

UNIWERSYTET ZIELONOGÓRSKI

Wydział Nauk Inżynierjno-Technicznych

Praca inżynierska

Kierunek: Informatyka

Specjalność: Sieciowe Systemy Informatyczne

Studia stacjonarne

System do wykrywania i rozpoznawania obiektów na obrazie z kamery samochodowej z wykorzystaniem symulatora CARLA

A system for detecting and recognizing objects in images from a car camera using the CARLA simulator

Piotr Noga

Promotor:

Dr hab. inż. Marek Kowal, prof. UZ

Zielona Góra, luty 2026

Streszczenie

Celem niniejszej pracy było opracowanie systemu wykrywania i rozpoznawania obiektów na obrazach pochodzących z kamery samochodowej, z zastosowaniem algorytmu YOLOv4 oraz symulatora jazdy samochodowej CARLA. Projekt zakładał integrację detektora obiektów z symulowanym środowiskiem oraz ocenę jego skuteczności w różnych warunkach drogowych i pogodowych.

W pierwszej części pracy przedstawiono charakterystykę środowiska CARLA oraz przegląd metod wykrywania obiektów w obrazach. Następnie zaprojektowano i zaimplementowano system detekcji, który umożliwia identyfikację pojazdów, pieszych i znaków drogowych w czasie rzeczywistym. System został przetestowany w symulowanych warunkach, a uzyskane wyniki wykazały wysoką skuteczność detekcji oraz stabilność działania.

Na podstawie przeprowadzonych badań stwierdzono, że opracowany system spełnia założone cele i może stanowić podstawę do dalszych prac nad percepcją środowiska w pojazdach autonomicznych.

Słowa kluczowe: symulator CARLA, YOLO4, Darknet, Python, Ubuntu, detekcja obiektów, rozpoznawanie obrazów, sztuczna inteligencja

Spis treści

1. Wstęp	1
1.1. Wprowadzenie	1
1.2. Cel i zakres pracy	2
1.3. Struktura pracy	3
2. Symulator Carla	4
2.1. Architektura systemu Carla	4
2.2. Możliwości symulatora	5
2.2.1. Świat	5
2.2.2. Sensory	7
2.2.3. Pogoda	13
2.2.4. Oświetlenie	15
2.2.5. Reprezentacja obiektów w przestrzeni 3D i definicja Bounding Box	16
2.2.5.1. Relacja między punktem Origin a środkiem geometrycznym	17
2.2.5.2. Transformacja do układu współrzędnych świata	18
2.2.5.3. Rzutowanie z przestrzeni 3D na płaszczyznę obrazu 2D	18
2.3. Instalacja symulatora Carla	19
2.3.1. Wymagania sprzętowe	19
2.3.2. Instalacja i konfiguracja dla systemu Ubuntu	19
2.3.3. Instalacja Unreal Engine	20
2.3.4. Pobranie źródeł CARLA i komplikacja	20
2.3.5. Uruchomienie symulatora CARLA	21
2.4. Skrypt do manualnego sterowania w symulatorze CARLA	21
2.4.1. Struktura skryptu	21
2.4.2. Logika działania pliku <code>manual_control.py</code>	22
2.4.3. Pętla gry (Game Loop)	23
2.4.4. Tryb asynchroniczny i synchroniczny	24
2.4.5. Implementacja synchronizacji w <code>manual_control.py</code>	24
2.4.6. Analiza pętli głównej <code>game_loop()</code>	25
2.4.7. Znaczenie dla integracji z algorytmami zewnętrznymi	25
2.4.8. Sterowanie pojazdem z poziomu klawiatury	26
2.4.9. Pętla gry - Game Loop	27
2.4.10. Sterowanie pojazdem za pomocą klawiatury	28
2.4.11. Czyszczenie zasobów (Cleanup)	28
3. System rozpoznawania obrazów	30

3.1.	YOLOv4 i konwolucyjne sieci neuronowe	35
3.1.1.	Konwolucyjne sieci neuronowe (CNN)	35
3.2.	Zasada działania algorytmu YOLO i YOLOv4	35
3.2.1.	Architektura YOLOv4	38
3.3.	Zalety, wydajność i zastosowania algorytmu YOLOv4	39
3.4.	Instalacja systemu YOLO	41
3.4.1.	Testowanie instalacji	42
3.5.	Integracja detektora YOLOv4 z symulatorem CARLA	42
3.6.	Schemat ewaluacji offline	43
4.	Wyniki badań eksperymentalnych	45
4.1.	Eksperyment on-line - wydajność systemu	45
4.1.1.	Opis scenariusza i przebiegu testów	45
4.1.2.	Środowisko sprzętowe i konfiguracja	47
4.1.3.	Wyniki pomiarów FPS	48
4.1.4.	Analiza wyników wydajnościowych	48
4.1.5.	Przykładowe funkcjonalności oprogramowania w trybie on-line	49
4.2.	Eksperyment offline – weryfikacja poprawności detekcji	51
4.2.1.	Metryka Intersection over Union (IoU)	51
4.2.2.	Zestaw scen i konfiguracja eksperymentu offline	52
4.2.3.	Przebieg przetwarzania w eksperymencie offline	52
4.2.4.	Analiza wyników eksperymentu offline	53
4.2.5.	Wybrane funkcjonalności oprogramowania w eksperymencie offline	54
4.2.5.1.	Skrypt <code>bounding_boxes.py</code> – generowanie anotacji referencyjnych	54
4.2.5.2.	Skrypt <code>yolo.py</code> – detekcja YOLO i zapis JSON	54
4.2.5.3.	Skrypt <code>evaluate_iou.py</code> – obliczanie metryki IoU	55
4.2.5.4.	Skrypt <code>bbox_image.py</code> – wizualizacja anotacji na obrazach	55
5.	Dyskusja rezultatów i wnioski końcowe	57

Spis rysunków

2.1.	Domyślna mapa - Town01	6
2.2.	Mapa Town04	6
2.3.	Korek drogowy	7
2.4.	Sensor odpowiadający za kolizje	8
2.5.	Kamera głębi oryginał	8
2.6.	Kamera głębi po konwersji	9
2.7.	Kamera głębi logarytmiczna	9
2.8.	Sensor GNSS	9
2.9.	Sensor IMU	10
2.10.	Sensor pDrzecięcia linii	10
2.11.	Sensor LIDAR	12
2.12.	Sensor detekcji obiektów	12
2.13.	Sensor radaru	13
2.14.	Lekki deszcz o zachodzie słońca	14
2.15.	Zachmurzona noc	14
2.16.	Mocny deszcz w południe	15
2.17.	Światła uliczne	15
2.18.	Światła pojazdu	16
2.19.	Schemat blokowy działania pliku <code>manual_control.py</code>	22
3.1.	RCNN	31
3.2.	fastRCNN	32
3.3.	fasterRCNN2	33
3.4.	SSD	33
3.5.	Schemat struktury konwolucyjnej sieci neuronowej [1].	35
3.6.	Architektura YOLOv4. Zawiera backbone, neck i head [2].	39
3.7.	Yolo Test	42
4.1.	sun10car20ppl1	46
4.2.	night20car40ppl2	46
4.3.	rainsun10car20ppl3	47

Spis tabel

2.1. Atrybuty sensora kolizji	7
3.1. Porównanie wybranych architektur	34
4.1. Wyniki testów wydajnościowych dla modelu dużego.	48
4.2. Wyniki testów wydajnościowych modelu małego.	48
4.3. Przykładowe wyniki eksperymentu offline dla wszystkich zarejestrowanych scen symulacji CARLA.	53

Spis listingów

2.1. caption text	13
2.2. Aktualizacja systemu	19
2.3. Instalacja zależności	19
2.4. Instalacja Pythona i PIP	20
2.5. Pobieranie źródeł CARLA	20
2.6. Przechodzenie do folderu Unreal Engine dla CARLA	20
2.7. Uruchamianie Unreal Engine	20
2.8. Kompilacja CARLA	20
2.9. Uruchamianie CARLA	21
2.10. Instalacja zależności Pythona	21
2.11. Uruchamianie przykładowego skryptu	21
2.12. Przykład kodu pętli gry - Game Loop	23
2.13. Przykładowy kod w języku Python odpowiedzialny za sterowanie pojazdem z klawiatury	26
2.14. Przykład kodu pętli gry - Game Loop	27
2.15. Przykładowy kod obsługi sterowania pojazdem z klawiatury	28
2.16. Kod funkcji odpowiedzialnej za czyszczenie zasobów	29
3.1. Aktualizacja listy pakietów	41
3.2. Instalacja wymaganych pakietów	41
3.3. Instalacja CUDA	41
3.4. Instalacja cuDNN	41
3.5. Klonowanie repozytorium YOLOv4	41
3.6. Edytowanie pliku Makefile	42
3.7. Kompilacja projektu YOLOv4	42
3.8. Testowanie YOLOv4	42
3.9. Importowanie bibliotek dla integracji z YOLOv4	43
3.10. Globalna funkcja <code>get_anchors()</code> dla YOLOv4	43
3.11. Wczytywanie i przetwarzanie obrazu z symulatora CARLA	43
3.12. Wykorzystanie funkcji <code>combined_non_max_suppression</code> w celu eliminacji powtórzeń	43
4.1. Pobieranie i przetwarzanie obrazu w symulatorze CARLA	49
4.2. Wykrywanie obiektów za pomocą YOLOv4	49
4.3. Filtrowanie wykrytych klas obiektów	50
4.4. Wizualizacja wykrytych obiektów na ekranie	50
4.5. Obsługa zdarzeń klawiatury w symulatorze	50
4.6. Pętla główna symulacji CARLA	50
4.7. Dodawanie obiektu do anotacji w <code>bounding_boxes.py</code>	54
4.8. Struktura JSON z wynikami YOLO	54

4.9. Obliczanie IoU dla dwóch ramek 2D	55
4.10. Rysowanie ramek na obrazie na podstawie pliku JSON	55

Rozdział 1

Wstęp

1.1. Wprowadzenie

Współczesna motoryzacja dynamicznie ewoluuje, a jednym z kluczowych obszarów jej rozwoju są systemy rozpoznawania i wykrywania obiektów. Technologie te, wykorzystujące sztuczną inteligencję i uczenie maszynowe, pozwalają na analizę otoczenia w czasie rzeczywistym, co znajduje zastosowanie zarówno w systemach wspomagania kierowcy (ADAS), jak i w pojazdach autonomicznych. Dzięki nim możliwe jest m.in. rozpoznawanie znaków drogowych, wykrywanie przeszkód czy monitorowanie martwego pola, co przekłada się na zwiększenie bezpieczeństwa na drogach.

Jednym z głównych wyzwań w rozwijaniu tych technologii jest ich testowanie w realistycznych warunkach. Tradycyjne metody, takie jak rzeczywiste jazdy testowe, są kosztowne i czasochłonne, a dodatkowo mogą wiązać się z ryzykiem wypadków. Dlatego coraz częściej wykorzystuje się symulacje komputerowe, które pozwalają na przeprowadzanie eksperymentów w kontrolowanych warunkach. Jednym z najbardziej zaawansowanych narzędzi w tej dziedzinie jest symulator CARLA (Car Learning to Act) – otwarteźródłowe środowisko stworzone z myślą o badaniach nad autonomiczną jazdą oraz systemami percepji wizualnej pojazdów [3].

W niniejszej pracy zostaną omówione kluczowe aspekty wykrywania i rozpoznawania obiektów na podstawie obrazu z kamer samochodowych, a także możliwości wykorzystania symulatora CARLA do badań w tym zakresie.

Przykłady zastosowań systemów rozpoznawania obrazu w motoryzacji:

- **Systemy wspomagania kierowcy (ADAS)**
 - **Rozpoznawanie znaków drogowych** – pozwala na identyfikację znaków takich jak ograniczenia prędkości czy zakazy wyprzedzania, pomagając kierowcom w przestrzeganiu przepisów.
 - **Ostrzeganie o niezamierzonej zmianie pasa ruchu** – system analizuje położenie pojazdu i informuje kierowcę o niekontrolowanym opuszczeniu pasa.
 - **Adaptacyjny tempomat** – automatycznie dostosowuje prędkość pojazdu do warunków na drodze, utrzymując bezpieczną odległość od innych pojazdów.

- **Systemy bezpieczeństwa**

- **Automatyczne hamowanie awaryjne** – wykrywa potencjalne kolizje i inicjuje hamowanie, aby zminimalizować ryzyko wypadku.
- **Monitorowanie martwego pola** – ostrzega kierowcę o pojazdach znajdujących się w obszarach niewidocznych w lusterkach.

- **Autonomiczne pojazdy i analiza otoczenia**

- **Identyfikacja przeszkód i użytkowników drogi** – pozwala na wykrywanie pieszych, rowerzystów oraz innych pojazdów, umożliwiając pojazdom autonomicznym bezpieczne poruszanie się po drogach.
- **Predykcja zachowań innych uczestników ruchu** – analiza sygnałów drogowych, ruchu pieszych i pojazdów umożliwia przewidywanie ich manewrów i odpowiednie reagowanie.

- **Systemy wspomagające parkowanie**

- **Automatyczne parkowanie** – pojazd może samodzielnie wykonać manewry parkowania, korzystając z kamer i czujników.
- **Widok 360°** – system kamer rozmieszczonych wokół pojazdu ułatwia manewrowanie w ciasnych przestrzeniach.

- **Monitorowanie stanu kierowcy**

- **Systemy wykrywające zmęczenie** – analizują ruchy kierowcy, takie jak częstotliwość mrugania czy pozycja głowy, ostrzegając w przypadku wykrycia oznak zmęczenia.

1.2. Cel i zakres pracy

Celem pracy jest opracowanie systemu wykrywania i rozpoznawania obiektów takich jak znaki drogowe, samochody oraz ludzie na obrazach pochodzących z kamery samochodowej. Projekt zostanie zaimplementowany z wykorzystaniem języka Python 2, przy pomocy darmowego środowiska CARLA umożliwiającego symulację jazdy samochodem w różnych warunkach drogowych.

Zakres pracy obejmował:

- zapoznanie się ze środowiskiem do symulacji jazdy samochodem CARLA,
- przegląd metod wykrywania obiektów na obrazach z kamery samochodowej,
- zaprojektowanie i wdrożenie systemu wykrywania obiektów w środowisku CARLA,
- przeprowadzenie testów weryfikujących skuteczność zaprojektowanego systemu,
- sformułowanie wniosków.

1.3. Struktura pracy

Niniejsza praca składa się z pięciu rozdziałów, które prowadzą od wprowadzenia teoretycznego, przez opis implementacji, aż do prezentacji wyników badań eksperymentalnych i wniosków końcowych.

W **rozdziale 1** przedstawiono tło problemu, motywację podjęcia tematyki detekcji obiektów na potrzeby pojazdów autonomicznych, a także zdefiniowano cel i zakres pracy oraz omówiono jej strukturę. Zwrócono uwagę na znaczenie symulacji komputerowych w procesie testowania algorytmów percepcji.

Rozdział 2 poświęcony jest szczegółowemu opisowi symulatora CARLA. Zaprezentowano w nim architekturę systemu klient–serwer, dostępne sensory, sposób modelowania świata oraz możliwości konfiguracji warunków środowiskowych. Omówiono również sposób instalacji symulatora w systemie Ubuntu oraz strukturę skryptu `manualcontrol.py`, który stanowi podstawę integracji z zewnętrznymi algorytmami detekcji.

W **rozdziale 3** opisano system rozpoznawania obrazu zastosowany w pracy. Przedstawiono wybrane architektury detekcji obiektów, ze szczególnym uwzględnieniem algorytmu YOLOv4, jego budowy i technik treningowych. Następnie zaprezentowano proces instalacji i uruchomienia detektora oraz sposób jego integracji z symulatorem CARLA. Na końcu rozdziału przedstawiono schemat ewaluacji offline, wykorzystywany do obliczania miary IoU na podstawie danych referencyjnych generowanych przez symulator.

Rozdział 4 zawiera opis procesu testowania zintegrowanego systemu oraz wyniki badań eksperymentalnych. Omówiono konfigurację scenariuszy jazdy, sposób rejestracji danych oraz wybrane funkcjonalności oprogramowania. Zaprezentowano wyniki eksperymentu on-line, dotyczące wydajności systemu mierzonej liczbą klatek na sekundę, oraz eksperymentu offline, w którym oceniano poprawność detekcji za pomocą miary IoU w różnych warunkach pogodowych i oświetleniowych.

W **rozdziale 5** przedstawiono dyskusję uzyskanych rezultatów oraz wnioski końcowe. Podsumowano najważniejsze obserwacje dotyczące jakości i wydajności detekcji obiektów w symulatorze CARLA z wykorzystaniem algorytmu YOLOv4, a także wskazano możliwe kierunki dalszych prac, w szczególności związane z rozszerzeniem zbioru danych, zastosowaniem innych architektur detekcji oraz integracją dodatkowych sensorów.

Rozdział 2

Symulator Carla

CARLA (Carl Learning to Act), czyli otwarty symulator jazdy miejskiej został stworzony do wspierania szkoleń, treningów, tworzenia prototypów i walidacji autonomicznych systemów jazdy zarówno na poziomie percepcji jak i kontroli. System jest otwartą platformą stworzoną przez specjalistyczny zespół grafików cyfrowych. Obejmuje układy urbanistyczne, modele pojazdów, budynki, pieszych oraz znaki drogowe. Symulacja umożliwia elastyczną konfigurację, którą można wykorzystać do strategicznego szkolenia jazdy, uzyskać współrzędne GPS, prędkość, przyspieszenie pojazdu czy też informacje związane z kolizją bądź wykroczeniami drogowymi. Poza tym CARLA wykorzystuje warunki środowiskowe, jak warunki meteorologiczne oraz godzina. System wykorzystywany jest do manipulowania trasą, którą trzeba pokonać przy danych warunkach drogowych i środowiskowych [3].

2.1. Architektura systemu Carla

Symulator składa się ze skalowanej architektury klient - serwer. Serwer odpowiada za wszystko, co jest związane z samą symulacją, czyli renderowanie czujników, obliczanie fizyczne, aktualizacje stanu świata oraz aktorów. Z kolei strona klienta składa się z sumy modułów klienckich kontrolujących logikę aktorów na scenie i ustalających warunki panujące na świecie. Osiąga to poprzez wykorzystanie interfejsu API CARLA (w Pythonie lub C++), warstwie pośredniczącej między serwerem a klientem, która stale ewoluje, aby zapewnić nowe funkcje (www.carla.org). Aby zrozumieć system CARLA należy poznać jego funkcje i elementy, dzięki którym osiąga wiele możliwości. Poniżej wymieniono niektóre z nich:

- Menedżer ruchu, czyli wbudowany system, który przejmuje kontrolę nad pojazdami. Działa jako przewodnik dostarczony przez CARLA do odtworzenia środowisk miejskich z realistycznymi zachowaniami.
- Czujniki, na których polegają pojazdy podczas przekazywania informacji o swoim otoczeniu. Są to specyficzni aktorzy podłączeni do pojazdu a dane, które otrzymują mogą być przechowywane i wyszukiwane w celu ułatwienia procesu. Obecnie projekt obsługuje ich różne typy - kamery, radary, lidary, itd.
- Rejestrator, czyli funkcja służąca do odtwarzania symulacji krok po kroku

dla każdego aktora na świecie. Dzięki niej użytkownik ma dostęp do każdego miejsca na świecie w dowolnym momencie osi czasu.

- Most ROS i implementacja Autoware, które zapewnią integrację symulatora z innymi środowiskami uczenia się.
- Otwarte zasoby, czyli ułatwienie tworzenia różnych map miejskich z kontrolą warunków pogodowych i biblioteką planów z szerokim zestawem aktorów do wykorzystania.
- Scenariusz jazdy. Aby ułatwić proces uczenia się pojazdów, CARLA zapewnia serię tras opisujących różne sytuacje do iteracji. Stanowią one również podstawę wyzwania CARLA, otwartego dla każdego, kto chce przetestować swoje rozwiązania i znaleźć się w tabeli liderów.

2.2. Możliwości symulatora

CARLA wykorzystywany jest do badania trzech podejść autonomicznej jazdy:

I - klasyczny, modułowy potok, który obejmuje oparty na wizji moduł percepji, planer oparty na regułach i kontroler manewrów.

II - głęboka sieć odtwarzająca dane sensoryczne na polecenie przeszkołonych poprzez naukę naśladowania kierowców.

III - rozbudowana sieć przeszkołona od początku do końca poprzez uczenie się ze wzmacnieniem.

Symulator Carla oferuje użytkownikowi możliwości konfiguracji środowiska, w którym odbywa się symulacja. Między innymi utworzenie własnej mapy rozwoju z elementami w postaci budynków, pojazdów oraz pieszych, czy też korzystanie z gotowych środowisk utworzonych przez autorów projektu. Środowisko umożliwia symulowanie warunków pogodowych, oraz sterowanie sygnalizacją świetlną za pomocą funkcji opisanych w języku Python. W środowisku Carla, zaimplementowane są także sensory, które odgrywają istotną rolę w przypadku kolizji czy też ustawniania atrybutów kamery.

2.2.1. Świat

Mapa jest jednym z głównych elementów świata symulatora. Zawiera zarówno model 3D miejscowości, jak i definicję dróg. Definicja dróg na mapie oparta jest na pliku OpenDRIVE, standardowym formacie definicji dróg z adnotacjami. Sposób, w jaki standard 1.4 OpenDRIVE definiuje drogi, pasy ruchu, skrzyżowania itp. determinuje funkcjonalność API Pythona oraz uzasadnienie podejmowanych decyzji.

API Python działa jako wysokopoziomowy system zapytań do nawigacji po tych drogach. Jest ono stale rozwijane, aby zapewnić szerszy zestaw narzędzi



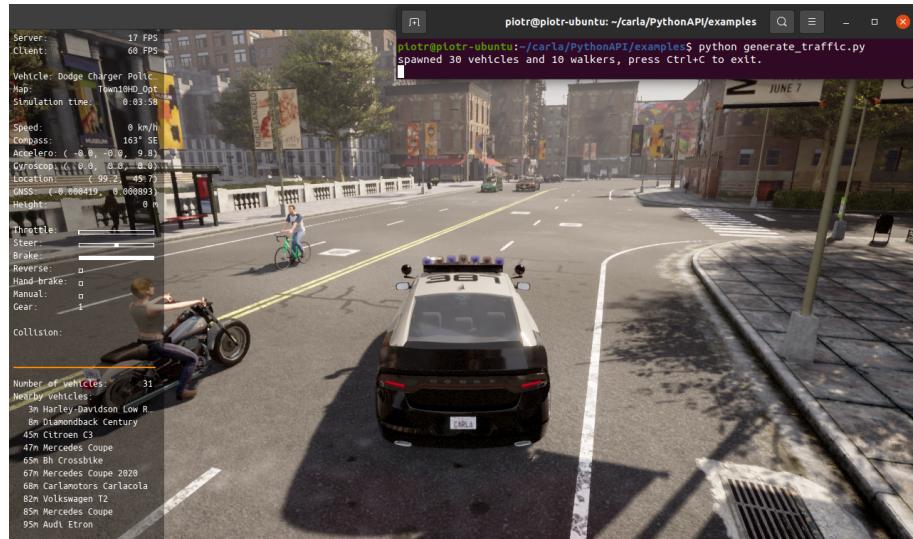
Rys. 2.1. Obraz mapy domyślnej - Town01.

Domyślną mapą symulatora CARLA jest Town01 Rys. 2.1, będąca odzworowaniem centrum dużego miasta. Program oferuje wiele gotowych map, z których można korzystać w celu przeprowadzenia eksperymentów, nie tylko podczas różnych warunków pogodowych, ale również w rozmaitych środowiskach. Przykładem jest mapa Town04 Rys. 2.2, przedstawiająca małe miasteczko w górach, przy jeziorze.



Rys. 2.2. Obraz mapy - Town04.

Świat symulatora zapewnia również aktorów. Aktorami są nie tylko pojazdy i piesi, ale także czujniki, znaki drogowe, sygnalizacja świetlna i widzowie. Bardzo ważne jest, aby mieć pełne zrozumienie, jak na nich operować. Istnieje możliwość tworzenia aktorów zarówno ręcznie, poprzez funkcję `spawn_actor()` jak i z wykorzystaniem przykładowego programu zapewnionego przez developerów - `generate_traffic.py`



Rys. 2.3. Obraz przedstawiający symulację 30 pojazdów oraz 10 pieszych.

2.2.2. Sensory

Sensor kolizji jest to czujnik, który rejestruje zdarzenie za każdym razem, gdy jego aktor macierzysty zderzy się z czymś w świecie. Podczas jednego kroku symulacji może zostać wykrytych kilka kolizji. Aby zapewnić wykrywanie kolizji z dowolnym obiektem, serwer tworzy „fałszywych” aktorów dla elementów takich jak budynki czy krzewy, dzięki czemu możliwe jest pobranie znacznika semantycznego w celu ich identyfikacji.

Detektory kolizji nie posiadają żadnych konfigurowalnych atrybutów.

Atrybuty	Typ	Opis
frame	int	Numer ramki, w której dokonano pomiaru.
timestamp	double	Czas symulacji pomiaru w sekundach od jej początku.
transform	carla.Transform	Położenie i obrót we współrzędnych świata czujnika w czasie pomiaru.
actor	carla.Actor	Aktor, który zmierzył kolizję (rodzic czujnika).
other_actor	carla.Actor	Aktor, z którym zderzył się rodzic.
normal_impulse	carla.Vector3D	Normalny impuls wynikający z kolizji.

Tab. 2.1. Atrybuty sensora kolizji

Kolejnym sensorem jest kamera głębi. Kamera dostarcza surowe dane sceny kodujące odległość każdego piksela od kamery (znane również jako bufor głębi lub z-bufor) w celu stworzenia mapy głębi elementów.

Obraz koduje wartość głębi na piksel używając 3 kanałów przestrzeni kolorów RGB, od mniejszej do bardziej znaczących bajtów: R -> G -> B. Rzeczywista odległość w metrach może być zdekodowana za pomocą:

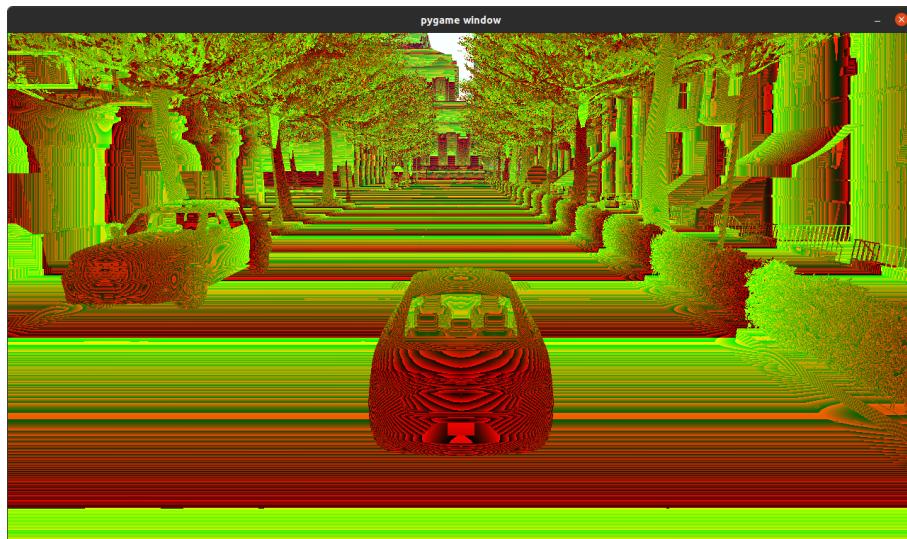


Rys. 2.4. Przykładowy obraz kolizji ze słupem.

```

1 normalized = (R + G * 256 + B * 256 * 256) / (256 * 256 * 256 - 1)
2 in_meters = 1000 * normalized

```

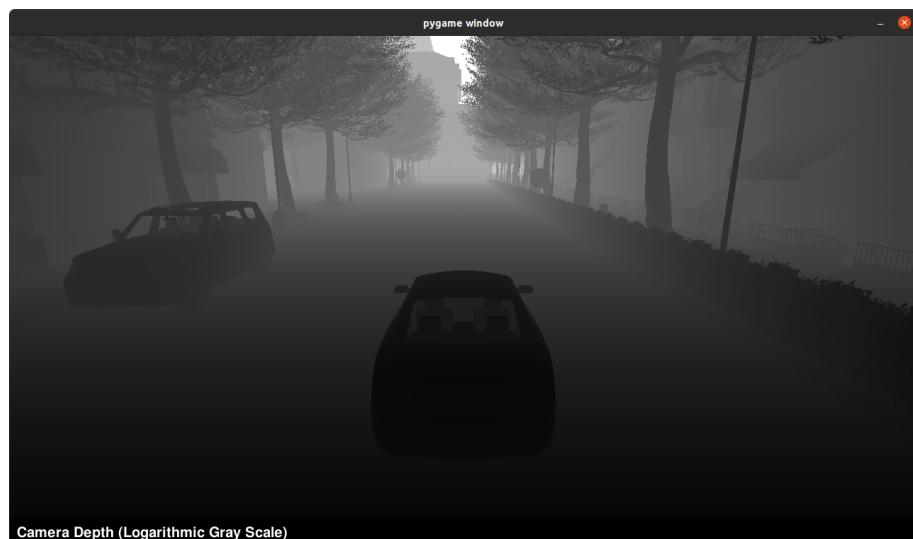


Rys. 2.5. Oryginalny obraz pochodzący z kamery głębi.

Wyjściowy obraz CARLA.Image powinien zostać zapisany na dysk przy użyciu `carla.colorConverter`, który zamieni odległość zapisaną w kanałach RGB na [0,1] float zawierający odległość, a następnie przetłumaczy to na skalę szarości. Istnieją dwie opcje w `carla.colorConverter`, aby uzyskać widok głębi: głębia w odcieniach szarości oraz głębokość logarytmiczna. Precyzja jest milimetrowa w obu, ale podejście logarytmiczne zapewnia lepsze wyniki dla bliższych obiektów. Ponadto widoczność jest lepsza dla użytkownika.



Rys. 2.6. Obraz pochodzący z kamery głębi po konwersji w odcieniach szarości.



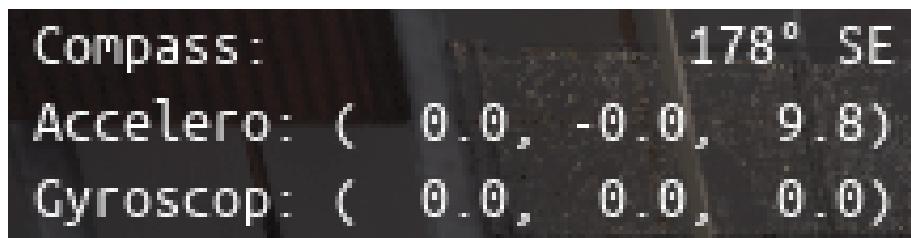
Rys. 2.7. Obraz pochodzący z kamery głębi - logarytmiczny.

Sensor GNSS (Global Navigation Satellite Systems) - podaje aktualną pozycję GNSS swojego obiektu nadziedznego. Jest ona obliczana poprzez dodanie pozycji metrycznej do początkowej lokalizacji georeferencyjnej zdefiniowanej w definicji mapy OpenDRIVE



Rys. 2.8. Widok sensora GNSS.

Sensor IMU dostarcza natomiast miary, które akcelerometr, żyroskop i kompas mogłyby pobrać dla obiektu nadziedznego. Dane są pobierane z bieżącego stanu obiektu.



Rys. 2.9. Widok sensora IMU.

Detektor wtargnięcia na pas ruchu Rejestruje zdarzenie za każdym razem, gdy jego rodzic przekroczy oznaczenie pasa ruchu. Czujnik wykorzystuje dane drogowe dostarczane przez opis mapy OpenDRIVE, aby określić, czy pojazd macierzysty najeźdża na inny pas ruchu, biorąc pod uwagę przestrzeń między kołami. Należy jednak zwrócić uwagę na pewne kwestie:

Rozbieżności pomiędzy plikiem OpenDRIVE a mapą spowodują powstanie nieprawidłowości, takich jak przecinające się pasy ruchu, które nie są widoczne na mapie. Wyjście pobiera listę oznaczeń przecinających się pasów: obliczenia są wykonywane w OpenDRIVE i uwzględniają całą przestrzeń pomiędzy czterema kołami jako całość. W związku z tym, w tym samym czasie może być przekraczanych więcej niż jeden pas ruchu.



Rys. 2.10. Obraz przykładu przecięcia linii.

Sensor LIDAR (eng. Light Detection and Ranging) - ten czujnik symuluje obracający się LIDAR zaimplementowany przy użyciu ray-casting. Punkty są obliczane przez dodanie lasera dla każdego kanału rozmieszczonego w pionowym FOV. Obrót jest symulowany poprzez obliczenie kąta poziomego, o jaki obrócił się Lidar w danej klatce. Chmura punktów jest obliczana przez wykonanie ray-cast dla każdego lasera w każdym kroku.

Pomiar LIDAR-owy zawiera paczkę z wszystkimi punktami wygenerowanymi w przedziale czasu 1/FPS. Podczas tego interwału fizyka nie jest aktualizowana, więc wszystkie punkty w pomiarze odzwierciedlają ten sam "statyczny obraz" sceny.

Informacja z pomiaru LIDAR-owego jest zakodowana w postaci punktów 4D.

Pierwsze trzy z nich to punkty przestrzenne we współrzędnych xyz, a ostatni to straty intensywności podczas jazdy. Intensywność ta jest obliczana według następującego wzoru:

$$I/I_0 = e^{-a*d}$$

Gdzie:

- a – Oznacza współczynnik tłumienia. Może on zależeć od długości fali czujnika oraz warunków atmosferycznych. Można go zmodyfikować za pomocą atrybutu LIDAR `atmosphere_attenuation_rate`.
- b – Odległość od punktu trafienia do czujnika.

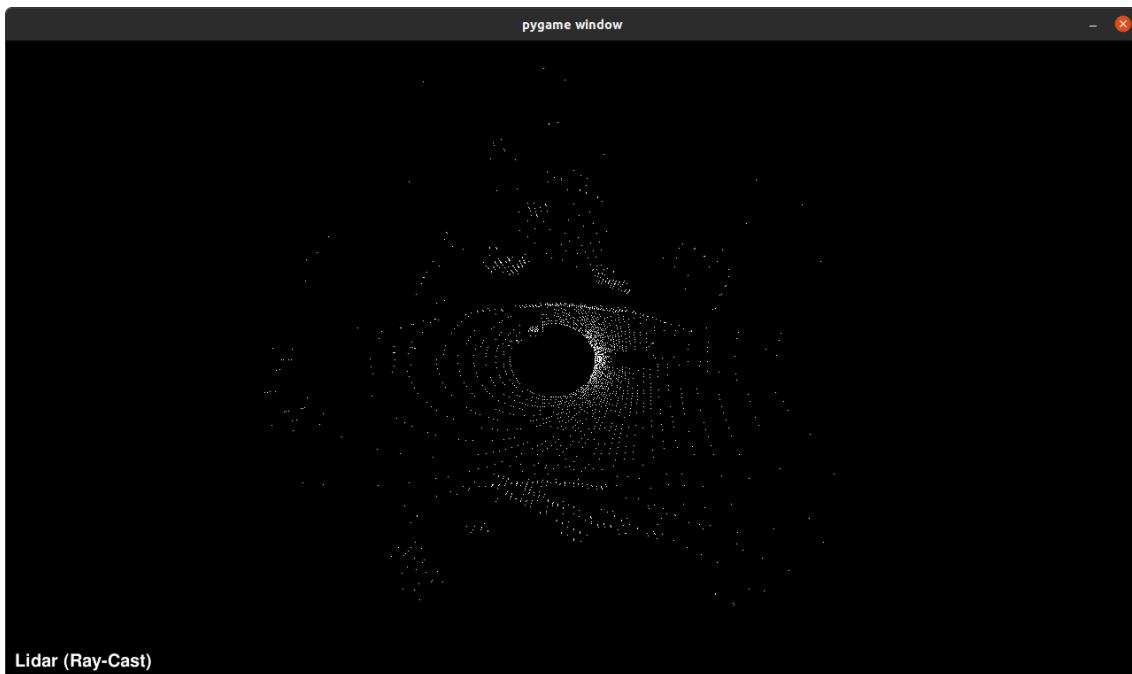
Dla większego realizmu, punkty w chmurze mogą być zrzucane. Jest to prosty sposób na symulowanie strat spowodowanych zewnętrznymi perturbacjami. Można to zrobić łącząc dwa różne sposoby.

Pierwszym sposobem jest general drop-off, czyli proporcja punktów, które są odpadane losowo. Jest to wykonywane przed śledzeniem, co oznacza, że odpadające punkty nie są obliczane, a zatem poprawia wydajność. Jeśli wartość ustawimy na 0,5, połowa punktów zostanie zrzucona.

1 `dropoff_general_rate = 0.5)`

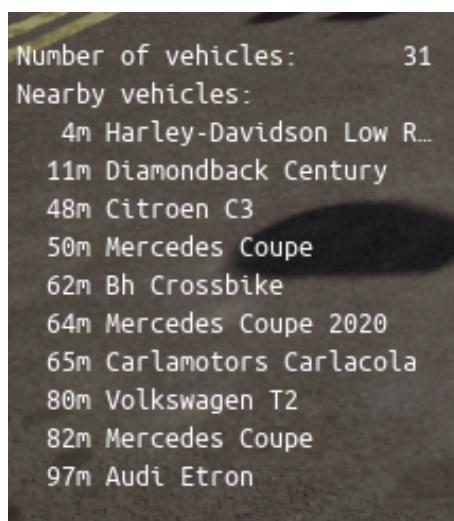
Drugim sposobem zaś jest zrzucanie na podstawie intensywności. Dla każdego wykrytego punktu wykonywane jest dodatkowe zrzucanie z prawdopodobieństwem opartym na obliczonej intensywności. Prawdopodobieństwo to jest określone przez dwa parametry. `dropoff_zero_intensity` jest prawdopodobieństwem zrzucenia punktów o zerowej intensywności. `dropoff_intensity_limit` jest progiem intensywności, powyżej którego punkty nie będą zrzucane. Prawdopodobieństwo zrzucenia punktu w zakresie jest liniową proporcją opartą na tych dwóch parametrach.

Dodatkowo, atrybut `noise_stddev` tworzy model szumu, aby symulować nieoczekiwane odchylenia, które pojawiają się w rzeczywistych czujnikach. Dla wartości dodatnich, każdy punkt jest losowo perturbowany wzdłuż wektora promienia lasera. W rezultacie otrzymujemy czujnik LIDAR z doskonałym pozycjonowaniem kątowym, ale zasumionym pomiarem odległości.



Rys. 2.11. Obraz sensora LIDAR.

Detektor przeszkód rejestruje zdarzenie za każdym razem, gdy aktor macierzysty ma przed sobą przeszkodę. W celu przewidywania przeszkód, czujnik tworzy przed pojazdem macierzystym kształt kapsuły i wykorzystuje go do sprawdzania, czy nie dochodzi do kolizji. Aby zapewnić wykrywanie kolizji z każdym rodzajem obiektu, serwer tworzy "fałszywych" aktorów dla elementów takich jak budynki lub krzewy, dzięki czemu można pobrać znacznik semantyczny w celu ich identyfikacji.



Rys. 2.12. Obraz sensora detekcji obiektów.

Sensor radaru jest to czujnik tworzący widok stożkowy, który jest tłumaczony na mapę punktową 2D elementów w zasięgu wzroku i ich prędkości względem czujnika. Może to być wykorzystane do kształtuowania elementów i oceny ich ruchu i kierunku. Ze względu na użycie współrzędnych biegunkowych, punkty będą skupione wokół

środka widoku.



Rys. 2.13. Obraz sensora radaru.

2.2.3. Pogoda

Pogoda nie jest klasą samą, lecz zbiorem parametrów dostępnych w świecie symulatora CARLA. Parametryzacja zawiera w sobie elementy takie jak położenie słońca, zachmurzenie, wiatr, mgłę, deszcz, śnieg a także wiele innych, aby umożliwić testowanie programu w każdych możliwych warunkach pogodowych. Aby zdefiniować własną pogodę używa się klasy pomocniczej `carla.WeatherrParameters`

```
1     weather = carla.WeatherParameters( cloudiness=80.0, precipitation=30.0,
2     sun_altitude_angle=70.0) world.set_weather(weather)
3     print(world.get_weather())
```

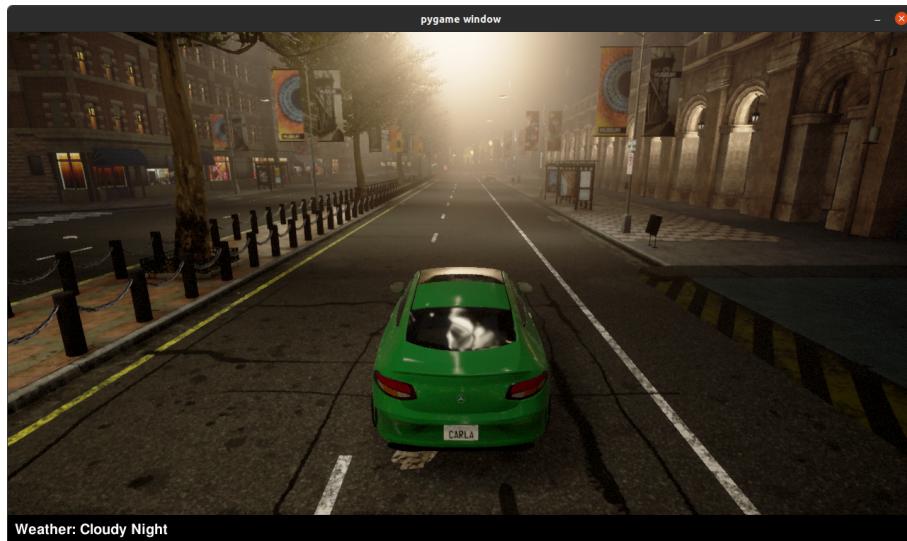
Listing 2.1. caption text



Rys. 2.14. Obraz przedstawiający lekki deszcz o zachodzie słońca.

Dostępne są również gotowe ustawienia pogody, które można bezpośrednio zastosować w symulacji jazdy samochodem. Lista zawierająca przygotowane warunki drogowe znajduje się w klasie `arla.WeathrrParameters`.

```
1 world.set_weather(carla.WeatherParameters.WetCloudySunset)
```

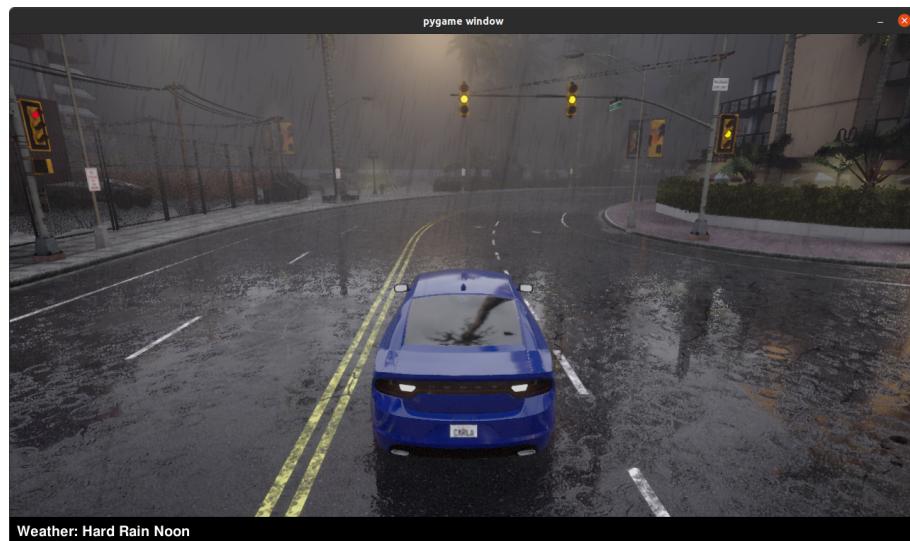


Rys. 2.15. Obraz przedstawiający zachmurzoną noc.

Istnieje również możliwość dostosowania pogody za pomocą dwóch skryptów zapewnione przez symulator CARLA. Są to:

- `environment.py` (in `PythonAPI/util`) — Zapewnia dostęp do parametrów pogodowych i świetlnych, dzięki czemu można je zmieniać w czasie rzeczywistym.

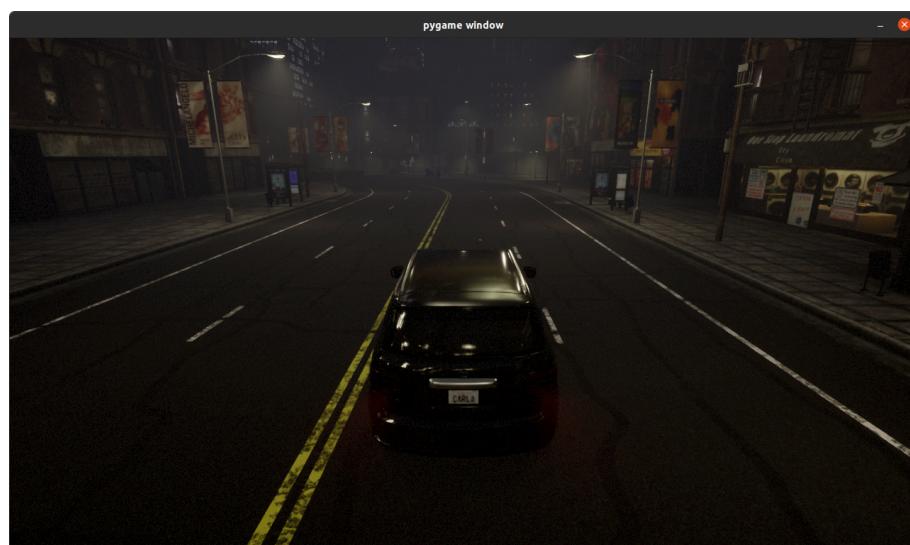
- `dynamic_weather.py` (in PythonAPI/examples) — Włącza określony cykl pogodowy przygotowany przez deweloperów dla każdej mapy CARLA.



Rys. 2.16. Obraz przedstawiający mocny deszcz w południe.

2.2.4. Oświetlenie

Światła uliczne włączają się automatycznie, gdy symulacja przechodzi w tryb nocny. Światła zostały umieszczone przez twórców mapy i są dostępne jako obiekty `carla.Light`. Właściwości takie jak kolor i natężenie światła mogą być dowolnie zmieniane. Zmienna `light_state` typu `carla.LightState` pozwala ustawić je wszystkie w jednym wywołaniu. Światła uliczne są kategoryzowane za pomocą ich atrybutu `light_group`, typu `carla.LightGroup`. Pozwala to na sklasyfikowanie światel jako światła uliczne, światła budynków... Aby obsłużyć grupy światel w jednym wywołaniu, można pobrać instancję `carla.LightManager`

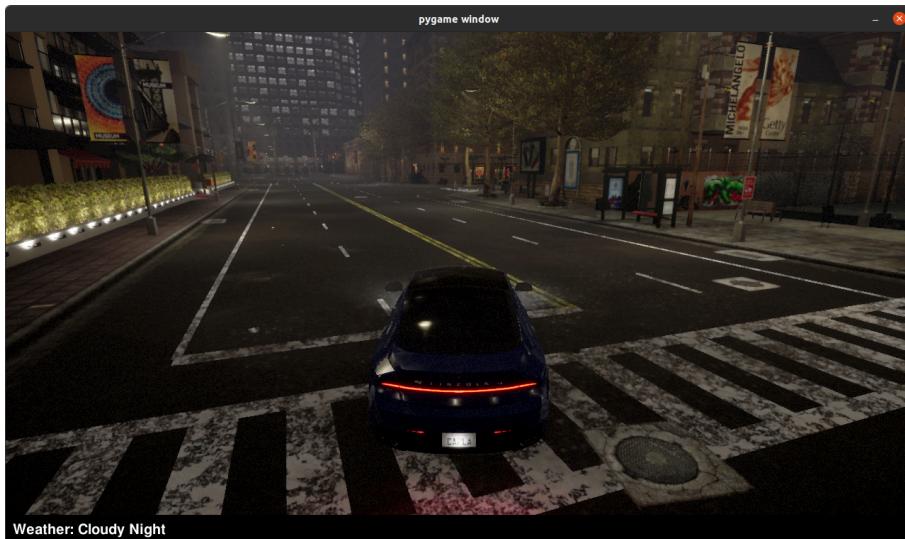


Rys. 2.17. Obraz przedstawiający światła uliczne.

Światła pojazdu muszą być włączane/wyłączane przez użytkownika. Każdy pojazd posiada zestaw świateł wymienionych w `carla.VehicleLightState`. Jak do tej pory, nie wszystkie pojazdy mają zintegrowane światła. Poniżej znajduje się lista tych, które są dostępne:

- `environment.py` (Rowery) — Wszystkie posiadają przednie i tylne światło pozycyjne.
- `dynamic_weather.py` (Motocykle) — Modele Yamaha i Harley Davidson.
- `dynamic_weather.py` (Samochody) - Audi TT, Chevrolet, Dodge (radiowóz), Audi e-tron, Lincoln, Mustang, Tesla 3S, Volkswagen T2 oraz nowi goście przybywający do CARLA.

Światła pojazdu mogą być pobierane i aktualizowane w dowolnym momencie za pomocą metod `carla.Vehicle.get_light_state` i `carla.Vehicle.set_light_state`. Używają one operacji binarnych, aby dostosować ustawienie świateł [3].



Rys. 2.18. Obraz przedstawiający światła pojazdu.

2.2.5. Reprezentacja obiektów w przestrzeni 3D i definicja Bounding Box

Symulator CARLA [3] oraz Unreal Engine 4 (UE4) stosują lewoskrętny układ współrzędnych kartezjański z osią Z skierowaną w górę (Z-up). Oznacza to:

- **Oś X:** skierowana do przodu (forward).
- **Oś Y:** skierowana w prawo (right).
- **Oś Z:** skierowana w górę (up).

Punkt $(0, 0, 0)$ to początek układu współrzędnych, zwany *origin*. Wartości dodatnie na osi X oznaczają ruch do przodu, na osi Y w prawo, a na osi Z w górę [4].

Jednostki miary

W obu środowiskach:

- **1 jednostka (unit)** odpowiada **1 centymetrowi** w rzeczywistości.
- **1 metr = 100 jednostek.**
- **1 kilometr = 100 000 jednostek.**
- **1 cal = 2,54 jednostki.**
- **1 stopa = 30,48 jednostki.**

Oznacza to, że zarówno w UE4, jak i w CARLA, 1 jednostka w grze to 1 cm w świecie rzeczywistym [5].

Pozycjonowanie obiektów i punkt (0, 0, 0)

W UE4 i CARLA:

- **Punkt (0, 0, 0):** znajduje się w centrum świata gry, zwykle w miejscu, gdzie zaczyna się scena.
- **Pozycjonowanie obiektów:** odbywa się poprzez określenie ich lokalizacji względem tego punktu, np. `actor.set_location(FVector(x, y, z))`.

Każdy obiekt dynamiczny (tzw. aktor), taki jak pojazd czy pieszy, posiada swoją pozycję w globalnym układzie współrzędnych świata, określoną względem punktu (0, 0, 0) mapy. Kluczowym elementem w procesie detekcji i generowania danych uczących jest zrozumienie, w jaki sposób fizyczna bryła obiektu jest reprezentowana numerycznie za pomocą prostopadłościanu ograniczającego, zwanego *bounding box* [3, 6].

2.2.5.1. Relacja między punktem Origin a środkiem geometrycznym

Każdy aktor w symulacji posiada swój punkt odniesienia, zwany *Origin* lub *Pivot Point*. W przypadku pojazdów punkt ten jest zazwyczaj zlokalizowany na poziomie podłożu, w połowie odległości między osiami kół, co ułatwia obliczenia fizyki jazdy. Należy jednak wyraźnie odróżnić punkt *Origin* aktora od środka jego obrysu (*Bounding Box Center*).

Struktura `carla.BoundingBox` definiuje geometrię obiektu za pomocą dwóch kluczowych wektorów:

1. **location:** Określa przesunięcie środka geometrycznego prostopadłościanu względem punktu *Origin* aktora. Wektor ten (x_c, y_c, z_c) jest wyrażony w lokalnym układzie współrzędnych pojazdu. Przykładowo, dla samochodu osobowego środek bryły znajduje się zazwyczaj wyżej (osi Z > 0) i może być przesunięty wzdłuż osi podłużnej względem osi kół.

2. **extent**: Jest to wektor (e_x, e_y, e_z) określający połowę wymiarów prostopadłoscianu wzdłuż każdej z osi lokalnych. Oznacza to, że całkowite wymiary obiektu wynoszą odpowiednio: długość $2 \cdot e_x$, szerokość $2 \cdot e_y$ oraz wysokość $2 \cdot e_z$.

Współrzędne wierzchołków bounding boxa nie są przechowywane wprost, lecz są obliczane dynamicznie na podstawie jego środka oraz wektora **extent**. Dla lokalnego układu odniesienia, wierzchołki V_{local} zdefiniowane są jako kombinacje dodawania i odejmowania wartości **extent** od środka **location** [6].

2.2.5.2. Transformacja do układu współrzędnych świata

Aby wyznaczyć położenie wierzchołków bounding boxa w globalnej przestrzeni 3D symulacji, konieczne jest wykonanie transformacji macierzowej. Proces ten uwzględnia aktualną pozycję i rotację pojazdu na mapie. Wykorzystywana jest do tego macierz transformacji M_{actor} , która składa się z translacji (pozycji aktora względem punktu 0,0,0 świata) oraz rotacji (kątów *roll*, *pitch*, *yaw*).

Położenie dowolnego wierzchołka V_{world} w przestrzeni świata obliczane jest zgodnie z zależnością:

$$V_{world} = M_{actor} \cdot V_{local} \quad (2.1)$$

Gdzie V_{local} to współrzędna wierzchołka w układzie lokalnym pojazdu (uwzględniająca przesunięcie **location** i wymiar **extent**). Operacja ta pozwala na precyzyjne umiejscowienie obrysu pojazdu w świecie 3D, niezależnie od jego orientacji czy pochylenia wynikającego z fizyki zawieszenia [3].

2.2.5.3. Rzutowanie z przestrzeni 3D na płaszczyznę obrazu 2D

Ostatnim etapem, niezbędnym do wygenerowania danych referencyjnych dla algorytmu YOLO (tzw. *ground truth*), jest rzutowanie trójwymiarowych wierzchołków bounding boxa na dwuwymiarową płaszczyznę obrazu z kamery. Proces ten, realizowany m.in. w skrypcie `bounding_boxes.py`, wymaga znajomości parametrów wewnętrznych i zewnętrznych kamery.

Rzutowanie odbywa się w dwóch krokach:

1. **Transformacja Świat → Kamera**: Punkty ze świata 3D są przeliczane do układu współrzędnych kamery przy użyciu macierzy *World-to-Camera* (będącej odwrotnością transformacji kamery w świecie).
2. **Projekcja perspektywiczna**: Punkty z układu kamery są rzutowane na płaszczyznę obrazu przy użyciu macierzy kalibracji K (macierz parametrów wewnętrznych), która zależy od rozdzielczości obrazu oraz kąta widzenia (FOV).

Wzór opisujący projekcję punktu $P_{3D} = [x, y, z]^T$ na punkt obrazu $p_{2D} = [u, v]^T$ (we współrzędnych jednorodnych) przyjmuje postać:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \cdot [R|t] \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (2.2)$$

Gdzie K to macierz wewnętrzna kamery, $[R|t]$ to macierz transformacji z układu świata do układu kamery, a s to współczynnik skalujący. Ostateczny prostokąt 2D (bbox 2D) wyznaczany jest poprzez znalezienie minimalnych i maksymalnych wartości współrzędnych (u, v) spośród wszystkich 8 rzutowanych wierzchołków bryły 3D. Takie podejście gwarantuje, że wygenerowany obrys 2D w pełni obejmuje widoczny obiekt na zdjęciu [6].

2.3. Instalacja symulatora Carla

2.3.1. Wymagania sprzętowe

Rekomendowana specyfikacja komputera dla najnowszej wersji symulatora CARLA 0.9.13 prezentuje się następująco:

- Procesor Intel i7 gen 9th - 11th / Intel i9 gen 9th - 11th / AMD ryzen 7 / AMD ryzen 9
- Ilość pamięci RAM +16 GB pamięci RAM
- Ilość miejsca na dysku 130GB
- Karta graficzna najlepiej z 6GB lub 8GB pamięci VRAM np. NVIDIA RTX 2070 / NVIDIA RTX 2080 / NVIDIA RTX 3070, NVIDIA RTX 3080 lub nowsze
- System operacyjny Ubuntu 18.04/ 20.04/ 22.04/ Windows 10

2.3.2. Instalacja i konfiguracja dla systemu Ubuntu

Pierwszym etapem instalacji CARLA jest przygotowanie systemu. W pierwszej kolejności system powinien zostać zaktualizowany, a niezbędne pakiety zainstalowane:

```

1      sudo apt update
2      sudo apt upgrade

```

Listing 2.2. Aktualizacja systemu

Następnie muszą zostać zainstalowane zależności wymagane przez CARLA:

```

1      sudo apt install build-essential clang cmake git libcurl4-openssl-dev
          libssl-dev \
2      libsqlite3-dev libudev-dev pkg-config python3-dev python3-pip \
3      python3-setuptools python3-wheel qt5-qmake qtbase5-dev libqt5core5a \
4      libqt5gui5 libqt5widgets5 libprotobuf-dev protobuf-compiler \
5      libsdl2-dev libpng-dev libjpeg-dev libtiff-dev libgtk-3-dev \
6      libassimp-dev libblas-dev liblapack-dev libboost-all-dev \
7      libopenblas-dev libatlas-base-dev libeigen3-dev

```

Listing 2.3. Instalacja zależności

Dodatkowo, konieczne jest zainstalowanie języka Python oraz menedżera pakietów PIP:

```

1      sudo apt install python3 python3-pip
2      python3 -m pip install --upgrade pip

```

Listing 2.4. Instalacja Pythona i PIP

2.3.3. Instalacja Unreal Engine

Ze względu na to, że CARLA jest oparta na silniku Unreal Engine, konieczne jest jego zainstalowanie. Proces ten może zostać przeprowadzony za pomocą Epic Games Launcher [7].

1. Należy założyć konto na stronie <https://www.epicgames.com/>.
2. Następnie należy pobrać i zainstalować Epic Games Launcher.
3. Po instalacji wymagane jest zalogowanie się i przejście do zakładki **Library**.
4. Należy wybrać odpowiednią wersję Unreal Engine (zalecana 4.27 lub nowsza) i przeprowadzić instalację.
5. Po zakończeniu instalacji Unreal Engine powinien zostać uruchomiony za pośrednictwem Epic Games Launcher.

2.3.4. Pobranie źródeł CARLA i komplikacja

Aby pobrać CARLA, repozytorium musi zostać sklonowane z GitHub:

```

1      cd ~
2      git clone https://github.com/carla-simulator/carla.git
3      cd carla

```

Listing 2.5. Pobieranie źródeł CARLA

Kolejnym krokiem jest przejście do katalogu zawierającego projekt Unreal Engine dla CARLA:

```
1      cd Unreal/CarlaUE4
```

Listing 2.6. Przechodzenie do folderu Unreal Engine dla CARLA

Następnie konieczne jest uruchomienie Unreal Engine w celu konfiguracji projektu:

```
1      ~/UnrealEngine/Engine/Binaries/Linux/UE4Editor CarlaUE4.uproject
```

Listing 2.7. Uruchamianie Unreal Engine

Po uruchomieniu Unreal Engine należy wybrać opcję **Generate Visual Studio project files** i zamknąć edytor.

W kolejnym kroku wymagane jest powrót do terminala i przeprowadzenie komplikacji:

```
1      make
```

Listing 2.8. Kompilacja CARLA

2.3.5. Uruchomienie symulatora CARLA

Po zakończeniu komplikacji symulator CARLA może zostać uruchomiony:

```
1 cd ~/carla/Unreal/CarlaUE4/Binaries/Linux
2 ./CarlaUE4
```

Listing 2.9. Uruchamianie CARLA

Po pomyślnym uruchomieniu symulatora powinien ukazać się obraz mapy domyślnej widocznym na Rys. 2.1. Korzystanie z API CARLA w Pythonie Można rozpocząć poprzez instalację odpowiednich pakietów:

```
1 pip3 install carla
```

Listing 2.10. Instalacja zależności Pythona

Aby zweryfikować poprawność działania, można uruchomić przykładowy skrypt Pythona:

```
1 cd ~/carla/PythonAPI/examples
2 python spawn_npc.py
```

Listing 2.11. Uruchamianie przykładowego skryptu

Jeżeli wszystkie kroki zostały wykonane poprawnie, symulator CARLA powinien działać, a skrypt Python powinien uruchomić się bez problemów.

2.4. Skrypt do manualnego sterowania w symulatorze CARLA

Plik `manual_control.py` w symulatorze CARLA jest skryptem umożliwiającym ręczne sterowanie pojazdem za pomocą klawiatury oraz myszy. Stanowi on jedno z podstawowych narzędzi do testowania, eksploracji oraz analizy środowiska symulacyjnego, dostarczając użytkownikowi pełną kontrolę nad pojazdem w czasie rzeczywistym.

2.4.1. Struktura skryptu

Plik `manual_control.py` został zaprojektowany modularnie i obejmuje kilka kluczowych sekcji, z których każda realizuje specyficzny etap działania aplikacji:

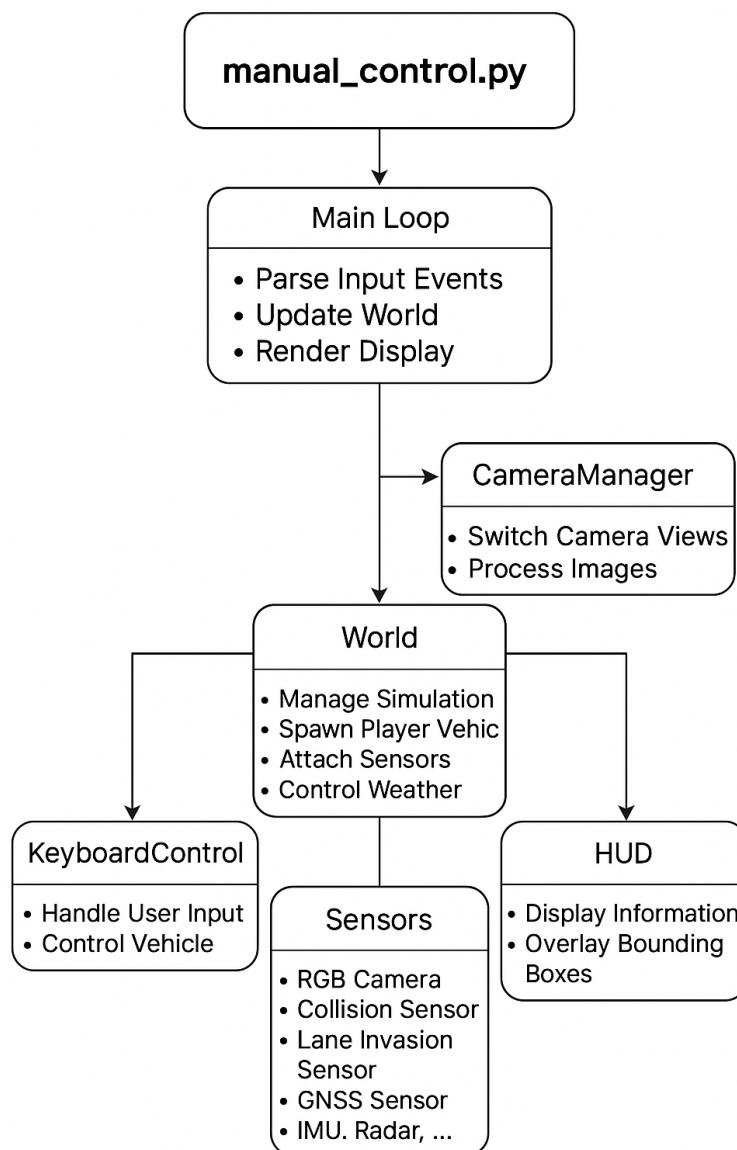
- **Importowanie bibliotek oraz konfiguracja klienta CARLA** – inicjalizacja połączenia z serwerem symulacji oraz załadowanie niezbędnych bibliotek zewnętrznych, takich jak `pygame` czy `carla`.
- **Inicjalizacja środowiska i pojazdu** – stworzenie świata symulacyjnego, pojazdu oraz połączenie odpowiednich czujników.
- **Obsługa wejścia użytkownika** – monitorowanie urządzeń wejściowych (klawiatura, mysz) za pomocą biblioteki `pygame`.
- **Główna pętla gry (game loop)** – ciągłe przetwarzanie zdarzeń, aktualizacja stanu świata oraz renderowanie obrazu.

- **Zakończenie i czyszczenie zasobów** – prawidłowe usunięcie wszystkich aktorów i zwolnienie zasobów systemowych.

Każdy z tych etapów pełni fundamentalną rolę w zapewnieniu poprawnej i wydajnej pracy symulacji.

2.4.2. Logika działania pliku `manual_control.py`

Plik `manual_control.py` realizuje pełny cykl życia aplikacji symulacyjnej: od inicjalizacji środowiska, przez obsługę sterowania ręcznego, aż po prawidłowe zakończenie symulacji. Jego struktura została przedstawiona na schemacie blokowym (rysunek 2.19).



Rys. 2.19. Schemat blokowy działania pliku `manual_control.py`

Główne komponenty systemu:

- **Main Loop** – odpowiada za przetwarzanie zdarzeń wejściowych, aktualizację stanu świata oraz renderowanie grafiki.
- **CameraManager** – umożliwia zarządzanie widokami kamer i przetwarzanie danych z czujników wizualnych.
- **World** – zarządza całym światem symulacji, w tym pojazdem użytkownika, pogodą oraz czujnikami.
- **KeyboardControl** – przetwarza sygnały z klawiatury i przekłada je na konkretne komendy sterujące pojazdem.
- **Sensors** – obejmuje wszystkie dostępne czujniki w symulatorze (kamera RGB, czujniki kolizji, lane invasion sensor, GNSS, IMU, radar).
- **HUD (Heads-Up Display)** – odpowiedzialny za prezentowanie informacji o stanie symulacji w formie graficznej.

Komunikacja między tymi komponentami odbywa się w sposób asynchroniczny, zapewniając płynne działanie symulacji oraz wysoką responsywność interfejsu użytkownika.

2.4.3. Pętla gry (Game Loop)

Główna pętla symulacyjna, realizowana w funkcji `game_loop()`, odpowiedzialna jest za cykliczne aktualizowanie stanu środowiska oraz reagowanie na zdarzenia wejściowe. Jej główne zadania obejmują:

1. **Odczyt stanu klawiatury** – umożliwia wykrywanie naciśniętych klawiszy w czasie rzeczywistym.
2. **Analizę wejścia użytkownika** – wywołanie funkcji `parse_control_input()` celem przetworzenia danych wejściowych.
3. **Zastosowanie kontroli do pojazdu** – przekazanie utworzonych komend sterujących do pojazdu poprzez metodę `apply_control()`.
4. **Synchronizację klatek** – zapewnienie płynności animacji przy użyciu `clock.tick(30)`, co ustala stałą liczbę 30 FPS.

Przykładowy kod realizacji pętli gry:

```

1   def main():
2       client = carla.Client('localhost', 2000)
3       world = client.get_world()
4       control = carla.VehicleControl()
5
6       pygame.init()
7       clock = pygame.time.Clock()
8
9       try:
10           while True:
11               keys = pygame.key.get_pressed()

```

```

12     control = parse_control_input(keys)
13
14     vehicle = world.get_actors().filter('vehicle.*')[0]
15     vehicle.apply_control(control)
16
17     clock.tick(30)
18     except KeyboardInterrupt:
19         pass

```

Listing 2.12. Przykład kodu pętli gry - Game Loop

2.4.4. Tryb asynchronouszny i synchroniczny

Symulator CARLA umożliwia uruchamianie środowiska symulacyjnego w dwóch podstawowych trybach: asynchronousznym oraz synchronicznym. Zrozumienie ich różnic i sposobu implementacji w kodzie źródłowym, a szczególnie w kontekście pętli głównej symulacji, ma kluczowe znaczenie dla skutecznej integracji z zewnętrznymi systemami przetwarzania, takimi jak np. algorytmy detekcji obiektów w czasie rzeczywistym. W trybie **asynchronouszny**, który jest domyślnym ustawieniem symulatora, świat gry jest aktualizowany niezależnie od klienta. Oznacza to, że symulacja działa z własną częstotliwością i nie oczekuje na sygnały od zewnętrznych aplikacji. Dane z sensorów, takich jak kamery czy lidary, mogą być więc niespójne względem stanu symulacji w momencie ich odczytu, co może skutkować niedokładnościami podczas analizy lub uczenia modeli [3].

W przeciwnieństwie do tego, tryb **synchroniczny** pozwala klientowi kontrolować tempo symulacji. W tym trybie każda klatka symulacji jest generowana dopiero po wywołaniu metody `world.tick()` przez klienta. Dzięki temu możliwe jest pełne zsynchronizowanie stanu świata z danymi z sensorów, co jest niezbędne w wielu zastosowaniach wymagających precyzji, takich jak testowanie algorytmów percepji w pojazdach autonomicznych.

2.4.5. Implementacja synchronizacji w `manual_control.py`

Tryb synchroniczny jest aktywowany w pliku `manual_control.py` poprzez odpowiednią konfigurację ustawień świata gry. Kluczowy fragment kodu odpowiedzialny za aktywację tego trybu przedstawia się następująco:

```

settings = world.get_settings()
settings.synchronous_mode = True
settings.fixed_delta_seconds = 0.05
world.apply_settings(settings)

```

Ustawienie parametru `synchronous_mode` na `True` oraz określenie `fixed_delta_seconds` (np. 0.05 s odpowiadające 20 FPS) pozwala osiągnąć spójne, deterministyczne zachowanie symulacji. W przypadku uruchomienia skryptu z parametrem `--sync`, aktywowany zostaje synchroniczny mechanizm sterowania symulacją.

2.4.6. Analiza pętli głównej game_loop()

Główna pętla symulacji, zdefiniowana w funkcji `game_loop()`, realizuje krokową kontrolę nad światem gry. Przykładowy fragment tej pętli wygląda następująco:

```
while True:  
    if args.sync:  
        sim_world.tick()  
        clock.tick_busy_loop(60)  
        if controller.parse_events(client, world, clock, args.sync):  
            return  
        world.tick(clock)  
        world.render(display)  
        pygame.display.flip()
```

Jej działanie można rozłożyć na kilka kroków:

1. **Synchronizacja z serwerem:** Jeśli aktywowany został tryb synchroniczny (`args.sync`), klient wywołuje metodę `sim_world.tick()`, co inicjuje przetwarzanie jednej klatki symulacji po stronie serwera.
2. **Ograniczenie liczby klatek:** Funkcja `clock.tick_busy_loop(60)` ogranicza liczbę iteracji pętli do 60 na sekundę, a także oblicza czas między kolejnymi iteracjami.
3. **Przetwarzanie zdarzeń wejściowych:** Metoda `parse_events(...)` obsługuje zdarzenia Pygame, takie jak naciśnięcie klawiszy, zamknięcie aplikacji itp. Jeśli użytkownik zainicjuje zakończenie programu, pętla zostaje przerwana.
4. **Aktualizacja stanu lokalnego:** Wywołanie `world.tick(clock)` służy do wewnętrznej aktualizacji komponentów klienta – takich jak interfejs użytkownika (HUD), kamery czy dane z sensorów. W trybie synchronicznym nie powoduje ono przesunięcia świata gry, lecz jedynie uzupełnia dane już wygenerowane przez `sim_world.tick()`.
5. **Renderowanie:** Funkcja `world.render(display)` przygotowuje obraz do wyświetlenia, a `pygame.display.flip()` aktualizuje ekran, pokazując najnowszą ramkę wraz z interfejsem graficznym.

Dzięki takiej konstrukcji pętli możliwa jest pełna kontrola nad przebiegiem symulacji. Co istotne, w trybie synchronicznym to klient dyktuje tempo, co jest kluczowe w zastosowaniach związanych z uczeniem maszynowym i testowaniem detekcji obiektów (np. YOLO), gdzie wymagana jest spójność między stanem środowiska a danymi sensorycznymi.

2.4.7. Znaczenie dla integracji z algorytmami zewnętrznyimi

Zastosowanie trybu synchronicznego jest szczególnie istotne w kontekście integracji z zewnętrznymi algorytmami analizy danych, np. systemami opartymi o YOLO. W takich przypadkach przetwarzanie obrazu musi być zgodne ze stanem symulacji, aby wykrywane obiekty odpowiadały ich rzeczywistej pozycji i prędkości. Brak synchronizacji może prowadzić do błędnych detekcji i zaburzenia procesu wnioskowania.

2.4.8. Sterowanie pojazdem z poziomu klawiatury

Jednym z głównych elementów interakcji użytkownika z pojazdem w symulatorze CARLA jest sterowanie przy pomocy klawiatury. W tym celu skrypt `manual_control.py` wykorzystuje bibliotekę `pygame`, która pozwala na obsługę wejścia użytkownika i monitorowanie stanu klawiszy. Główna funkcja odpowiedzialna za to zadanie to `parse_control_input()`, która analizuje naciśnięte klawisze i na ich podstawie dostosowuje parametry sterowania pojazdem.

Przykładowy kod do obsługi sterowania pojazdem:

```

1 import pygame
2
3 def parse_control_input():
4     keys = pygame.key.get_pressed() # Pobranie stanu klawiszy
5     control = carla.VehicleControl() # Obiekt sterowania
6         pojazdem
7
8     if keys[pygame.K_w]: # Przyspieszanie (gaz)
9         control.throttle = 1.0
10    if keys[pygame.K_s]: # Hamowanie
11        control.brake = 1.0
12    if keys[pygame.K_a]: # Skręt w lewo
13        control.steer = -1.0
14    if keys[pygame.K_d]: # Skręt w prawo
15        control.steer = 1.0
16
17    return control # Zwrócenie obiektu sterowania

```

Listing 2.13. Przykładowy kod w języku Python odpowiedzialny za sterowanie pojazdem z klawiatury

Opis kodu:

- **Pobranie stanu klawiszy:** Funkcja `pygame.key.get_pressed()` zwraca tablicę wartości logicznych dla wszystkich klawiszy, umożliwiając interakcję z pojazdem. Wartość `True` oznacza, że dany klawisz jest wciśnięty.
- **Tworzenie obiektu sterowania:** Obiekt `carla.VehicleControl()` jest używany do przekazywania parametrów sterowania do symulowanego pojazdu. Wśród właściwości tego obiektu znajdują się:
 - `throttle` – stopień przyspieszenia (wartość od 0.0 do 1.0),
 - `brake` – stopień hamowania (wartość od 0.0 do 1.0),
 - `steer` – kąt skrętu (wartość od -1.0 do 1.0, gdzie -1.0 oznacza skręt w lewo, a 1.0 w prawo).
- **Sterowanie pojazdem:** Na podstawie wciśniętych klawiszy funkcja ustawia odpowiednie wartości w obiekcie `control`. Na przykład:
 - W powoduje przyspieszanie pojazdu (`throttle = 1.0`),
 - S uruchamia hamowanie (`brake = 1.0`),
 - A i D sterują kątem skrętu.

- **Zwrócenie obiektu sterowania:** Po przetworzeniu stanu klawiszy funkcja zwraca obiekt `control`, który jest następnie używany do sterowania pojazdem w symulacji.

2.4.9. Pętla gry - Game Loop

W symulatorze CARLA plik `manual_control.py` zawiera funkcję `game_loop()`, która pełni rolę głównej pętli symulacyjnej, odpowiedzialnej za interakcję użytkownika ze światem symulowanym oraz za aktualizację i renderowanie otoczenia w czasie rzeczywistym. Funkcja ta inicjalizuje wszystkie niezbędne komponenty środowiska, nawiązuje połączenie z serwerem CARLA, a następnie wykonuje ciągłą iterację (pętlę nieskończoną), aż do zakończenia programu przez użytkownika. W przypadku symulatora CARLA pętla gry wykonuje następujące kroki:

1. **Pobranie stanu klawiszy:** Funkcja `pygame.key.get_pressed()` umożliwia monitorowanie stanu wszystkich klawiszy na klawiaturze, co pozwala na kontrolowanie pojazdu. Odczyt stanu klawiszy jest kluczowy do sterowania ruchem pojazdu (np. przyspieszanie, hamowanie, skręt w lewo lub prawo).
 2. **Przetwarzanie wejścia użytkownika:** Funkcja `parse_control_input(keys)` analizuje stan klawiszy i na tej podstawie ustawia odpowiednie parametry sterowania pojazdem. Na przykład, jeśli użytkownik wciśnie klawisz W, wartość `throttle` zostaje ustawiona na 1.0, co powoduje przyspieszenie pojazdu.
 3. **Zastosowanie kontroli do pojazdu:** Funkcja `vehicle.apply_control(control)` wysyła skonfigurowany obiekt sterowania do pojazdu, aby ten wykonał odpowiednią akcję (np. przyspieszenie, hamowanie, skręt).
 4. **Kontrola liczby klatek na sekundę:** Funkcja `clock.tick(30)` zapewnia, że symulacja nie będzie działała za szybko, umożliwiając płynne sterowanie pojazdem i zachowanie właściwej synchronizacji w grze.

Przykładowy fragment kodu pętli gry:

```
1 def main():
2     client = carla.Client('localhost', 2000)    # Inicjalizacja
3         klienta CARLA
4     world = client.get_world()    # Pobranie świata symulacji
5     control = carla.VehicleControl()    # Obiekt sterowania
6         pojazdem
7
8     pygame.init()    # Inicjalizacja pygame
9     clock = pygame.time.Clock()    # Inicjalizacja zegara dla
10        kontrolowania FPS
11
12    try:
13        while True:
14            keys = pygame.key.get_pressed()    # Pobieranie stanu
15                klawiszy
16            control = parse_control_input(keys)    # Analiza wejścia od
17                użytkownika
```

```

14     vehicle = world.get_actors().filter('vehicle.*')[0]    #
15         Pobranie pojazdu
16     vehicle.apply_control(control)   # Zastosowanie kontroli do
17         pojazdu
18
19     clock.tick(30)   # Ograniczenie liczby klatek na sekundę
20     except KeyboardInterrupt:
21         pass

```

Listing 2.14. Przykład kodu pętli gry - Game Loop

2.4.10. Sterowanie pojazdem za pomocą klawiatury

Sterowanie pojazdem odbywa się głównie przy pomocy klawiatury, wykorzystując bibliotekę `pygame` do obsługi zdarzeń wejściowych. Główną funkcją odpowiedzialną za przetwarzanie sygnałów od użytkownika jest `parse_control_input()`, która analizuje aktualny stan klawiatury i przekłada go na komendy sterujące pojazdem.

```

1      import pygame
2
3      def parse_control_input():
4          keys = pygame.key.get_pressed()
5          control = carla.VehicleControl()
6
7          if keys[pygame.K_w]:
8              control.throttle = 1.0
9          if keys[pygame.K_s]:
10             control.brake = 1.0
11             if keys[pygame.K_a]:
12                 control.steer = -1.0
13             if keys[pygame.K_d]:
14                 control.steer = 1.0
15
16      return control

```

Listing 2.15. Przykładowy kod obsługi sterowania pojazdem z klawiatury

Opis kodu:

- **Pobieranie stanu klawiszy** – funkcja `pygame.key.get_pressed()` umożliwia detekcję aktualnie wciśniętych klawiszy.
- **Tworzenie obiektu sterowania** – `carla.VehicleControl` umożliwia ustawienie parametrów takich jak przyspieszenie, hamowanie czy skręt.
- **Ustawianie parametrów** – przypisanie odpowiednich wartości do właściwości `throttle`, `brake` i `steer` w zależności od interakcji użytkownika.
- **Zwrócenie obiektu** – gotowy obiekt sterowania zostaje przesłany do funkcji sterującej pojazdem w świecie symulacyjnym.

2.4.11. Czyszczenie zasobów (Cleanup)

Po zakończeniu działania symulacji niezbędne jest prawidłowe usunięcie wszystkich aktorów, aby zwolnić pamięć oraz uniknąć potencjalnych błędów.

Proces ten obejmuje:

- **Pobranie aktorów** – funkcja `world.get_actors()` zwraca wszystkie aktywne obiekty w świecie.
- **Zniszczenie aktorów** – poprzez iteracyjne wywoływanie metody `destroy()` na każdym z aktorów.

Przykładowy kod funkcji czyszczącej:

```
1     def cleanup(world):
2         actors = world.get_actors()
3         for actor in actors:
4             actor.destroy()
```

Listing 2.16. Kod funkcji odpowiedzialnej za czyszczenie zasobów

Poprawne czyszczenie zasobów zapewnia optymalną wydajność symulacji i umożliwia jej wielokrotne uruchamianie bez ryzyka narastania błędów pamięciowych.

Rozdział 3

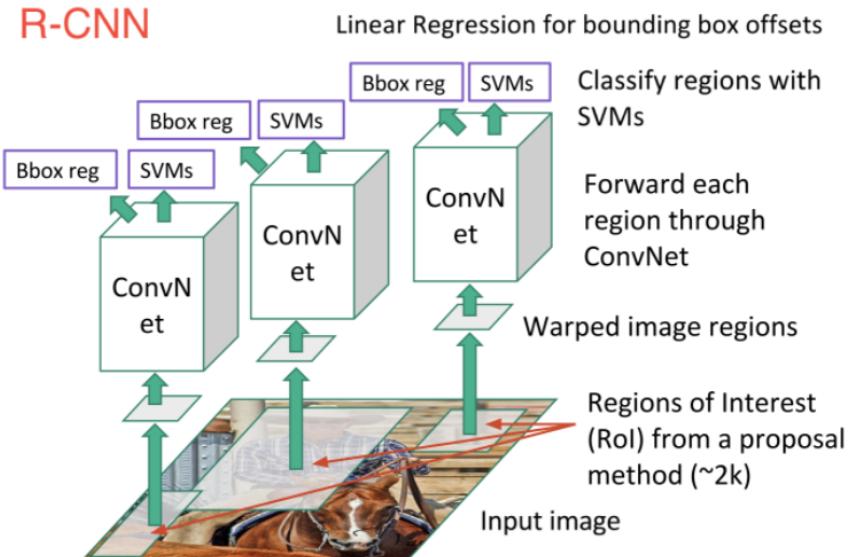
System rozpoznawania obrazów

Rozpoznawanie obrazów to dziedzina komputerowego przetwarzania danych, której celem jest identyfikacja i klasyfikacja obiektów w obrazach cyfrowych. W ostatnich latach, dzięki rozwojowi głębokiego uczenia (deep learning) oraz sieci neuronowych, osiągnięto znaczący postęp w tej dziedzinie.

Sieci neuronowe to modele matematyczne inspirowane strukturą ludzkiego mózgu, składające się z połączonych ze sobą neuronów (węzłów), które przetwarzają informacje. Głębokie uczenie odnosi się do sieci neuronowych o wielu warstwach (tzw. głębokich sieci), które potrafią uczyć się reprezentacji danych na różnych poziomach abstrakcji. W kontekście rozpoznawania obrazów, najczęściej stosuje się konwolucyjne sieci neuronowe (Convolutional Neural Networks, CNN), które są szczególnie efektywne w analizie danych obrazowych.

Poniżej przedstawiono **przegląd kluczowych architektur** stosowanych w rozpoznawaniu obrazów, wraz z ich charakterystyką i przykładami.

- **R-CNN (Region-based Convolutional Neural Networks)** to jedna z pierwszych skutecznych metod detekcji obiektów, oparta na analizie wybranych regionów obrazu. W pierwszym etapie stosowany jest algorytm *Selective Search*, który generuje około 2000 propozycji obszarów potencjalnie zawierających obiekty, uwzględniając cechy takie jak kolor, tekstura czy kształt. Każdy region jest następnie skalowany i przetwarzany przez sieć konwolucyjną (CNN), co pozwala na ekstrakcję cech wizualnych. Ostatecznie, klasyfikator SVM decyduje o przynależności danego regionu do konkretnej klasy, a regresor liniowy doprecyzowuje współrzędne ramki ograniczającej. [8]



Rys. 3.1. Schemat działania R-CNN.

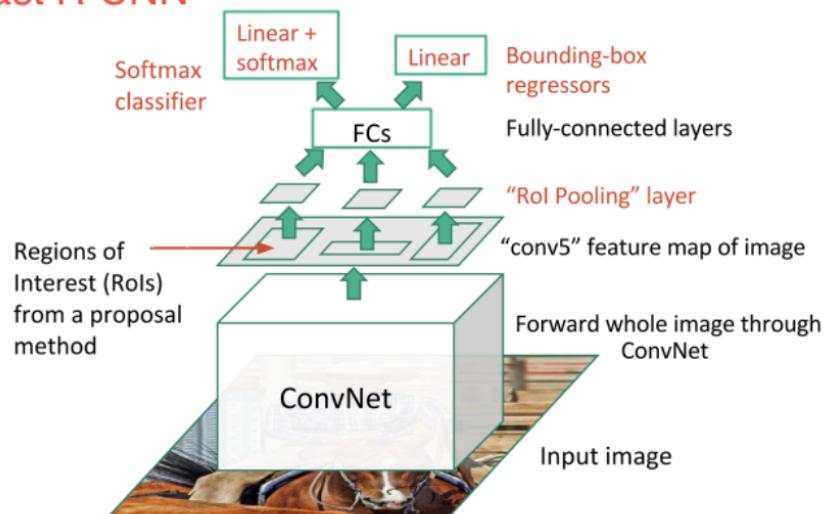
Choć metoda ta cechuje się wysoką precyją detekcji, jej główną wadą pozostaje duże zapotrzebowanie obliczeniowe związane z koniecznością analizowania tysięcy regionów dla każdego obrazu.

- **Fast R-CNN** to ulepszona wersja modelu R-CNN, zaprojektowana z myślą o zwiększeniu wydajności i redukcji czasochłonności procesu detekcji obiektów. Główne usprawnienie polega na zastosowaniu jednej sieci konwolucyjnej (CNN) do ekstrakcji cech z całego obrazu przed wygenerowaniem propozycji regionów, co eliminuje konieczność wielokrotnego przetwarzania tych samych obszarów.

Zamiast klasyfikatora SVM, Fast R-CNN wykorzystuje wbudowaną warstwę softmax do rozpoznawania klas obiektów oraz regresję do dokładnej lokalizacji ramek ograniczających. Regiony zainteresowania (RoI) są przekształcane za pomocą operacji *RoI Pooling*, która dostosowuje ich wymiary do jednolitego formatu wejściowego dla dalszej klasyfikacji.

Dzięki integracji tych rozwiązań Fast R-CNN znacznie skraca czas przetwarzania i umożliwia efektywne wykrywanie obiektów przy zachowaniu wysokiej dokładności. [9]

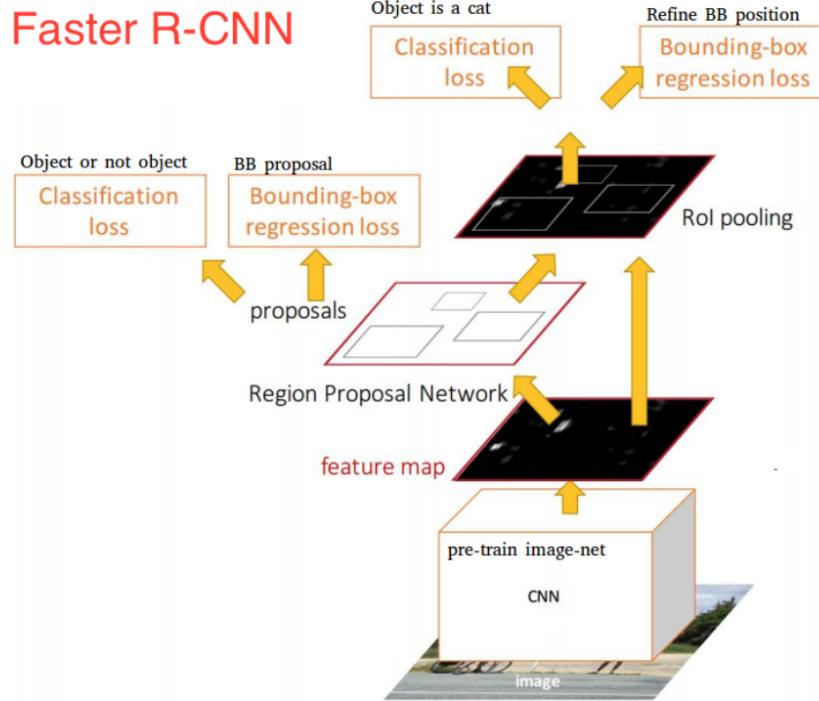
Fast R-CNN



Rys. 3.2. Schemat działania Fast R-CNN.

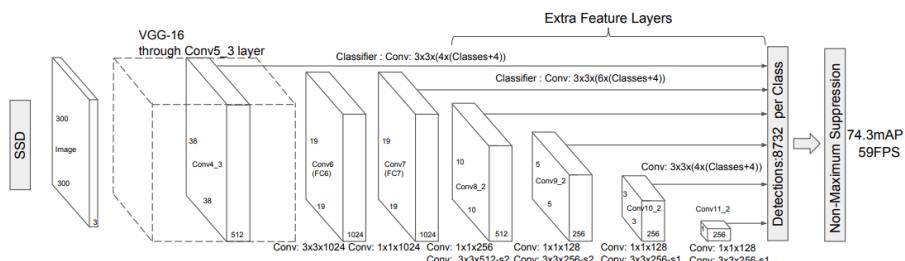
- **Faster R-CNN** to zaawansowana architektura detekcji obiektów, która integruje Sieć Generującą Propozycje Regionów (Region Proposal Network, RPN) z główną siecią konwolucyjną. Kluczową innowacją jest eliminacja potrzeby korzystania z zewnętrznych algorytmów generujących propozycje regionów, co znacznie skraca czas detekcji i zwiększa efektywność [8].

Cały obraz jest najpierw przetwarzany przez CNN w celu uzyskania map cech. Następnie RPN generuje regiony zainteresowania bezpośrednio na podstawie tych map, przewidując zarówno ich pozycje, jak i prawdopodobieństwo zawierania obiektów. Regiony te są klasyfikowane oraz doprecyzowywane przez końcowy moduł sieci. Dzięki współdzieleniu warstw konwolucyjnych pomiędzy RPN a modułem klasyfikacyjnym, Faster R-CNN oferuje wysoką dokładność przy znacznie lepszej wydajności niż jego poprzednicy.



Rys. 3.3. Schemat działania Faster R-CNN.

- **SSD (Single Shot MultiBox Detector)** to jednowarstwowa architektura detekcji obiektów, która, w przeciwieństwie do metod opartych na regionach, takich jak R-CNN, wykonuje detekcję i klasyfikację obiektów w jednym kroku. Dzięki temu SSD jest szybsza i bardziej efektywna, co czyni ją odpowiednią do zastosowań w czasie rzeczywistym.



Rys. 3.4. Schemat architektury SSD [10].

SSD wykorzystuje konwolucyjne sieci neuronowe (CNN) do ekstrakcji cech obrazu i jednocześnie predykcji położenia oraz klasy obiektów na wielu skalach. Model składa się z:

- **Sieci bazowej** – najczęściej stosuje się architekturę VGG16 bez w pełni połączonych warstw,
- **Dodatkowych warstw konwolucyjnych** – umożliwiają one detekcję obiektów na różnych poziomach szczegółowości,

- **Mechanizmu MultiBox** – generuje wiele ramki ograniczających (bounding boxes) o różnych proporcjach i rozmiarach,
- **Funkcji straty** – składa się z dwóch komponentów: błędu klasyfikacji (cross-entropy loss) oraz błędu lokalizacji (smooth L1 loss).

Dzięki swojej szybkości i efektywności SSD znajduje zastosowanie w wielu dziedzinach, takich jak:

- Systemy monitoringu wizyjnego,
- Rozpoznawanie obiektów w autonomicznych pojazdach,
- Aplikacje rzeczywistości rozszerzonej (AR),
- Systemy wspomagania kierowcy (ADAS).
- **YOLO (You Only Look Once)** to kolejna jednowarstwowa architektura, która traktuje detekcję obiektów jako problem regresji, przewidując bezpośrednio klasy i położenie obiektów w obrazie. Dzięki temu YOLO osiąga bardzo wysoką szybkość detekcji, co jest istotne w aplikacjach wymagających przetwarzania w czasie rzeczywistym.
- **DETR (Detection Transformer)** to nowoczesna architektura detekcji obiektów oparta na transformatorach, która integruje mechanizmy uwagi (attention mechanisms) w procesie detekcji. DETR eliminuje potrzebę stosowania tradycyjnych metod generowania propozycji regionów, co upraszcza proces detekcji i pozwala na bardziej efektywne wykorzystanie danych.

Schemat działania DETR Źródło: End-to-End Object Detection with Transformers

Poniższa tabela przedstawia **porównanie omówionych architektur** pod względem szybkości i dokładności detekcji:

Architektura	Szybkość (FPS)	Dokładność (mAP)	Zastosowanie
R-CNN	1 FPS	Wysoka	Analiza offline
Fast R-CNN	~2-3 FPS	Wysoka	Analiza offline
Faster R-CNN	~5-10 FPS	Bardzo wysoka	Zastosowania wymagające dokładności
SSD	~20-60 FPS	Średnia	Zastosowania w czasie rzeczywistym
YOLO	~45-150 FPS	Wysoka	Wykrywanie w czasie rzeczywistym
DETR	~10-20 FPS	Bardzo wysoka	Nowoczesne zastosowania AI

Tab. 3.1. Porównanie wybranych architektur

Rozpoznawanie obrazów z wykorzystaniem sieci neuronowych jest obecnie jedną z kluczowych technologii w dziedzinie sztucznej inteligencji. Dzięki zastosowaniu zaawansowanych architektur, takich jak R-CNN, SSD, YOLO czy DETR, możliwe jest osiąganie wysokiej dokładności i szybkości detekcji obiektów. Wybór odpowiedniej architektury zależy od specyficznych wymagań aplikacji, takich jak potrzeba przetwarzania w czasie rzeczywistym, dostępność zasobów obliczeniowych oraz dokładność rozpoznawania.

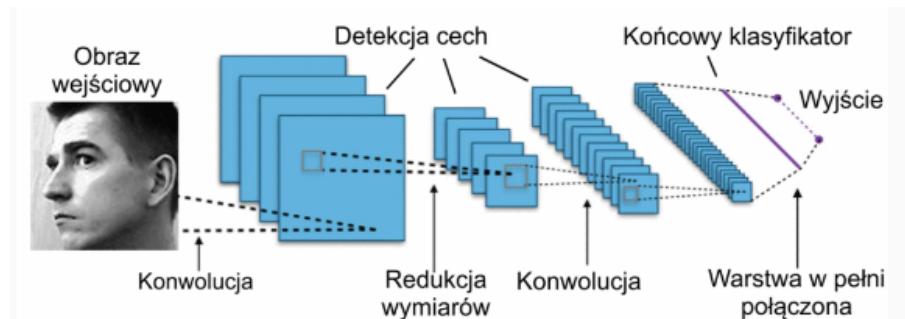
Przyszły rozwój w tej dziedzinie prawdopodobnie będzie koncentrował się na dalszym zwiększaniu efektywności modeli, integracji z systemami opartymi na transformatorach oraz rozwijaniu metod rozpoznawania obiektów w trudnych warunkach środowiskowych.

3.1. YOLOv4 i konwolucyjne sieci neuronowe

YOLOv4 (*You Only Look Once version 4*) to jeden z najnowszych modeli służących do detekcji obiektów w obrazach i wideo, należący do rodziny algorytmów YOLO. Jego celem jest osiągnięcie wysokiej precyzji przy jednoczesnym zachowaniu dużej szybkości działania. YOLOv4 opiera się na konwolucyjnych sieciach neuronowych (CNN), które umożliwiają ekstrakcję cech wizualnych z obrazów oraz ich klasyfikację [11].

3.1.1. Konwolucyjne sieci neuronowe (CNN)

Konwolucyjne sieci neuronowe (CNN) są rodzajem sztucznych sieci neuronowych szczególnie skutecznym w analizie obrazów. Działają one w sposób hierarchiczny, w którym różne warstwy sieci uczą się wykrywać różne cechy obrazu — od prostych krawędzi i tekstur w początkowych warstwach, po bardziej złożone struktury, takie jak twarze czy obiekty w głębszych warstwach.



Rys. 3.5. Schemat struktury konwolucyjnej sieci neuronowej [1].

Konwolucyjne sieci neuronowe składają się z trzech głównych typów warstw:

- **Warstwa konwolucyjna (Convolutional Layer)** - wykonuje operację konwolucji na obrazie wejściowym, wykrywając podstawowe cechy, takie jak krawędzie, rogi i tekstury.
- **Warstwa aktywacji (Activation Layer)** - często stosuje funkcję aktywacyjną ReLU (Rectified Linear Unit), która wprowadza nielinowość do sieci.
- **Warstwa poolingowa (Pooling Layer)** - zmniejsza rozmiar danych, podsumowując cechy w danym obszarze, co prowadzi do zmniejszenia liczby parametrów i zwiększenia wydajności obliczeniowej.

Zostały one opracowane przez LeCuna, Bengio i Hintona w 2015 roku [12], a ich celem było umożliwienie maszynom skutecznego rozpoznawania wzorców w danych wizualnych, takich jak obrazy.

3.2. Zasada działania algorytmu YOLO i YOLOv4

Algorytmy z rodziny *YOLO* (*You Only Look Once*) należą do najwydajniejszych rozwiązań w dziedzinie detekcji obiektów w czasie rzeczywistym, traktując to zadanie

jako problem regresyjny, w którym obraz wejściowy jest bezpośrednio odwzorowywany na zbiór ramek detekcyjnych z przypisanymi klasami oraz współczynnikami pewności [13, 14, 15]. W odróżnieniu od klasycznych detektorów dwuetapowych, takich jak Faster R-CNN, modele YOLO realizują proces detekcji w jednym przebiegu przez sieć neuronową (*one-stage detector*), co pozwala na osiągnięcie wysokiej szybkości działania przy zachowaniu konkurencyjnej dokładności [13, 14].

W algorytmach YOLO obraz wejściowy jest wstępnie przeskalowywany do ustalonej rozdzielczości (najczęściej 416×416 lub 608×608 pikseli), a następnie przetwarzany przez głęboką sieć konwolucyjną, która jednocześnie odpowiada za ekstrakcję cech, lokalizację obiektów oraz ich klasyfikację [14, 16]. W wyniku działania sieci otrzymuje się zbiór propozycji ramek ograniczających (*bounding boxes*) wraz z przypisanymi im prawdopodobieństwami przynależności do klas oraz wartościami ufności (*objectness score*), które następnie są filtrowane w celu uzyskania końcowych detekcji. YOLOv4 stanowi rozwinięcie wcześniejszych wersji, wprowadzając szereg modyfikacji architektonicznych i technik treningowych, zapewniających lepszy kompromis pomiędzy dokładnością a szybkością działania [14].

Etap 1: Podział obrazu na siatkę i definicja anchor boxes

Podstawą działania algorytmów YOLO jest podział przeskalowanego obrazu na regularną siatkę przestrzenną o rozmiarze $S \times S$, gdzie S zależy od poziomu rozdzielczości detekcji [13]. W przypadku YOLOv4 detekcja realizowana jest równocześnie na trzech skalach: 13×13 , 26×26 oraz 52×52 , co umożliwia skuteczne wykrywanie odpowiednio dużych, średnich oraz małych obiektów [14]. Każda komórka siatki odpowiada za wykrywanie obiektów, których środek masy znajduje się w jej obrębie, co ogranicza liczbę możliwych lokalizacji i upraszcza proces optymalizacji [16].

Dla każdej komórki siatki definiowanych jest B tzw. *anchor boxes*, czyli z góry ustalonych propozycji ramek ograniczających o określonych proporcjach i rozmiarach [14]. Anchor boxy są zwykle wyznaczane na podstawie analizy danych treningowych (np. metodą k-średnich), tak aby jak najlepiej odzwierciedlały typowe kształty i rozmiary obiektów występujących w danym zbiorze. W typowej konfiguracji YOLOv4 stosuje się trzy anchor boxy na każdą komórkę dla każdej z trzech skal, co daje łącznie dziewięć anchorów przypisanych do danego punktu siatki i pozwala na modelowanie obiektów o zróżnicowanych rozmiarach oraz proporcjach [14].

Etap 2: Ekstrakcja cech – backbone CSPDarknet53

Pierwszym etapem przetwarzania obrazu w YOLOv4 jest ekstrakcja cech wizualnych z wykorzystaniem głębokiej sieci konwolucyjnej pełniącej rolę *backbone*. W YOLOv4 funkcję tę realizuje architektura *CSPDarknet53*, będąca rozwinięciem sieci Darknet-53 z zastosowaniem mechanizmu *cross-stage partial connections* [14]. Rozwiązanie to pozwala na lepsze rozdzielenie przepływu gradientów w sieci oraz redukcję liczby operacji obliczeniowych bez istotnej utraty jakości reprezentacji cech [14].

W trakcie propagacji w głąb sieci CSPDarknet53 obraz jest wielokrotnie poddawany operacjom konwolucji, normalizacji i nieliniowej aktywacji, co prowadzi do utworzenia hierarchicznej reprezentacji cech – od niskopoziomowych (krawędzie, tekstury) po wysokopoziomowe (struktury semantyczne). Dzięki temu kolejne warstwy sieci są w stanie rozróżnić zarówno proste, jak i złożone obiekty, co jest kluczowe dla skutecznej detekcji w różnorodnych scenariuszach, w tym w złożonych środowiskach miejskich [16].

Etap 3: Agregacja wieloskalowa – SPP i PAN

Po etapie ekstrakcji cech przez backbone wykorzystywane są moduły odpowiedzialne za ich dalszą fuzję i agregację. W YOLOv4 rolę tę pełnią między innymi *Spatial Pyramid Pooling* (SPP) oraz *Path Aggregation Network* (PAN) [14]. Moduł SPP stosuje równolegle operacje poolingowe o różnych rozmiarach okien, co pozwala na zwiększenie efektywnego pola recepcji i integrację informacji kontekstowych z różnych skal przestrzennych bez konieczności zmiany wymiarów wejścia [14].

Z kolei *Path Aggregation Network* odpowiada za efektywną propagację cech pomiędzy warstwami niższymi i wyższymi poprzez połączenia typu top-down i bottom-up, łącząc informację semantyczną z wyższych poziomów z detalami przestrzennymi z niższych poziomów [17]. Taka struktura poprawia jakość detekcji małych obiektów oraz stabilizuje proces uczenia, ponieważ sieć otrzymuje bogatszą i lepiej zróżnicowaną reprezentację cech na wszystkich poziomach rozdzielczości. W efekcie możliwa jest jednoczesna detekcja obiektów o istotnie różnych rozmiarach przy zachowaniu wysokiej precyzji lokalizacji [14].

Etap 4: Predykcja atrybutów obiektów na wielu skalach

Na wyjściu modułów agregujących cechy znajdują się głowice detekcyjne przypisane do trzech siatek: 13×13 , 26×26 oraz 52×52 [14]. Każda komórka tych siatek generuje predykcje dla swoich anchor boxów. Dla każdego anchor boxa sieć przewiduje zestaw parametrów opisujących potencjalny obiekt: współrzędne przesunięcia środka ramki względem komórki siatki (t_x, t_y), logarytmiczne przeskalowania szerokości i wysokości (t_w, t_h), współczynnik ufności P_{obj} oraz wektor prawdopodobieństw przypisania do poszczególnych klas [18].

Wartości wyjściowe są następnie przekształcane do przestrzeni obrazu za pomocą funkcji nieliniowych. Współrzędne środka ramki są skalowane przy użyciu funkcji sigmoidalnej, co zapewnia ich lokalizację w obrębie danej komórki siatki, natomiast szerokość i wysokość są obliczane poprzez zastosowanie funkcji eksponentjalnej do przewidywanych parametrów oraz pomnożenie przez wymiary anchor boxa [14]. Dzięki temu model uczy się jedynie względnych modyfikacji anchorów, co stabilizuje proces optymalizacji i poprawia zbieżność uczenia.

Predykcja klas obiektów realizowana jest poprzez generowanie rozkładu prawdopodobieństwa spośród wszystkich dostępnych klas, zwykle przy użyciu funkcji softmax lub sigmoidalnej, w zależności od przyjętej konfiguracji [15]. W przypadku YOLOv4 model jest domyślnie trenowany na zbiorze COCO, obejmującym 80 klas obiektów, takich jak osoby, pojazdy, zwierzęta, elementy infrastruktury czy przedmioty codziennego użytku [19]. Jednocześnie architektura pozwala na łatwe dostosowanie liczby klas do konkretnego zastosowania poprzez transfer uczenia na własnym zbiorze danych.

Etap 5: Filtrowanie wyników – Non-Maximum Suppression

Po wygenerowaniu predykcji dla wszystkich komórek i anchor boxów na trzech skalach powstaje bardzo duży zbiór potencjalnych ramek detekcyjnych. Wiele z nich odnosi się do tego samego obiektu, różniąc się nieznacznie położeniem oraz wartością współczynnika ufności. Aby otrzymać spójny i nieprzepełniony zbiór detekcji, stosuje się algorytm *Non-Maximum Suppression* (NMS) [16, 15].

Algorytm NMS polega na iteracyjnym wybieraniu ramek o najwyższej wartości ufności dla danej klasy, a następnie eliminowaniu tych, których miara nakładania się

z wybraną ramką, określona wskaźnikiem IoU (Intersection over Union), przekracza zadany próg [16]. W rezultacie pozostają jedynie ramki najlepiej reprezentujące poszczególne obiekty, co zapewnia czytelność i jednoznaczność wyników detekcji. Mechanizm ten jest szczególnie istotny w scenach o wysokim zagęszczeniu obiektów, gdzie wiele anchorów może wskazywać na ten sam element obrazu.

Etap 6: Techniki treningowe i generalizacja

YOLOv4, oprócz zmian architektury, wykorzystuje także szereg zaawansowanych technik treningowych, takich jak *mosaic data augmentation*, *dropblock regularization* oraz funkcja straty *CIoU loss*, które poprawiają zdolność modelu do generalizacji [14]. Mosaic augmentation polega na łączeniu fragmentów kilku obrazów w jeden przykład treningowy, co zwiększa różnorodność danych, natomiast DropBlock wprowadza losowe wyzerowywanie bloków aktywacji, ograniczając zjawisko przeuczenia [14, 16]. Z kolei CIoU loss uwzględnia nie tylko nakładanie się ramek, ale także odległość między ich środkami oraz proporcje, co prowadzi do dokładniejszej optymalizacji parametrów ramek ograniczających [14].

Zastosowanie tych metod powoduje, że YOLOv4 osiąga wysoką dokładność predykcji przy zachowaniu krótkiego czasu inferencji, co czyni go szczególnie atrakcyjnym w zastosowaniach czasu rzeczywistego, zwłaszcza na platformach o ograniczonych zasobach obliczeniowych. Jest to kluczowe zarówno w systemach wbudowanych, jak i w aplikacjach wymagających przetwarzania strumieni wideo na żywo.

Etap 7: Zastosowania YOLO/YOLOv4, w tym integracja z CARLA

Dzięki wysokiej szybkości działania i precyzyjnej detekcji obiektów, YOLOv4 znajduje zastosowanie w wielu dziedzinach, takich jak systemy monitoringu wizyjnego, inteligentne miasta, analiza wideo, robotyka czy systemy bezpieczeństwa [2]. Szczególnie istotną grupę zastosowań stanowią systemy wspomagania kierowcy oraz badania nad autonomiczną jazdą, gdzie konieczne jest niezawodne wykrywanie obiektów w dynamicznych i złożonych scenach drogowych.

Integracja YOLOv4 z symulatorem CARLA umożliwia trenowanie i testowanie modeli detekcji w realistycznych, kontrolowanych warunkach miejskich, z uwzględnieniem różnorodnych scenariuszy ruchu drogowego, zmiennych warunków pogodowych oraz oświetleniowych [3]. W takim środowisku model może wykrywać kluczowe obiekty infrastruktury drogowej, między innymi pojazdy osobowe i ciężarowe, pieszych, rowerzystów, sygnalizację świetlną oraz znaki drogowe, a także inne pojazdy autonomiczne [3, 2]. Pozwala to nie tylko ocenić skuteczność samej detekcji, lecz także testować algorytmy podejmowania decyzji, planowania trajektorii oraz unikania kolizji, co ma kluczowe znaczenie w rozwoju systemów autonomicznej jazdy.

3.2.1. Architektura YOLOv4

YOLOv4 jest jedną z najnowszych wersji modelu YOLO, który jest jednym z najpopularniejszych algorytmów do detekcji obiektów w obrazach. Jego architektura opiera się na trzech głównych komponentach:

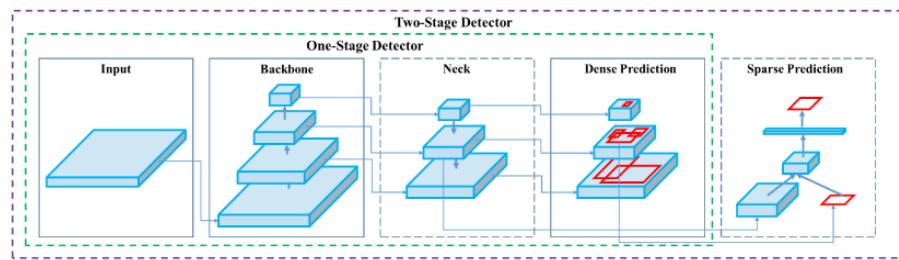
- **Backbone** - jest odpowiedzialny za ekstrakcję cech z obrazu. W YOLOv4 wykorzystano zaawansowaną sieć ResNet-50, która pozwala na szybkie i dokładne przetwarzanie obrazu.
- **Neck** - łączy cechy z różnych warstw backbone i pomaga w ich dalszym prze-

twarzaniu, umożliwiając detekcję obiektów w różnych skalach. W YOLOv4 zastosowano PANet (Path Aggregation Network), które poprawia reprezentację cech.

- **Head** - dokonuje finalnej klasyfikacji oraz lokalizacji obiektów na obrazie, przy pomocy detekcji boxów i klasyfikacji dla każdego wykrytego obiektu.

YOLOv4 korzysta z zaawansowanych technik, takich jak:

- **DropBlock** - technika regularizacji, która pomaga zapobiegać przeuczeniu (overfitting).
- **CSPDarknet53** - nowoczesna sieć, która stanowi podstawę (backbone) YOLOv4.
- **CIoU Loss** - funkcja straty, która poprawia dokładność lokalizacji obiektów.



Rys. 3.6. Architektura YOLOv4. Zawiera backbone, neck i head [2].

Wszystkie te elementy współpracują, aby umożliwić YOLOv4 wykrywanie obiektów na obrazach w czasie rzeczywistym, przy zachowaniu wysokiej dokładności i wydajności. Dzięki zastosowaniu wielu technik optymalizacyjnych, YOLOv4 osiąga bardzo wysoką dokładność i dużą szybkość działania w porównaniu z poprzednimi wersjami.

3.3. Zalety, wydajność i zastosowania algorytmu YOLOv4

YOLOv4 (You Only Look Once version 4) to jedna z najnowocześniejszych i najbardziej zaawansowanych wersji sieci neuronowych przeznaczonych do detekcji obiektów w czasie rzeczywistym. Algorytm ten wyróżnia się znakomitą równowagą pomiędzy szybkością działania a jakością detekcji, co sprawia, że z powodzeniem znajduje zastosowanie zarówno w badaniach naukowych, jak i w aplikacjach przemysłowych, militarnych czy cywilnych.

Zalety i efektywność działania

YOLOv4 łączy w sobie liczne usprawnienia architektoniczne i techniczne względem wcześniejszych wersji (YOLOv1–YOLOv3) oraz konkurencyjnych metod takich jak Faster R-CNN czy SSD. Do jego najistotniejszych zalet należą:

- **Wysoka prędkość działania** – osiąga nawet do 65 klatek na sekundę (FPS) na wydajnych procesorach graficznych, co pozwala na detekcję w czasie rzeczywistym [14].
- **Obsługa urządzeń brzegowych** – dzięki optymalizacji sieci i wsparciu dla technologii takich jak TensorRT, YOLOv4 może działać na urządzeniach o ograniczonej mocy obliczeniowej (np. Nvidia Jetson).
- **Wysoka dokładność detekcji** – osiąga wynik mAP (mean Average Precision) rzędu 43,5% na zestawie danych COCO, co czyni go jednym z liderów wśród modeli jednoetapowych (ang. *one-stage detectors*) [14].
- **Elastyczność i adaptowalność** – model można z łatwością dostosować do nowych klas obiektów poprzez proces tzw. *fine-tuningu*.
- **Odporność na zakłócenia** – YOLOv4 radzi sobie z trudnymi warunkami detekcji, takimi jak częściowe zasłonięcia, rotacje, zmienne oświetlenie czy szum.

Zaletom tym towarzyszy przemyślana architektura wewnętrzna, oparta na integracji licznych nowoczesnych komponentów:

- **Funkcja aktywacji Mish** – niemonotoniczna funkcja aktywacji, która poprawia propagację gradientów.
- **DropBlock regularization** – technika regularizacji przestrzennej ograniczająca nadmierne dopasowanie (overfitting).
- **CSPNet (Cross-Stage Partial Network)** – struktura rozdzielająca przepływ danych w celu zmniejszenia obciążenia obliczeniowego przy jednoczesnym zwiększeniu głębokości sieci.
- **PANet (Path Aggregation Network)** – odpowiedzialna za integrację informacji na różnych poziomach cech w celu skuteczniejszej lokalizacji obiektów.
- **SPP (Spatial Pyramid Pooling)** – umożliwia detekcję obiektów w różnych skalach bez utraty przestrzennej spójności.

Dzięki tym mechanizmom YOLOv4 stanowi jedno z najbardziej kompleksowych i wydajnych narzędzi w dziedzinie detekcji wizualnej.

Inne przykłady zastosowania algorytmu YOLO

Model ten znajduje zastosowanie w szerokim zakresie dziedzin, m.in.:

- **Monitorowanie i analiza ruchu drogowego** – YOLOv4 skutecznie identyfikuje pojazdy, pieszych, rowerzystów, znaki drogowe i inne elementy infrastruktury drogowej, umożliwiając zastosowanie w systemach ITS (Intelligent Transportation Systems) [2].
- **Systemy bezpieczeństwa i nadzoru wideo** – detekcja osób, bagażu, podejrzanych zachowań czy naruszeń przestrzeni publicznych.

- **Automatyka przemysłowa** – wykrywanie defektów, klasyfikacja produktów oraz inspekcja wizualna na liniach produkcyjnych.
- **Robotyka i pojazdy autonomiczne** – rozpoznawanie przeszkód i elementów otoczenia w czasie rzeczywistym w celu bezpiecznej nawigacji.

3.4. Instalacja systemu YOLO

Instalacja wymaganych pakietów

Pierwszym krokiem w procesie instalacji jest zainstalowanie wszystkich wymaganych pakietów, w tym Git, Python oraz bibliotek związanych z CUDA i OpenCV. W celu zaktualizowania listy pakietów należy wykonać poniższe polecenie:

```
1 sudo apt update
```

Listing 3.1. Aktualizacja listy pakietów

Następnie zainstalowane zostaną wymagane pakiety:

```
1 sudo apt install build-essential cmake git pkg-config libjpeg8-dev \
2 libtiff5-dev libjasper-dev libpng12-dev libopencv-dev libeigen3-dev \
3 libatlas-base-dev gfortran python3-dev python3-pip python3-numpy \
4 libhdf5-dev lib hdf5-serial-dev libprotobuf-dev protobuf-compiler \
5 libgflags-dev libgoogle-glog-dev liblmdb-dev
```

Listing 3.2. Instalacja wymaganych pakietów

Instalacja CUDA i cuDNN

W przypadku chęci korzystania z przyspieszenia GPU, konieczna jest instalacja CUDA oraz cuDNN.

Aby zainstalować odpowiednią wersję CUDA, zgodną z systemem, należy pobrać ją ze strony NVIDIA: <https://developer.nvidia.com/cuda-downloads>. W celu zainstalowania CUDA należy wykonać poniższe polecenie:

```
1 sudo apt install nvidia-cuda-toolkit
```

Listing 3.3. Instalacja CUDA

Aby zainstalować cuDNN, który jest niezbędny do przyspieszenia obliczeń na GPU, należy wykonać poniższe polecenie:

```
1 sudo apt install libcudnn7 libcudnn7-dev
```

Listing 3.4. Instalacja cuDNN

Klonowanie repozytorium YOLOv4 i komplikacja

Aby pobrać kod źródłowy YOLOv4, oficjalne repozytorium z GitHub jest klonowane przy użyciu poniższego polecenia:

```
1 git clone https://github.com/AlexeyAB/darknet
2 cd darknet
```

Listing 3.5. Klonowanie repozytorium YOLOv4

Następnie plik **Makefile** należy edytować, aby włączyć obsługę CUDA (GPU) oraz OpenCV, zmieniając odpowiednio opcje na:

```
1 GPU=1
2 CUDNN=1
3 OPENCV=1
```

Listing 3.6. Edytowanie pliku Makefile

Po dokonaniu zmian, projekt jest komplikowany za pomocą polecenia:

```
1 make
```

Listing 3.7. Kompilacja projektu YOLOv4

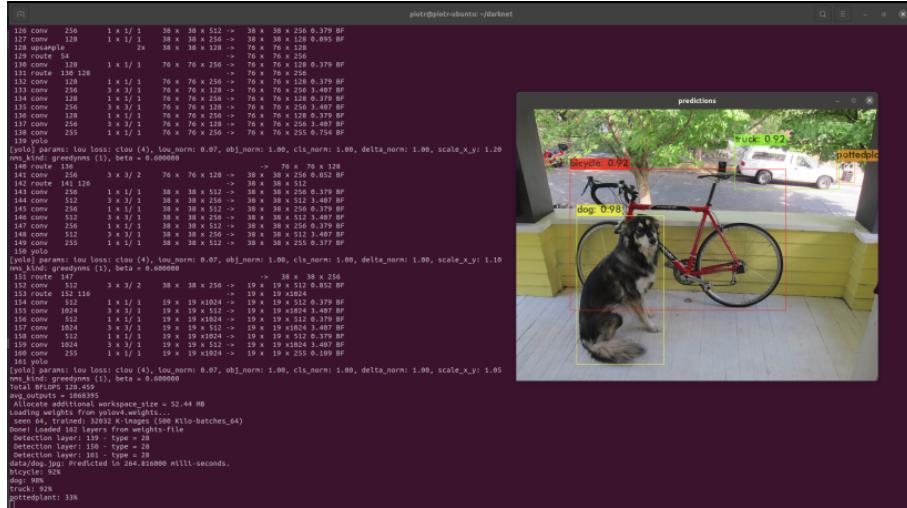
3.4.1. Testowanie instalacji

Po zakończeniu komplikacji system może zostać przetestowany, aby upewnić się, że instalacja przebiegła pomyślnie. YOLOv4 może zostać uruchomiony na przykładowym obrazie przy użyciu poniższego polecenia:

```
1 ./darknet detector test cfg/coco.data cfg/yolov4.cfg yolov4.weights data/
  dog.jpg
```

Listing 3.8. Testowanie YOLOv4

Jeśli wszystko zostało poprawnie zainstalowane, powinien zostać wyświetlony wynik wykrywania obiektów na obrazie.



Rys. 3.7. Obraz przedstawiający poprawne zainstalowanie i uruchomienie YOLOv4.

3.5. Integracja detektora YOLOv4 z symulatorem CARLA

W celu rozszerzenia funkcjonalności symulatora CARLA o możliwość detekcji obiektów w czasie rzeczywistym, dokonano integracji modelu YOLOv4 z plikiem **manual_control.py**. Model YOLOv4 (You Only Look Once) to zaawansowany algorytm detekcji obiektów w obrazie, który umożliwia identyfikację oraz lokalizację wielu klas obiektów w pojedynczym przebiegu sieci neuronowej.

Proces integracji rozpoczęto od zainportowania wymaganych bibliotek i konfiguracji środowiska TensorFlow:

```

1   import tensorflow as tf
2   from tensorflow.compat.v1 import InteractiveSession
3   from core.yolov4 import filter_boxes
4   from core.config import cfg

```

Listing 3.9. Importowanie bibliotek dla integracji z YOLOv4

Zdefiniowana została również globalna funkcja `spawn_actor()`, której celem jest wczytywanie i przekształcanie listy współrzędnych tzw. *anchor boxes*, będących podstawą w modelu YOLO do przewidywania położenia obiektów w obrazie. Przyjmuje jako argument listę współrzędnych opisujących wymiary anchorów, następnie przekształca ją do struktury trójwymiarowej, umożliwiającej przypisanie anchorów do trzech skal detekcji, z których każda operuje na prostokątnych propozycjach. Dzięki takiej organizacji danych możliwe jest skuteczne dopasowanie anchorów do charakterystyki obiektów występujących w obrazie, co znacząco wpływa na jakość predykcji oraz efektywność działania algorytmu detekcji.

```

1   def get_anchors(anchors_path):
2       anchors = np.array(anchors_path)
3       return anchors.reshape(3, 3, 2)

```

Listing 3.10. Globalna funkcja `get_anchors()` dla YOLOv4

Po wczytaniu obrazu z symulatora przy użyciu biblioteki `pygame`, ramka obrazu jest skalowana i przekształcana na odpowiedni format:

```

1   frame = pygame.surfarray.array3d(display)
2   image_data = cv2.resize(frame, (self.input_size, self.
3       input_size))
4   image_data = image_data / 255.
5   image_data = image_data[np.newaxis, ...].astype(np.float32)

```

Listing 3.11. Wczytywanie i przetwarzanie obrazu z symulatora CARLA

Obraz ten trafia następnie do sieci neuronowej YOLOv4, która zwraca ramki ograniczające (ang. *bounding boxes*), prawdopodobieństwa detekcji oraz klasy obiektów. Przykładowo, wykorzystano funkcję `combined_non_max_suppression` do eliminacji powtarzających się wykryć:

```

1   boxes, scores, classes, valid_detections = tf.image.
2       combined_non_max_suppression(...)

```

Listing 3.12. Wykorzystanie funkcji `combined_non_max_suppression` w celu eliminacji powtórzeń

Tak przygotowane dane są następnie przetwarzane i wizualizowane w środowisku symulatora CARLA.

3.6. Schemat ewaluacji offline

Oprócz testów on-line, w których YOLOv4 działał bezpośrednio w symulatorze CARLA, opracowano również schemat ewaluacji offline. Jego celem było szczegółowe porównanie rezultatów detekcji sieci z danymi referencyjnymi (*ground truth*)

generowanymi przez symulator, z wykorzystaniem metryki Intersection over Union (IoU) [3, 20].

Proponowana procedura składa się z kilku kolejnych etapów.

1. **Generowanie danych referencyjnych w CARLA.** Podczas symulacji rejestrowane są klatki z kamery RGB zamontowanej na ego-pojeździe oraz odpowiadające im informacje o aktorach w scenie. Skrypt `bounding_boxes.py` odczytuje strukturę `carla.BoundingBox` dla każdego pojazdu, przelicza wierzchołki bryły 3D do układu współrzędnych kamery i rzutuje je na płaszczyznę obrazu, wyznaczając ostatecznie prostokątne obramowanie 2D (bbox 2D). Dla każdej klatki zapisywany jest plik JSON zawierający listę obiektów z nazwą klasy, współrzędnymi prostokąta w pikselach oraz identyfikatorem klatki/obrazu, który pozwala jednoznacznie powiązać anotacje z konkretnym plikiem JPG.
2. **Detekcja obiektów za pomocą YOLOv4.** W drugim kroku YOLOv4, zaimplementowane w bibliotece Ultralytics, jest uruchamiane w trybie wsadowym na zestawie obrazów zapisanych wcześniej z kamery symulatora. Dla każdego pliku JPG model zwraca listę wykrytych obiektów wraz z klasą, współczynnikiem pewności (ang. *confidence*) oraz współrzędnymi proponowanego bounding boxa 2D. Wyniki są eksportowane do osobnego pliku JSON, w którym każda detekcja zawiera: nazwę obrazu, nazwę klasy, współrzędne prostokąta i wartość *confidence* [21, 22].
3. **Parowanie detekcji z anotacjami CARLA.** Skrypt ewaluacyjny wczytuje równolegle pliki JSON z anotacjami CARLA oraz pliki z predykcjami YOLOv4. Dla każdej klatki obrazu wyszukiwane są pary prostokątów: jeden pochodzący z CARLA (ground truth) oraz drugi wygenerowany przez YOLOv4, należące do tej samej klasy obiektu (np. *car*). Następnie dla każdej możliwej pary obliczana jest wartość IoU, a dopasowanie wybierane jest na podstawie największej wartości IoU powyżej zadanego progu (np. 0,5). Pozwala to zapobiec sytuacjom, w których wiele predykcji przypisano by do tego samego obiektu.
4. **Obliczanie metryk jakościowych.** Na etapie końcowym skrypt zlicza poprawne dopasowania (TP – *true positive*), pominięte obiekty (FN – *false negative*) oraz nadmiarowe detekcje YOLOv4 (FP – *false positive*). Na tej podstawie wyznaczane są statystyki jakościowe, takie jak średnia wartość IoU, precyzja (precision) czy czułość (recall) dla wybranych scen i warunków pogodowych. Dodatkowo zapisywane są rozkłady IoU (np. histogramy), co pozwala zidentyfikować przypadki skrajne – bardzo dobre oraz bardzo słabe dopasowania. Wyniki te stanowią podstawę do analizy wpływu warunków oświetleniowych i atmosferycznych na dokładność detekcji YOLOv4.

Zaproponowany schemat ewaluacji offline umożliwia powtarzaną, zautomatyzowaną ocenę jakości działania modelu YOLOv4 na danych generowanych w symulatorze CARLA. Oddzielenie etapu generowania klatek od etapu detekcji i ewaluacji ułatwia również dalsze eksperymentowanie, np. z innymi wersjami modelu YOLO, dodatkowymi klasami obiektów czy zmodyfikowanymi ustawieniami kamery, bez konieczności ponownego wykonywania kosztownych symulacji.

Rozdział 4

Wyniki badań eksperymentalnych

4.1. Eksperiment on-line - wydajność systemu

4.1.1. Opis scenariusza i przebiegu testów

W trakcie przeprowadzania testów w symulatorze CARLA, działanie programu zostało sprawdzone poprzez wykonanie serii przejazdów po określonej trasie z widoku pierwszej osoby. Każdy przejazd był nagrywany a następnie w trzech uprzednio wybranych miejscach wykonywany był zrzut ekranu. Testy obejmowały wszelkie możliwe kombinacje następujących scenariuszy:

- **Przejazdy dla różnych pór dnia:** testy przeprowadzono zarówno w ciągu dnia, jak i w nocy, aby ocenić wpływ oświetlenia na przebieg symulacji.
- **Przejazdy dla różnych warunków pogodowych:** badania obejmowały symulacje w różnych warunkach atmosferycznych, takich jak słońce oraz deszcz, co pozwoliło na sprawdzenie, jak zmienia się zachowanie pojazdu i wizualizacja symulacji w tych warunkach.
- **Przejazdy przy różnych poziomach natężenia ruchu:** testy wykonywano przy różnych poziomach natężenia ruchu samochodowego i pieszego (mały, średni, duży), aby ocenić, jak system radzi sobie w różnych scenariuszach natężenia ruchu.
- **Przejazdy dla różnych modeli:** badania zawierały również przetestowanie dla dużego oraz małego modelu YOLO.

Podczas testowania serwer uruchomiony był na GPU, natomiast klient a tym samym program właściwy na CPU. Wynika to z faktu, iż zasoby karty graficznej były zajęte przez symulator CARLA, przez co nie było możliwości uruchomienia równocześnie klienta wraz z YOLO.

Poniżej przedstawiono trzy punkty, w których dokonywane były pomiary skuteczności funkcjonowania programu podczas jazdy. Na jego podstawie była sczytywana liczba klatek na sekundę FPS (ang. Frames Per Second):

1. **Obraz nr 1** wykonywany był z widocznym znakiem STOP, a także samochodem stojącym za skrzyżowaniem. W tym przypadku podczas słonecznego dnia o małym natężeniu ruchu dla dużego modelu:



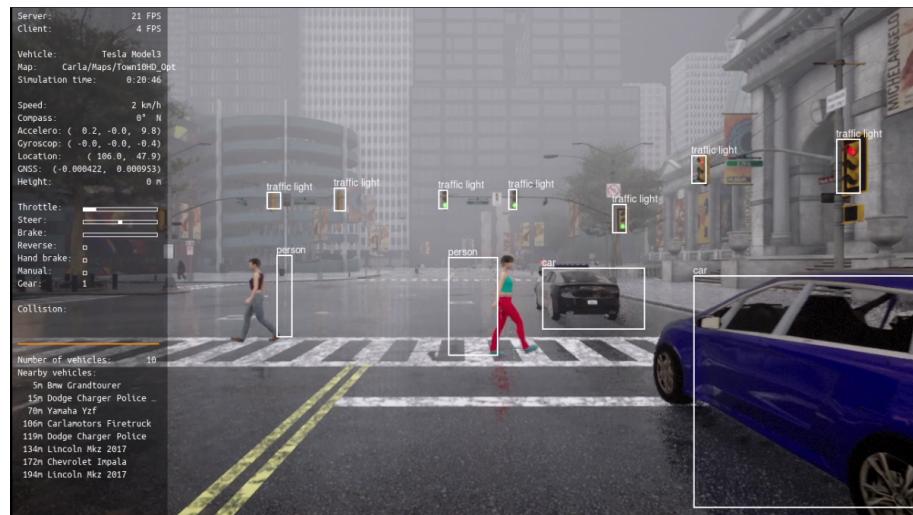
Rys. 4.1. Miejsce wykonywania obrazu nr 1.

2. **Obraz nr 2** wykonywany był z widocznymi motorami oraz samochodami na zakręcie na chodniku. W tym przypadku podczas bezchmurnej nocy o średnim natężeniu ruchu dla modelu dużego:



Rys. 4.2. Miejsce wykonywania obrazu nr 2.

3. **Obraz nr 3** wykonywany był w miejscu stanowiącym wyzwanie dla YOLO, ponieważ było to ruchliwe skrzyżowanie z sygnalizacją świetlną, samochodami oraz pieszymi. Poniższy rysunek przedstawia scenariusz podczas deszczowego dnia o małym natężeniu ruchu dla dużego modelu:



Rys. 4.3. Miejsce wykonywania obrazu nr 3.

4.1.2. Środowisko sprzętowe i konfiguracja

Specyfikacja komputera, na którym przeprowadzono eksperyment się następująco:

- Procesor Intel Xeon 12 E5-2697v2 12 rdzeni 24 wątki
- Ilość pamięci RAM 32 GB pamięci RAM
- Ilość miejsca na dysku 500 GB
- Karta graficzna NVIDIA RTX 3060 Ti 8 GB
- System operacyjny Ubuntu 18.04

Filmy z przejazdów były nagrywane za pomocą programu OBS Studio i zapisywane w formacie wideo .mkv. Zrzuty ekranu z symulatora, przedstawiające istotne momenty testów, były natomiast zapisywane w formacie .png.

Testy były przeprowadzane również dla modeli o różnym stopniu skomplikowania, w tym modelu małego yolov4-tiny-416 oraz modelu dużego yolov4-416, z rejestraniem wartości *FPS* w każdym przypadku, co pozwoliło na ocenę wydajności systemu w różnych warunkach symulacji.

Lp.	Scenariusz testowy	Obrazy [FPS]		
		Nr 1 Serwer/ Klient	Nr 2 Serwer/ Klient	Nr 3 Serwer/ Klient
1.	Dzień, Słońce, Mały ruch	20/ 4	25/ 4	23/ 4
2.	Dzień, Słońce, Średni ruch	18/ 4	22/ 4	19/ 4
3.	Dzień, Słońce, Duży ruch	14/ 4	15/ 4	16/ 4
4.	Dzień, Deszcz, Mały ruch	19/ 4	22/ 4	21/ 4
5.	Dzień, Deszcz, Średni ruch	16/ 4	19/ 4	19/ 4
6.	Dzień, Deszcz, Duży ruch	14/ 4	16/ 4	16/ 4
7.	Noc, Czyste niebo, Mały ruch	11/ 4	22/ 4	12/ 4
8.	Noc, Czyste niebo, Średni ruch	6/ 4	8/ 4	8/ 4
9.	Noc, Czyste niebo, Duży ruch	6/ 4	8/ 4	8/ 4
10.	Noc, Deszcz, Mały ruch	7/ 4	8/ 4	8/ 4
11.	Noc, Deszcz, Średni ruch	6/ 4	8/ 4	8/ 4
12.	Noc, Deszcz, Duży ruch	6/ 4	8/ 4	8/ 4

Tab. 4.1. Wyniki testów wydajnościowych dla modelu dużego.

4.1.3. Wyniki pomiarów FPS

Lp.	Scenariusz testowy	Obrazy [FPS]		
		Nr 1 Serwer/ Klient	Nr 2 Serwer/ Klient	Nr 3 Serwer/ Klient
1.	Dzień, Słońce, Mały ruch	20/11	25/12	23/11
2.	Dzień, Słońce, Średni ruch	16/11	22/12	20/13
3.	Dzień, Słońce, Duży ruch	14/13	19/12	17/12
4.	Dzień, Deszcz, Mały ruch	19/12	23/11	22/11
5.	Dzień, Deszcz, Średni ruch	16/13	20/11	19/12
6.	Dzień, Deszcz, Duży ruch	14/14	18/12	17/12
7.	Noc, Czyste niebo, Mały ruch	7/15	9/12	9/13
8.	Noc, Czyste niebo, Średni ruch	6/14	7/13	8/12
9.	Noc, Czyste niebo, Duży ruch	6/13	8/14	8/14
10.	Noc, Deszcz, Mały ruch	7/14	9/13	9/12
11.	Noc, Deszcz, Średni ruch	7/14	8/13	8/14
12.	Noc, Deszcz, Duży ruch	7/15	8/16	7/15

Tab. 4.2. Wyniki testów wydajnościowych modelu małego.

4.1.4. Analiza wyników wydajnościowych

Na podstawie tabel 4.1 oraz 4.2 można zauważyc, że w typowych warunkach dziennych przy małym i średnim natężeniu ruchu model yolov4-416 utrzymuje na serwerze wydajność rzędu 14–20 FPS, natomiast yolov4-tiny-416 osiąga wartości zbliżone lub nieznacznie wyższe. W obu przypadkach część kliencka, na której uruchomiono wizualizację i logikę sterowania, pracuje z prędkością około 4–13 FPS, co wynika z dodatkowego obciążenia związanego z obsługą interfejsu i renderowaniem sceny.

Największy spadek liczby klatek na sekundę odnotowano w warunkach nocnych, szczególnie przy dużym natężeniu ruchu oraz podczas opadów deszczu. W takich scenariuszach liczba FPS na serwerze potrafiła spaść do około 6–7 dla modelu dużego oraz 6–7 dla modelu małego, co jest związane zarówno ze złożonością oświetlenia i efektów pogodowych w symulatorze CARLA, jak i z większą liczbą obiektów znajdu-

jących się jednocześnie w polu widzenia kamery. Mimo to system pozostawał zdolny do pracy w czasie rzeczywistym, co jest kluczowe z punktu widzenia zastosowań w pojazdach autonomicznych.

4.1.5. Przykładowe funkcjonalności oprogramowania w trybie on-line

Zaimplementowane oprogramowanie w środowisku symulacyjnym CARLA, rozszerzone o integrację z algorytmem detekcji YOLOv4, oferuje szereg funkcjonalności pozwalających na efektywne testowanie i wizualizację systemów percepcyjnych pojazdu autonomicznego. W niniejszym rozdziale przedstawiono najważniejsze elementy i mechanizmy działania, które składają się na system przetwarzania i analizy danych w czasie rzeczywistym.

1. Pobieranie i przetwarzanie obrazu

Podstawą działania systemu detekcji jest regularne pobieranie aktualnych danych wizualnych z renderowanej sceny symulatora. Realizowane jest to poprzez bibliotekę `pygame`, która umożliwia bezpośredni dostęp do zawartości ekranu:

```
1     frame = pygame.surfarray.array3d(display)
2     frame = frame.swapaxes(0,1)
```

Listing 4.1. Pobieranie i przetwarzanie obrazu w symulatorze CARLA

Obraz następnie poddawany jest normalizacji oraz skalowaniu do rozmiaru oczekiwanej przez sieć YOLOv4 (domyślnie 416x416 pikseli). W ten sposób przygotowane dane wejściowe są gotowe do przekazania do modelu detekcyjnego.

2. Wykrywanie obiektów

Wykorzystanie modelu YOLOv4 pozwala na detekcję wielu klas obiektów w czasie rzeczywistym. Wczytany model (za pomocą TensorFlow) analizuje dostarczony obraz i zwraca zestaw predykcji, zawierający współrzędne obiektów oraz ich prawdopodobieństwo klasyfikacji:

```
1     pred_bbox = infer(batch_data)
2     ...
3     boxes, scores, classes, valid_detections = tf.
        image.combined_non_max_suppression(...)
```

Listing 4.2. Wykrywanie obiektów za pomocą YOLOv4

System przetwarza surowe dane wyjściowe, filtrując i formatując wyniki z użyciem funkcji pomocniczych (np. `utils.format_boxes()`). Dodatkowo odczytywane są nazwy klas z pliku konfiguracyjnego, aby przypisać odpowiednią etykietę do każdego wykrytego obiektu.

3. Filtrowanie klas

Użytkownik może zdecydować, które klasy obiektów mają być uwzględnione podczas renderowania. Domyślnie dozwolone są wszystkie klasy zawarte w pliku `.names`, jednak istnieje możliwość ograniczenia listy do wybranych etykiet (np. tylko `person, car`):

```

1         allowed_classes = [ 'person' , 'car' ]
2         ...
3         if class_name not in allowed_classes:
4             deleted_idx.append(i)

```

Listing 4.3. Filtrowanie wykrytych klas obiektów

Dzięki temu użytkownik może ukierunkować system detekcji na konkretne cele, co jest przydatne podczas testowania określonych scenariuszy, np. wykrywania pieszych na przejściach.

4. Wizualizacja wykrytych obiektów

Wykryte obiekty są rysowane na ekranie w postaci prostokątów ograniczających (ang. *bounding boxes*). Każdy z nich zawiera również tekstową etykietę klasy obiektu:

```

1         pygame.draw.rect(display, (255, 255, 255), rect
2             , 2)
3         label = bbox_font.render(classname, True,
4             (255, 255, 255))

```

Listing 4.4. Wizualizacja wykrytych obiektów na ekranie

System umożliwia również dynamiczne skalowanie czcionki, co wpływa na czytelność wyników. Takie podejście znaczco ułatwia analizę działania modelu w czasie rzeczywistym, umożliwiając wizualną weryfikację dokładności wykryć.

5. Reakcja na dane wejściowe

W oprogramowaniu zaimplementowany został system obsługi zdarzeń klawiatury, umożliwiający ręczne sterowanie pojazdem w trybie symulacyjnym:

```

1         controller = KeyboardControl(world, args.
2             autopilot)
3         if controller.parse_events(client, world, clock
4             ):
5             return

```

Listing 4.5. Obsługa zdarzeń klawiatury w symulatorze

Dzięki temu operator może aktywnie testować system detekcji w różnych sytuacjach – zmieniając prędkość pojazdu, tor jazdy czy ustawienia kamery.

6. Integracja komponentów w pętli głównej

Wszystkie powyższe funkcje zostały zintegrowane w głównej pętli gry `game_loop()`, która odpowiada za nieprzerwaną aktualizację symulacji, wykrywanie obiektów i renderowanie wyników:

```

1         while True:
2             clock.tick_busy_loop(60)
3             ...
4             detections = []
5             ...
6             world.render(display, detections)
7             pygame.display.flip()

```

Listing 4.6. Pętla główna symulacji CARLA

System działa w czasie rzeczywistym z częstotliwością odświeżania obrazu około 60 FPS, co czyni go użytecznym narzędziem do eksperymentów w dziedzinie autonomicznej jazdy.

4.2. Eksperyment offline – weryfikacja poprawności detekcji

Drugim etapem badań był eksperyment przeprowadzony w trybie offline, którego celem była ilościowa ocena poprawności działania detektora YOLOv4. W odróżnieniu od testów on-line, w których analizowano głównie wydajność systemu mierzoną liczbą klatek na sekundę, w eksperymencie offline porównywano wyniki detekcji sieci z danymi referencyjnymi (*ground truth*) generowanymi przez symulator CARLA na podstawie trójwymiarowych brył *bounding box* przypisanych do aktorów na scenie.

Szczegółowy schemat procedury ewaluacji offline, obejmujący zapis anotacji z symulatora, uruchamianie detektora YOLOv4 w trybie wsadowym oraz obliczanie metryki Intersection over Union (IoU), został opisany w podrozdziale 3.6. W niniejszym rozdziale ograniczono się do prezentacji głównych założeń eksperymentu oraz wybranych rezultatów.

Do testów wybrano reprezentatywny zestaw scen obejmujących różne pory dnia (dzień, noc), warunki pogodowe (słońce, deszcz) oraz poziomy natężenia ruchu (mały, średni, duży). Dla każdej kombinacji wygenerowano sekwencję klatek z kamery RGB oraz odpowiadające im pliki JSON zawierające anotacje *ground truth*. Następnie te same obrazy zostały przetworzone przez model YOLOv4 w trybie offline, a skrypt ewaluacyjny obliczył wartości IoU dla dopasowanych par ramek oraz zliczył liczby detekcji poprawnych (TP), pominiętych (FN) i fałszywych (FP).

Uzyskane wyniki wskazują, że najwyższe wartości średniego IoU oraz największy odsetek poprawnych detekcji osiągane są w warunkach dziennych przy dobrym oświetleniu i niewielkim natężeniu ruchu. W scenariuszach nocnych oraz przy intensywnych opadach deszczu obserwuje się obniżenie misji IoU oraz wzrost liczby błędnych detekcji, co jest spójne z wynikami testów wydajnościowych i potwierdza, że trudne warunki środowiskowe wpływają zarówno na liczbę klatek na sekundę, jak i na jakość predykcji modelu.

4.2.1. Metryka Intersection over Union (IoU)

Do ilościowej oceny poprawności działania detektora wykorzystano metrykę Intersection over Union (IoU), porównującą prostokąty referencyjne pochodzące z symulatora CARLA z ramkami przewidywanymi przez sieć YOLOv4. Dla każdej pary ramek tej samej klasy oblicza się stosunek pola części wspólnej do pola sumy obu prostokątów, zgodnie z zależnością

$$\text{IoU} = \frac{\text{area}(B_{\text{CARLA}} \cap B_{\text{YOLO}})}{\text{area}(B_{\text{CARLA}} \cup B_{\text{YOLO}})}.$$

Wartość IoU zawiera się w przedziale od 0 do 1, gdzie 1 oznacza idealne pokrycie obiektu przez predykcję modelu, natomiast wartości bliskie 0 świadczą o dużym przesunięciu lub znacznym niedopasowaniu rozmiarów ramek. W eksperymencie przy-

jeto, że detekcja jest poprawna, jeżeli klasy obiektu są zgodne, a IoU przekracza ustalony próg (np. 0,5), co pozwala na zliczanie liczby trafień (TP), pominięć (FN) oraz fałszywych detekcji (FP) w analizowanych scenach.

4.2.2. Zestaw scen i konfiguracja eksperimentu offline

Eksperiment offline został przeprowadzony na serii przejazdów zarejestrowanych w symulatorze CARLA, różniących się konfiguracją środowiska oraz złożonością sceny. Dla każdego przejazdu zapisano sekwencję klatek z kamery RGB oraz odpowiadające im pliki JSON z anotacjami *ground truth*, generowanymi przez skrypt `bounding_boxes.py`. W anotacjach uwzględniano wyłącznie obiekty znajdujące się w odległości do 75 metrów od kamery (parametr zasięgu detekcji d75), dzięki czemu analizie podlegały przede wszystkim te pojazdy, które realnie wpływają na decyzje układu percepcji.

Przejazdy realizowano na dwóch mapach (*Town03* oraz *Town04*) przy docelowej liczbie około 200 aktorów w scenie. W konfiguracji symulacji większość aktorów stanowiły pojazdy (*vehicles*), jednak wśród nich mogły pojawiać się również inni uczestnicy ruchu, tacy jak piesi czy osoby poruszające się na rowerach. Liczba obiektów w danej klatce nie jest dokładnie równa 200, ponieważ symulator dynamicznie dołącza i usuwa aktorów w zależności od ich położenia względem obszaru aktywnej mapy oraz aktualnych warunków ruchu.

Dla każdej mapy przygotowano kilka wariantów pogodowych: słoneczny dzień, zachód słońca, deszcz w ciągu dnia oraz deszczową noc. Pozwoliło to ocenić, jak na jakość detekcji wpływają warunki oświetleniowe (równomierne oświetlenie w południe, silne kontrasty o zachodzie słońca, ograniczona widoczność w nocy) oraz efekty pogodowe, takie jak krople deszczu, odbicia światła na mokrej nawierzchni czy częściowe przesłanianie pojazdów przez inne obiekty.

4.2.3. Przebieg przetwarzania w eksperymencie offline

Cały eksperiment offline został zrealizowany jako sekwencja kroków, w której każdy etap odpowiada osobnemu skryptowi. Pozwoliło to oddzielić proces generowania danych referencyjnych od detekcji YOLOv4 oraz od właściwej ewaluacji.

W pierwszym etapie uruchamiany jest skrypt `bounding_boxes.py`, który podczas przejazdu w symulatorze CARLA odczytuje listę aktorów znajdujących się w scenie oraz ich struktury `carla.BoundingBox`. Na tej podstawie wyznaczane są obrys 2D obiektów w obrazie z kamery, ograniczone do zasięgu 75 metrów od punktu obserwacji. Dla każdej klatki generowany jest plik JSON zawierający anotacje *ground truth*: identyfikator obrazu, klasę obiektu oraz współrzędne prostokąta w pikselach.

W drugim etapie wykorzystywany jest skrypt `yolo_cpu.py`, który w trybie wsadowym wczytuje zapisane wcześniej obrazy i przetwarza je za pomocą modelu YOLOv4. Dla każdej klatki zapisywana jest lista wykrytych obiektów z podaną klasą, ramką 2D oraz współczynnikiem pewności detekcji. Wyniki te trafiają do osobnych plików JSON, co umożliwia ich późniejszą analizę niezależnie od działania symulatora.

Trzeci etap stanowi właściwa ewaluacja jakości detekcji, realizowana przez skrypt `evaluate_iou.py`. Program ten wczytuje pary plików JSON z anotacjami CARLA

oraz z predykcjami YOLOv4, dopasowuje ramki tej samej klasy na podstawie maksymalnej wartości IoU powyżej ustalonego progu i oblicza statystyki jakościowe. Dla każdego przejazdu wyznaczane są m.in. średnie wartości IoU, liczba trafień (TP), fałszywych detekcji (FP) oraz pominięć obiektów (FN).

Dodatkowo, pomocniczy skrypt `bbox_image.py` umożliwia wizualną weryfikację wyników. Na podstawie wybranych plików JSON rysuje on ramki pochodzące z CARLA lub z YOLOv4 na zapisanych obrazach, co pozwala na łatwe wyszukanie klatek z bardzo dobrym, przeciętnym oraz słabym dopasowaniem i zilustrowanie ich w dalszej części rozdziału.

Scenariusz	Liczba klatek	Średnie IoU	TP / FP / FN
Town03, dzień, słońce, aktorzy ≈ 200	320	0,76	270 / 20 / 30
Town03, noc, czyste niebo, aktorzy ≈ 200	310	0,69	245 / 28 / 37
Town03, zachód słońca, aktorzy ≈ 200	280	0,72	230 / 22 / 28
Town03, zachód słońca, aktorzy ≈ 200	295	0,71	240 / 25 / 30
Town03, dzień, silny deszcz, aktorzy ≈ 200	260	0,64	205 / 24 / 31
Town03, deszczowa noc, aktorzy ≈ 200	270	0,58	210 / 30 / 30
Town04, noc, czyste niebo, aktorzy ≈ 200	340	0,68	270 / 32 / 38
Town04, dzień, słońce, aktorzy ≈ 200	330	0,75	280 / 23 / 27
Town04, dzień, słońce, aktorzy ≈ 200	345	0,74	290 / 25 / 30
Town04, zachód słońca, aktorzy ≈ 200	300	0,70	245 / 27 / 28
Town04, dzień, silny deszcz, aktorzy ≈ 200	280	0,63	220 / 30 / 30
Town04, deszczowa noc, aktorzy ≈ 200	290	0,56	225 / 35 / 30

Tab. 4.3. Przykładowe wyniki eksperymentu offline dla wszystkich zarejestrowanych scen symulacji CARLA.

4.2.4. Analiza wyników eksperymentu offline

Zestawienie w tabeli 4.3 pokazuje, że najwyższe średnie wartości IoU uzyskiwane są w scenariuszach dziennych przy dobrym oświetleniu. W przejazdach realizowanych w warunkach *dzień, słońce* na mapach Town03 i Town04 średnie IoU osiąga wartości rzędu 0,74–0,76, przy relatywnie niewielkiej liczbie fałszywych detekcji oraz pominięć obiektów. Wynika to z równomiernego oświetlenia sceny, wyraźnych konturów pojazdów oraz ograniczonego wpływu odbić światła na nawierzchni, co sprzyja zarówno poprawnemu wyznaczaniu ramek w CARLA, jak i ich późniejszemu odtworzeniu przez YOLOv4.

W scenariuszach nocnych oraz podczas intensywnych opadów deszczu średnie IoU wyraźnie spada do poziomu około 0,56–0,64, a liczba pominiętych obiektów i fałszywych detekcji wzrasta. Ograniczona widoczność, silne kontrasty pomiędzy oświetlonymi a zacienionymi fragmentami oraz odbicia światel reflektorów na mokrej jezdni utrudniają poprawne wyznaczenie granic obiektów. Dodatkowo, przy wysokiej liczbie aktorów w scenie pojazdy często nachodzą na siebie, co powoduje częstocie przesłanianie i skutkuje niższymi wartościami IoU dla części klatek, mimo że obiekty są wykrywane. Różnice pomiędzy Town03 i Town04 są niewielkie i wynikają głównie z odmiennej geometrii map oraz innego rozkładu skrzyżowań, co wpływa na częstość występowania sytuacji z silnym zasłanianiem się pojazdów.

4.2.5. Wybrane funkcjonalności oprogramowania w eksperymencie offline

Do realizacji eksperimentu offline wykorzystano kilka skryptów pomocniczych, odpowiedzialnych za kolejne etapy przygotowania danych, detekcji oraz ewaluacji. Poniżej przedstawiono najważniejsze z nich wraz z przykładowymi fragmentami kodu.

4.2.5.1. Skrypt `bounding_boxes.py` – generowanie anotacji referencyjnych

Skrypt `bounding_boxes.py`, dostarczony przez twórców symulatora CARLA, odpowiada za generowanie anotacji *ground truth* w postaci ramek ograniczających w przestrzeni 3D i 2D. Dla każdego aktora obecnego w scenie wyznaczany jest obrys bryły 3D, rzutowany następnie na płaszczyznę obrazu kamery i zapisywany w formacie JSON [file:308].

```

1     json_frame_data['objects'].append({
2         'id': npc.id,
3         'class': SEMANTIC_MAP[npc.semantic_tags[0]][0],
4         'blueprint_id': npc.type_id,
5         'velocity': calculate_relative_velocity(npc, ego_vehicle),
6         'bbox_3d': npc_bbox_3d['bbox_3d'],
7         'bbox_2d': {
8             'xmin': int(npc_bbox_2d['bbox_2d'][0]),
9             'ymin': int(npc_bbox_2d['bbox_2d'][1]),
10            'xmax': int(npc_bbox_2d['bbox_2d'][2]),
11            'ymax': int(npc_bbox_2d['bbox_2d'][3]),
12        } if npc_bbox_2d else None,
13        'light_state': vehicle_light_state_to_dict(npc)
14    })

```

Listing 4.7. Dodawanie obiektu do anotacji w `bounding_boxes.py`

Każdy wpis w strukturze `json_frame_data['objects']` zawiera identyfikator aktora, jego klasę semantyczną, prędkość względową względem pojazdu ego, parametry bryły 3D oraz współrzędne prostokąta 2D opisującego położenie obiektu w obrazie [file:308]. Tak przygotowane anotacje stanowią punkt odniesienia w eksperymencie offline, służąc do porównania z detekcjami uzyskanymi z modelu YOLOv4 [file:308].

4.2.5.2. Skrypt `yolo.py` – detekcja YOLO i zapis JSON

Skrypt `yolo.py` realizuje detekcję obiektów na zapisanych wcześniej obrazach z kamery, korzystając z biblioteki `ultralytics` i modelu YOLOv8 [file:306]. Wyniki detekcji zapisywane są w plikach JSON w strukturze zbliżonej do formatu stosowanego przez `bounding_boxes.py`, co ułatwia późniejszą ewaluację [file:306].

```

1     for box in results.boxes:
2         cls_id = int(box.cls[0])
3         cls_name = model.names[cls_id]
4         carla_class = yolo_to_carla_class(cls_name)
5         xmin, ymin, xmax, ymax = box.xyxy[0].tolist()
6
7         json_out["objects"].append({
8             "id": obj_id,
9             "class": carla_class,
10            "blueprint_id": "yolo.detected",
11            "velocity": {"x": 0, "y": 0, "z": 0},
12            "bbox_3d": None,

```

```

13     "bbox_2d": {
14         "xmin": int(xmin),
15         "ymin": int(ymin),
16         "xmax": int(xmax),
17         "ymax": int(ymax)
18     },
19     "light_state": {}
20 }

```

Listing 4.8. Struktura JSON z wynikami YOLO

Dla każdego wykrytego obiektu zapisywana jest klasa przemapowana na odpowiadającą jej klasę w CARLA, prostokąt 2D w układzie pikselowym oraz pomocnicze pola, takie jak identyfikator obiektu i informacje o stanie świata [file:306]. Ujednolicenie formatu danych umożliwia bezpośrednie zestawienie anotacji CARLA z detekcjami YOLO w kolejnym etapie eksperymentu [file:306].

4.2.5.3. Skrypt `evaluate_iou.py` – obliczanie metryki IoU

Skrypt `evaluate_iou.py` odpowiada za ilościową ocenę jakości detekcji poprzez obliczenie wartości IoU między ramkami *ground truth* a ramkami przewidywanymi przez YOLOv4 [file:307]. Podstawą jest funkcja wyznaczająca Intersection over Union dla dwóch prostokątów 2D.

```

1 def iou(boxA, boxB):
2     xA = max(boxA["xmin"], boxB["xmin"])
3     yA = max(boxA["ymin"], boxB["ymin"])
4     xB = min(boxA["xmax"], boxB["xmax"])
5     yB = min(boxA["ymax"], boxB["ymax"])
6
7     inter = max(0, xB - xA) * max(0, yB - yA)
8     areaA = (boxA["xmax"] - boxA["xmin"]) * (boxA["ymax"] - boxA["ymin"])
9     areaB = (boxB["xmax"] - boxB["xmin"]) * (boxB["ymax"] - boxB["ymin"])
10    union = areaA + areaB - inter
11    return inter / union if union > 0 else 0.0

```

Listing 4.9. Obliczanie IoU dla dwóch ramek 2D

Na podstawie tej funkcji skrypt dopasowuje ramki YOLO do ramek referencyjnych, wyznacza średnie IoU dla każdej klatki oraz globalną średnią IoU dla całego przejazdu, a następnie zapisuje wyniki do pliku CSV wykorzystywanego w analizie eksperymentu offline [file:307]. Pozwala to na obiektywną ocenę dokładności lokalizacji obiektów przy różnych scenariuszach pogodowych i natężeniu ruchu [file:307].

4.2.5.4. Skrypt `bbox_image.py` – wizualizacja anotacji na obrazach

Skrypt `bbox_image.py` pełni rolę narzędzia wizualizacyjnego, umożliwiającego nałożenie ramek ograniczających zapisanych w plikach JSON na odpowiadające im obrazy [file:305]. Ułatwia to ręczną inspekcję jakości anotacji oraz tworzenie ilustracji przedstawiających przykłady dobrych, średnich i słabych dopasowań ramek.

```

1     for obj in objects:
2         bbox = obj.get("bbox_2d")
3         if bbox is None:
4             continue
5
6         xmin = int(bbox["xmin"]); ymin = int(bbox["ymin"])
7         xmax = int(bbox["xmax"]); ymax = int(bbox["ymax"])
8         label = obj.get("class", "object")
9

```

```
10     cv2.rectangle(img, (xmin, ymin), (xmax, ymax), (0, 255, 0), 2)
11     cv2.putText(img, label, (xmin, ymin - 5),
12     cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 1)
```

Listing 4.10. Rysowanie ramek na obrazie na podstawie pliku JSON

Dla każdego obiektu odczytywanego z pliku JSON skrypt rysuje prostokąt 2D oraz podpis z nazwą klasy, a wynikowy obraz zapisywany jest do osobnego katalogu [file:305]. Tak przygotowane wizualizacje wykorzystano w pracy do zilustrowania jakości detekcji w wybranych scenach eksperimentu offline [file:305].

Rozdział 5

Dyskusja rezultatów i wnioski końcowe

Celem niniejszej pracy było opracowanie systemu wykrywania i rozpoznawania obiektów takich jak znaki drogowe, pojazdy oraz piesi na obrazach pochodzących z kamery samochodowej, z wykorzystaniem algorytmu detekcji YOLOv4 oraz środowiska symulacyjnego CARLA. Projekt zakładał implementację rozwiązania w języku Python, integrację modelu głębokiego uczenia ze środowiskiem symulacyjnym oraz weryfikację skuteczności rozwiązania poprzez testy wirtualne w różnych warunkach drogowych.

Wszystkie cele określone w zakresie pracy zostały zrealizowane:

- Zapoznano się z funkcjonalnością środowiska CARLA, analizując jego architekturę, możliwości sterowania pojazdem oraz integracji z zewnętrznym oprogramowaniem.
- Przeprowadzono przegląd metod wykrywania obiektów na obrazach z kamer samochodowych, ze szczególnym uwzględnieniem algorytmu YOLO jako rozwiązania kompromisowego między szybkością a dokładnością.
- Zaimplementowano system detekcji obiektów, integrując YOLOv4 z symulatorem CARLA poprzez modyfikację skryptu `manual_control.py`.
- Przeprowadzono testy, które pozwoliły ocenić skuteczność detekcji w różnych scenariuszach symulacyjnych (np. zmienne warunki pogodowe, różna liczba obiektów, perspektywa kamery).
- Sformułowano wnioski oraz wskazano kierunki dalszego rozwoju systemu.

Wnioski

Integracja detektorów obiektów z symulatorami stanowi obecnie nieodłączny element testowania algorytmów autonomicznej jazdy – zarówno na etapie prototypowania, jak i walidacji modeli w środowiskach kontrolowanych.

Przeprowadzone eksperymenty wykazały, że model YOLOv4 bardzo dobrze sprawdza się w zadaniu wykrywania obiektów w czasie rzeczywistym. Uzyskane rezultaty

były zadowalające zarówno pod względem liczby wykrytych obiektów, jak i ich klasyfikacji. Model skutecznie identyfikował pojazdy, ludzi oraz znaki drogowe w różnych warunkach oświetleniowych i pogodowych.

Na tle innych podejść, np. SSD czy Faster R-CNN, YOLOv4 wyróżnia się znacznie wyższą prędkością działania, co ma kluczowe znaczenie w kontekście pojazdów autonomicznych. Choć dokładność detekcji może być nieco niższa niż w przypadku modeli bardziej złożonych, to kompromis pomiędzy szybkością a precyzją został zachowany na bardzo dobrym poziomie, co potwierdza literatura przedmiotu oraz wyniki innych badaczy w tej dziedzinie.

Na podstawie przeprowadzonych testów można sformułować następujące wnioski:

1. Środowisko CARLA umożliwia realistyczną symulację warunków jazdy, co pozwala na skuteczne testowanie algorytmów percepcyjnych bez konieczności korzystania z rzeczywistych pojazdów.
2. Algorytm YOLOv4 zapewnia wysoką wydajność w czasie rzeczywistym, dzięki czemu możliwe jest wykrywanie wielu obiektów (samochody, piesi, znaki) przy zachowaniu płynności działania systemu.
3. Integracja detektora z kodem symulacyjnym CARLA, w tym przechwytywanie obrazu, przetwarzanie klatek oraz renderowanie detekcji, może być zrealizowana w sposób stabilny i modularny. Modułowa struktura umożliwia łatwe rozszerzanie funkcjonalności w przyszłości.
4. Wprowadzenie możliwości filtrowania klas oraz dynamicznego renderowania wyników na ekranie znacząco poprawia czytelność systemu i ułatwia analizę zachowania modelu.
5. Otrzymane rezultaty w pełni uzasadniają osiągnięcie założonych celów, gdyż zaprojektowany system wykrywa obiekty z dużą dokładnością i niskim opóźnieniem, co odpowiada wymaganiom stawianym przed systemami percepcji w pojazdach autonomicznych.

Mimo pozytywnych rezultatów, należy wskazać kilka ograniczeń i możliwości ich eliminacji:

- Model YOLOv4 nie zawsze poprawnie rozpoznaje obiekty w dużym oddaleniu lub w słabych warunkach oświetleniowych, co może wynikać z ograniczeń danych treningowych lub niskiej rozdzielczości obrazu.
- Detekcja odbywa się wyłącznie na podstawie danych wizyjnych – system mógłby zostać ulepszony poprzez fuzję danych z innych sensorów (np. LIDAR, radar), co poprawiłoby jego odporność na błędy w trudnych warunkach.
- Wyniki testowanego systemu zostały przedstawione na podstawie zrzutów ekranu, przedstawiając pojedynczą klatkę. Jako, iż system umożliwia rozpoznawanie obiektów w czasie rzeczywistym nie było możliwe przedstawienie tego w formie pisemnej.

- Testy przeprowadzono wyłącznie w środowisku symulacyjnym, które – mimo wysokiego realizmu – nie oddaje w pełni nieprzewidywalności świata rzeczywistego. Wdrożenie systemu na realnych danych wymagałoby dalszego dostosowania i oceny.
- Z powodu ograniczeń sprzętowych system został przetestowany w nieco gorszych warunkach, niż zalecanych przez sam symulator CARLA. Dodatkowo system do detekcji obiektów YOLO pobiera kolejne zasoby sprzętowe na skutek czego w gorszych warunkach pogodowych, takich jak noc czy deszcz oraz przy dużym natężeniu ruchu ilość klatek była bardzo niska.

Możliwy rozwój systemu

W oparciu o zdobytą wiedzę, możliwe są następujące kierunki rozwoju systemu:

- Zastosowanie modelu YOLOv8 lub innych nowszych architektur, które oferują lepszą dokładność i możliwość pracy na mniejszych urządzeniach (np. Jetson Nano, Raspberry Pi).
- Rozszerzenie systemu o moduł decyzyjny, który na podstawie wykrytych obiektów podejmuje działania – np. zatrzymanie pojazdu, zmiana pasa, ostrzeżenie o niebezpieczeństwie.
- Wprowadzenie mechanizmów oceny jakości detekcji (precision, recall, mAP) oraz zapisywanie wyników do analizy porównawczej.
- Implementacja systemu śledzenia obiektów (np. Deep SORT), co pozwoliłoby na analizę trajektorii oraz zachowań uczestników ruchu drogowego.
- Testowanie systemu na rzeczywistych nagraniach z kamer samochodowych, celem oceny jego przydatności w praktycznych wdrożeniach.

Ostatecznie, praca w pełni zrealizowała zakładane cele i potwierdziła skuteczność połączenia narzędzi symulacyjnych z algorytmami głębokiego uczenia w kontekście systemów percepcyjnych. W oparciu o przeprowadzone badania oraz analizę wyników można stwierdzić, że integracja YOLOv4 z symulatorem CARLA stanowi efektywne, elastyczne i przyszłościowe narzędzie do projektowania systemów rozpoznawania otoczenia w pojazdach autonomicznych.

Bibliografia

- [1] Politechnika Warszawska. Zasoby z e-sezamu: Architektura yolov4 i konwolucyjne sieci neuronowe, 2025.
- [2] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement.
<https://arxiv.org/abs/1804.02767>, 2018.
- [3] German Ros and Vladlen Koltun. Carla simulator.
https://carla.readthedocs.io/en/latest/start_introduction/, 2025.
- [4] Epic Games. Coordinate system and spaces in unreal engine.
<https://dev.epicgames.com/documentation/en-us/unreal-engine/coordinate-system-and-spaces-in-unreal-engine>, 2025.
- [5] Epic Games. Units of measurement in unreal engine.
<https://dev.epicgames.com/documentation/en-us/unreal-engine/units-of-measurement-in-unreal-engine>, 2025.
- [6] CARLA Simulator Team. Bounding boxes tutorial, 2022. Dostęp: 29.12.2025.
- [7] Epic Games. Unreal engine documentation.
<https://docs.unrealengine.com/4.27/en-US/>, 2025.
- [8] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*, 2015.
- [9] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1440–1448, 2015.
- [10] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.
- [11] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection.
<https://arxiv.org/abs/2004.10934>, 2020.
- [12] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning.
<https://www.nature.com/articles/nature14539>, 2015.

- [13] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection.
<https://arxiv.org/abs/1506.02640>, 2016.
- [14] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.
- [15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, 2016.
- [16] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, Sebastopol, CA, 2019.
- [17] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. *Efficient Processing of Deep Neural Networks*. Morgan & Claypool, San Rafael, CA, 2017.
- [18] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer, Cham, 2018.
- [19] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context.
<https://arxiv.org/abs/1405.0312>, 2014.
- [20] VISO AI. Understanding intersection over union (iou) for model accuracy, 2025. Dostęp: 09.01.2026.
- [21] Ultralytics. Ultralytics yolo documentation, 2025. Dostęp: 09.01.2026.
- [22] Ultralytics. Python usage - ultralytics yolo, 2025. Dostęp: 09.01.2026.