_error(H, corners1, corners2)
print(f"Sredni błąd reprojekcji: {reproj_error:.3f}")
# Pokaz obrazów
cv.namedWindow("Image 1 - detected corners", cv.WINDOW_NORMAL)
cv.imshow("Image 1 - detected corners", img1_draw)

cv.namedWindow("Image 2 - detected corners", cv.WINDOW_NORMAL)
cv.imshow("Image 2 - detected corners", img2_draw)

cv.namedWindow("Warped image 1 to image 2", cv.WINDOW_NORMAL)
cv.imshow("Warped image 1 to image 2", warped_img)

return H, warped_img

def save_homo(homography, homography_filename):
    """
    Zapis homografii do pliku. Będzie zapisane w tym samym folderze, co i plik kontrolera
    (najlepiej jeżeli wszystkie pliki .py są w tym samym folderze).
    """
    np.save(homography_filename, homography)
    print(f"Matrix saved as {homography_filename}")

def solve_chess_size(image, name, pattern_size, pattern_size2, second=False):
    """
    """

    #h,w = shape #rozmiar docelowego obrazu
    #warped = warp_with_cuda(image, homography, name, h, w, pattern_size)
    #w_m,h_m = chess_real_size
    gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    ret, corners = cv.findChessboardCorners(gray, pattern_size,
                                             cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_NORMALIZE_IMAGE)
    criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 100, 0.0001)
    scale_x = None
    scale_y = None
    if ret:
        corners_new = cv.cornerSubPix(gray, corners, (30, 30), (-1, -1), criteria)
        #print(corners)

        img_draw = cv.drawChessboardCorners(image, pattern_size, corners_new, ret)

        # Pokaz obrazów

    """
    avg_w, avg_h, w, h = average_chessboard_size(corners_new, pattern_size)

    print(f"Szerokość klatki: {avg_w}")
    print(f"Wysokość klatki: {avg_h}")
    print(f"Szerokość wysokość szachownicy w pikselach: {(w,h)}")

    scale_x = w_m/w
    scale_y = h_m/h
    print(f"Jak w,y przeskalać obraz: {((scale_x,scale_y))}")
    """
    cols, rows = pattern_size

    top_left = corners_new[0]
    top_right = corners_new[cols - 1]
    bottom_left = corners_new[(rows - 1) * cols]
    bottom_right = corners_new[rows * cols - 1]

    corners_4 = np.array([
        top_left,
        top_right,
        bottom_right,
        bottom_left
    ], dtype=np.float32)

    if second:
        x1, y1, w1, h1 = cv.boundingRect(corners_4)
        x1_crop, y1_crop, w1_crop, h1_crop = expand_bbox_fixed(x1, y1, w1, h1, 200, gray.shape)
        img_masked = mask_out_chessboard(gray.copy(), (x1_crop, y1_crop, w1_crop, h1_crop), margin=100)
        pattern_size_corrected = pattern_size2 #TUTAJ ZMIENIAMY
        # Spróbuj znaleźć kolejną szachownicę

```

```

ret_, corners_ = cv.findChessboardCorners(img_masked, pattern_size_corrected,
                                          cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_NORMALIZE_IMAGE)

corners_combined = []
if ret_ and (not np.array_equal(corners_new, corners_)):
    corners_ = cv.cornerSubPix(img_masked, corners_, (21, 21), (-1, -1), criteria)
    cols, rows = pattern_size_corrected

    top_left = corners_[0]
    top_right = corners_[cols - 1]
    bottom_left = corners_[(rows - 1) * cols]
    bottom_right = corners_[rows * cols - 1]

    corners_4_ = np.array([
        top_left,
        top_right,
        bottom_right,
        bottom_left
    ], dtype=np.float32)
    corners_combined = np.concatenate((corners_4_, corners_4_), axis=0)
    img2_draw = cv.drawChessboardCorners(img_draw, pattern_size_corrected, corners_, ret_)
    top_left_min_ = min(corners_4_, key=lambda x: (x[0][0], x[0][1])) # Pierwszy element to współrzędne
x,y
    cv.circle(img2_draw, (int(top_left_min_[0][0]), int(top_left_min_[0][1])), 20, (0, 0, 255), -1)
    cv.namedWindow(f"detected corners {name}", cv.WINDOW_NORMAL)
    cv.imshow(f"detected corners {name}", img2_draw)

    return corners_combined
else:
    print(f"Nie znaleziono drugiej szachownicy lub jest identyczna {name}")
    return corners_4_
cv.namedWindow(f"detected corners {name}", cv.WINDOW_NORMAL)
cv.imshow(f"detected corners {name}", img_draw)
return corners_4

else:
    print(f"Nie znaleziono szachownicy {name}")
    return None

```

Plik visualise.py

Dany plik zawiera funkcje wizualizacyjne do widoku „z lotu ptaka”, przekształcenia obrazów za pomocą CUDA OpenCV, wizualizacji czujników ultradźwiękowych.

```
"""
plik z funkcjami pomocniczymi wizualizacyjnymi
"""

import numpy as np
from scipy.interpolate import splprep, splev
from scipy.ndimage import uniform_filter1d
import matplotlib.pyplot as plt
import cv2 as cv
import camera_calibration as cc
from scipy.linalg import logm, expm, sqrtm

# Vehicle parameters
TRACK_FRONT = 1.628
TRACK_REAR = 1.628
WHEELBASE = 2.995
MAX_WHEEL_ANGLE = 0.5 # rad
CAR_WIDTH = 1.95
CAR_LENGTH = 4.85

global s
s = 2 # skala, jeżeli obrazy były wykonane przy innej rozdzielcości

def draw_cones(ax,fig,distances):
    """
    Do wizualizacji strefy detekcji czujników ultradźwiękowych.
    wywoływana za każdym razem w pętli kontrolera - rysuje na pustej kanwie
    matplotlib'a i
    """
    ax.clear()

    front_sensor_angles = [90,45,15,-15,-45,-90]
    rear_sensor_angles = [90,135,180,180,-135,-90]

    sensor_positions = np.array([
        [ 3.515873,  0.865199], # front left side
        [ 3.588074,  0.81069 ], # front left
        [ 3.799743,  0.375011], # front lefter
        [ 3.799743, -0.375011], # front righter
        [ 3.588074, -0.81069 ], # front right
        [ 3.515873, -0.865199], # front right side
        [-0.505871,  0.923198], # left side
        [-0.845978,  0.798194], # left
        [-0.929999,  0.32    ], # lefter
        [-0.930001, -0.32    ], # righter
        [-0.840982, -0.789534], # right
        [-0.505875, -0.9232  ] # right side
    ])

    sensor_angles = front_sensor_angles + rear_sensor_angles
    sensor_fovs_front = [45]*6
    sensor_fovs_rear = [45]*6
    sensor_fovs = sensor_fovs_front + sensor_fovs_rear
    # obrót do osi współrzędnych
    rot = np.array([[0,-1],[1,0]])
    sensor_positions_rot = sensor_positions @ rot.T
    sensor_angles_rot = np.array(sensor_angles) + 90

    for pos, angle_deg, fov_deg, dist in zip(sensor_positions_rot, sensor_angles_rot, sensor_fovs, distances):
        angle_rad = np.radians(angle_deg)
        fov_rad = np.radians(fov_deg)

        theta = np.linspace(angle_rad - fov_rad/2, angle_rad + fov_rad/2, 30)
        x = pos[0] + dist * np.cos(theta)
        y = pos[1] + dist * np.sin(theta)

        ax.fill(np.append(pos[0], x), np.append(pos[1], y), color='lightblue', alpha=0.3)
        ax.plot([pos[0], x[0]], [pos[1], y[0]], 'b:', alpha=0.5)
        ax.plot([pos[0], x[-1]], [pos[1], y[-1]], 'b:', alpha=0.5)
        ax.plot(pos[0], pos[1], 'ko')

    ax.add_patch(plt.Rectangle(
```

```

        (-CAR_WIDTH/2, -1), CAR_WIDTH, CAR_LENGTH,
        edgecolor='gray', facecolor='lightgray', alpha=0.8
    )))
    ax.set_xlim(-8, 10)
    ax.set_ylim(-10, 10)
    ax.set_aspect('equal')
    ax.set_title("Czujniki ultradźwiękowe")
    ax.grid(True)
    #fig.canvas.draw()
    #fig.canvas.flush_events()

def collect_homo(names_images,homographies,carstreams):
    """
    DRUGI SPOSÓB
    """

    h, w = int(3600/s),int(3600/s)

    (stream1,stream2,stream3,stream4,
    stream5,stream6,stream7,stream8,
    stream9,stream10,stream11,stream12,stream13) = streams
    (front_H,right_H,right_fender_H,
    rear_H, left_fender_H, left_H)= homographies
    imgs = []
    # warp CUDA wszystkie kamery
    left = warp_with_cuda(names_images["camera_left_pillar"], left_H, "left homo", h, w,stream1)
    right = warp_with_cuda(names_images["camera_right_pillar"], right_H, "right homo", h, w,stream2)
    rear = warp_with_cuda(names_images["camera_rear"], rear_H, "rear homo", h, w,stream3)
    front = warp_with_cuda(names_images["camera_front_bumper_wide"], front_H, "front homo", h, w,stream4)
    # front_wind = warp_with_cuda(names_images["camera_front_top"], front_wind_H, "front wind homo", h, w)
    right_fender = warp_with_cuda(names_images["camera_right_fender"], right_fender_H, "right fender homo", h,
w,stream5)
    left_fender = warp_with_cuda(names_images["camera_left_fender"], left_fender_H, "left fender homo", h, w,stream6)
    #imgs.extend([left,right,rear,front,
    S = np.array([[1/s,0,0],[0,1/s,0],[0,0,1]]).astype(np.float32)

    right_to_front_H = np.array([[ 1.0103254e+00,  8.9369901e-03,  2.4237607e+03],
[-4.8033521e-03, -1.0200684e+00 , 1.8881282e+03],
[-1.9079014e-06,  2.8290551e-06,  1.0000000e+00]]).astype(np.float32)
    right_to_front_H = S @ right_to_front_H @ np.linalg.inv(S)

    left_to_front_H = np.array([[ 1.0053335e+00 , -6.1842590e-04, -2.4580974e+03],
[-5.9892777e-03,  1.0021796e+00 , 1.8966011e+03],
[-2.172203e-06, -4.8802093e-07,  1.0000000e+00]]).astype(np.float32)
    left_to_front_H = S @ left_to_front_H @ np.linalg.inv(S)

    left_to_rear_H = np.array([[ 9.7381920e-01, -2.3735706e-03, -2.2782476e+03],
[ 4.3283560e-04,  9.7134042e-01, -1.7275361e+03],
[-3.2049848e-06,  5.5537777e-07,  1.0000000e+00]]).astype(np.float32)

    right_to_rear_H = np.array([[ 9.8992777e-01,  1.6606528e-02,  4.6191958e+03],
[-2.1194941e-03 , 9.9684310e-01, -1.7071634e+01],
[-7.0434027e-07,  3.0691269e-06 , 1.0000000e+00]]).astype(np.float32)

    left_to_rear_H = S @ left_to_rear_H @ np.linalg.inv(S)
    right_to_rear_H = S @ right_to_rear_H @ np.linalg.inv(S)

    rear_to_front_H = np.array([[ 1.1127317e+00,  7.0112962e-03 ,-8.0307449e+01],
[ 5.2037694e-02,  1.0801761e+00 , 4.4567339e+03],
[ 1.0615680e-05,  6.2511299e-07,  1.0000000e+00]]).astype(np.float32)
    rear_to_front_H = S @ rear_to_front_H @ np.linalg.inv(S)

    canvas_front = blend_warp_GPUONLY(front,right,right_to_front_H,stream7)
    canvas_front = blend_warp_GPUONLY(canvas_front,left,left_to_front_H,stream8)
    canvas_rear = blend_warp_GPUONLY(rear, left_fender, left_to_rear_H,stream9)
    canvas_rear = blend_warp_GPUONLY(canvas_rear,right_fender,right_to_rear_H,stream10)
    canvas = blend_warp_GPUONLY(canvas_front,canvas_rear,rear_to_front_H,stream11)

    canvas = canvas.download()

    cv.namedWindow("bev",cv.WINDOW_NORMAL)
    cv.imshow("bev",canvas)
    #cv.imwrite("img_vis.jpg",cv.cvtColor(canvas,cv.COLOR_BGR2RGB))
    """
    crop_scale = 0.5
    margin = 0

```

```

ch,cw = canvas.shape[0], canvas.shape[1]
crop_h = int(ch * crop_scale / 2)
crop_w = int(cw * crop_scale / 2)

center_h, center_w = ch // 2, cw // 2

y1 = max(center_h - crop_h - margin, 0)
y2 = min(center_h + crop_h + margin, h)
x1 = max(center_w - crop_w - margin, 0)
x2 = min(center_w + crop_w + margin, w)

cropped = canvas[y1:y2, x1:x2]

"""
#DZIAŁA TYLKO DLA FULLHD OBRAZÓW, S=2
cropped = canvas
h, w = cropped.shape[:2]

crop_top_px    = int(0.184 * h)
crop_bottom_px = int(0.215 * h)
crop_left_px   = int(0.14 * w)
crop_right_px  = int(0.14 * w)

y1 = crop_top_px
y2 = h - crop_bottom_px
x1 = crop_left_px
x2 = w - crop_right_px
cropped = cropped[y1:y2, x1:x2]

scalex = 0.18 # np. 20% szerokości BEV
scaley = 0.45
bev_h, bev_w = cropped.shape[:2]
new_w = int(bev_w * scalex)
new_h = int(bev_h * scaley)
car_resized = cv.resize(car, (new_w, new_h), interpolation=cv.INTER_AREA)

# 3. Oblicz pozycję tak, by wstawić go na środku
x_offset = (bev_w - new_w) // 2 + 15
y_offset = (bev_h - new_h) // 2 - 15

cropped[y_offset:y_offset+new_h, x_offset:x_offset+new_w] = car_resized

cv.namedWindow("bev",cv.WINDOW_NORMAL)
cv.imshow("bev",cropped)
cv.imwrite("img_vis.png",cropped)
#H,_ = cc.chess_homography(canvas,names_images["camera_helper"],(10,6))

return canvas

def alt_collect_homo(names_images,homographies,carstreams):
    """
    TRZECI SPOSOB
    """
    h, w = int(3600/s),int(3600/s)

    (stream1,stream2,stream3,stream4,
    stream5,stream6,stream7,stream8,
    stream9,stream10,stream11,stream12,stream13) = streams
    (front_H,right_H,right_fender_H,
    rear_H,left_fender_H,left_H)= homographies
    imgs = []
    # warp CUDA wszystkie kamery
    left = warp_with_cuda(names_images["camera_left_pillar"], left_H, "left homo", h, w,stream1)
    right = warp_with_cuda(names_images["camera_right_pillar"], right_H, "right homo", h, w,stream2)
    rear = warp_with_cuda(names_images["camera_rear"], rear_H, "rear homo", h, w,stream3)
    front = warp_with_cuda(names_images["camera_front_bumper_wide"], front_H, "front homo", h, w,stream4)
    # front_wind = warp_with_cuda(names_images["camera_front_top"], front_wind_H, "front wind homo", h, w)
    right_fender = warp_with_cuda(names_images["camera_right_fender"], right_fender_H, "right fender homo", h, w,stream5)
    left_fender = warp_with_cuda(names_images["camera_left_fender"], left_fender_H, "left fender homo", h, w,stream6)

    #CZĘŚĆ KODU DO SPORZĄDZENIA HOMOGRAFII RZUTOWANIA NA KANWE

    canvas_size = (6000,6000)
    canvas_cpu = np.zeros((canvas_size[1], canvas_size[0], 3), dtype=np.uint8)
    canvas = cv.cuda_GpuMat()
    canvas.upload(canvas_cpu,stream7)

    px = np.array([[0,0],[6000/s,0],[6000/s,6000/s],[0,6000/s]]).astype(np.float32)
    met = np.array([[10,10],[10,-10],[-10,-10],[-10,10]],dtype = np.float32) #

    H_px_to_m_bevev,_ = cv.findHomography(met,px, cv.RANSAC,5.0)

```

```

S = np.array([[1/s,0,0],[0,1/s,0],[0,0,1]]).astype(np.float32)
H_left_to_bev= np.array([[5.78196165e-01, 1.36768422e-03 ,5.75901552e+02],
[2.08805960e-03, 5.78244815e-01, 1.31412439e+03],
[9.24389963e-07 ,7.19948103e-07, 1.00000000e+00]],dtype=np.float32)
H_left_to_bev = S @ H_left_to_bev @ np.linalg.inv(S)

H_right_to_bev = np.array([[ 5.74277218e-01, -8.39567193e-05, 3.34911956e+03],
[ 3.93911249e-06, 5.73836613e-01, 1.31777314e+03],
[ 1.31049336e-07 ,-1.42342160e-07, 1.00000000e+00]],dtype=np.float32)
H_right_to_bev = S @ H_right_to_bev @ np.linalg.inv(S)

H_left_fender_to_bev = np.array([[ 5.60635651e-01, 3.13768463e-02, 6.29376393e+02],
[-1.14353803e-02, 6.03217079e-01, 2.80548010e+03],
[-4.06125598e-06, 1.18455527e-05, 1.00000000e+00]],dtype=np.float32)
H_corr_lf = np.array(
[[ 9.95078761e-01, -5.70508469e-04, 5.23265217e+00],
[-3.64430112e-03, 9.96849037e-01, 7.87846839e+00],
[-1.0446326e-06,-2.72402103e-07, 1.00000000e+00]]).astype(np.float32)
H_left_fender_to_bev = H_corr_lf @ H_left_fender_to_bev
H_left_fender_to_bev = S @ H_left_fender_to_bev @ np.linalg.inv(S)

H_right_fender_to_bev = np.array([[ 5.68654217e-01, 3.69548635e-02, 3.30027369e+03],
[-5.18751953e-03, 6.04909446e-01 ,2.80305923e+03],
[-2.20235231e-06, 1.11229040e-05 ,1.00000000e+00]],dtype=np.float32)
H_corr_rf = np.array(
[[ 9.86315188e-01, -3.03481721e-04, 2.78378345e+01],
[-5.49422481e-03, 9.93028602e-01 ,2.23023238e+01],
[-1.63834578e-06, -8.44384115e-08 ,1.00000000e+00]]).astype(np.float32)
H_right_fender_to_bev = H_corr_rf @ H_right_fender_to_bev
H_right_fender_to_bev = S @ H_right_fender_to_bev @ np.linalg.inv(S)

H_rear_to_bev = np.array([[5.72595558e-01, 2.77600165e-02 ,1.96983903e+03],
[2.49365825e-04, 6.06721803e-01, 3.77306058e+03],
[6.15294083e-08, 9.20599720e-06, 1.00000000e+00]],dtype=np.float32)
H_corr_rear = np.array([[ 9.86009607e-01, -5.12444388e-05, 4.22106295e+01],
[ 6.09014276e-05, 9.87099849e-01 ,4.70971490e+01],
[ 1.79688786e-08, -1.22849686e-08 ,1.00000000e+00]]).astype(np.float32)
H_rear_to_bev = H_corr_rear @ H_rear_to_bev
H_rear_to_bev = S @ H_rear_to_bev @ np.linalg.inv(S)

H_front_to_bev = np.array([[5.78772185e-01, 3.67819798e-03, 1.95954679e+03],
[2.99076766e-04, 5.83709774e-01, 2.15103801e+02],
[1.77376103e-07 ,1.22892995e-06, 1.00000000e+00]],dtype = np.float32)
H_front_to_bev = S @ H_front_to_bev @ np.linalg.inv(S)

bev_left = blend_warp_GPUONLY(canvas, left,H_left_to_bev,stream7,canvas_size=(6000//s,6000//s))
bev_right = blend_warp_GPUONLY(bev_left,right,H_right_to_bev,stream8,canvas_size=(6000//s,6000//s)) =
bev_left_fender
blend_warp_GPUONLY(bev_right, left_fender,H_left_fender_to_bev,stream9,canvas_size=(6000//s,6000//s)) =
bev_right_fender
blend_warp_GPUONLY(bev_left_fender,right_fender,H_right_fender_to_bev,stream10,canvas_size=(6000//s,6000//s))
bev_rear = blend_warp_GPUONLY(bev_right_fender,rear,H_rear_to_bev,stream11,canvas_size=(6000//s,6000//s))
bev_front = blend_warp_GPUONLY(bev_rear,front,H_front_to_bev,stream12,canvas_size=(6000//s,6000//s)) =

bev = bev_front.download()

# Granice w metrach (obszar 8x6 metrów -> prostokąt)
meters = np.array([
[ 5.6, 6],
[ 5.6, -6],
[-5.95, -6],
[-5.95, 6]
], dtype=np.float32)

# Dodaj trzecią współrzędną (homogeniczne)
meters_hom = np.hstack([meters, np.ones((meters.shape[0], 1))])

# Przekształć na piksele
pxs = (H_px_to_m_bev @ meters_hom.T).T

# Normalizuj (dziel przez ostatni element, jeśli nie równe 1)
#pxs /= pxs[:, [2]]

# Rzutuj na int (piksele)
pxs_int = pxs[:, :2].astype(int)

# Ustal minimalne i maksymalne współrzędne pikseli (x_min, x_max, y_min, y_max)
x_min = np.min(pxs_int[:, 0])
x_max = np.max(pxs_int[:, 0])
y_min = np.min(pxs_int[:, 1])
y_max = np.max(pxs_int[:, 1])

```

```

# UWAGA: Sprawdzenie czy zakres jest w granicach obrazu
x_min = max(x_min, 0)
y_min = max(y_min, 0)
x_max = min(x_max, canvas_cpu.shape[1])
y_max = min(y_max, canvas_cpu.shape[0])

# Obcinanie obrazu (ROI -> Region of Interest)
cropped = bev[y_min:y_max, x_min:x_max]

# ---- 1) Oblicz skalę px/m na podstawie homografii metr>piksel ----
# 1) Dwa punkty homogeniczne w metrach dla osi X
e_x = np.array([[0,0,1],
                [1,0,1]], dtype=np.float32)
# 2) Dwa punkty homogeniczne w metrach dla osi Y
e_y = np.array([[0,0,1],
                [0,1,1]], dtype=np.float32)

# Rzut tych punktów na piksele:
p_x = (H_px_to_m_bev @ e_x.T).T
p_x /= p_x[:, [2]]
p_y = (H_px_to_m_bev @ e_y.T).T
p_y /= p_y[:, [2]]

# Skale:
scale_x = np.linalg.norm(p_x[1,:2] - p_x[0,:2]) # px na 1 m w osi X
scale_y = np.linalg.norm(p_y[1,:2] - p_y[0,:2]) # px na 1 m w osi Y

# ---- 2) Oblicz rozmiar car w pikselach ----
W_real = 1.95 # szerokość samochodu w metrach
L_real = 4.95 # długość samochodu w metrach
w_car_px = int(round(W_real * scale_x))
h_car_px = int(round(L_real * scale_y))

# 2) Inny rozmiar
car_resized = cv.resize(car, (w_car_px, h_car_px), interpolation=cv.INTER_AREA)

# 3) Wylicz środek ROI jako środek samego cropped
Hc, Wc = cropped.shape[:2]
cx_px = Wc // 2
cy_px = Hc // 2

# 5) Przesunąć jeżeli trzeba
d_forward = 0.1 # np. 0.5 m do przodu
pix_offset = int(round(d_forward * scale_y))
# w obrazie „do przodu” = w górę, czyli y maleje:
cy_px -= pix_offset

# 6) Oblicz ROI w pikselach:
x0 = cx_px - w_car_px//2
y0 = cy_px - h_car_px//2
x1 = x0 + w_car_px
y1 = y0 + h_car_px

# 7) Przytnij do granic i wklej:
x0c, y0c = max(x0,0), max(y0,0)
x1c, y1c = min(x1,Wc), min(y1,Hc)
sx0, sy0 = x0c - x0, y0c - y0
sx1, sy1 = sx0 + (x1c-x0c), sy0 + (y1c-y0c)

cropped[y0c:y1c, x0c:x1c] = car_resized[sy0:sy1, sx0:sx1]

cv.namedWindow("bev",cv.WINDOW_NORMAL)
cv.imshow("bev",cropped)
return cropped

def chain_collect_homo(names_images,homographies,car,streams):
    h, w = int(3600/s),int(3600/s)

    (stream1,stream2,stream3,stream4,
     stream5,stream6,stream7,stream8,
     stream9,stream10,stream11,stream12,stream13) = streams
    (front_H,right_H,right_fender_H,
     rear_H, left_fender_H, left_H)= homographies
    imgs = []
    # warp CUDA wszystkie kamery
    left = warp_with_cuda(names_images["camera_left_pillar"], left_H, "left homo", h, w,stream1)
    right = warp_with_cuda(names_images["camera_right_pillar"], right_H, "right homo", h, w,stream2)
    rear = warp_with_cuda(names_images["camera_rear"], rear_H, "rear homo", h, w,stream3,show=True)
    front = warp_with_cuda(names_images["camera_front_bumper_wide"], front_H, "front homo", h, w,stream4)
    # front_wind = warp_with_cuda(names_images["camera_front_top"], front_wind_H, "front wind homo", h, w)

```

```

    right_fender = warp_with_cuda(names_images["camera_right_fender"], right_fender_H, "right fender homo", h,
w,stream5)
    left_fender = warp_with_cuda(names_images["camera_left_fender"], left_fender_H, "left fender homo", h, w,stream6)

#cv.imwrite("lewa_kolumna_6.jpg",left)
#cv.imwrite("lewy_blotnik_6.jpg",left_fender)

#H1 - lewa do frontalnej,
#H2 - H1 do prawej
#H3 - H2 do prawego blotnika
#H4 - tylna zarowno do prawego, jak i lewego blotnika (dołączono )
#H5 - lewy blotnik ostatni,
S = np.array([[1/s,0,0],[0,1/s,0],[0,0,1]]).astype(np.float32)

H1 = np.array([[ 1.013,  0.0036699,       -2443],
[ 0.008471,   1.0221,      1880.2],
[ 3.1404e-06,  5.2511e-06,           1]]).astype(np.float32)
H1 = S @ H1 @ np.linalg.inv(S)
H2 = np.array([[ 0.99331,  0.024914 ,     4847.5],
[-0.0033566,  1.0098,      1888],
[-1.2409e-06,  4.6596e-06,           1]]).astype(np.float32)
H2 = S @ H2 @ np.linalg.inv(S)
H3 = np.array([[ 0.99215,  0.027157,     4769.2],
[-0.0035934,  1.0118 ,     4438.5],
[-1.3227e-06,  5.0418e-06,           1]]).astype(np.float32)
H3 = S @ H3 @ np.linalg.inv(S)
H4 = np.array([[ 1.0145,  0.034837,     2429.5],
[ 0.01286,   1.038,      6119.2],
[ 1.771e-06,  7.0018e-06,           1]]).astype(np.float32)
H4 = S @ H4 @ np.linalg.inv(S)
H5 = np.array([[1.0146343e+00, 1.3305261e-02, 6.5544266e+01],
[1.1184645e-02, 1.0057985e+00, 4.4326597e+03],
[1.5989363e-06 ,4.4307935e-06, 1.0000000e+00]]).astype(np.float32)
H5 = S @ H5 @ np.linalg.inv(S)
"""

H6 = np.array([[ 1.0074713e+00, 1.0635464e-03, 3.0394157e+01],
[-2.9231559e-03, 1.0117992e+00 , 2.5832144e+03],
[-7.9523011e-07, 3.8797717e-07, 1.0000000e+00]]).astype(np.float32)
H6 = S @ H6 @ np.linalg.inv(S)
"""

canvas = blend_warp_GPUONLY(front, left, H1, stream7)
canvas = blend_warp_GPUONLY(canvas, right, H2, stream8)
canvas = blend_warp_GPUONLY(canvas, right_fender, H3, stream9)
canvas = blend_warp_GPUONLY(canvas, rear, H4, stream10)
canvas = blend_warp_GPUONLY(canvas, left_fender, H5, stream11)

canvas = canvas.download()

cropped = canvas
h, w = cropped.shape[:2]

crop_top_px = int(0.1965 * h)
crop_bottom_px = int(0.177 * h)
crop_left_px = int(0.11 * w)
crop_right_px = int(0.11 * w)

y1 = crop_top_px
y2 = h - crop_bottom_px
x1 = crop_left_px
x2 = w - crop_right_px
cropped = cropped[y1:y2, x1:x2]

scalex = 0.18 # np. 20% szerokości BEV
scaley = 0.45
bev_h, bev_w = cropped.shape[:2]
new_w = int(bev_w * scalex)
new_h = int(bev_h * scaley)
car_resized = cv.resize(car, (new_w, new_h), interpolation=cv.INTER_AREA)

# 3. Oblicz pozycję tak, by wstawić go na środku
x_offset = (bev_w - new_w) // 2 + 5
y_offset = (bev_h - new_h) // 2 - 19

cropped[y_offset:y_offset+new_h, x_offset:x_offset+new_w] = car_resized

cv.namedWindow("bev",cv.WINDOW_NORMAL)
cv.imshow("bev",cropped)

return cropped

def warp_with_cuda(image, H, name,h,w,stream, gpu=True, show=False,first_time=True):
"""

```

Funkcja pozwalająca na wyprostowanie obrazu za pomocą homografii H. Można zarówno jak i wykorzystywać ją do gładkiego przetwarzania na GPU, jak i z pobieraniem z powrotem do CPU. Również show pozwala pokazać w oddzielnym oknie obraz.

```

"""
if first_time:
    gpu_img = cv.cuda_GpuMat()
    gpu_img.upload(image, stream=stream)
else:
    gpu_img = image
H_corrected = np.array(np.zeros((3,3))).astype(np.float32)

if name == "left fender homo":
    #pierwsza w kolumnie translacji - x, druga liczba - y;
    translation = np.array([
        [1, 0, 24],
        [0, 1, 0],
        [0, 0, 1]
    ], dtype=np.float32)
    H_corrected = translation @ H
    #print(H_corrected)
else:
    H_corrected = H

warped_gpu = cv.cuda.warpPerspective(gpu_img, H_corrected, (w, h), stream=stream)
warped = cv.cuda.cvtColor(warped_gpu, cv.COLOR_BGR2RGB, stream=stream)

if not gpu:
    warped = warped.download()
else:
    warped = warped
if show:
    if gpu:
        warp = warped.download()
    else:
        warp=warped
    cv.namedWindow(name, cv.WINDOW_NORMAL)
    cv.imshow(name, warp)
#stream.waitForCompletion()
return warped

def warp_and_blend_gpu(img1, img2, H, canvas_size=None, alpha=0.8):
    """
    Fast GPU-only homography warp + simple blend:
    1) compute output canvas bounds & translation
    2) warp img1, img2 onto that canvas
    3) build binary masks on GPU via threshold
    4) do a weighted blend in the overlap, and bitwise OR outside

    img1, img2 : BGR uint8
    H           : float32 homography (3x3) mapping img2 → img1 frame
    canvas_size: (w,h) to force output size, or None to auto-compute
    alpha       : blend weight for img2 in the overlap region
    """
    # upload
    g1 = cv.cuda_GpuMat(); g1.upload(img1)
    g2 = cv.cuda_GpuMat(); g2.upload(img2)
    h1, w1 = img1.shape[:2]
    h2, w2 = img2.shape[:2]

    # compute translation & canvas size if needed
    if canvas_size is None:
        pts1 = np.float32([[0,0],[w1,0],[w1,h1],[0,h1]]).reshape(-1,1,2)
        pts2 = np.float32([[0,0],[w2,0],[w2,h2],[0,h2]]).reshape(-1,1,2)
        pts2t = cv.perspectiveTransform(pts2, H)
        all_pts = np.vstack([pts1, pts2t])
        x_min, y_min = np.int32(all_pts.min(axis=0).ravel() - 0.5)
        x_max, y_max = np.int32(all_pts.max(axis=0).ravel() + 0.5)
        trans = np.array([[1,0,-x_min],[0,1,-y_min],[0,0,1]], dtype=np.float32)
        out_w, out_h = x_max - x_min, y_max - y_min
    else:
        trans = np.eye(3, dtype=np.float32)
        out_w, out_h = canvas_size

    # warp both onto GPU canvas
    g1w = cv.cuda.warpPerspective(g1, trans, (out_w, out_h))
    g2w = cv.cuda.warpPerspective(g2, trans @ H, (out_w, out_h))

    # build masks by thresholding grayscale on GPU
    gray1 = cv.cuda.cvtColor(g1w, cv.COLOR_BGR2GRAY)
    gray2 = cv.cuda.cvtColor(g2w, cv.COLOR_BGR2GRAY)
    _, m1w = cv.cuda.threshold(gray1, 1, 255, cv.THRESH_BINARY)

```

```

_, m2w = cv.cuda.threshold(gray2, 1, 255, cv.THRESH_BINARY)

# overlap region mask
overlap = cv.cuda.bitwise_and(m1w, m2w)

# simple blend in overlap
blend = cv.cuda.addWeighted(g1w, 1-alpha, g2w, alpha, 0)

# non-overlap contributions
inv2 = cv.cuda.bitwise_not(m2w)
inv1 = cv.cuda.bitwise_not(m1w)
part1 = cv.cuda.bitwise_and(g1w, cv.cuda.cvtColor(inv2, cv.COLOR_GRAY2BGR))
part2 = cv.cuda.bitwise_and(g2w, cv.cuda.cvtColor(inv1, cv.COLOR_GRAY2BGR))
partB = cv.cuda.bitwise_and(blend, cv.cuda.cvtColor(overlap, cv.COLOR_GRAY2BGR))

tmp = cv.cuda.add(part1, part2)
out = cv.cuda.add(tmp, partB)

return out.download()

def blend_warp_GPUONLY(g1, g2, H, stream, canvas_size=None, alpha=0.85):
    """
    Szybkie mieszanie i prostowanie dwóch obrazów na GPU.
    Wykorzystuje
    Fast GPU-only homography warp + simple blend:
        1) compute output canvas bounds & translation
        2) warp img1, img2 onto that canvas
        3) build binary masks on GPU via threshold
        4) do a weighted blend in the overlap, and bitwise OR outside

    img1, img2 : BGR uint8
    H           : float32 homography (3x3) mapping img2 → img1 frame
    canvas_size: (w,h) to force output size, or None to auto-compute
    alpha       : blend weight for img2 in the overlap region
    """
    # upload

    w1, h1 = g1.size()
    w2, h2 = g2.size()

    # compute translation & canvas size if needed
    if canvas_size is None:
        pts1 = np.float32([[0,0],[w1,0],[w1,h1],[0,h1]]).reshape(-1,1,2)
        pts2 = np.float32([[0,0],[w2,0],[w2,h2],[0,h2]]).reshape(-1,1,2)
        pts2t = cv.perspectiveTransform(pts2, H)
        all_pts = np.vstack([pts1, pts2t])
        x_min, y_min = np.int32(all_pts.min(axis=0).ravel() - 0.5)
        x_max, y_max = np.int32(all_pts.max(axis=0).ravel() + 0.5)
        trans = np.array([[1,0,-x_min],[0,1,-y_min],[0,0,1]], dtype=np.float32)
        out_w, out_h = x_max - x_min, y_max - y_min
    else:
        trans = np.eye(3, dtype=np.float32)
        out_w, out_h = canvas_size

    # warp both onto GPU canvas
    g1w = cv.cuda.warpPerspective(g1, trans, (out_w, out_h), stream=stream)
    g2w = cv.cuda.warpPerspective(g2, trans @ H, (out_w, out_h), stream=stream)

    # build masks by thresholding grayscale on GPU
    gray1 = cv.cuda.cvtColor(g1w, cv.COLOR_BGR2GRAY, stream=stream)
    gray2 = cv.cuda.cvtColor(g2w, cv.COLOR_BGR2GRAY, stream=stream)
    _, m1w = cv.cuda.threshold(gray1, 1, 255, cv.THRESH_BINARY, stream=stream)
    _, m2w = cv.cuda.threshold(gray2, 1, 255, cv.THRESH_BINARY, stream=stream)

    # overlap region mask
    overlap = cv.cuda.bitwise_and(m1w, m2w, stream=stream)

    # simple blend in overlap
    blend = cv.cuda.addWeighted(g1w, 1-alpha, g2w, alpha, 0, stream=stream)

    # non-overlap contributions
    inv2 = cv.cuda.bitwise_not(m2w, stream=stream)
    inv1 = cv.cuda.bitwise_not(m1w, stream=stream)
    g1w = cv.cuda.bitwise_and(g1w, cv.cuda.cvtColor(inv2, cv.COLOR_GRAY2BGR), stream=stream)
    g2w = cv.cuda.bitwise_and(g2w, cv.cuda.cvtColor(inv1, cv.COLOR_GRAY2BGR), stream=stream)
    blend = cv.cuda.bitwise_and(blend, cv.cuda.cvtColor(overlap, cv.COLOR_GRAY2BGR), stream=stream)

    # sum them up
    out = cv.cuda.add(g1w, g2w, stream=stream)
    out = cv.cuda.add(out, blend, stream=stream)
    #stream.waitForCompletion()
    return out

```

Plik park_algo.py

W tym pliku są funkcje dotyczące parkowania równoległego.

```
import math
import numpy as np
import matplotlib.pyplot as plt
import time
import matplotlib.animation as animation

# Vehicle parameters
TRACK_FRONT = 1.628
TRACK_REAR = 1.628
WHEELBASE = 2.995
MAX_WHEEL_ANGLE = 0.5 # rad
CAR_WIDTH = 1.95
CAR_LENGTH = 4.85
MAX_SPEED = -3.0
class Parking:
    # Domyślne marginesy startu i końca manewru
    marg_start = 0.2
    mard_end = 0.4

    def __init__(self, driver, side, times,
                 min_width=CAR_WIDTH*1.1,
                 min_length=CAR_LENGTH*1.25,
                 threshold=1*CAR_WIDTH):
        # Inicjalizacja parametrów parkingu
        self.min_width = min_width      # minimalna szerokość miejsca
        self.min_length = min_length    # minimalna długość miejsca
        self.threshold = threshold      # próg zmiany odległości

        self.state = "searching_start" # aktualny stan maszyny stanów
        self.start_pose = None         # miejsce wykrycia początku luki
        self.spots = []                # wykryte miejsca parkingowe
        self.driver = driver           # interfejs do samochodu w Webots

        # Pomocnicze zmienne do detekcji zmian odległości
        self.prev_distance_front = 0.0
        self.prev_distance_rear = 0.0
        self.dist_start_far = 0.0
        self.dist_start_cl = 0.0
        self.dist_end_far = 0.0
        self.dist_end_cl = 0.0

        self.side = side      # strona parkowania: "left" lub "right"
        self.x = 0.0          # os x w układzie globalnym
        self.y = 0.0          # os y w układzie globalnym
        self.yaw = 0.0         # orientacja pojazdu
        self.last_time = times # czas ostatniej aktualizacji

        # Parametry regulatora PID dla sterowania
        self.Kp = 1.2
        self.Kd = 0.001
        self.Kl = 0.5
        self.prev_yaw_err = 0.0

    def update_odometry(self, yaw):
        """
        Aktualizuje pozycję (x,y) na podstawie prędkości i kąta yaw.
        """
        # Pobierz prędkość w km/h i przelicz na m/s
        v = self.driver.getCurrentSpeed() / 3.6

        # Integracja prostokątna z krokiem czasowym 0.05s
        self.yaw = yaw
        dx = v * math.cos(self.yaw) * 0.05
        dy = v * math.sin(self.yaw) * 0.05
        self.x += dx
        self.y += dy

        return (self.x, self.y, self.yaw)

    def update_state(self, dists_names, yaw):
        """
        Maszyna stanów wykrywająca luki parkingowe.
        """
        # Wybór odpowiednich czujników w zależności od strony
        if self.side == "left":
```

```

        distance_front = dists_names["distance sensor left front side"]
        distance_rear = dists_names["distance sensor left side"]
    else: # self.side == "right"
        distance_front = dists_names["distance sensor right front side"]
        distance_rear = dists_names["distance sensor right side"]

    # Oblicz zmiany odległości
    delta_front = distance_front - self.prev_distance_front
    delta_rear = distance_rear - self.prev_distance_rear
    self.prev_distance_front = distance_front
    self.prev_distance_rear = distance_rear

    # Aktualizuj pozycję z odometrii
    odom_pose = self.update_odometry(yaw)
    x, y, yaw = odom_pose

    # Stan: poszukiwanie początku luki
    if self.state == "searching_start":
        if delta_front > self.threshold:
            # Znaleziono gwałtowny wzrost odległości ⇒ początek luki
            self.start_pose = (x - self.marg_start, y, yaw)
            self.dist_start_far = distance_front
            self.dist_start_cl = self.prev_distance_front
            self.state = "searching_progress"
            print("Kandydat na miejsce znalezione.")

    # Stan: poszukiwanie końca luki
    elif self.state == "searching_progress":
        if -delta_front > self.threshold:
            # Znaleziono gwałtowny spadek odległości ⇒ koniec luki
            end_pose = (x + self.marg_end, y, yaw)
            self.dist_end_far = distance_front
            self.dist_end_cl = distance_front - delta_front

            # Utwórz obiekt miejsca parkingowego
            spot = self._make_spot(self.start_pose, end_pose)
            if spot:
                self.spots.append(spot)
                print("Miejsce znalezione!!!", spot)
                self.state = "waiting_for_park"
                return odom_pose, spot
            else:
                print("Miejsce okazało się za małe.")
                self.state = "searching_start"

    def exec_path(self, curr_pose, end_pose, lateral_dist):
        """
        Generuje sterowanie pojazdem do osiągnięcia end_pose.
        """
        x, y, yaw = curr_pose
        x_e, y_e, _ = end_pose

        # Oblicz kąt do punktu docelowego i błąd yaw
        target_yaw = math.atan2(y_e - y, x_e - x)
        yaw_err = normalize_angle(target_yaw - yaw)

        # Odległość do celu
        dist_forward = math.hypot(x_e - x, y_e - y)

        # Regulator PD dla yaw
        steer = self.Kp * yaw_err + self.Kd * (yaw_err - self.prev_yaw_err)
        self.prev_yaw_err = yaw_err

        # Ustaw kąty sterowania i prędkość
        self.driver.setSteeringAngle(steer)
        self.driver.setCruisingSpeed(MAX_SPEED)

        # Dodatkowa korekcja boczna (trzymanie od krawężnika)
        deltalat = lateral_dist - (CAR_WIDTH/2 + 0.1)
        steer += self.Kl * deltalat
        self.driver.setSteeringAngle(steer)

    def _make_spot(self, start, end):
        """
        Tworzy opis miejsca parkingowego na podstawie pozycji start/end.
        """
        sp_len = end[0] - start[0]
        if sp_len < self.min_width:
            return None

        # Szerokość miejsca na podstawie różnicy odczytów sonaru
        sp_wid = ((self.dist_start_far - self.dist_start_cl) +
                  (self.dist_end_far - self.dist_end_cl)) / 2

```

```

# Środek luki parkingowej
sp_cen_x = start[0] + sp_len / 2
sp_cen_y = start[1] + (self.dist_start_cl + self.dist_start_far) / 2

# Ustawienie w zależności od strony parkowania
if self.side == "left":
    sen_pos = [3.515873,  0.865199,  90]
else:
    sen_pos = [3.515873, -0.865199, -90]

# Punkt końcowy manewru
x_end = sp_cen_x + sen_pos[0] - 1.425
y_end = sp_cen_y + sen_pos[1]

return (x_end, y_end, sen_pos[2])

def get_spots(self):
    """Zwraca listę wykrytych miejsc parkingowych."""
    return self.spots

class LivePlotter:
    def __init__(self, ax, max_points=100):
        # Inicjalizacja wykresu: ograniczenia i etykiety
        self.max_points = max_points
        self.data = []
        self.ax = ax
        self.line, = self.ax.plot([], [], 'r-')
        self.ax.set_xlim(0, self.max_points)
        self.ax.set_ylim(-10, 10)
        self.ax.set_title("Wartości z czujnika na żywo")
        self.ax.set_xlabel("Pomiar")
        self.ax.set_ylabel("Wartość")
        self.val = 0.0

    def update(self, frame=None):
        """
        Aktualizacja danych na wykresie w czasie rzeczywistym.
        """
        sensor_value = self.val          # pobierz aktualną wartość czujnika
        self.data.append(sensor_value)
        if len(self.data) > self.max_points:
            self.data.pop(0)
        xdata = list(range(len(self.data)))
        ydata = self.data

        # Zaktualizuj dane linii i zakres osi
        self.line.set_data(xdata, ydata)
        self.ax.set_xlim(0, max(self.max_points, len(self.data)))
        self.ax.set_ylim(min(ydata) - 1, max(ydata) + 1)
        return (self.line,)

#def run(self):
#    ani = animation.FuncAnimation(self.fig, self.update, interval=100, blit=True)
#    plt.show()

```

Plik stereo_yolo.py

Znajdują się w tych skryptach funkcje do estymacji pozycji obiektów za pomocą YOLO i stereowizj.

```
import numpy as np
import cv2
#from ultralytics import YOLO
import math

"""
W tym pliku są pomocnicze funkcje, które zajmowały niepotrzebne miejsce
w pliku kontrolera. Odpowiadają za narysowanie brył i punktów pochodzących
ze stereowizji i z YOLO
"""

def calculate_intrinsic_matrix(width, height, fov_rad):
    """
    Calculate the camera intrinsic matrix for given resolution and field of view.

    :param width: Image width in pixels
    :param height: Image height in pixels
    :param fov_rad: Horizontal field of view in radians
    :return: 3x3 intrinsic matrix
    """

    fx = (width / 2) / math.tan(fov_rad / 2)
    fy = fx # Assuming square pixels (can be adjusted if needed)
    cx = width / 2
    cy = height / 2

    K = np.array([
        [fx, 0, cx],
        [0, fy, cy],
        [0, 0, 1]
    ])

    return K

"""
Dwie poniższe funkcje są do zbudowania pozy kamery i szachownicy
w postaci macierzy przekształcenia jednorodnego.
"""

def build_homogeneous_transform(R, t):
    T = np.eye(4)
    T[:3, :3] = R
    T[:3, 3] = t.flatten()
    return T

def build_pose_matrix(position, yaw_deg):

    yaw_rad = np.deg2rad(yaw_deg)
    R = np.array([
        [np.cos(yaw_rad), -np.sin(yaw_rad), 0],
        [np.sin(yaw_rad), np.cos(yaw_rad), 0],
        [0, 0, 1]
    ])
    T = np.eye(4)
    T[:3, :3] = R
    T[:3, 3] = position
    return T

"""
Poniższa funkcja jest dla przeliczenia punktu na płaszczyźnie
drogi zgodnie z położeniem kamery w układzie samochodu -
przelicza kliknięty punkt w pikselach na punkt
w metrach odnośnie samochodu.
"""

def get_click_position(event, x, y, flags, param):
    global click_position,image

    T_center_to_camera,K = param
    if event == cv2.EVENT_LBUTTONDOWN:
        click_position = (x, y)
        print(f"Kliknięto na pozycji: {click_position}")

        # Rysowanie punktu na obrazie
        cv2.circle(image, (x, y), 5, (0, 0, 255), -1)
        cv2.imshow("kamerka", image)
```

```

point_on_ground = pixel_to_world(x, y, K, T_center_to_camera)
print("Punkt na ziemi w układzie BEV:", point_on_ground)

"""
Funkcja pixel_to_world robi to, co wskazuje jej nazwa -
przelicza piksel na obrazie kamery do współrzędnych globalnych
na podstawie parametrów wewnętrznych i zewnętrznych kamery.
Parametry zewnętrzne są określone macierzą T_center_to_camera -
położenie kamery w układzie samochodu.
"""

def pixel_to_world(u, v, K, T_center_to_camera):
    pixel = np.array([u, v, 1.0]) # Piksel w przestrzeni obrazu (homogeniczny)
    ray_camera = np.linalg.inv(K) @ pixel
    ray_camera = ray_camera / np.linalg.norm(ray_camera) # Normalizowanie

    ray_world = T_center_to_camera[:3, :3] @ ray_camera
    camera_position = T_center_to_camera[:3, 3]

    if ray_world[2] == 0:
        return None # Promień równoległy do ziemi, brak przecięcia

    t = -camera_position[2] / ray_world[2]
    point_on_ground = camera_position + t * ray_world

    return point_on_ground # Punkt na ziemi (X, Y, 0)

click_position = None
global image

"""
Przelicza punkty z układu globalnego 3D na obraz kamery 2D.
Posługuje się macierzą kamery oraz jej położeniem
w układzie globalnym (samochodu).
"""

def project_points_world_to_image(points_world, T_world_to_camera, K):
    projected_points = []

    # Inverse -> Camera <- World
    T_camera_to_world = np.linalg.inv(T_world_to_camera)

    for point in points_world:
        point_h = np.append(point, 1) # homogeneous
        point_in_camera = T_camera_to_world @ point_h
        Xc, Yc, Zc = point_in_camera[:3]

        if Zc <= 0:
            continue # behind camera

        # Project
        p_image = K @ np.array([Xc, Yc, Zc])
        u = p_image[0] / p_image[2]
        v = p_image[1] / p_image[2]
        projected_points.append((int(u), int(v)))

    return projected_points

def disp_to_cam3D(u, v, d, K, B):
    if d <= 0:
        return None # brak danych

    f = K[0, 0]
    cx = K[0, 2]
    cy = K[1, 2]

    Z = f * B / d
    X = (u - cx) * Z / f
    Y = (v - cy) * Z / f
    return np.array([X, Y, Z]) # w układzie kamery

def points_from_mask_to_3D(mask_resized, filtered_disp, K, baseline, T_cam_to_car):
    """
    Given a segmentation mask and filtered disparity map, compute 3D coordinates of
    the left and right extreme points of the object.
    """

    Args:
        mask_resized (np.ndarray): Binary mask of shape (H, W)
        filtered_disp (np.ndarray): Disparity map (float32) of shape (H, W)
        K (np.ndarray): Intrinsic matrix of the right (or left) camera
        baseline (float): Stereo baseline in meters (e.g., 0.03 m)
        T_cam_to_car (np.ndarray): 4x4 transformation matrix from camera to car frame

```

```

>Returns:
    tuple: Two 3D points (np.ndarray) in car coordinate frame or (None, None) if no points found
"""

# Step 1: Find leftmost and rightmost mask points

ys, xs = np.where(mask_resized > 0)
if len(xs) == 0:
    return None, None

left_x = np.min(xs)
right_x = np.max(xs)

left_y = int(np.median(ys[xs == left_x]))
right_y = int(np.median(ys[xs == right_x]))


def get_valid_disparity(disp_map, x, y, window=30):
    h, w = disp_map.shape
    x0 = max(0, x - window)
    x1 = min(w, x + window + 1)
    y0 = max(0, y - window)
    y1 = min(h, y + window + 1)
    patch = disp_map[y0:y1, x0:x1]
    valid = patch[patch > 0]
    if valid.size == 0:
        return None
    return np.median(valid)

# Step 2: Get disparity at these points

disp_left = get_valid_disparity(filtered_disp, left_x, left_y)
disp_right = get_valid_disparity(filtered_disp, right_x, right_y)

if disp_left is None or disp_right is None:
    return None, None
if disp_left <= 0 or disp_right <= 0:
    return None, None

# Step 3: Reproject using pinhole stereo model
f = K[0, 0]
cx = K[0, 2]
cy = K[1, 2]

# Point 1 (left)
Z1 = f * baseline / disp_left
X1 = (left_x - cx) * Z1 / f
Y1 = (left_y - cy) * Z1 / f

point_cam1 = np.array([X1, Y1, Z1, 1.0])

# Point 2 (right)
Z2 = f * baseline / disp_right
X2 = (right_x - cx) * Z2 / f
Y2 = (right_y - cy) * Z2 / f

point_cam2 = np.array([X2, Y2, Z2, 1.0])

# Step 4: Transform to car frame
point_car1 = T_cam_to_car @ point_cam1
point_car2 = T_cam_to_car @ point_cam2

return point_car1[:3], point_car2[:3]

```

Plik parking_parallel_new.py

Plik główny, zawierający pętle kontrolera i powiązany przez Webots API do wykonywania wszystkich operacji (jeżeli umieścimy pętlę w innym pliku, nie odwoła się do niej, dopóki nie wspomnimy tego w głównym pliku).

```
import numpy as np
import cv2
import math
import os
from controller import Robot, Camera,
GPS, Keyboard, DistanceSensor,
Gyro, InertialUnit, Supervisor,
Display)
from vehicle import Driver
#from transformers import AutoImageProcessor, AutoModelForDepthEstimation
#from transformers import DepthProImageProcessorFast, DepthProForDepthEstimation
#import torch._dynamo
#torch._dynamo.config.suppress_errors = True
#import torch
#import torchvision.transforms as T

import visualise as vis
import camera_calibration as cc
import park_algo as paig
import stereo_yolo as sy
from ultralytics import YOLO

#from scipy.interpolate import splprep, splev
#from scipy.ndimage import uniform_filter1d
import matplotlib.pyplot as plt
import matplotlib._pylab_helpers
import time
import threading, queue

# ----- Stałe -----
TIME_STEP = 128
NUM_DIST_SENSORS = 12
NUM_CAMERAS = 8
MAX_SPEED = 250.0
CAMERA_HEIGHT=2160
CAMERA_WIDTH=3840
global parking

SENSOR_INTERVAL = 0.06
IMAGE_INTERVAL = 0.2
KEYBOARD_INTERVAL = 0.04

# Parametry samochodu, charakterystyki z symulatora
TRACK_FRONT = 1.628
TRACK_REAR = 1.628
WHEELBASE = 2.995
MAX_WHEEL_ANGLE = 0.5 # rad
CAR_WIDTH = 2.302
CAR_LENGTH = 4.85

# ----- Zmienne globalne -----
driver = Driver()

#robot = Robot()
#supervisor = Supervisor()
display = Display('display')
keyboard = Keyboard()
keyboard.enable(TIME_STEP)
gps = None
Driver.synchronization = False
cameras = []
print(driver.synchronization)
camera_names = []
cam_matrices = {}
images = []
distance_sensors = []

speed = 0.0
steering_angle = 0.0
manual_steering = 0

previous_error = 0.0
```

```

integral = 0.0
homography_matrices = {}

# ----- Pomocnicze -----
def print_help():
    print("Samochód teraz jeździ.")
    print("Proszę użyć klawiszy UP/DOWN dla zwiększenia prędkości lub LEFT/RIGHT dla skrętu")
    print("Naciśnij klawisz P, aby rozpocząć poszukiwanie miejsca")
    print("Podczas parkowania, wciśnij Q aby szukać z prawej strony")
    print("albo E aby szukać miejsca z lewej strony")

def set_speed(kmh):
    global speed
    speed = min(kmh, MAX_SPEED)
    driver.setCruisingSpeed(speed)
    print(f"Ustawiona prędkość {speed} km/h")

def set_steering_angle(wheel_angle):
    global steering_angle
    # Clamp steering angle to [-0.5, 0.5] radians (per vehicle constraints)
    wheel_angle = max(min(wheel_angle, MAX_WHEEL_ANGLE), -MAX_WHEEL_ANGLE)
    steering_angle = wheel_angle
    driver.setSteeringAngle(steering_angle)
    print(f"Skręcam {steering_angle} rad")

def change_manual_steering_angle(inc):
    global manual_steering
    new_manual_steering = manual_steering + inc
    if -25.0 <= new_manual_steering <= 25.0:
        manual_steering = new_manual_steering
        set_steering_angle(manual_steering * 0.02)

#-----Sensor functions-----
camera_names = [
    "camera_front_bumper_wide", "camera_rear",
    "camera_left_fender", "camera_right_fender",
    "camera_left_pillar", "camera_right_pillar",
    "camera_front_top", "camera_front_top_add",
    "camera_helper"
]

def get_camera_image(camera):
    width = camera.getWidth()
    height = camera.getHeight()
    img = camera.getImage()
    if img is None:
        return None

    img_array = np.frombuffer(img, np.uint8).reshape((height, width, 4))[:, :, :3]
    img_array = cv2.cvtColor(img_array, cv2.COLOR_BGR2RGB)
    return img_array

sensor_names = [
    "distance sensor left front side", "distance sensor front left", "distance sensor front lefter",
    "distance sensor front righter", "distance sensor front right", "distance sensor right front side",
    "distance sensor left side", "distance sensor left", "distance sensor lefter",
    "distance sensor righter", "distance sensor right", "distance sensor right side"
]

def process_distance_sensors(sen):
    l_dist = sen.getLookupTable()
    a_dist = (l_dist[0]-l_dist[3])/(l_dist[1]-l_dist[4])
    b_dist = l_dist[3]-l_dist[4]*a_dist
    value = sen.getValue()
    distance = a_dist*value+b_dist
    sigma = l_dist[2]
    noisy_distance = distance + np.random.normal(0, sigma)
    return noisy_distance

# ----- Główna pętla -----
def main():

    names_dists = {}
    dists = []
    for name in sensor_names:
        sensor = driver.getDevice(name)
        if sensor:
            sensor.enable(TIME_STEP)

```

```

        distance_sensors.append(sensor)

    print(f"Found sensor: {name}")
else:
    print(f"Sensor not found: {name}")

# Inicjalizuj kamery
for name in camera_names:
    cam = driver.getDevice(name)
    if cam:
        cam.enable(TIME_STEP)
        cameras.append(cam)
        width = cam.getWidth()
        height = cam.getHeight()
        fov_rad = cam.getFov()
        K = sy.calculate_intrinsic_matrix(width, height, fov_rad)
        cam_matrices[name] = K
        print(f"Odnaleziono kamerę: {name}")

# GPS inicjalizacja
gps = driver.getDevice("gps")
if gps:
    gps.enable(TIME_STEP)
    print("GPS enabled")
#Inicjalizacja IMU
imu = driver.getDevice("inertial unit")
if imu:
    imu.enable(TIME_STEP)
    print("IMU enabled")

print_help()

print("reading homographies...")
right_H = np.load("right_homo.npy").astype(np.float32)
left_H = np.load("left_homo.npy").astype(np.float32)
front_H = np.load("front_homo.npy").astype(np.float32)
right_fender_H = np.load("right_fender_homo.npy").astype(np.float32)
left_fender_H = np.load("left_fender_homo.npy").astype(np.float32)
rear_H = np.load("rear_homo.npy").astype(np.float32)

homographies = []
homographies.extend([front_H,right_H,right_fender_H,
rear_H,left_fender_H,left_H])

s = 2 # skala
S = np.array([[1/s,0,0],[0,1/s,0],[0,0,1]]).astype(np.float32)
homographies = [S @ H @ np.linalg.inv(S) for H in homographies]

print("reading transformation matrices...")

front_T = np.load("camera_front_bumper_wide_T_global.npy").astype(np.float32)
left_T = np.load("camera_left_pillar_T_global.npy").astype(np.float32)
right_T = np.load("camera_right_pillar_T_global.npy").astype(np.float32)
left_fender_T = np.load("camera_left_fender_T_global.npy").astype(np.float32)
right_fender_T = np.load("camera_right_fender_T_global.npy").astype(np.float32)
rear_T = np.load("camera_rear_T_global.npy").astype(np.float32)
front_top_T = np.load("camera_front_top_T_global.npy").astype(np.float32)

prev_x = 0
prev_y = 0

stream1 = cv2.cuda.Stream()
stream2 = cv2.cuda.Stream()
stream3 = cv2.cuda.Stream()
stream4 = cv2.cuda.Stream()
stream5 = cv2.cuda.Stream()
stream6 = cv2.cuda.Stream()
stream7 = cv2.cuda.Stream()
stream8 = cv2.cuda.Stream()
stream9 = cv2.cuda.Stream()
stream10 = cv2.cuda.Stream()
stream11 = cv2.cuda.Stream()
stream12 = cv2.cuda.Stream()
stream13 = cv2.cuda.Stream()
streams = (stream1,stream2,stream3,stream4,stream5,
stream6,stream7,stream8,stream9,stream10,stream11,stream12,stream13)

```

```

car = cv2.imread("bmw.png", flags=cv2.IMREAD_COLOR)

model = YOLO("yolo11m-seg.pt")
#
#
#
last_sensor_time = driver.getTime()
last_image_time = driver.getTime()
last_key_time = driver.getTime()

T_center_to_camera = front_top_T
name_right = "camera_front_top"
name_left = "camera_front_top_add"

# Obiekt stereo matcher
stereo_left = cv2.StereoSGBM_create(minDisparity=0,
numDisparities=32,
blockSize=20,
disp12MaxDiff=1,
uniquenessRatio=10,
speckleWindowSize=100,
speckleRange=8)
# matcher dla prawego obrazu - trzeba użyć createRightMatcher z ximgproc
stereo_right = cv2.ximgproc.createRightMatcher(stereo_left)

#----- Dalej wątki -----
plt.ion()
fig, ax_cones, ax_live = None, None, None
parking = False
parker = None
def check_keyboard():
    nonlocal parker,parking, fig, ax_cones, ax_live
    key = keyboard.getKey()
    if key == Keyboard.UP:
        set_speed(speed + 0.5)
    elif key == Keyboard.DOWN:
        set_speed(speed - 0.5)
    elif key == Keyboard.RIGHT:
        change_manual_steering_angle(+2)
    elif key == Keyboard.LEFT:
        change_manual_steering_angle(-2)
    elif key == ord('P') or key == ord('p'):
        parking = not parking
    if parking:
        # Utwórz okno i osie tylko raz

        fig, (ax_cones,ax_live) = plt.subplots(1, 2, figsize=(20,10))
        #odkomentować niżej jeżeli chce się tylko wszystkie czujniki
        #fig, ax_cones = plt.subplots(1, 1, figsize=(12,12))
        fig.suptitle("Parkowanie")

        print("Rozpoczęto parking")
    else:

        plt.close(fig)
        cv2.destroyAllWindows()
        print("Ukończono parking")
if parker is not None:
    if parking and parker.state != "parking":
        if key in (ord('Y'),ord('y')):
            parker.state = "parking"
        elif key in (ord('N'),ord('n')):
            parker.state == "searching_start"
            print("Rozpoczynam od nowa...")
        elif key in (ord('Q'),ord('q')):
            print("Szukam z lewej strony...")
            parker.side = "left"
        elif key in (ord('E'),ord('e')):
            print("Szukam z prawej strony...")
            parker.side = "right"

first_call = True
odom = 0.0
spot = 0.0
while driver.step() != -1:

    now = driver.getTime()

    if parking:

```

```

# jeżeli z klawiatury włączono parkowanie, to:

if now - last_sensor_time >= SENSOR_INTERVAL:
    # co SENSOR_INTERVAL (zadany na początku) wykonujemy:
    last_sensor_time = now
    if first_call:
        plotter = palg.LivePlotter(ax_live)
        parker = palg.Parking(driver, "left", now)
        yaw_init = imu.getRollPitchYaw()[2]
    first_call = False
    # odczyt odległości
    dists = [process_distance_sensors(s) for s in distance_sensors]
    names = dict(zip(sensor_names, dists))

    # draw_cones na ax_cones
    vis.draw_cones(ax_cones, fig, dists)

    # live-plot na ax_live
    if (parker.side == "left"):
        plotter.val = names["distance sensor left front side"]
    elif (parker.side == "right"):
        plotter.val = names["distance sensor right front side"]
    plotter.update(0)

    # automaty parkowania
    yaw = imu.getRollPitchYaw()[2] - yaw_init
    parker.update_state(names, yaw)

    if parker.state == "parking":
        odom, spot = parker.update_state(names, yaw)
        if (parker.side == "left"):
            parker.exec_path(odom, spot, names["distance sensor left front side"])
        elif (parker.side == "right"):
            parker.exec_path(odom, spot, names["distance sensor right front side"])
    #to niżej do wizualizacji - zakomentować razem z "plotter"
    #i "vis.draw_cones" jeżeli nie chce się wizualizacji
    fig.canvas.draw_idle()
    fig.canvas.flush_events()

    # co IMAGE_INTERVAL - przetwarzanie obrazów
    if now - last_image_time >= IMAGE_INTERVAL:
        last_image_time = now
        images = [get_camera_image(c) for c in cameras]
        names_images = dict(zip(camera_names, images))

        viss = vis.alt_collect_homo(names_images, homographies, car, streams)
        #cv2.imwrite("img3_vis.png",viss)

        cv2.waitKey(1)

    elif not parking:
        first_call = True
    if now - last_key_time >= KEYBOARD_INTERVAL:
        last_key_time = now
        check_keyboard()

if __name__ == "__main__":
    main()

```