AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ

Projekt dyplomowy

System do detekcji przeszkód zrealizowany z wykorzystaniem kamery zdarzeniowej

Obstacles detection system using dynamic vision sensor

Autor: *Roman Nowak*
Kierunek studiów: *Automatyka i Robotyka*
Opiekun pracy: *dr inż. Tomasz Kryjak*

Kraków, 2025

Streszczenie

W ramach niniejszej pracy zrealizowano system detekcji obiektów – potencjalnych przeszkód dla robota mobilnego (drona) z wykorzystaniem kamery zdarzeniowej (ang. *Dynamic Vision Sensor (DVS)*). Czujnik ten wykorzystano w celu zapewnienia poprawnego działania w różnych warunkach oświetleniowych. Przeanalizowano sposoby rozwiązania zagadnienia w innych projektach dostępnych w literaturze i na tej podstawie stworzono własny algorytm detekcji. Przygotowano symulację SiL (ang. *Software in the Loop*) umożliwiającą testowanie algorytmu w świetle dziennym oraz w nocnych warunkach, oraz tworzenie różnorodnych scen z przeszkodami. Umieszczono w niej drona wyposażonego w symulowaną kamerę zdarzeniową. Przeprowadzono testy systemu wykrywania przeszkód, wykorzystując metodę SiL oraz gotowe zbiory danych z DVS. Przedstawiono i omówiono ich wyniki.

Abstract

In this work an object detection system – obstacles for a mobile robot (drone) using an event camera (DVS – Dynamic Vision Sensor) was created. This sensor was applied to ensure correct operation under different lighting conditions. Ways of solving the issue in other designs available in the literature were analysed and a custom detection algorithm was created. A SiL (Software in the Loop) simulation was prepared to test the algorithm in daylight and night conditions in variety of scenes with obstacles. A drone equipped with a simulated event camera was deployed in the scene of testing environment. Tests of the obstacle detection system were carried out using the SiL method and available datasets with DVS data. Their results were presented and discussed.

Spis treści

1. Wprowadzenie

Autonomiczne sterowanie pojazdami jest współcześnie szeroko rozwijanym i wykorzystywanym zagadnieniem. Realizacja takiego samodzielnego sterowania wymaga integracji precyzyjnego systemu. Jednym z jego kluczowych elementów, zapewniających skuteczność i bezpieczeństwo działania, jest system detekcji przeszkód. Może on być stworzony na różne sposoby, dzięki zastosowaniu kilku typów czujników. Częstym rozwiązaniem jest użycie kamer lub LiDAR-u (ang. *Light Detection and Ranging*). Obiecującym, nowoczesnym czujnikiem jest kamera zdarzeniowa, której wykorzystanie pozwala na osiągnięcie wymiernych korzyści. Wymaga to jednak opracowania algorytmów dostosowanych do działania tego czujnika, przykładowo do detekcji przeszkód na trasie ruchu autonomicznego pojazdu.

Kamery zdarzeniowe to nowoczesne czujniki, wykorzystywane w systemach wizyjnych. Słowo „kamera" może być mylące – ich działanie znacznie różni się od tradycyjnych kamer. DVS (ang. *Dynamic Vision Sensor*, czyli kamera zdarzeniowa) nie przechwytuje pełnych klatek obrazu. Jej sygnałem wyjściowym jest asynchroniczny strumień tzw. zdarzeń (ang. *event*). Są one generowane asynchronicznie dla każdego piksela DVS w wyniku zmiany jasności o zadany próg. Zmiany jasności pikseli pojawiają się, jeżeli w polu widzenia wystąpił ruch lub zmieniło się oświetlenie sceny. DVS często można traktować jako czujnik rejestrujący poruszające się obiekty.

Sposób działania kamer zdarzeniowych sprawia, że dysponują one szeregiem istotnych przewag nad tradycyjnymi kamerami:

- Wysoka rozdzielczość czasowa, czyli częstotliwość, z jaką generowane są dane, dzięki temu w danych z kamery zdarzeniowej nie występuje rozmycie ruchu,
- Niska latencja, co pozwala na dostarczanie dokładnych danych nawet o szybko poruszających się obiektach,
- Niskie zużycie energii,
- Szeroka rozpiętość tonalna (ang. *high dynamic range*), czyli zdolność do rejestracji zdarzeń w scenie o bardzo dużych różnicach w natężeniu światła,
- Wysoka czułość na występowanie zmian jasności pikseli.

Na wady i ograniczenia kamer zdarzeniowych składają się:

– Wrażliwość na występowanie szumu, czyli generowania fałszywych zdarzeń,

– Nietypowe dane wyjściowe – do ich przetwarzania wymagane są specjalistyczne algorytmy,

– Brak zdolności do rejestracji statycznych obiektów,

– Wysoka cena.

Wspomniane cechy tworzą z kamer zdarzeniowych bardzo dobry czujnik do wykorzystania wszędzie tam, gdzie występują szybko poruszające się obiekty, lub gdy mamy do czynienia z ograniczonym lub zmiennym oświetleniem. Wykorzystywane są w systemach wizyjnych samochodów, a także robotów mobilnych i dronów. Dzięki zastosowaniu DVS można tworzyć dla nich wydajne i efektywne systemy percepcji.

Zaprojektowanie systemu detekcji przeszkód, z użyciem kamery zdarzeniowej, pozwala w pełni wykorzystać jej zalety. Poprawnie zaimplementowany system detekcji potencjalnych zagrożeń staje się jednym z podstawowych elementów robota mobilnego, umożliwiających jego autonomiczne sterowanie. Dzięki dostarczaniu możliwie aktualnych i precyzyjnych danych o przeszkodach, znajdujących się na ścieżce robota, zyskuje się możliwość jej korekty. W konsekwencji otrzymuje się kompletny, bezpieczny i skuteczny system sterowania robotem mobilnym.

1.1. Cele projektu

Celem projektu była realizacja systemu detekcji obiektów – potencjalnych przeszkód dla robota mobilnego – z wykorzystaniem kamery zdarzeniowej. Czujnik ten powinien zapewnić poprawne działanie w różnych niekorzystnych warunkach oświetleniowych. Dodatkowym celem była próba implementacji zaprojektowanego systemu na platformie eGPU lub SoC FPGA.
Realizacja projektu została podzielona na kilka etapów:

W **pierwszym** etapie należało wykonać przegląd literatury na temat detekcji obiektów z wykorzystaniem kamer zdarzeniowych i na tej podstawie wybrać podejście do implementacji programowej.

W **drugim** etapie, symulacji SiL (ang. *Software-in-the-Loop*), należało wybrać symulator, narzędzie do realizacji oraz język programowania. Następnie w symulacji powinien zostać umieszczony model robota (drona) wyposażonego w kamerę zdarzeniową. Powinna zostać także zapewniona możliwość sterowania nim oraz tworzenia zróżnicowanych scen z przeszkodami. Należało również opracować kilka scenariuszy testowych o różnym stopniu skomplikowania.

Celem **trzeciego** etapu była implementacja algorytmu detekcji przeszkód i jego ewaluacja w stworzonym środowisku wraz z oceną skuteczności oraz złożoności obliczeniowej.

Celem **ostatniego** etapu projektu było przeniesienie systemu na wbudowaną platformę obliczeniową – w pierwszej kolejności na eGPU Jetson, a w drugiej, opcjonalnej, na SoC FPGA. System należało przetestować w formie HiL (ang. *Hardware-in-the-Loop*). W tym celu konieczne było opracowanie sposobu wymiany danych pomiędzy środowiskiem symulacyjnym a platformą obliczeniową. Oczekiwanym

rezultatem tego etapu była weryfikacja działania systemu i porównanie z modelem programowym. Dodatkowo, należało podjąć próbę uruchomienia rozwiązania na rzeczywistym pojeździe.

1.2. Zawartość pracy

Rozdział **drugi** (2) zawiera informacje teoretyczne, niezbędne do pełnego zrozumienia treści pracy. Opisano w nim kamery zdarzeniowe, wbudowane platformy obliczeniowe oraz wyjaśniono pojęcia takie jak: ROS (ang. *Robot Operating System*), *Software in the Loop*, *Hardware in the Loop* czy triangulacja.

W rozdziale **trzecim** pracy (3) przeprowadzono przegląd literatury. Przeanalizowano rozwiązania zastosowane w podobnych projektach.

Rozdział **czwarty** (4) to opis realizacji symulacji, stworzonej w celu testowania rozwijanego systemu. Zawarte są w nim informacje o przygotowaniu środowiska symulacyjnego oraz przedstawione i opisane wyniki.

W rozdziale **piątym** (5) opisano zaprojektowany algorytm detekcji przeszkód. Wyjaśniono etapy jego działania.

W rozdziale **szóstym** (6) przedstawiono przeprowadzone w różnych scenariuszach testy systemu. Zaprezentowano i oceniono ich wyniki.

Rozdział **siódmy** (7) zawiera podsumowanie całej pracy. Opisane są w nim efekty i wyniki. Na ich podstawie sformułowane są wnioski. Zdefiniowano również plany dalszego rozwoju stworzonego w ramach pracy projektu.

2. Wstęp teoretyczny

Rozdział zawiera podstawowe informacje teoretyczne niezbędne do zrozumienia opisywanych w pracy zagadnień. Wyjaśniono kluczowe pojęcia używane w tekście oraz przedstawiono metody wykorzystane w części praktycznej projektu.

2.1. Kamery zdarzeniowe

Kamery zdarzeniowe (ang. *Dynamic Vision Sensors (DVS)*) to urządzenia rejestrujące obraz w sposób odmienny od tradycyjnych kamer klatkowych, które zapisują obrazy w regularnych odstępach czasu (np. 30 klatek na sekundę). W odróżnieniu od nich, kamery zdarzeniowe rejestrują jedynie zmiany jasności w poszczególnych pikselach matrycy. Porównanie sposobu, w jaki rejestrowane są dane w kamerze tradycyjnej i zdarzeniowej, przedstawione jest na rysunku 2.1. Wyjście zwykłej kamery to klatki, pojawiające się co pewien stały czas i zawierające pełny widok, podczas gdy jedynym obiektem pojawiającym się na wyjściu DVS jest poruszający się obiekt, rejestrowany jako osobne, często występujące asynchroniczne zdarzenia.

Każdy piksel DVS działa niezależnie i rejestruje zdarzenie w momencie, gdy różnica jasności padającego na nie światła przekroczy określony próg.

Szczegółowe informacje na temat działania i sposobów wykorzystania tych kamer można znaleźć w artykule [1].

Kamera zdarzeniowa podaje na wyjście asynchroniczny strumień zdarzeń (ang. *event*), z których każdy jest krotką w formacie:

$$(t, x, y, p)$$

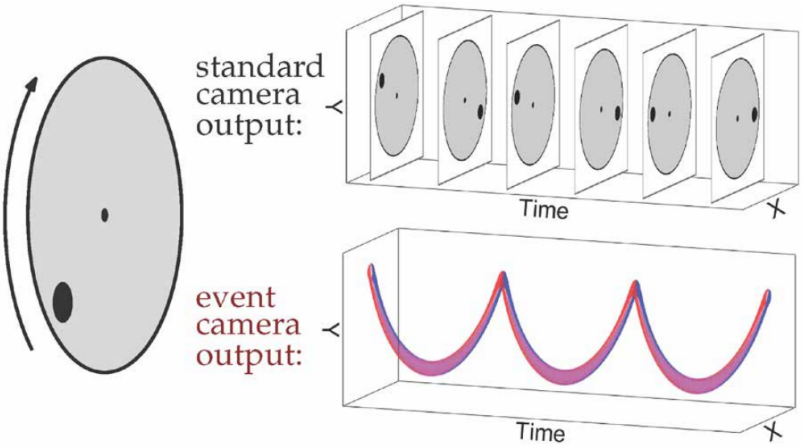
gdzie:

t – czas wystąpienia zdarzenia (ang. *timestamp*),

x, y – położenie piksela w matrycy,

p – polaryzacja zdarzenia (ang. *polarity*) – mówi o kierunku zmiany jasności piksela, przyjmuje wartości -1 lub 1 (czasami przyjmuje się wartości 0 oraz 1).

W celu wizualizacji zdarzeń na płaszczyźnie obrazu stosuje się metody ich reprezentacji w postaci tzw. *event frame*'ów, czyli pewnego rodzaju klatek zdarzeniowych. Istnieje kilka podejść, ale w tym projekcie są one tworzone przy wykorzystaniu reprezentacji *exponentially decaying time surface* [2]. W



Rys. 2.1. Porównanie sposobu rejestracji danych przez zwykłą kamerę i DVS.

metodzie tej wartość piksela na obrazie obliczana jest na podstawie jego *timestampu*, przy zastosowaniu funkcji (2.1). Dzięki temu jasność piksela na obrazie wskazuje na to, jak dawno wystąpiło tam zdarzenie.

$$f(t,u) = p_u * e^{\frac{t_u - t}{\tau}} \tag{2.1}$$

gdzie:

- u, p_u, t_u – odpowiednio: zdarzenie, jego polaryzacja i czas wystąpienia,
- $f(t,u)$ – wartość jasności piksela na *event frame*'ie,
- τ – czas, który obejmuje klatka, czyli różnica czasu wystąpienia najstarszego i najmłodszego zdarzenia.

DVS pozbawione są ograniczeń związanych z rejestracją danych ze stosunkowo niewielką częstotliwością – tradycyjne kamery odczytują obraz z częstotliwością kilkudziesięciu Hz, natomiast kamery zdarzeniowe działają z minimalnym opóźnieniem, sięgającym mikrosekund (np. dla DAVIS346 – ok. 12 µHz), czyli teoretyczna częstotliwość odczytu zdarzeń sięga dziesiątek kHz. Efektem takiego sposobu działania jest niska latencja. Dzięki rejestracji wyłącznie pojedynczych zdarzeń zamiast całych klatek (brak redundancji danych), kamery zdarzeniowe pozwalają na oszczędzanie pamięci i energii. Efekt ten jest szczególnie wyraźny w przypadku statycznych scen, w których pojawia się niewiele ruchomych obiektów. Kolejną zaletą jest wysoka odporność na występowanie zmiennego oświetlenia, dzięki szerokiej rozpiętości tonalnej (ang. *high dynamic range*), osiągniętej dzięki asynchroniczności i niezależności działania pikseli.

Ograniczeniem DVS jest wrażliwość na występowanie szumu, czyli pojawianie się na wyjściu fałszywych zdarzeń. Kolejną wadą jest nietypowy format danych wyjściowych, który uniemożliwia bezpośrednie wykorzystanie do ich przetwarzania algorytmów stosowanych dla zwykłych kamer. Warto również wspomnieć o ograniczonych możliwościach dostarczania informacji o statycznych scenach, gdy ruch

nie występuje lub jest bardzo powolny. Koszt zakupu kamery zdarzeniowej znacznie przewyższa cenę tradycyjnej kamery.

2.2. Robot Operating System (ROS)

ROS2 (ang. *Robot Operating System 2*) to otwartoźródłowy *framework*, który ułatwia projektowanie, rozwój i wdrażanie systemów robotyki. Jest platformą programistyczną zapewniającą zestaw gotowych funkcji, bibliotek i narzędzi wspomagających komunikację, percepcję, sterowanie oraz współpracę robotów. ROS2 można traktować jako „warstwę pośrednią", która zarządza wymianą informacji między różnymi elementami robota. Jego głównym zadaniem jest zapewnienie płynnego przepływu danych między tymi elementami i umożliwienie ich współpracy. ROS2 wspiera języki programowania C++ oraz Python. W projekcie wykorzystany został ten drugi.

Działanie ROS-a bazuje na równolegle pracujących i asynchronicznie komunikujących się węzłach (ang. *nodes*), czyli jednostkach wykonawczych, które realizują konkretne zadania w systemie. Każdy węzeł działa jako niezależny proces i może komunikować się z innymi węzłami. Podstawowy sposób komunikacji to *topic*, który jest kanałem komunikacyjnym, na którym węzły mogą publikować dane lub je subskrybować, umożliwiając asynchroniczną wymianę informacji. Dane przesyłane przez *topic* mają postać wiadomości (ang. *messages*), które posiadają predefiniowany typ, np. dane obrazu, pozycja robota czy informacje tekstowe. Dzięki tej architekturze węzły mogą niezależnie wymieniać informacje, co zapewnia modułowość i elastyczność systemu.

Przykład działania:

– Węzeł A (np. kamera) **publikuje** (czyli regularnie nadaje do węzłów odbierających (ang. *subscriber*)) dane obrazu na *topic* – na przykład */camera/image_raw*,

– Węzeł B (np. algorytm przetwarzania obrazu) **subskrybuje** ten sam *topic*, aby odbierać dane,

– Dane (wiadomości) zaczynają być przesyłane z węzła A do węzła B.

Inne sposoby na komunikację między węzłami to:

– usługi (ang. *services*) – pozwalają na dwustronną komunikację, w której jeden węzeł wysyła żądanie i otrzymuje od drugiego odpowiedź,

– akcje (ang. *actions*) – pozwalają na wykonywanie zadań długotrwałych, gdzie węzeł może otrzymywać aktualizacje postępu i anulować zadanie w trakcie jego realizacji.

2.3. Testowanie systemów wbudowanych

W procesie rozwoju systemów oprogramowania i algorytmów dla systemów wbudowanych, ważna jest możliwość testowania i oceny działania na wszystkich etapach pracy. W zależności od stopnia postępów, wykorzystuje się w tym celu różne metody.

2.3.1. Software in the Loop

Software in the Loop (SiL) to metoda testowania stosowana na wczesnym etapie rozwijania systemu wbudowanego. W procesie SiL cały system jest uruchamiany w środowisku komputerowym, gdzie wszystkie fizyczne komponenty są symulowane, np. przy użyciu wirtualnych czujników, silników czy urządzeń wejścia/wyjścia. Następnie algorytmy sterowania, przetwarzania danych czy komunikacji są testowane w tym wirtualnym środowisku, co pozwala na wczesne wykrycie błędów i optymalizację oprogramowania przed przejściem do bardziej kosztownych etapów testów.

Zalety metody *Software in the Loop* to:

– Skrócenie czasu testowania – umożliwia szybkie testowanie i weryfikację algorytmów bez potrzeby korzystania z fizycznych komponentów, co przyspiesza proces rozwoju,

– Niskie koszty – testowanie w środowisku symulacyjnym eliminuje koszty związane z wykorzystaniem sprzętu oraz ryzyko jego uszkodzenia podczas testów,

– Bezpieczeństwo – pozwala na testowanie potencjalnie niebezpiecznych scenariuszy bez ryzyka uszkodzenia rzeczywistych urządzeń lub zagrożenia dla ludzi,

– Łatwość w modyfikacjach – symulowane środowisko pozwala na łatwe wprowadzanie zmian,

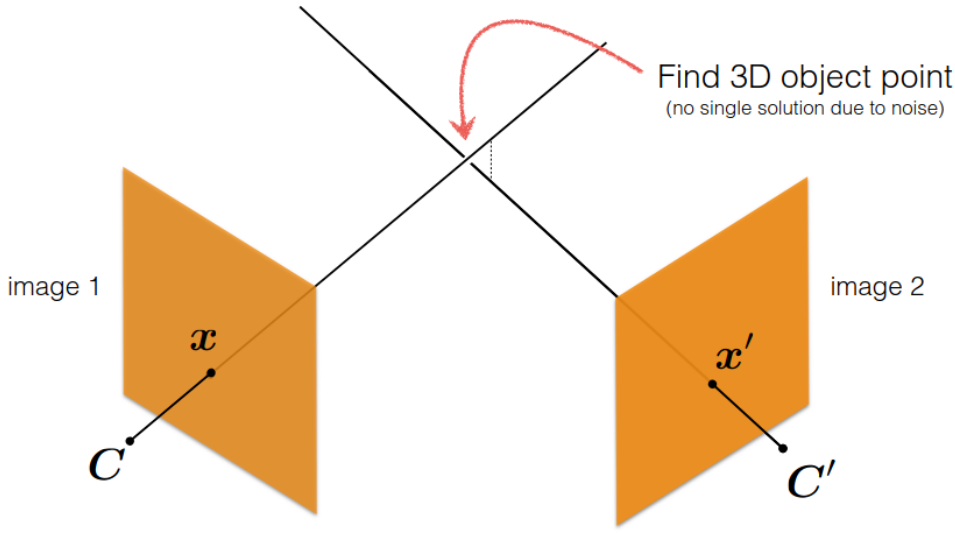
– Szybka identyfikacja błędów – umożliwia szybkie wykrycie problemów w kodzie.

2.3.2. Hardware in the Loop

Hardware in the Loop (HiL) to metoda testowania, w której rzeczywisty sprzęt jest połączony z symulowanym środowiskiem, co pozwala na weryfikację działania systemu na docelowej platformie obliczeniowej. W tej metodzie część systemu jest symulowana komputerowo (dynamika obiektów, emulacja czujników), a część działa na rzeczywistym sprzęcie. Dzięki temu można testować interakcje między sprzętem a oprogramowaniem, co umożliwia dokładne sprawdzenie działania systemu przed jego pełnym wdrożeniem.

HiL pozwala na realistyczne testowanie algorytmów sterowania i interakcji między różnymi elementami systemu, bez potrzeby uruchamiania pełnego prototypu w rzeczywistych. To podejście zapewnia większe bezpieczeństwo i zmniejsza ryzyko uszkodzenia sprzętu, ponieważ testy mogą odbywać się w kontrolowanym środowisku. Ponadto umożliwia optymalizację algorytmów i dokładną walidację działania systemu.

Przewaga nad SiL polega na możliwości sprawdzenia działania oprogramowania na docelowym sprzęcie – warunki są więc bardziej zbliżone do rzeczywistych.



Rys. 2.2. Idea triangulacji. C i C' to punkty centralne na matrycach kamer. x i x' to położenia tego samego obiektu (punktu w przestrzeni 3D) na płaszczyznach obrazu. Długość odcinka między punktami C i x jest określona przez ogniskowe kamer. W idealnym przypadku (brak szumu, dokładnie ten sam czas rejestracji obu klatek) półproste będące przedłużeniem tych odcinków powinny przeciąć się jednym punkcie – rozwiązaniu. Źródło: [3].

2.4. Triangulacja jako metoda estymacji głębi

Aby przeprowadzić estymację głębi punktu widocznego na obrazie z dwóch kamer, w pierwszej kolejności konieczne jest odczytanie ich parametrów. Dostarczają one informacji na temat położenia kamery w przestrzeni oraz o sposobie przekształcania trójwymiarowych punktów świata rzeczywistego na dwuwymiarowy obraz.

Parametry wewnętrzne:

– macierz K (2.2) – macierz, która zawiera informacje o parametrach wewnętrznych takich jak: ogniskowa dla osi x – f_x oraz y – f_y i współrzędne punktu głównego (centrum obrazu) – c_x i c_y,

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

– wektor dystorsji (2.3) – opisuje zniekształcenie obrazu. Współczynniki k to wartość dystorsji radialnej, a p – tangencjalnej. Jeżeli wektor ten nie jest zerowy, to należy użyć go do korekcji zniekształceń obrazu.

$$d = \begin{bmatrix} k_1 & k_2 & k_3 & p_1 & p_2 \end{bmatrix} \tag{2.3}$$

Parametry zewnętrzne:

– translacja (2.4) – położenie kamery w przestrzeni 3D,

$$t = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.4}$$

– orientacja – zwykle w formie macierzy R o rozmiarze 3×3. Dostarcza informacji o rotacji kamery.

Zasada triangulacji przedstawiona jest na rysunku 2.2. Należy znaleźć współrzędne punktu przecięcia półprostych poprowadzonych przez punkty C i x obu kamer. W tym celu należy wyznaczyć ich macierze projekcji (2.5), które łączą parametry zewnętrzne i wewnętrzne.

$$P = K \cdot [R|t] \tag{2.5}$$

Macierzy projekcji można użyć do sformułowania równań, w których niewiadomymi będą współrzędne poszukiwanego punktu $\mathbf{X}$.

$$P \cdot \mathbf{X} = x, \quad P' \cdot \mathbf{X} = x' \tag{2.6}$$

gdzie:

P i P', to macierze projekcji odpowiednio pierwszej i drugiej kamery – wartości znane,

$\mathbf{X}$ to wektor współrzędnych szukanego punktu w trójwymiarowej przestrzeni,

x i x' to położenie obrazu obiektu na płaszczyznach kamer w przestrzeni 3D – wartości znane.

Pojedyncze równanie z (2.6) można zapisać jako:

$$\begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \cdot \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} \tag{2.7}$$

Współrzędne punktów x i x' można otrzymać przez przekształcenie ich współrzędnych z obrazu do przestrzeni kamery za pomocą równania (2.8).

$$\begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} = K^{-1} \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{2.8}$$

gdzie:

u i v to współrzędne obiektu na obrazie.

W celu wyznaczenia głębi, trzeba znaleźć rozwiązanie najlepiej spełniające równania (2.6). Jednoznaczne rozwiązanie prawdopodobnie nie istnieje z uwagi na niedokładności wynikające chociażby z powodu występowania szumu.

Jeżeli układem odniesienia w trójwymiarowej przestrzeni, w jakim rozpatrywane były wszystkie współrzędne punktów, była jedna z kamer, to odległość do niej jest równa współrzędnej Z_0 wynikowego wektora $\mathbf{X}$.

Układ stereo kamer oznacza dwie kamery skonfigurowane do wspólnej pracy w celu estymacji głębi. Często umieszczone są one w jednej obudowie w taki sposób, by zachowywały te same parametry (z wyjątkiem translacji). Oznacza to, że przekształcają one punkty z przestrzeni 3D na płaszczyznę obrazu w ten sam sposób i nie są obrócone względem siebie. Pozwala to na uproszczenie obliczeń.

2.5. Platformy sprzętowe

Aby zintegrować system wbudowany na rzeczywistym dronie lub robocie mobilnym, należy zaimplementować go na wbudowanej platformie obliczeniowej, która zapewni możliwość działania oprogramowania.

W celu implementacji systemów wizyjnych, często stosowana platforma – mikrokontroler, wyposażony w mikroprocesor, może okazać się nieodpowiednim rozwiązaniem. Powodem tego jest duża ilość danych przetwarzana przez algorytmy wizyjne. Szczególnie kiedy wymagane jest niezawodne działanie systemu w czasie rzeczywistym, należy skorzystać z innych platform obliczeniowych, które pozwalają na zrównoleglenie obliczeń i w konsekwencji przyśpieszą czas ich wykonywania.

Pierwszym rozwiązaniem jest zastosowanie układu SoC (ang. *System on Chip*) z wbudowanym procesorem graficznym (ang. *Graphics Processing Unit (GPU)*). Takim układem jest płytka z serii eGPU Jetson. SoC oznacza układ zawierający kompletny system elektroniczny – w tym przypadku mikrokontroler, GPU i peryferia. Takie rozwiązanie pozwala na tworzenie aplikacji o wysokiej wydajności i implementację złożonych algorytmów przetwarzania obrazu lub realizację uczenia maszynowego na platformie wbudowanej.

Sposobem, który daje nawet większe możliwości zrównoleglenia obliczeń, jest zastosowanie platformy FPGA (ang. *Field Programmable Gate Array*). Termin ten oznacza programowalną matrycę bramek logicznych, która pozwala na projektowanie i implementowanie specyficznych układów cyfrowych dostosowanych do konkretnych zadań. FPGA składa się z tysięcy (a czasem milionów) programowalnych bloków logicznych i połączeń.

Budowa FPGA:

– Bloki logiczne – są to podstawowe jednostki obliczeniowe FPGA, które mogą implementować funkcje logiczne lub pełnić rolę komórki pamięci; każdy blok logiczny składa się z LUT (ang. *Look-Up Table*) oraz przerzutników i multiplekserów,

– Macierz połączeń – umożliwia programowalne połączenia między blokami logicznymi, wejściami/wyjściami (I/O) i innymi komponentami,

– Wejścia/wyjścia – służą do komunikacji z zewnętrznymi urządzeniami i peryferiami,

– Specjalizowane komponenty – większość nowoczesnych FPGA zawiera dodatkowe elementy, takie jak: bloki DSP (ang. *Digital Signal Processing*) do szybkich obliczeń numerycznych, pamięć RAM, zintegrowane kontrolery dla standardów komunikacyjnych (np. Ethernet, PCIe).

FPGA umożliwia równoległe przetwarzanie danych, zapewniając wysoką wydajność w aplikacjach czasu rzeczywistego. Są doskonałe do prototypowania, eliminując potrzebę kosztownej produkcji ASIC (ang. *Application-Specific Integrated Circuit* – specjalizowany układ scalony), i gwarantują niskie opóźnienia. Wady FPGA obejmują pracochłonny (w porównaniu do innych platform sprzętowych) proces implementacji systemów, a także ograniczone zasoby pamięciowe.

3. Przegląd literatury

Problem skutecznej detekcji oraz unikania przeszkód przez roboty mobilne lub drony często pojawia się w projektach z dziedziny robotyki opisanych w literaturze. Potrzeba integracji takich systemów wynika z chęci zwiększenia ich autonomiczności – każdy system sterowania robotem mobilnym, który ma działać samodzielnie, bez ingerencji operatora, musi być wyposażony w system pełniący taką funkcję.

Do detekcji zagrożeń na trasie robota często wykorzystuje się proste czujniki zbliżeniowe lub czujniki odległości. W ten sposób jednak uzyskuje się jedynie stosunkowo późną informację o obecności lub dodatkowo o odległości do przeszkody. Takie podejście okaże się wystarczające, jeśli robot porusza się wolno, przeszkody są duże i nieruchome, a wymagana reakcja robota na wystąpienie barier na trasie nie wymaga żadnych dodatkowych danych (przykładowo – robot skręca w prawo o 90 stopni, za każdym razem gdy napotka przeszkodę).

Realizowany w ramach projektu system powinien jednak mieć szersze możliwości. Oprócz dostarczania danych o wystąpieniu przeszkody, ma on za zadanie informować także o jej położeniu, kształcie oraz prędkości, a wszystko to odpowiednio wcześnie, tak aby możliwe było zaplanowanie trajektorii i zapewniona możliwość szybkiego przemieszczania się robota. Takie podejście do problemu daje możliwości reakcji na przeszkody będące w ruchu oraz na realizację bardziej złożonych algorytmów sterowania w celu ich skutecznego uniknięcia.

Sposoby wykrywania przeszkód zebrane i opisane są w artykułach [4], [5] oraz w [6]. Detekcja obiektów może być realizowana na podstawie danych pochodzących z różnych typów czujników. Najczęściej wykorzystywane sensory to:

– LiDAR (ang. *Light Detection and Ranging*) – czujnik zbierający przestrzenne dane o otoczeniu (odległości do otaczających go obiektów), za pomocą krótkich sygnałów świetlnych. Główną zaletą jest wysoka precyzja, natomiast wadą – duża ilość danych, które należy przetworzyć, a więc wysokie wymagania obliczeniowe systemów opartych na tych czujnikach.

– RADAR (ang. *Radio Detection and Ranging*) – działa dzięki sygnałom radiowym. Charakteryzuje się potencjalnie dużym dystansem detekcji, ale może mieć problem z wykrywaniem obiektów niewielkich oraz takich, które słabo odbijają fale w tym zakresie.

– SONAR (ang. *Sound Navigation and Ranging*) – wykorzystuje sygnały dźwiękowe i używany jest głównie tam, gdzie warunki negatywnie wpływają na działanie innych czujników (np. pod wodą).

Rys. 3.1. Przykład manewru uniku z projektu [7].

– Czujniki ultradźwiękowe – to urządzenia wykorzystujące fale dźwiękowe o wysokiej częstotliwości (powyżej zakresu słyszalnego dla ludzkiego ucha, tj. > 20 kHz) do wykrywania obiektów i mierzenia odległości. Działają na zasadzie emisji ultradźwięków i analizy echa odbitego od przeszkody. Ich zaletą jest niezależność od oświetlenia sceny i niski koszt, natomiast wadą stosunkowo niewielki dystans detekcji.

– Kamery – niewątpliwą zaletą ich stosowania jest niska cena. Wykorzystując kamery do detekcji obiektów, stosuje się różne podejścia. Przeszkody można wykrywać na podstawie ich znanych właściwości – kształtu i rozmiaru, lub na podstawie ich ruchu i prędkości. Projektuje się systemy wykorzystujące pojedynczą kamerę, jak i takie z dwiema kamerami skonfigurowanymi w układ stereo, co daje większe możliwości przy obliczaniu głębi.

– **Kamery zdarzeniowe** – szerzej opisane w podrozdziale 2.1

W literaturze znaleźć można artykuły opisujące projekty o podobnej tematyce, w ramach których został zaprojektowany i zaimplementowany system detekcji i unikania przeszkód. W celu wyboru metody rozwiązania tego problemu przeanalizowano sposoby detekcji zastosowane w wybranych, najbardziej zbliżonych do tematu projektu publikacjach. Cele w tych artykułach częściowo pokrywają się z celami projektu (1.1), co pozwala na wzorowanie się na nich i czyni je szczególnie pomocnymi w realizacji zadania.

Dynamic obstacle avoidance for quadrotors with event cameras (*Davide Falanga, Kevin Kleber, Davide Scaramuzza*) [7] – projekt, w którym autorzy zaprojektowali i zaimplementowali system unikania przeszkód dla czterowirnikowego drona. Celem było ominięcie szybko poruszających się (do $10\ \frac{\mathrm{m}}{\mathrm{s}}$) w kierunku drona obiektów (na przykład rzuconej w niego piłki – rys. 3.1). Statyczne bariery na trasie drona są ignorowane. System działa zarówno na zewnątrz, jak i we wnętrzach budynków. Autorzy, oprócz skuteczności, skupiają się na osiągnięciu możliwie niskiej latencji w wykrywaniu przeszkód – końcowa wartość to $3.5\ ms$. Jest to możliwe właśnie dzięki zastosowaniu kamer zdarzeniowych.

Na system detekcji przeszkód z projektu [7] składają się kolejne etapy:

1. Kompensacja ruchu własnego drona – autorom zależy na wykrywaniu wyłącznie poruszających się przeszkód, jednak zdarzenia w DVS generowane są również przez ruch samego drona i obiekty statyczne. Na podstawie danych z czujnika IMU (ang. *inertial measurement unit*) – rotacji drona, odróżniane są statyczne elementy sceny od dynamicznych.

2. Segmentacja przeszkód – dzięki wynikom poprzedniego etapu możliwe jest oddzielenie zdarzeń odpowiadających za przeszkody od statycznej części sceny. Dalej przetwarzane są tylko obiekty w ruchu.

3. Obliczenie przepływu optycznego za pomocą algorytmu Lucasa-Kanade [8], w celu usprawnienia późniejszej klasteryzacji zdarzeń. Przepływ optyczny dostarcza dane o wektorach ruchu pikseli na obrazie, co ułatwia odróżnienie różnych obiektów, które znajdują się blisko siebie, ale poruszają się inaczej.

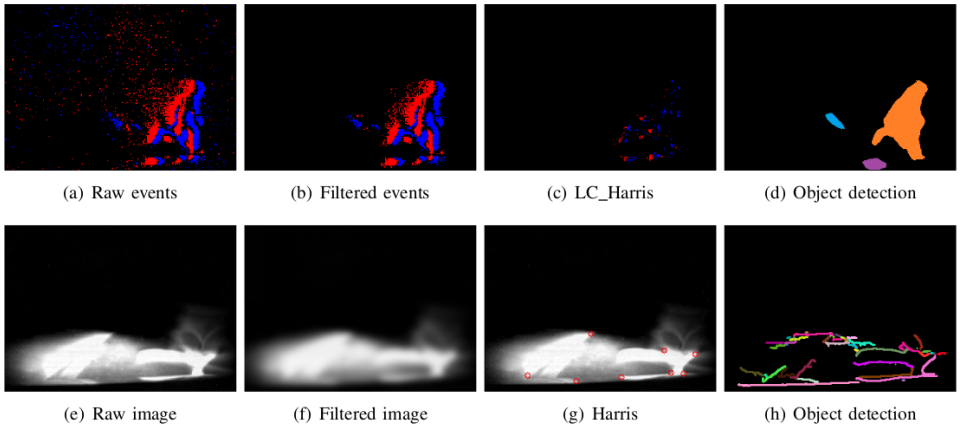
4. Klasteryzacja – za pomocą algorytmu DBScan [9] separuje się zdarzenia odpowiadające poszczególnym obiektom – scena może zawierać wiele przeszkód. Dodatkową rolą tego etapu jest usunięcie szumu.

5. Estymacja pozycji w przestrzeni 3D – w przypadku gdy używana jest tylko jedna kamera zdarzeniowa, autorzy w celu wyznaczenia głębi, ograniczają się do przeszkód o znanym rozmiarze. Wtedy ich odległość od kamery może być łatwo wyznaczona na podstawie ich szerokości w klatce obrazu. W przypadku dwóch kamer w układzie stereo [5], możliwa jest estymacja głębi dla wszystkich obiektów za pomocą triangulacji [10].

6. Projekcja pozycji przeszkód z układu współrzędnych kamery na układ współrzędnych świata i estymacja prędkości przeszkód.

EVDodgeNet: Deep Dynamic Obstacle Dodging with Event Cameras (*Nitin J. Sanket, Chethan M. Parameshwara, Chahat Deep Singh, Ashwin V. Kuruttukulam, Cornelia Fermüller, Davide Scaramuzza, Yiannis Aloimonos*) [11] – projekt, w którym problem zdefiniowany jest bardzo podobnie do [7] – unikanie przez drona szybko poruszających się obiektów różnych kształtów i rozmiarów. Podejście do implementacji znacznie się jednak różni. Autorzy do kolejnych etapów detekcji obiektów na podstawie danych z kamery zdarzeniowej wykorzystują szereg płytkich sieci neuronowych. Pierwsza z nich odpowiedzialna jest za poprawę jakości ramek zdarzeniowych (usunięcie rozmycia, redukcja szumów). Kolejna odpowiada za uzyskiwanie odometrii (estymacja pozycji). Zadaniem ostatniej jest segmentacja niezależnie poruszających się obiektów. Modele trenowane były wyłącznie przy wykorzystaniu symulacji komputerowej. Skuteczność gotowego systemu autorzy oceniają na 70%. System przetestowany został również w trudnych warunkach oświetleniowych, w których mógł efektywnie pracować, dzięki zastosowaniu DVS.

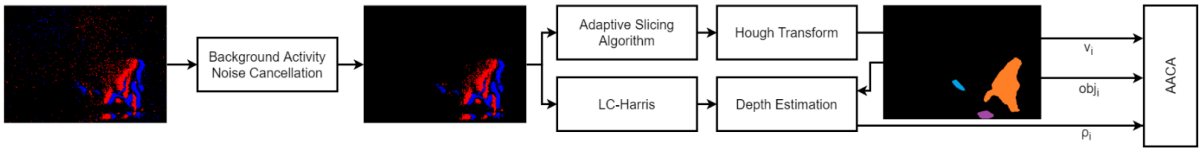
Night vision obstacle detection and avoidance based on Bio-Inspired Vision Sensors (*Jawad Naveed Yasin, Sherif Abdelmonem Sayed Mohamed, Mohammad Hashem Haghbayan, Jukka Heikkonen, Hannu Tenhunen, Muhammad Mehboob Yasin, Juha Plosila*) [12] - projekt, w którym autorzy koncentrują się na detekcji przeszkód w ograniczonych warunkach oświetleniowych – stąd wybór kamery zdarzeniowej. Zadanie to jest realizowane przy wykorzystaniu tradycyjnego podejścia – bez głębokich sieci neuronowych. Działanie gotowego systemu zostało przetestowane na gotowych zbiorach danych. Na koniec porównane zostały wyniki otrzymane dla tradycyjnej kamery oraz DVS.



(a) Raw events (b) Filtered events (c) LC_Harris (d) Object detection

(e) Raw image (f) Filtered image (g) Harris (h) Object detection

Rys. 3.2. Porównanie działania algorytmu detekcji przeszkód z artykułu [12] dla DVS i tradycyjnej kamery. Zastosowanie kamery zdarzeniowej umożliwia detekcję w nocnych warunkach.



Rys. 3.3. Schemat działania algorytmu detekcji z projektu [12].

System detekcji przeszkód z projektu [12] działa w następujący sposób:

1. Filtracja szumu tła – w danych otrzymywanych z kamery zdarzeniowej występują błędnie wygenerowane zdarzenia, które nie są częścią właściwego sygnału i które należy z niego usunąć w celu obniżenia wymagań obliczeniowych dla późniejszych etapów (mniej zdarzeń do przetworzenia) oraz zwiększenia niezawodności algorytmu. W tym celu użyto algorytmu kNN (ang. *k-nearest neighbour*) [13].

2. Podział odczytywanych zdarzeń na grupy, jako które będą dalej przetwarzane. Do każdej *klatki*, zapisywane jest N zdarzeń. Wartość N dobierana jest dynamicznie w zależności od prędkości obiektów. *D*opasowanie płaszczyzn w chmurze punktów – zdarzeń, do poszczególnych obiektów na scenie za pomocą algorytmu RHT (ang. *Randomised Hough Transform*).

3. Wyznaczenie punktów charakterystycznych za pomocą metody bazującej na algorytmie Harris [14].

4. Estymacja głębi dla każdej z przeszkód. Autorzy mimo zastosowania pojedynczej kamery, obliczają odległość za pomocą triangulacji, dzięki danym o pozycji i rotacji kamery.

Innym, często spotykanym podejściem do wykrywania obiektów za pomocą kamer zdarzeniowych, jest użycie uczenia maszynowego i głębokich sieci neuronowych. Tego typu modele do detekcji zostały

stworzone w ramach wielu dostępnych w literaturze projektów i stanowią bogatą bazę narzędzi. Przykładowo wymienić można algorytmy:

- *RED* [15] – wprowadza rekurencyjną architekturę, umożliwiającą uczenie na podstawie zdarzeń bez konieczności rekonstrukcji obrazu,
- *ASTMNet* [16] – wykorzystuje moduł pamięci do detekcji obiektów w sposób ciągły, eliminując potrzebę przekształcania zdarzeń na obrazy,
- *RVT* [17] – redukuje czas przetwarzania do $12\,\mu s$ przy zachowaniu wysokiej wydajności i efektywności parametrów,
- *DMANet* [18] – wykorzystuje podwójną pamięć (modeluje ją jako krótko i długotrwałą) do skutecznej agregacji zdarzeń dla lepszej detekcji obiektów,
- *TEDNet* [19] – wprowadza etykiety widoczności obiektów i strategie ich śledzenia, umożliwiając detekcję w przypadku braku relatywnego ruchu wobec kamery.

4. Symulacja Software in the Loop

Pierwszym elementem, który w ramach projektu należało stworzyć, jest symulacja SiL (ang. *Software in the Loop*). Celem jest otrzymanie narzędzia, które pozwoli na wygodne i szybkie testowanie tworzonego w późniejszym etapie algorytmu detekcji przeszkód.

Przed przystąpieniem do realizacji zadania, warto zdefiniować wymagania i oczekiwania, jakie są postawione przed gotowym oprogramowaniem:

1. Symulacja powinna umożliwić testowanie dla różnych scenariuszy (np. inne przeszkody w zmiennych pozycjach).

2. W symulacji należy umieścić robota/drona, wyposażonego w odpowiednie czujniki, z których dane powinny być łatwo dostępne.

   Jako pojazd przenoszący kamerę zdarzeniową, dla którego będą wykrywane przeszkody, został wybrany czterowirnikowy dron. Taka decyzja uwarunkowana była kilkoma czynnikami:

   – Zachowana zostaje analogia do analizowanych w ramach przeglądu literatury projektów, w których również wykorzystywano podobny pojazd. Ułatwia to wzorowanie się na zastosowanych w publikacjach metodach.

   – Dron ma szerokie możliwości unikania wykrytych przeszkód, a więc pełnego wykorzystania dostarczonych mu przez algorytm danych. Dzięki temu możliwości pojazdu nie będą ograniczeniem dla efektywności algorytmu sterującego, korzystającego z informacji o przeszkodach.

3. Użytkownik powinien mieć możliwość sterowania (za pomocą napisanego kodu) robotem/dronem poruszającym się po scenie symulacji.

4. Symulacja powinna dostarczać dane z kamery zdarzeniowej, umieszczonej na robocie/dronie.

4.1. Symulacja w Gazebo

Jako środowisko symulacyjne, w którym całość została zrealizowana, wybrano oprogramowanie Gazebo.

Gazebo jest otwartoźródłowym symulatorem robotyki, wykorzystywanym do testowania i rozwijania algorytmów sterowania robotami w realistycznym, trójwymiarowym środowisku. Zapewnia dostęp do zaawansowanego modelu fizyki oraz wielu czujników. Jest zintegrowany z platformą programistyczną ROS2. Gazebo umożliwia umieszczanie na scenie zarówno obiektów statycznych, jak i ruchomych. Symulator pozwala też na stworzenie nocnej sceny.

Oprócz wspomnianych funkcjonalności, Gazebo jest środowiskiem, które świetnie nadaje się do zastosowania w projekcie z dwóch głównych powodów:

– Gazebo jest wspierane przez inne narzędzia, których użycie jest konieczne; w szczególności ważna jest możliwość użycia wykorzystywanego w projekcie narzędzia do sterowania lotem drona w symulacji,

– Gazebo wykorzystuje i wspiera użycie *frameworka* ROS2, który jest konieczny do sterowania dronem oraz odczytywania danych z czujników.

W celu umieszczenia w symulacji drona oraz zapewnienia możliwości sterowania nim należy wykorzystać odpowiednie oprogramowanie. Do tego celu wybrany został system **PX4 Autopilot**. Jest to zaawansowane, *open-source*'owe oprogramowanie sterujące dla bezzałogowych statków latających (ale również dla innych robotów mobilnych). Jest jednym z najczęściej wykorzystywanych systemów autopilota w dziedzinie robotyki powietrznej i autonomicznych pojazdów, wspieranym przez aktywną społeczność i dużą liczbę integracji z różnorodnym sprzętem. Najważniejszymi dla projektu zaletami tego systemu są:

– Wsparcie dla czterowirnikowych dronów,

– Możliwość sterowania za pomocą zewnętrznego oprogramowania, wykorzystując ROS2 – tryb Offboard.

4.1.1. Przygotowanie środowiska

Pierwszym krokiem realizacji symulacji była instalacja i przygotowanie środowiska. Dla systemu Linux w dystrybucji Ubuntu 22.04 pobrane zostały:

– ROS2 w dystrybucji Humble,

– Symulator Gazebo,

– Python 3.10.

W celu umieszczenia drona w środowisku symulacyjnym i umożliwienia sterowania nim, wykorzystany został system PX4 Autopilot. Należy go odpowiednio skonfigurować do pracy z Gazebo i ROS-em, tak żeby było możliwe pisanie własnych węzłów (ang. *node*) w Pythonie. Są to procesy, których zadaniem będzie na przykład sterowanie dronem lub przetwarzanie danych z czujników.

Ponieważ konfiguracja wszystkich narzędzi, tak by skutecznie ze sobą współpracowały, jest zadaniem stosunkowo trudnym – szczególnie dla nowego użytkownika, warto skorzystać z gotowego przykładu i dołączonej do niego instrukcji konfiguracji. Dzięki temu, dopóki dokładnie wykonuje się opisane kroki, zyskuje się względną pewność, że błędy popełnione na tym etapie nie utrudnią realizacji zadania w przyszłości. Taki przykład dostępny jest na Githubie pod linkiem: *https://github.com/ARK-Electronics/ROS2_PX4_Offboard_Example*. Autorzy tego projektu poprawnie skonfigurowali go i przygotowali do dalszej rozbudowy dla innych użytkowników.

Zawarte w przykładzie funkcjonalności obejmują:

– Podstawowa, pusta scena w symulatorze Gazebo z umieszczonym w niej czterowirnikowym dronem dostępnym w PX4 Autopilot,

– Przykład sterowania dronem, w którym jako wartość zadana używany jest wektor prędkości – na tej podstawie nowemu użytkownikowi łatwiej jest stworzyć własny system sterowania,

– Zaimplementowane zostało sterowanie kierunkiem i prędkością ruchu drona za pomocą strzałek na klawiaturze – ta funkcjonalność nie jest wymagana w projekcie, jednak jest ciekawym przykładem, na podstawie którego można rozbudowywać symulację,

– Dodatkowy moduł wizualizacji, wykorzystujący narzędzie *rviz*, służący do wyświetlenia ścieżki, jaką dron przebył w czasie trwania symulacji – ten moduł również nie jest niezbędny, ale stanowi interesujący dodatek,

– Uruchamianie wszystkich modułów symulacji za pomocą jednej komendy, co znacząco ułatwia późniejsze z niej korzystanie.

Po dostosowaniu przykładu do zastosowania w projekcie, otrzymuje się gotową do dalszej pracy platformę, w której możliwe jest kontrolowanie drona za pomocą kodu pisanego w języku Python oraz obsługa danych z czujników.

Model drona wykorzystywany w symulacji to x500 (rys. 4.1) (nazwa używana w dokumentacji PX4). Jest on wyposażony w czujnik IMU (ang. *inertial measurement unit*), który składa się z akcelerometru i żyroskopu i dostarcza danych o ruchu i orientacji drona w przestrzeni, niezbędnych do skutecznego sterowania nim.

4.1.2. Losowe generowanie toru przeszkód

Dron powinien móc poruszać się w scenie zawierającej statyczne przeszkody, które mógłby wykrywać i omijać. W celu zapewnienia zmiennych warunków, w jakich algorytm będzie testowany, należy stworzyć losowo generowany tor przeszkód. Żeby to osiągnąć, przygotowano odpowiedzialny za to skrypt w Pythonie.

Świat symulacji w Gazebo generowany jest na podstawie kodu zawartego w pliku *.sdf*. To w nim definiuje się używane pluginy (czyli moduły odpowiedzialne za różne zadania), czujniki, a także umieszcza

Rys. 4.1. Model drona x500, dostępny w PX4 w symulacji w środowisku Gazebo.

obiekty i decyduje o ich położeniu. Skrypt generujący przeszkody uruchamia się za każdym razem, gdy włączana jest symulacja, i umieszcza obiekty w odpowiednim miejscu w kodzie w pliku *.sdf*.

Działanie skryptu generującego przeszkody:

- Przekazanie parametrów wejściowych:
  - Liczba przeszkód,
  - Lista nazw przeszkód – wybór z trzech dostępnych: drzewo iglaste, drewniany słup energetyczny, szeroki cylindryczny słup,
  - Obszar, który ma być pokryty przeszkodami,
  - Minimalny dystans między dwoma obiektami,
- Losowe wygenerowanie pozycji i orientacji dla każdej przeszkody, tak by zachowany był minimalny dystans między nimi,
- Odpowiednie sformatowanie tekstu i wpisanie go do pliku *.sdf*.

Oprócz przeszkód w symulacji umieszczony został jeszcze stolik – stanowisko startowe dla drona, oraz bramka oznaczająca metę – cel trasy drona. Wszystkie wykorzystane obiekty pochodzą z szerokiej biblioteki modeli dostępnych za darmo dla Gazebo: Gazebo Fuel. Łatwo dostępne modele dostosowane do szybkiego użycia w symulacji to kolejna z zalet tego środowiska.

Dodatkowo można symulować ograniczone warunki oświetleniowe przez ręczne zmiany intensywności oświetlenia w pliku *.sdf*.

Przykładowe sceny można zobaczyć na rysunku 4.2.

Rys. 4.2. Przykłady losowo wygenerowanego toru przeszkód dla drona.

4.1.3. Umieszczenie w symulacji kamery zdarzeniowej

Pomimo że w bibliotekach środowiska Gazebo dostępnych jest wiele czujników, brakuje gotowego symulatora kamery zdarzeniowej. Dla projektu, w którym stanowi ona podstawowy element, jest to poważny problem. Jednak dzięki znajomości sposobu działania DVS, można podjąć próbę przybliżenia otrzymywanych z tego czujnika danych za pomocą zwykłej kamery.

W pierwszej kolejności należy umieścić na dronie tradycyjną kamerę. Ta jest dostępna w Gazebo w ramach pakietu *gz-sensors*. Rozdzielczość kamery ustawiona jest na 240×180, co odpowiada rozdzielczości kamery zdarzeniowej dostępnej na rynku i powszechnie używanej (na przykład w [12]) DAVIS240. Wykorzystując ROS-a, stworzony został węzeł, w którym dane z kamery były na bieżąco odczytywane i przekształcane do postaci klatek obrazu, a następnie dalej przetwarzane.

Kamera zdarzeniowa generuje zdarzenia z każdą zmianą jasności piksela o stały próg. Najprostszy sposób na ich otrzymanie na podstawie klatek obrazu tradycyjnej kamery może więc wyglądać następująco:

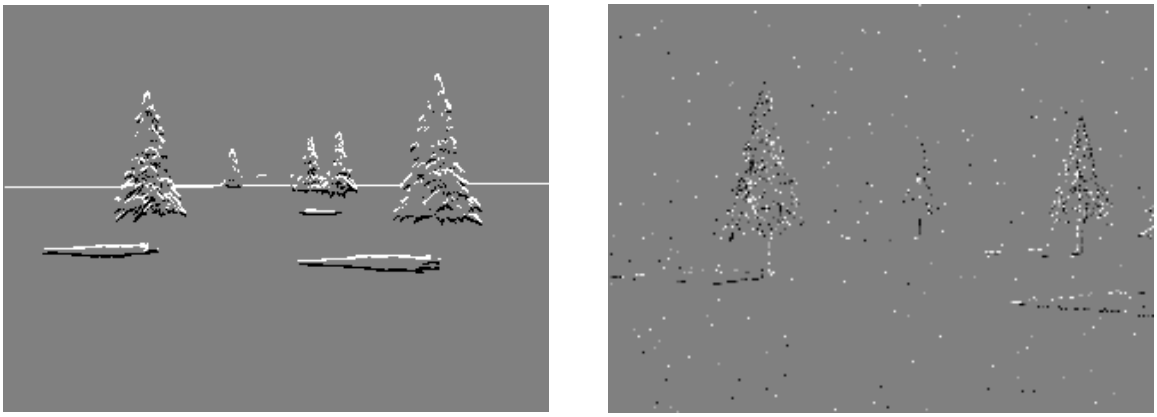
– Dwie kolejne klatki obrazu zostają od siebie odjęte (najpierw bieżąca klatka od poprzedniej, a następnie w odwrotnej kolejności, aby otrzymać zdarzenia o dodatniej i ujemnej zmianie jasności),

– Wynikowe obrazy są progowane przy zastosowaniu stałego progu,

– Z tak otrzymanych masek odczytywane są wartości położenia x i y, dla wszystkich niezerowych pikseli i zapisywane jako zdarzenie razem z informacją o aktualnym czasie (ang. *timestamp*) i polaryzacji (ang. *polarity*).

Takie proste podejście zostało zaimplementowane jako osobny moduł symulujący kamerę zderzeniową. Odczytuje on klatki z kamery, a następnie symuluje wyjście DVS, publikując przygotowane w tym celu własne wiadomości (ang. *messages*), zawierające odczytane zdarzenia.

Dodatkowo porównano wyjście symulowane tą metodą z wyjściem rzeczywistego DVS, za pomocą zbioru danych pochodzących z kamery zdarzeniowej DAVIS240 *shapes rotation* [20]. Przykładowe porównanie *event frame*'ów jest widoczne na rysunku 4.3. Na klatce, na której reprezentowane są zdarzenia



Rys. 4.3. Przykładowe *event frame*'y dla uproszczonego symulatora DVS (po lewej) oraz v2e (po prawej). Słabo widoczne kształty obiektów na klatce z v2e są wynikiem ograniczonej liczby zdarzeń na jednej klatce i szumu.

pochodzące z uproszczonej metody symulowania DVS, piksele (poza szarym tłem) przyjmują wyłącznie wartości skrajne (czarny, biały). Oznacza to, że nie ma różnicy w czasie ich powstania – wszystkie zostały wygenerowane w jednym momencie przez odjęcie dwóch klatek obrazu. Dodatkowo na klatce z v2e widoczny jest szum.

Niestety, mimo że w korzystnych warunkach oświetleniowych i przy niskich prędkościach względnych obiektów, *event frame* wygląda bardzo podobnie do tego otrzymanego z rzeczywistych zdarzeń, to tak mocno uproszczone podejście nie może być zastosowane w celu symulacji DVS. Decyduje o tym kilka czynników:

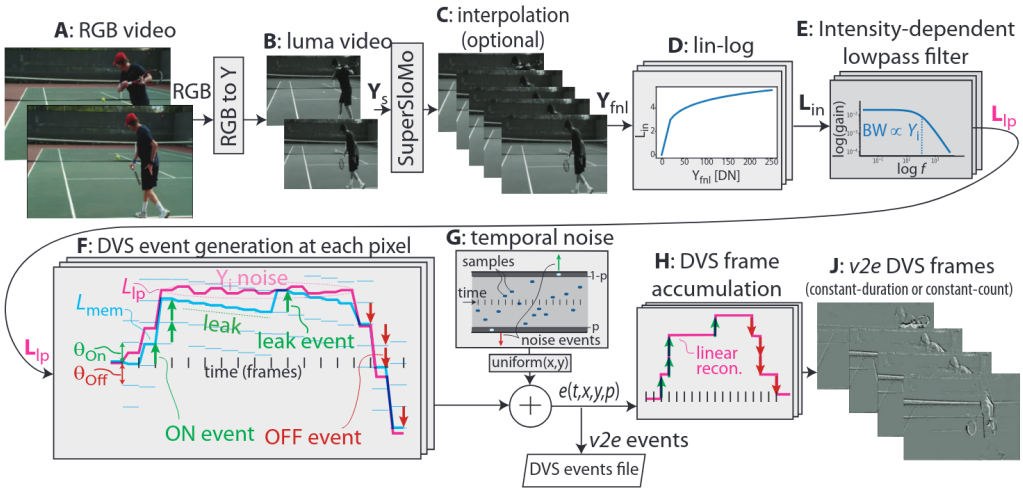
– Jak łatwo zauważyć, w ten sposób traci się wszystkie z istotnych zalet kamery zdarzeniowej – zdarzenia rejestrowane są dokładnie wtedy kiedy klatki obrazu i z taką samą czułością, jaką dysponuje tradycyjna kamera,

– W przypadku szybciej poruszających się obiektów, liczba klatek na sekundę, jaką rejestruje kamera, może okazać się niewystarczająca – wtedy zdarzenia pozytywne i negatywne rozdzielają się, co zmniejsza jakość otrzymywanych danych,

– W takim podejściu brakuje uwzględnienia szumów, jakie występują w rzeczywistej kamerze zdarzeniowej.

Oczywiście nie wszystkie z tych problemów da się całkowicie wyeliminować, ponieważ tak naprawdę czujnikiem zbierającym dane jest zwykła kamera. Można je natomiast ograniczyć przez zastosowanie bardziej dopracowanego systemu symulującego DVS.

Zaawansowane i dopracowane sposoby na konwersję klatek obrazu z tradycyjnej kamery na zdarzenia można znaleźć w literaturze. Kilka z nich jest zaimplementowanych jako gotowe do użycia narzędzia. Przykładem takiego projektu może być ESIM [21] lub v2e [22].



Rys. 4.4. Schemat działania symulacji kamery zdarzeniowej w v2e [22].

Do zastosowania w projekcie wybrane zostało v2e, ponieważ oprócz funkcji otrzymywania zdarzeń z wcześniej nagranego filmu, oferuje ono przygotowaną do użycia w Pythonie bibliotekę, która umożliwia konwersję kolejnych klatek obrazu na zdarzenia w czasie rzeczywistym. Dzięki temu v2e można zastosować w węźle ROS-a.

v2e to *framework* stworzony w Pythonie, który umożliwia konwersję klatek obrazu ze zwykłej kamery na możliwie realistyczny i odpowiadający pierwowzorowi strumień zdarzeń. Proces symulacji (na rys. 4.4) jest wieloetapowy i obejmuje:

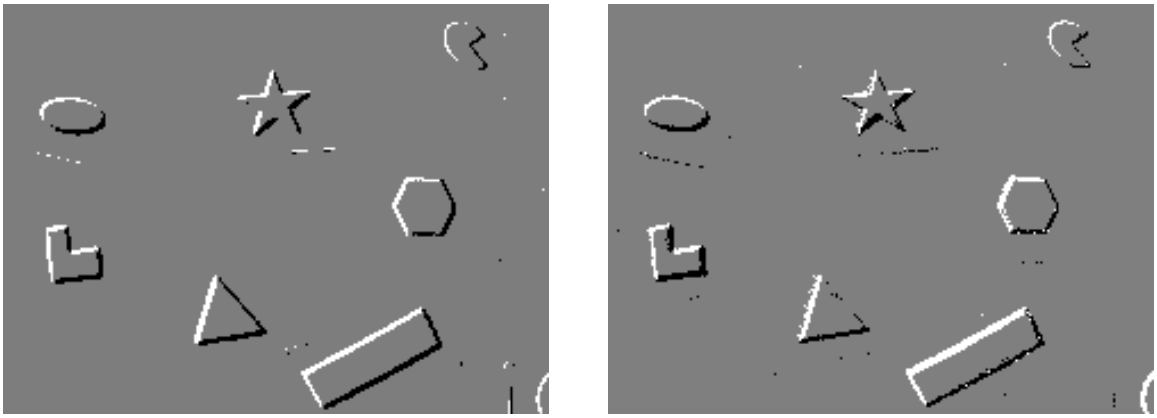
– Interpolację klatek, w celu zwiększenia ich częstotliwości, za pomocą narzędzia SuperSloMo [23],

– Uwzględnienie logarytmicznej skali, w jakiej działają piksele DVS,

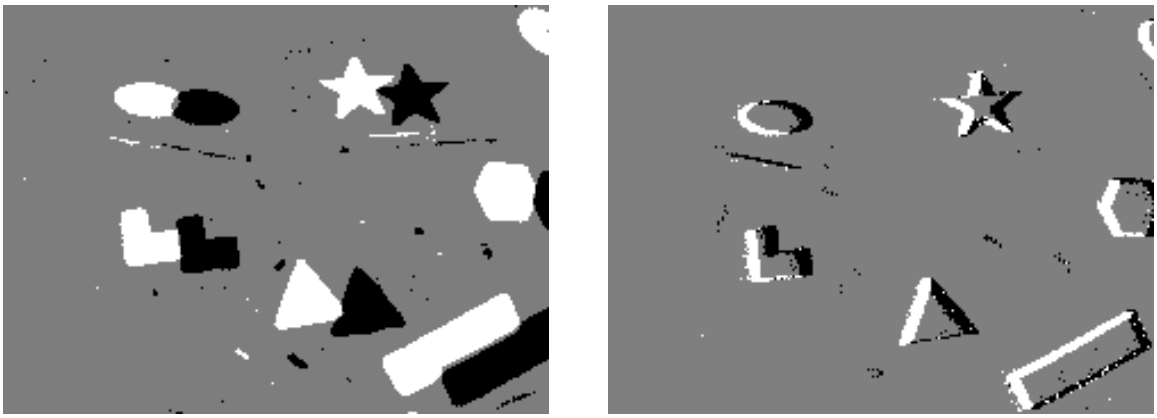
– Generację zdarzeń dla każdego piksela na podstawie zmian jasności na poszczególnych klatkach,

– Dodanie do sygnału szumów, występujących w rzeczywistych kamerach zdarzeniowych.

Wspomniany emulator ma istotną wadę – nie pozwala w pełni wykorzystać zalet systemu v2e. Ponieważ narzędzie SuperSloMo [23], stosowane do interpolacji klatek i zwiększenia FPS (ang. *frames per second* – klatki na sekundę) sygnału wejściowego, wymaga zarówno obecnej, jak i przyszłej klatki obrazu, nie jest możliwe jego zastosowanie bez wprowadzenia dodatkowej latencji. Wobec tego sposobem na poprawę jakości danych wejściowych kamery jest podniesienie jej FPS. Im większa dynamika sceny (prędkość obiektów), tym bardziej rośnie wymagana wartość FPS. Niestety, wzrost tej wartości znacząco wpływa na wymagania obliczeniowe symulacji i jakość jej wykonania. Wobec tego, FPS ustawiane jest na najwyższą możliwą wartość, która nie spowalnia działania symulacji zbyt mocno. Testy przeprowadzane były przy wartości 100 – zdecydowanie zbyt niskiej dla szybko poruszających się obiektów, jednak należało wziąć pod uwagę ograniczenia sprzętowe.



Rys. 4.5. *Event frame*'y otrzymane z klatek obrazu za pomocą uproszczonej metody symulacji DVS (po lewej) i z danych *shapes rotation* [20] (po prawej), dla małej prędkości ruchu kamery.



Rys. 4.6. *Event frame*'y otrzymane z klatek obrazu za pomocą uproszczonej metody symulacji DVS (po lewej) i z danych *shapes rotation* [20] (po prawej), gdy kamera porusza się szybciej.

4.2. Rezultaty

Rysunki 4.5 oraz 4.6 zawierają porównanie uproszczonego sposobu na symulowanie DVS z *event frame*'ami uzyskanymi na podstawie zbioru danych. Szczególnie obserwując 4.5, można ulec wrażeniu, że prosta metoda jest wystarczająco dobra i klatki niewiele się od siebie różnią. Wystarczy jednak spojrzeć na 4.6, żeby uświadomić sobie, że uproszczenie to jest zbyt daleko posunięte, żeby ta metoda nadawała się do zastosowania w symulacji. Przy szybkim ruchu kamery obiekty rozdzielają się, tworząc dwa osobne kształty. Generowana jest duża liczba zdarzeń. Negatywnie wpływa to zarówno na złożoność obliczeniową algorytmu detekcji, jak i jego skuteczność – jedna przeszkoda może być wykryta jako dwie, jej rozmiar i położenie mogą nie odpowiadać rzeczywistemu.

5. Algorytm detekcji przeszkód

Celem tego etapu projektu było zaprojektowanie, zaimplementowanie w języku programowania Python, a na koniec przetestowanie algorytmu detekcji przeszkód.

Przyjęte założenia i cele:

1. Algorytm powinien pozwalać na wykrywanie zarówno statycznych (względem układu odniesienia świata), jak i poruszających się przeszkód.

2. Powinien być możliwie nieskomplikowany obliczeniowo w celu zachowania możliwości późniejszej implementacji na wbudowanej platformie obliczeniowej. Ze względu na dużą ilość przetwarzanych danych, systemy wizyjne często charakteryzują się znaczną złożonością obliczeniową, dlatego warto projektować je tak, żeby ją ograniczyć.

3. Musi dostarczać dane o obecności, położeniu i odległości do przeszkody.

4. Ma za zadanie wykrywać wiele obiektów znajdujących się w polu widzenia kamery w tym samym czasie.

5.1. Wybrane podejście

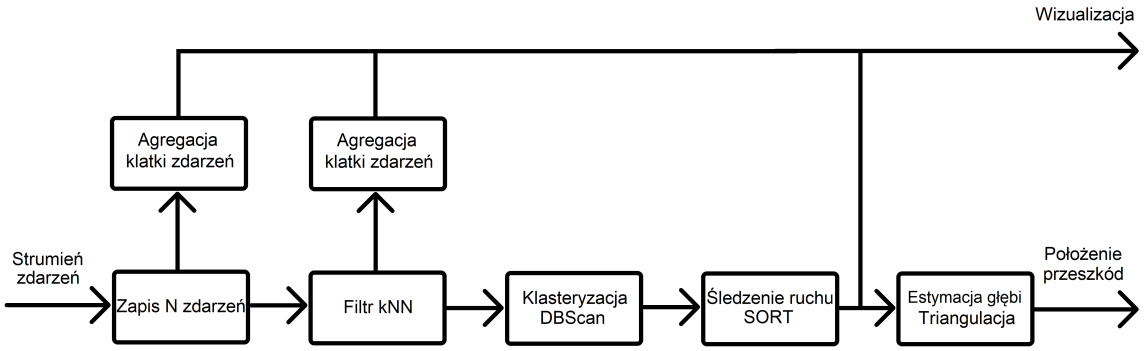
W celu ułatwienia późniejszej implementacji na platformie wbudowanej, wszystkie podejścia bazujące na głębokich sieciach neuronowych zostały odrzucone. Algorytmy te charakteryzują się wysoką złożonością obliczeniową. Dla platform wbudowanych, które dysponują ograniczonymi zasobami, ich implementacja stanowi duże wyzwanie, szczególnie przy wymogu realizacji obliczeń w czasie rzeczywistym.

Podejście do rozwiązania problemu detekcji zostało przygotowane na podstawie analizy dwóch podobnych projektów, w ramach których zaprojektowano zbliżone systemy: [7] oraz [12].

Algorytm wykrywania obiektów składa się z następujących etapów:

1. Odczyt i akumulacja N zdarzeń.

2. Filtracja sygnału, w celu eliminacji szumu tła (ang. *background activity noise*).



Rys. 5.1. Schemat blokowy działania algorytmu.

3. Podział zdarzeń składających się na sygnał na klastry odpowiadające oddzielnym przeszkodom.

4. Przypisanie do wykrytych obiektów indywidualnych identyfikatorów na podstawie algorytmu śledzącego ich ruch.

5. Estymacja głębi dla każdej przeszkody za pomocą triangulacji oraz ustalenie ich pozycji w trójwymiarowej przestrzeni.

Wyjścia z etapów pierwszego, drugiego i czwartego, opcjonalnie mogą być wizualizowane, co ułatwiło przeprowadzanie testów.

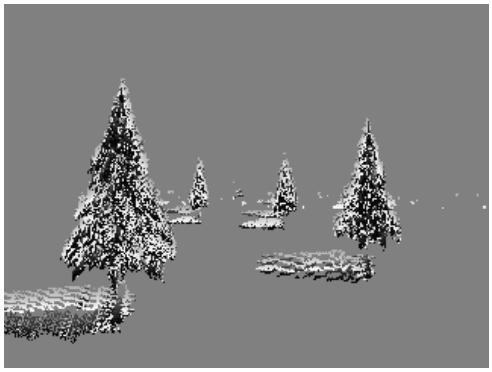
Na rysunku 5.1 przedstawiono blokowy schemat działania systemu. Strzałkami zaznaczony jest przepływ danych między kolejnymi modułami (etapami) algorytmu. Na wejście podawany jest asynchroniczny strumień zdarzeń, a na wyjściu otrzymywane są dane o położeniu i prędkości przeszkód. Dodatkowo odczytywać można obrazy wizualizujące przetwarzane dane na wyjściu wybranych etapów. Można je obserwować dzięki reprezentacji w postaci ramek zdarzeniowych, realizowanych osobno przed i po filtracji.

5.2. Implementacja

5.2.1. Akumulacja zdarzeń

Aby móc wykorzystać otrzymywane w asynchronicznym strumieniu zdarzenia do dalszej detekcji obiektów, najprostszym rozwiązaniem jest ich akumulacja i zrzutowanie na płaszczyznę obrazu. Do tego zagadnienia można podejść na dwa sposoby:

– Zbieranie zdarzeń z określonego interwału czasowego. To intuicyjne rozwiązanie ma swoją wadę – szczególnie jeśli czas akumulacji jest zbyt długi, na stworzonych za pomocą zgrupowanych w ten sposób zdarzeń ramkach może być widoczne rozmycie ruchu szybko poruszających się obiektów. Można to zauważyć na rysunku 5.2.



Rys. 5.2. Rozmycie ruchu widoczne, gdy czas odczytywania zdarzeń w celu ich podziału jest zbyt długi w stosunku do względnej prędkości obiektu.

– Zbieranie N zdarzeń – ten sposób pozwala ograniczyć wspomniany problem, dlatego to on został wykorzystany. Wartość N jest stała – jest parametrem detektora, dla symulacji w Gazebo została ustawiona na 1500. Jednak to rozwiązanie także ma swoją wadę – w przypadku zbyt dużej dynamiki obiektów, gdy generowane jest dużo zdarzeń, wzrasta częstotliwość tworzenia nowych *event frame*'ów, a wraz z nią spada liczba milisekund na ich przetworzenie przed pojawieniem się kolejnej ramki.

5.2.2. Filtracja sygnału

Ponieważ w danych z rzeczywistych kamer zdarzeniowych często obecny jest szum, należy go usunąć, co zwiększy skuteczność algorytmu dzięki odrzuceniu fałszywych zdarzeń oraz zmniejszeniu ich ogólnej liczby.

Dokładne metody filtracji szumu kamer zdarzeniowych opisane są w artykułach [13] oraz [24].

W projekcie zastosowany został stosunkowo prosty, ale skuteczny algorytm kNN (ang. *k-Nearest Neighbour*). Działa on według prostej zasady (rys. 5.3). Jako sygnał uznawane są tylko te zdarzenia, w których otoczeniu $n \times n$ w czasie nie większym niż t od chwili obecnej wystąpiło k zdarzeń. Wartości n, t oraz k to parametry filtra. Dla symulatora zostały one dobrane eksperymentalnie i ustawione jako:

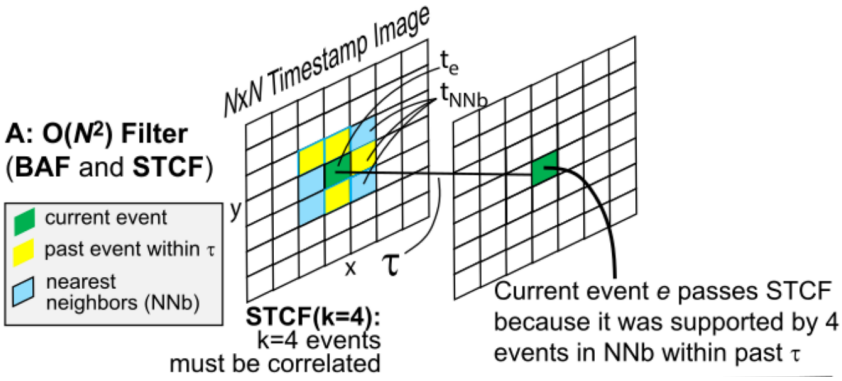
$$n = 3;\ \ t = \frac{1}{60}s;\ \ k = 3$$

Implementacja:

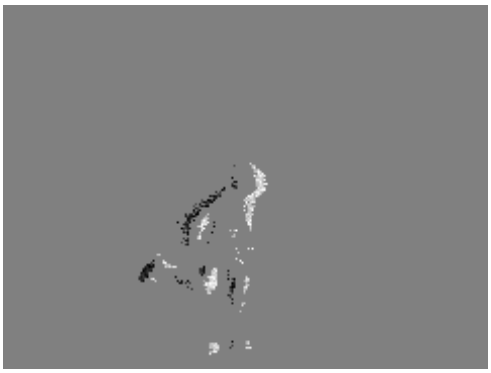
– Tworzona jest macierz SAE (ang. *Surface of Active Events*) o rozmiarze obrazu, w której przechowywany jest czas wystąpienia ostatniego zdarzenia dla każdego piksela,

– Dla każdego zdarzenia, podawanego na wejście detektora, sprawdzana jest odpowiednia dla niego ramka z macierzy SAE i liczeni są jego sąsiedzi,

– Jeśli jest ich co najmniej k, zdarzenie jest uznawany za poprawny, jeśli nie, jest odrzucany.



Rys. 5.3. Sposób działania filtra kNN. Źródło: [24].



Rys. 5.4. Przykład ramki zdarzeniowej uzyskanej z danych ze zbioru *night run* [25].

Zarówno oryginalne, jak i przefiltrowane zdarzenia, opcjonalnie mogą być zapisywane jako ramki zdarzeniowe w celu wizualizacji pracy algorytmu. Przykład wizualizacji zdarzeń po filtracji przedstawiony jest na rysunku 5.4.

5.2.3. Segmentacja obiektów

Celem tej fazy jest rozpoznanie w trójwymiarowej chmurze zdarzeń (wymiary: x, y i t) odrębnych obiektów.

W pierwszej kolejności do jego realizacji zostało wykorzystane podejście zaprezentowane w [12] – dopasowywanie płaszczyzn za pomocą algorytmu RHT (ang. *Randomised Hough Transform*) [26].

RHT to algorytm, który może być zastosowany do dopasowania równań (w tym przypadku równań płaszczyzn w postaci $ax + by + cz + d = 0$) do zbioru punktów. Bazuje on na transformacji Hougha, ale charakteryzuje się mniejszą złożonością obliczeniową. Ze zbioru punktów (tutaj zdarzeń, rozpatrywanych jako punkty w przestrzeni 3D o współrzędnych t, x oraz y) losowo wybierane są trzy, dla których obliczane są parametry płaszczyzny. Jest to powtarzane wielokrotnie, a obliczane płaszczyzny akumulowane są w przestrzeni Hougha. Jest to taka przestrzeń, w której wymiarami są parametry równania

(a, b, c, d), w której dla każdego ich zestawu zliczana jest liczba jego wystąpień. Za dopasowane płaszczyzny uznawane są te, które zebrały najwięcej głosów (wystąpiły najwięcej razy).

Algorytm RHT został zaimplementowany w Pythonie na podstawie artykułu [26]. Następnie rozwiązanie to zostało przetestowane. Niestety, nie udało się osiągnąć żadnych zadowalających wyników spełniających zdefiniowany dla tej fazy cel. Mogło to być spowodowane trudnościami z doborem parametrów algorytmu. Nie można też wykluczyć niewykrytych błędów w kodzie.

RHT zastąpiono innym podejściem obecnym w projekcie [7]. Do przypisania zdarzeń do poszczególnych obiektów wykorzystywany jest algorytm do klasteryzacji DBScan [9]. Charakteryzuje się on kilkoma ważnymi zaletami:

– Nie wymaga określenia z góry liczby klastrów,

– Znajduje klastry o dowolnym kształcie,

– Jest odporny na występowanie szumu,

– Jest zaimplementowany i dostępny do wykorzystania w Pythonie w bibliotece *scikit-learn*.

DBScan ma dwa parametry wejściowe: ϵ – maksymalny promień sąsiedztwa oraz $minP$ – minimalną liczbę punktów wchodzących w skład klastra. Dla symulacji zostały one ustawione na:

$$\epsilon = 7; \;\; minP = 30$$

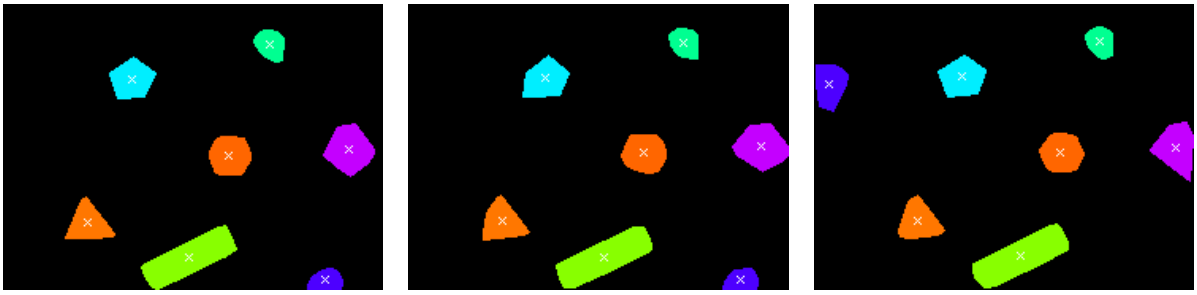
5.2.4. Śledzenie obiektów

Kolejny etap (estymacja głębi) wymaga zapewnienia możliwości odczytywania informacji o tej samej przeszkodzie, pochodzących z obecnej i z poprzednich iteracji algorytmu. Taką opcję zapewnia zastosowanie narzędzia śledzącego ruch wykrytych obiektów – SORT [27]. System ten na wejście przyjmuje detekcje obiektów w postaci opisanych na nich prostokątów. Dla każdego przewiduje ruch, dzięki zastosowaniu modelu predykcji, wykorzystującego filtr Kalmana. Jest powszechnie używany do estymacji przyszłego stanu obiektu (np. prędkości, położenia) na podstawie obserwacji. Następnie dopasowuje wejściowe detekcje do wcześniej śledzonych obiektów. Zwraca ich położenie wraz z przypisanym do niego indywidualnym numerem identyfikacyjnym. Dzięki temu do każdej wykrytej przeszkody przypisywany jest unikalny numer ID, co pozwala na dopasowanie do siebie obiektów między kolejnymi iteracjami algorytmu. Przykład działania algorytmu śledzenia obiektów widoczny jest na rysunku 5.5.

Na tym etapie możliwa jest opcjonalna wizualizacja wykrytych obiektów.

5.2.5. Estymacja głębi

Obliczenie odległości wykrytego obiektu od drona jest ważnym elementem algorytmu, ponieważ pozwala na określenie pozycji przeszkody w przestrzeni 3D.



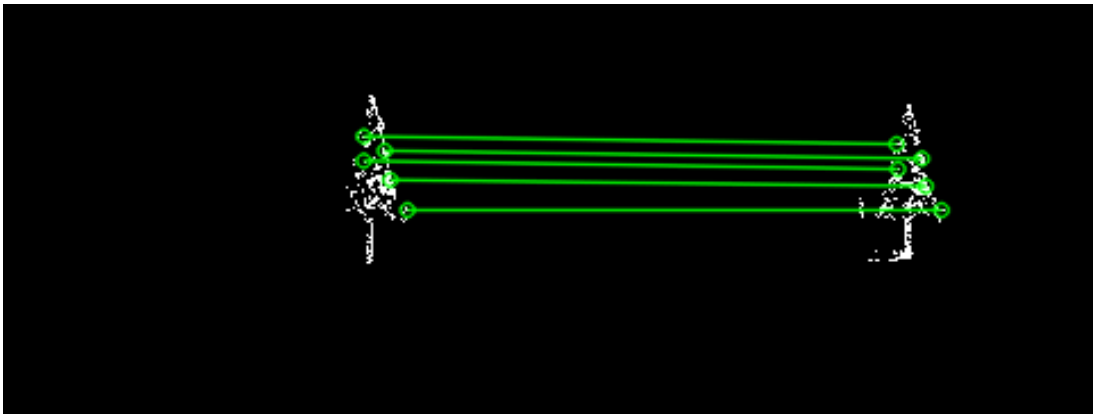
Rys. 5.5. Przeszkody wykryte w danych *shapes rotation* w trzech kolejnych iteracjach algorytmu. Widoczne kształty są efektem działania algorytmu klasteryzacji DBScan. Kolory, którymi zaznaczone są zajmowane przez obiekty obszary, są na stałe przypisane do indywidualnego ID każdej przeszkody. Na zestawianiu obrazków widoczny jest więc efekt działania algorytmu SORT – obiekty są skutecznie śledzone.

W przypadku gdy zastosowany jest układ kamer stereo, zadanie to można wykonywać za pomocą triangulacji – znając wzajemne położenie i orientację kamer, na podstawie przesunięcia obiektu na dwóch odpowiadających sobie klatkach (dysparycji) można ocenić odległość do niego.

W projekcie używana jest jednak jedna kamera zdarzeniowa. Mimo tego wypróbowano podobne podejście w celu otrzymania głębi, wykorzystywane w publikacji [12]. Odczytując dane o położeniu i rotacji drona i zamontowanej na nim kamery, możemy zapamiętać dwie kolejne klatki wraz z ich pozycjami oraz orientacjami i na podstawie ich położenia oraz pozycji obiektu na obrazach przeprowadzić triangulację analogicznie do sytuacji, w której używane byłyby dwie kamery. Pomysł polega na tym, że dla każdej z następujących po sobie ramek, dane zarejestrowane są w nieco innym położeniu unoszącego się w powietrzu drona. Ten nieznaczny ruch kamery powinien pozwolić na estymację dystansu do przeszkód. Takie podejście może być określone jako *Structure from Motion (SfM)*. Jest to technika, służąca do szacowania położenia obiektów w przestrzeni trójwymiarowej na podstawie sekwencji dwuwymiarowych obrazów.

W celu przeprowadzenia triangulacji należy zlokalizować odpowiadające sobie punkty na dwóch kolejnych klatkach. Wyszukiwanie i dopasowywanie ich zostało zrealizowane za pomocą biblioteki OpenCV i dostępnych w niej funkcji i modułów. Wykorzystany został detektor cech ORB (ang. *Oriented FAST and Rotated BRIEF*). Jest to wydajny detektor oraz deskryptor cech, który łączy metody FAST (ang. *Features from Accelerated Segment Test*) do wykrywania punktów charakterystycznych i BRIEF (ang. *Binary Robust Independent Elementary Features*) do ich opisu. Charakteryzuje się niską złożonością obliczeniową. FAST to algorytm wykrywania punktów charakterystycznych, który szybko je identyfikuje na podstawie analizy jasności pikseli w otoczeniu. BRIEF to metoda opisu wykrytych wcześniej punktów poprzez tworzenie binarnego deskryptora na podstawie porównań jasności losowych par pikseli w sąsiedztwie punktu.

Następnie punkty charakterystyczne są dopasowywane między obrazami tej samej przeszkody na dwóch kolejnych klatkach. Ostatnim krokiem jest pozbycie się błędnych dopasowań (ang. *outliers*)



Rys. 5.6. Wynik dopasowywania cech. Widoczne są dwa kolejno zarejestrowane obrazy binarne, zawierające jedną z wykrytych przeszkód. Dla par punktów w następnym etapie przeprowadzana jest triangulacja.

przez użycie algorytmu RANSAC (ang. *Random Sample Consensus*). Jest to iteracyjny algorytm używany do dopasowywania modelu matematycznego do danych, które mogą zawierać dużą liczbę odchyleń (*outliers*). Działanie algorytmu polega na losowym wyborze podzbioru danych, na podstawie którego dopasowywany jest model, oraz ocenie, ile punktów danych (ang. *inliers*) pasuje do tego modelu w ramach zadanej tolerancji. Proces ten jest powtarzany wiele razy, a najlepszy model to ten, który ma największą liczbę *inliers*.

Przykładowy efekt można zobaczyć na rysunku 5.6.

Dla każdej z uzyskanych par punktów przeprowadzana jest triangulacja i otrzymywane są odległości. Jako wartość dystansu do przeszkody przyjmowana jest najmniejsza z nich.

Dysponując wartościami odległości drona do przeszkody w kolejnych iteracjach, łatwo obliczyć składową prędkości obiektu w kierunku drona (prędkość względem drona) (5.1). Odejmując najnowszą i poprzednią wartość dystansu drona do danej przeszkody, otrzymuje się zmianę jej położenia, którą w celu otrzymania prędkości należy podzielić przez czas, w jakim ta zmiana nastąpiła – czyli różnicę czasów zarejestrowania przeszkody.

$$v_d = \frac{d_{prev} - d}{t - t_{prev}} \tag{5.1}$$

gdzie:

- d_{prev} i d, to odpowiednio poprzednia i obecna wartość odległości do przeszkody,
- t_{prev} i t, to odpowiednio poprzednia i obecna wartość czasu zarejestrowania przeszkody.

6. Wyniki i testy

Po implementacji algorytmu w języku Python sprawdzono jego działanie na dwa sposoby:

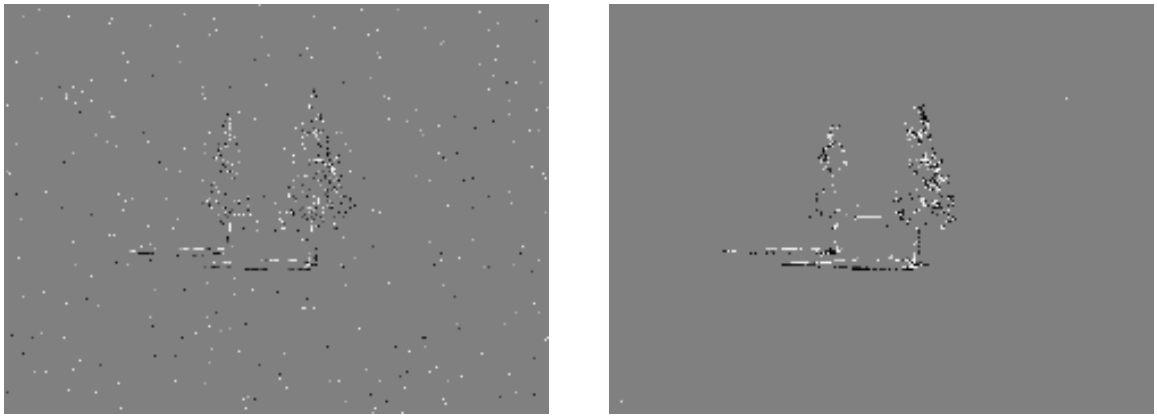
1. Na różnych zbiorach danych z rzeczywistych kamer zdarzeniowych:
   - Dla zbioru *shapes rotation*, pochodzącego z projektu [20]. Jest to zbiór danych zarejestrowany przez przesuwanie i obracanie kamerą zdarzeniową DAVIS240 nad powierzchnią z umieszczonymi na niej różnorakimi kształtami. Nie są to dane zbliżone do tych, na których algorytm ma docelowo działać, lecz obecność wielu osobnych obiektów pozwala na ocenę segmentacji i śledzenia.
   - Dla zbioru *night run*, pochodzącego z projektu [25]. Jest on dobrym sposobem na przetestowanie zdolności do wykrywania obiektów w ciemności. Został stworzony przez zarejestrowanie człowieka biegnącego nocną porą przed obiektywem kamery zdarzeniowej zamocowanej na stojącym samochodzie.

   Testy na uprzednio zarejestrowanych zbiorach danych pozwalają na sprawdzenie działania tylko części algorytmu – ponieważ nie ma w nich dostępu do informacji o parametrach zewnętrznych kamery (tj. pozycji i orientacji), nie można przeprowadzić estymacji głębi.

2. W środowisku symulacyjnym Gazebo, gdzie mogą zostać przetestowane wszystkie etapy algorytmu. Testy zostały przeprowadzane dla różnych scenariuszy testowych, obejmujących:
   - Sceny statyczne – przeszkody, jakie obecne są na ścieżce drona, nie poruszają się względem otoczenia.
   - Sceny dynamiczne – przeszkoda to obiekt będący w ruchu.

   W celu sprawdzenia działania algorytmu również przy ograniczonym oświetleniu, dla części sceny powtórzono test w warunkach symulujących księżycową noc – zachowano słabe źródło światła – księżyc.

Niestety, po uruchomieniu symulacji i odczytaniu wartości wyjściowych zauważono, że obliczane odległości drona do przeszkód nie są prawidłowe i nie zachowują wystarczającej dokładności, żeby można było w jakikolwiek sposób ich dalej użyć. Może być to spowodowane kilkoma czynnikami:



Rys. 6.1. Przykład ramki zdarzeniowej przed (po lewej) i po zastosowaniu filtracji (po prawej).

– Metoda ta wymaga bardzo dokładnych danych o pozycji i rotacji kamery; dodatkowo muszą być one pobrane w chwili wykrycia przeszkody,

– Błędy w dopasowywaniu punktów charakterystycznych – zdarzało się, że pary punktów nie były przypisane właściwie, co dodatkowo wpływało na przekłamania w danych o dystansie,

– Błędy w procesie śledzenia obiektów – aby metoda działała prawidłowo, wymagane są dane o przeszkodzie z bieżącej i poprzedniej iteracji,

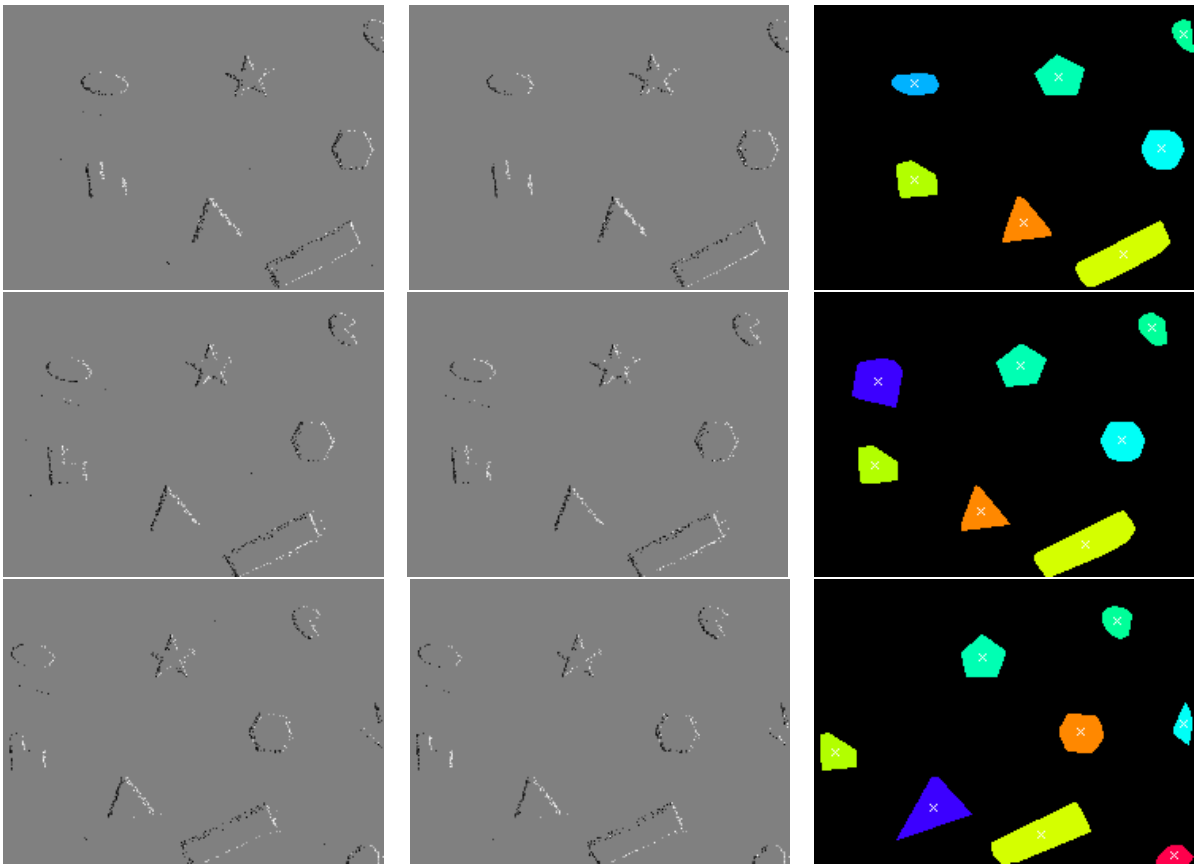
– Zbyt małe przemieszczenie drona między dwiema kolejnymi klatkami mogło wpłynąć na znaczne zmniejszenie dokładności – nie sposób wyznaczyć głębię za pomocą triangulacji, jeśli dron nie przemieścił się względem przeszkody w żadnym kierunku.

Mimo wielu prób, nie udało się uzyskać poprawnych wyników, zachowując przyjętą metodę. Brak prawidłowych danych o dystansie do wykrytych obiektów jest poważnym problemem, który w ostatecznej wersji algorytmu musi być rozwiązany w inny sposób. Błąd uniemożliwia otrzymanie pozycji przeszkody w trójwymiarowej przestrzeni, a także powoduje niepoprawne wartości obliczanych prędkości przeszkód.

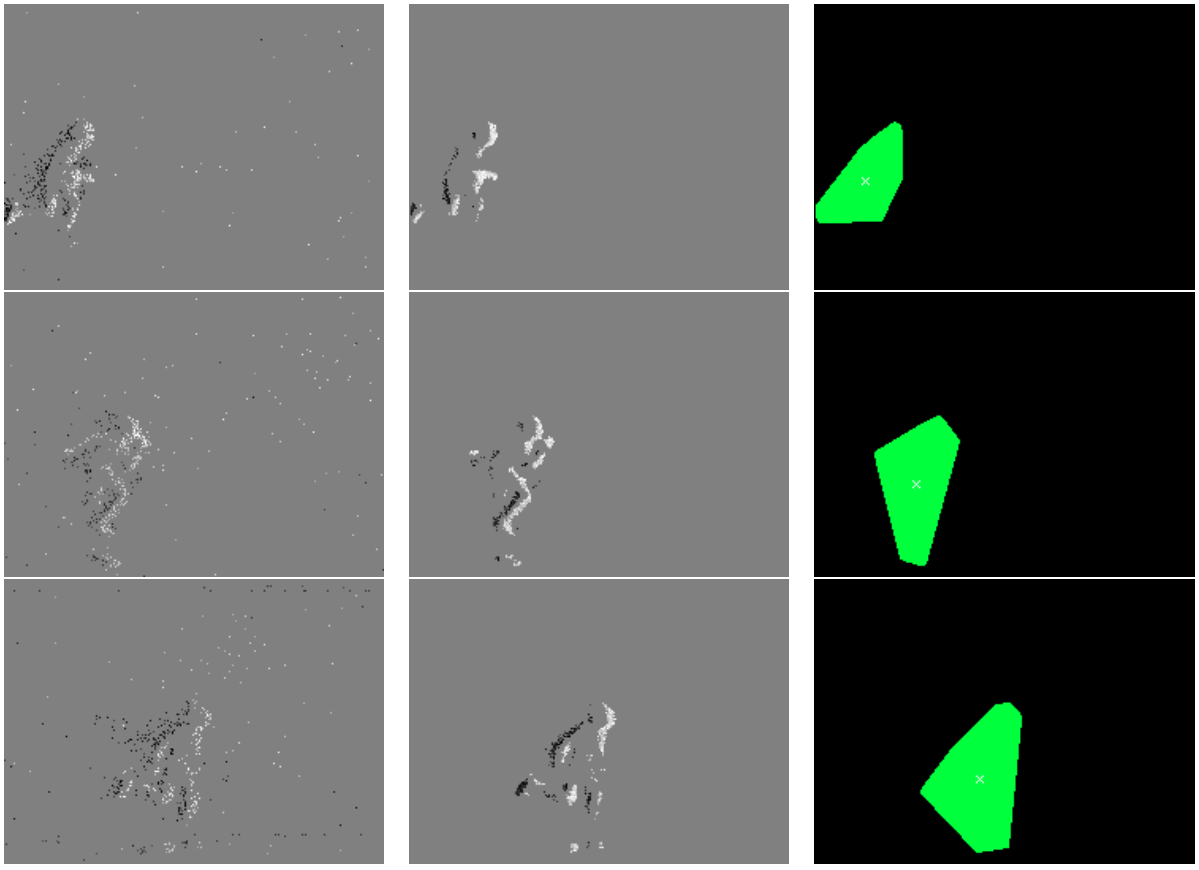
Algorytm realizuje zadanie detekcji obecności obiektów w polu widzenia kamery zdarzeniowej. Dostarcza danych o ich kształcie oraz rozmiarach i położeniu na dwuwymiarowej płaszczyźnie obrazu. Możliwe jest przetestowanie wcześniejszych etapów i ocena jakości samego wykrywania obiektów, pomimo błędnego działania ostatniej fazy algorytmu – estymacji głębi.

W pierwszej kolejności osobno sprawdzono działanie każdej z faz algorytmu.

Dzięki wizualizacji wyników filtrowania i porównania go z oryginalnymi zdarzeniami, można ocenić jego jakość oraz dobrać parametry. Wynik działania filtru na symulacji z dodanym szumem można zobaczyć na rysunku 6.1.



Rys. 6.2. Zestawienie wyników testowania algorytmu na zbiorze danych *shapes rotation* w kilku wybranych momentach. Od lewej strony kolejne obrazki przedstawiają: dane przed filtracją, dane po filtracji, wykryte obiekty.



Rys. 6.3. Zestawienie wyników testowania algorytmu na zbiorze danych *night run* w kilku wybranych momentach. Od lewej strony kolejne obrazki przedstawiają: dane przed filtracją, dane po filtracji, wykryte obiekty.

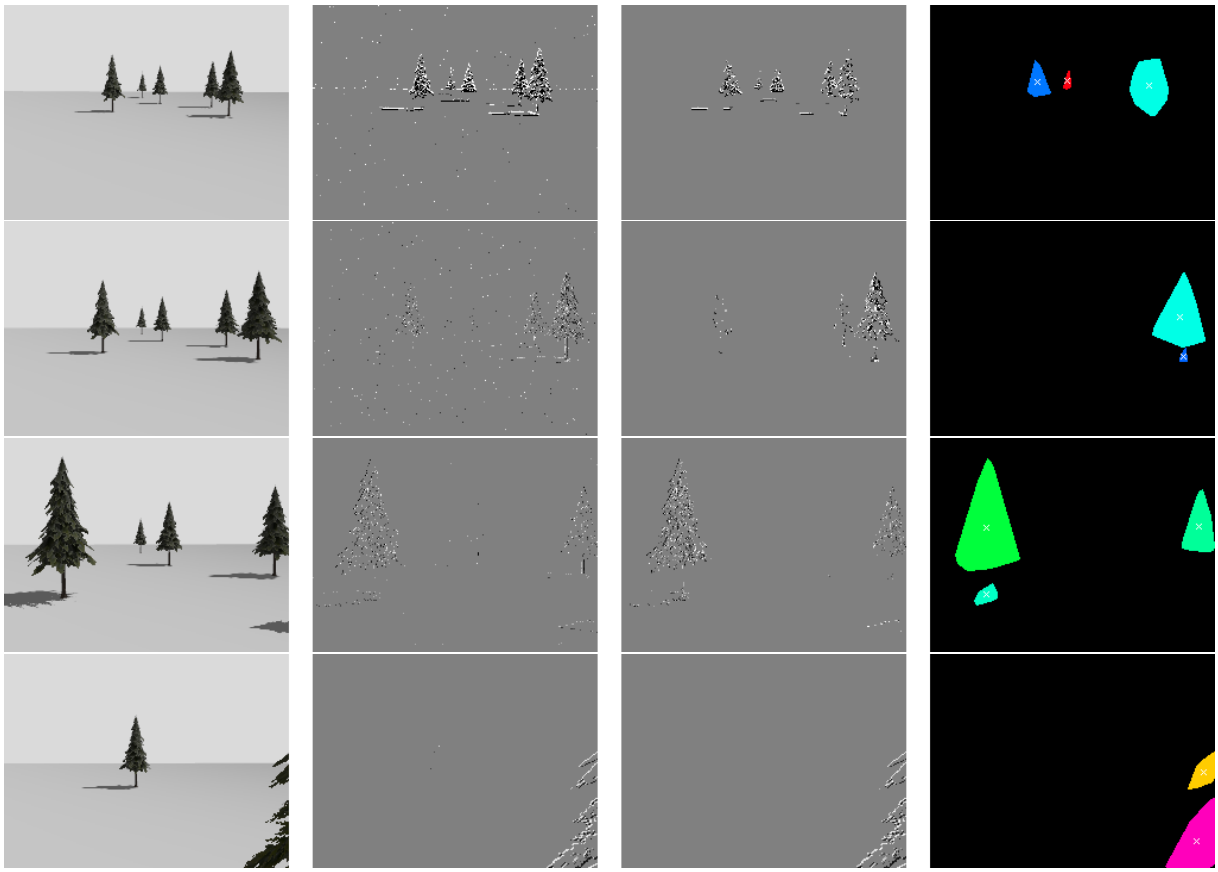
Jak można zauważyć na rysunku 6.2, zbiór danych *shapes rotation* charakteryzuje się niewielkim szumem. W śledzeniu obiektów czasem pojawiają się błędy – szczególnie w przypadku zmiany kierunku ruchu obiektu. Jest to widoczne przez zmianę jego koloru. Obiekt również przestaje być śledzony, gdy znajdzie się poza polem widzenia, a następnie w nie powróci (SORT nie został wyposażony w funkcjonalność obsługującą takie przypadki). Nie jest to jednak duży problem – algorytm wymaga jedynie możliwości porównania pozycji przeszkody z aktualnej i poprzedniej iteracji.

Na podstawie rysunku 6.3 można stwierdzić, że w danych *night run* występuje stosunkowo dużo szumu – aby go wyeliminować, ustawiono bardziej rygorystyczne wartości parametrów filtra:

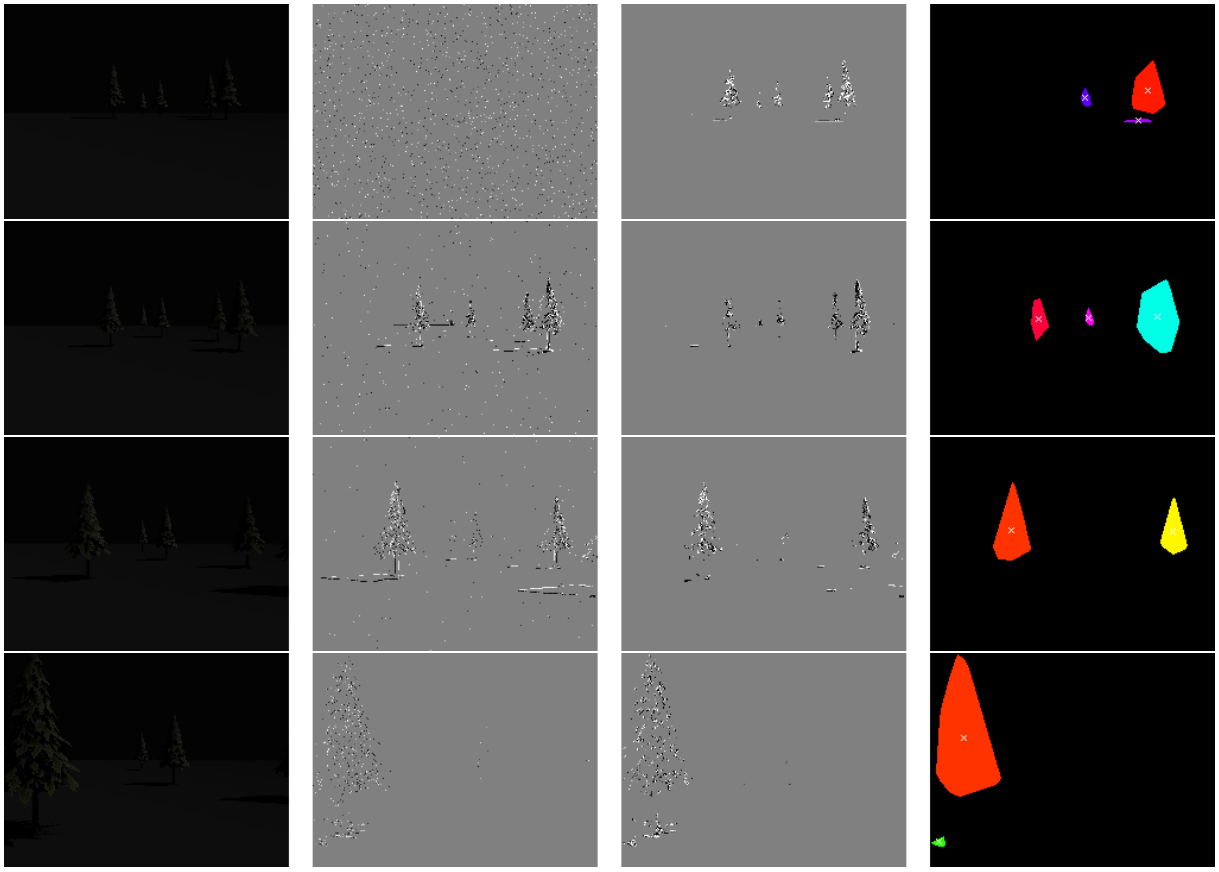
$$n = 3;\ \ t = \frac{1}{400}s;\ \ k = 6$$

W prostszym przypadku (tylko jeden poruszający się w jednym kierunku obiekt) nie występują żadne błędy w śledzeniu – przez cały test obiekt zachował ten sam kolor.

Na podstawie przebiegu testu sprawdzającego działanie systemu dla statycznej sceny w warunkach dziennych (rys. 6.4) można zauważyć kilka istotnych cech algorytmu:



Rys. 6.4. Pierwszy scenariusz testowy – statyczna scena w dziennym świetle. Rolę przeszkód pełnią drzewa umieszczone przed lecącym naprzód dronem. Kolejne obrazki od lewej zawierają: obraz z tradycyjnej kamery, nieprzefiltrowane dane z DVS, dane z DVS po filtracji, wykryte przeszkody.

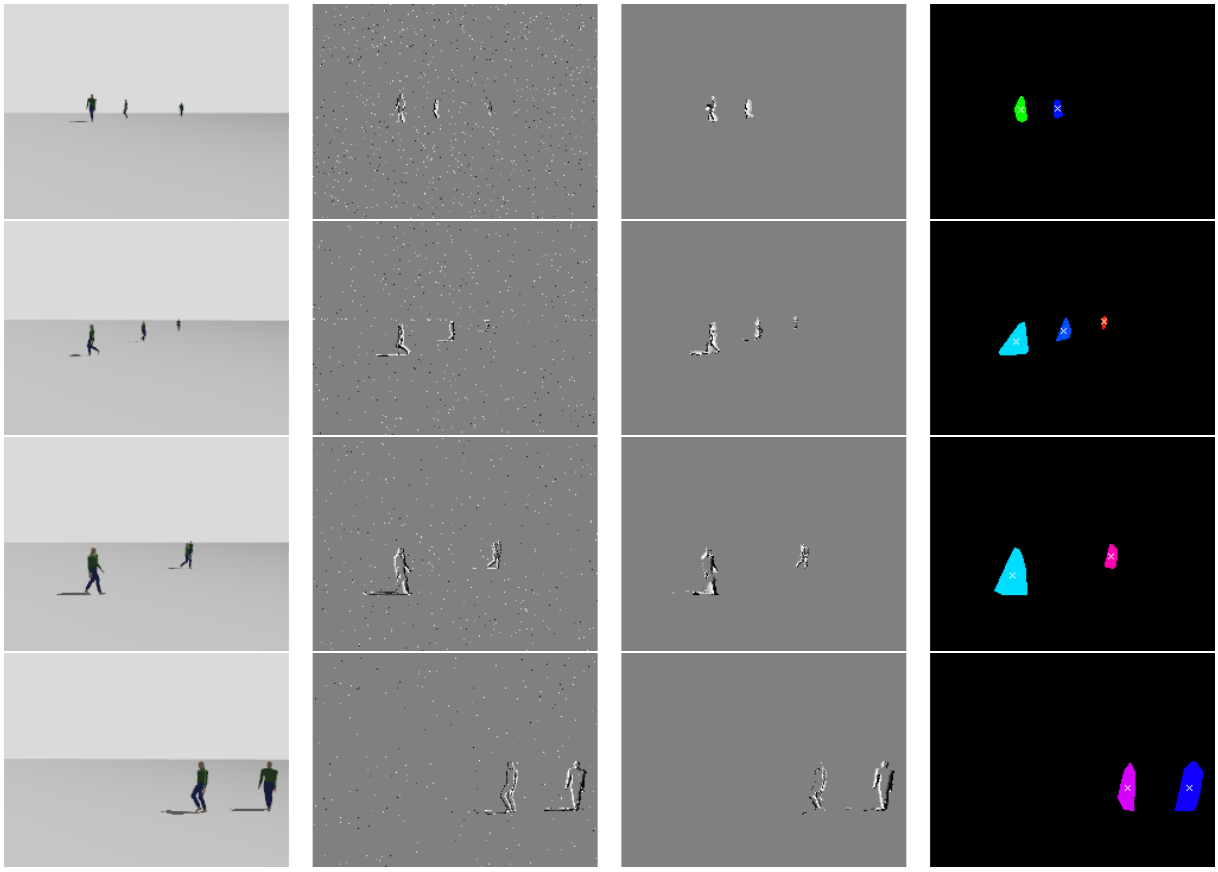


Rys. 6.5. Drugi scenariusz testowy – statyczna scena w nocy. Rolę przeszkód pełnią drzewa umieszczone przed lecącym naprzód dronem. Kolejne obrazki od lewej zawierają: obraz z tradycyjnej kamery, nieprzefiltrowane dane z DVS, dane z DVS po filtracji, wykryte przeszkody.

– Algorytm często pomija przeszkody znajdujące się dalej i mające mniejszą prędkość względem drona – jest to spowodowane głównie tym, że dane przetwarzane są w pakietach po N zdarzeń. Jeśli w polu widzenia znajduje się przeszkoda, która generuje większość z nich, to zdarzenia związane z pozostałymi przeszkodami stanowią bardzo niewielką część całości, co powoduje, że są uznawane za szum – jeśli nie na etapie filtracji, to podczas klasteryzacji. Nie jest to w jednoznaczny sposób wada – pomijane w ten sposób przeszkody znajdują się daleko od drona i/lub poruszają się względem niego wolno. Algorytm nadaje priorytet tym, które stanowią większe zagrożenie.

– W przypadku gdy przeszkodami są drzewa, ich pień bywa uznawany za osobny obiekt. Podział jednej przeszkody na kilka osobnych obiektów prowadzi do niepotrzebnego wzrostu kosztu obliczeniowego.

– Obiekty, które w rzeczywistości znajdują się daleko od siebie, ale w klatce obrazu są blisko, uznawane są za jedną przeszkodę – może to być jeden z czynników utrudniających triangulację.



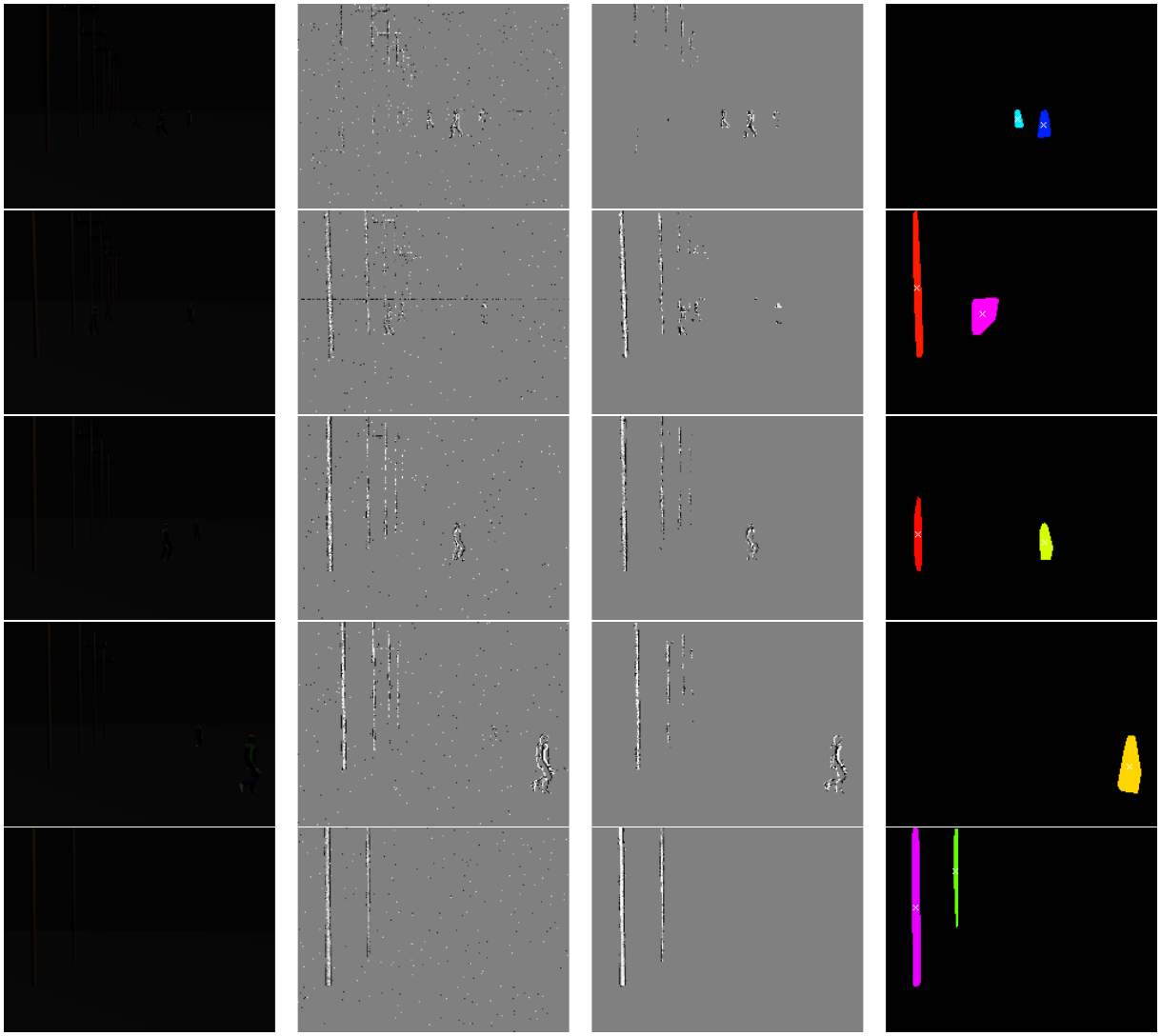
Rys. 6.6. Trzeci scenariusz testowy – dynamiczna scena w dzień. Rolę przeszkód pełnią poruszające się szybkim krokiem ($1 - 1,5\ m/s$) osoby. Kolejne obrazki od lewej zawierają: obraz z tradycyjnej kamery, nieprzefiltrowane dane z DVS, dane z DVS po filtracji, wykryte przeszkody.

Drugi scenariusz testowy (rys. 6.5) pokazuje, że nawet w ciemności algorytm zachowuje swoją funkcjonalność. Względem działania za dnia występują jednak pewne problemy:

– Zdarza się, że jako przeszkoda rozpoznawana jest jedynie część rzeczywistego obiektu,

– Zaburzenia pracy algorytmu, takie jak utrata śledzonego obiektu i przypisanie do niego innego ID czy wykrycie cienia obiektu jako przeszkody, pojawiają się częściej.

Na podstawie trzeciego testu (rys. 6.6) można stwierdzić, że poruszające się obiekty są wykrywane nawet skuteczniej niż te statyczne – widoczne są z dalszej odległości. Przyczyną tego jest większa liczba generowanych zdarzeń. Problemem, jaki można zaobserwować w działaniu algorytmu, jest utrata ciągłości w śledzeniu przeszkód, gdy ludzie zmieniają kierunek ruchu lub wychodzą poza pole widzenia kamery.



Rys. 6.7. Czwarty scenariusz testowy – dynamiczna scena w nocy. Żeby utrudnić zadanie oprócz spacerujących ludzi, do sceny dodano rząd słupów energetycznych. Względem drugiego scenariusza (rys. 6.5), światła jest jeszcze mniej. Kolejne obrazki od lewej zawierają: obraz z tradycyjnej kamery, nieprzefiltrowane dane z DVS, dane z DVS po filtracji, wykryte przeszkody.

Z ostatnim, najtrudniejszym zadaniem (rys. 6.7), algorytm nie poradził sobie w pełni zadowalająco. W części klatek przeszkody nie były wykrywane lub były wykrywane tylko częściowo. Śledzenie obiektów stosunkowo często zawodziło – co w znacznym stopniu utrudniałoby estymację głębi.

Na podstawie przeprowadzonych testów algorytmu można sformułować jego cechy – zalety i wady:

- Gdy w polu widzenia kamery znajduje się kilka obiektów, pośród których występują takie, które przez swoją większą prędkość lub bliższy dystans odpowiadają za generowanie wyższej liczby zdarzeń, pozostałe przeszkody nie są wykrywane. Zazwyczaj pomijane w wyniku tego obiekty stanowią mniejsze zagrożenie dla drona, ponieważ poruszają się one wolno i/lub znajdują się daleko, więc takie zachowanie można potraktować jako zaletę – ograniczana jest liczba wykrywanych obiektów, a razem z nią czas obliczeń.

- Algorytm dobrze radzi sobie w ograniczonych warunkach oświetleniowych. Jak można zaobserwować na rysunkach 6.5 oraz 6.7, w pierwszej kolumnie obrazków widok z tradycyjnej kamery jest całkowicie lub niemal całkowicie nieczytelny. Mimo to detekcja przeszkód ciągle jest możliwa, choć jej jakość nieco się zmniejsza.

- Typowe dla działania algorytmu błędy, których częstotliwość występowania nasila się w bardzo trudnych warunkach oświetleniowych, to:

  - Niewykrywanie obiektu na pojedynczych klatkach obrazu, co zaburza ciągłość jego działania i powoduje dalsze błędy w śledzeniu przeszkód,
  - Niewykrywanie całej przeszkody, a tylko jej części – problem dotyczy obiektów statycznych przy ograniczonym oświetleniu,
  - Wykrywanie jednego obiektu jako kilka mniejszych przeszkód, co utrudnia śledzenie oraz może niepotrzebnie podwyższać czas potrzebny na przetwarzanie klatki.

7. Podsumowanie

W ramach pracy nad projektem wykonane zostały:

– Przegląd literatury – zapoznano się z metodami detekcji i unikania przeszkód przy wykorzystaniu kamer zdarzeniowych. Przeanalizowano algorytmy wykrywania obiektów w wybranych artykułach i na tej podstawie stworzono własne podejście do problemu. Przeanalizowano publikacje pomocne w jego realizacji.

– Środowisko symulacyjne do testów SiL – stworzono różne scenariusze testowe (sceny), umożliwiające sprawdzenie działania algorytmu w dziennych lub nocnych warunkach na zróżnicowanych torach przeszkód. W środowisku umieszczono model drona, wyposażonego w kamerę zdarzeniową i rozwiązano problem sterowania nim.

– Algorytm detekcji przeszkód – zaprojektowany dzięki analizie literatury sposób działania algorytmu zaimplementowano w języku programowania Python. Przeprowadzono wstępne próby jego działania na zbiorach danych pochodzących z kamery zdarzeniowej DAVIS240. Używając *frameworka* ROS2, przetestowano algorytm w symulacji.

Niestety, nie wszystkie cele wymienione w rozdziale 1.1 zostały osiągnięte. Z powodu ograniczonego czasu oraz rozbudowanej formy projektu, nie udało się podjąć próby implementacji i przetestowania rozwiązania w formule *Hardware in the Loop* na platformie wbudowanej.

Na podstawie testów systemu, przedstawionych i opisanych w rozdziale **??**, można zdefiniować jego cechy:

– Algorytm dobrze radzi sobie w ograniczonych warunkach oświetleniowych,

– Błędy w działaniu algorytmu występują częściej w ograniczonych warunkach oświetleniowych. Zazwyczaj są to:

  – Niewykrywanie obiektu na pojedynczych klatkach obrazu,

  – Wykrywanie tylko części przeszkody,

  – Wykrywanie jednego obiektu jako kilka mniejszych przeszkód.

Dzięki testom *Software in the Loop*, możliwe było wykrycie wielu błędów i ich naprawa na wczesnym etapie pracy nad systemem detekcji obiektów – podczas implementacji modelu programowego.

W czasie testów SiL stwierdzony został poważny problem z wartościami odległości do wykrytych obiektów zwracanymi przez algorytm. Niestety okazały się one na tyle niedokładne i zawierały tyle błędów, że ich użycie do wyznaczania pozycji obiektów w przestrzeni 3D oraz ich prędkości okazało się niemożliwe. Mimo podjętych prób naprawy, zachowując przyjęte podejście (triangulacja z użyciem jednej kamery zdarzeniowej), nie udało się tego problemu rozwiązać.

Możliwe przyczyny błędnego działania tej fazy algorytmu:

– Niedokładne dane o pozycji i rotacji kamery,

– Błędy w dopasowywaniu punktów charakterystycznych,

– Błędy w procesie śledzenia obiektów,

– Zbyt małe przemieszczenie drona między dwiema kolejnymi klatkami.

Ponieważ zastosowanie triangulacji w przypadku pojedynczej kamery okazało się błędnym podejściem, w celu realizacji zadania rozpoznawania głębi, konieczna może okazać się zmiana metody. Proponowane rozwiązania:

– Ograniczenie wykrywania do obiektów o znanym rozmiarze – znacznie zmniejszy to potencjalne zastosowania systemu, ale pozwoli na otrzymywanie danych o głębi bez wyposażania drona w dodatkowe czujniki.

– Umieszczenie na dronie LiDAR-u, jako dodatkowego czujnika. W tym rozwiązaniu kamera zdarzeniowa pozwoli na skuteczniejsze wykrywanie obiektów, które szybko się poruszają oraz poprawi działanie w ciemności, a LiDAR umożliwi zwiększenie dokładności oraz pozwoli na precyzyjne odczytywanie głębi. Minusem takiego rozwiązania jest wzrost ceny oraz stopnia skomplikowania systemu. Znacznie zwiększyłaby się złożoność obliczeniowa z powodu dodatkowych danych do przetworzenia.

– Zastosowanie dwóch kamer zdarzeniowych w układzie stereo, dzięki czemu możliwe będzie przeprowadzenie triangulacji w poprawny i dokładny sposób.

– Wybór i zastosowanie w projekcie algorytmu estymacji głębi dedykowanego dla pojedynczych kamer zdarzeniowych. Takie rozwiązania można znaleźć w literaturze na przykład w artykułach [28] lub [29]. W porównaniu do prostej koncepcyjnie triangulacji, są to rozbudowane i bardziej złożone obliczeniowo systemy.

Projekt ma szerokie możliwości dalszego rozwoju i rozbudowy.

W ramach dalszej pracy nad projektem planowane jest:

– Rozwiązanie problemu otrzymywania poprawnych danych o głębi,

– Optymalizacja akumlacji zdarzeń do dalszego przetwarzania. Dwa podejścia do tego zagadnienia - zbieranie danych w pewnym przedziale czasowym oraz akumulacja pewnej ich liczby, można połączyć w jedno. Ten sposób pozwala na połączenie ich zalet i minimalizacje ich wad (odpowiednio możliwości wystąpienia rozmycia ruchu oraz za dużej do realizacji w czasie rzeczywistym częstotliwości ramek zdarzeniowych).

– Zaimplementowanie algorytmu na wbudowanej platformie obliczeniowej – eGPU Jetson i przetestowanie jego działania metodą HiL,

– Zaimplementowanie algorytmu na platformie z układem FPGA (ang. *Field Programmable Gate Array*). Przeprowadzenie testów i porównanie działania systemu uruchamianego na FPGA i eGPU Jetson. FPGA jest platformą, która może okazać się najskuteczniejsza dla stworzonego algorytmu ze względu na szerokie możliwości zrównoleglania wykonywanych obliczeń, co w przypadku systemów wizyjnych jest wyjątkowo efektywne.

– Zaprojektowanie i implementacja systemu sterowania dronem, tak by na podstawie danych otrzymywanych z procesu detekcji, umożliwić unikanie przeszkód,

– Zamontowanie systemu na rzeczywistym dronie i przetestowanie go na przygotowanym torze przeszkód.

Mimo że algorytm detekcji testowany był z wykorzystaniem czterowirnikowego drona, to może być zastosowany do wykrywania poruszających się względem kamery obiektów na innych rodzajach pojazdów i wszelkich robotach mobilnych.

Bibliografia

[1] Guillermo Gallego i in. „Event-Based Vision: A Survey". W: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.1 (sty. 2022), 154–180. ISSN: 1939-3539. DOI: *10.1109/tpami.2020.3008413*.

[2] Krzysztof Blachut i Tomasz Kryjak. „High-definition event frame generation using SoC FPGA devices". W: *2023 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*. IEEE, wrz. 2023, 106–111. DOI: *10.23919/spa59660.2023.10274447*.

[3] Kris Kitani. *Triangulation*. URL: *http://www.cs.cmu.edu/~16385/s17/Slides/11.4_Triangulation.pdf* (term. wiz. 2025-01-09).

[4] Yadwinder Singh i Lakhwinder Kaur. „Obstacle Detection Techniques in Outdoor Environment: Process, Study and Analysis". W: *International Journal of Image, Graphics and Signal Processing* 9 (maj 2017), s. 35–53. DOI: *10.5815/ijigsp.2017.05.05*.

[5] Nadra Ben Romdhane, Mohamed Hammami i Hanêne Ben-Abdallah. „A generic obstacle detection method for collision avoidance". W: *2011 IEEE Intelligent Vehicles Symposium (IV)*. 2011, s. 491–496. DOI: *10.1109/IVS.2011.5940503*.

[6] Rebecca Skantar. *Obstacle Detection Methods*. URL: *https://sites.tufts.edu/eeseniordesignhandbook/files/2022/05/Skantar_TechNotes.pdf* (term. wiz. 2025-01-11).

[7] Davide Falanga, Kevin Kleber i Davide Scaramuzza. „Dynamic obstacle avoidance for quadrotors with event cameras". W: *Science Robotics* 5.40 (2020), eaaz9712. DOI: *10.1126/scirobotics.aaz9712*. eprint: *https://www.science.org/doi/pdf/10.1126/scirobotics.aaz9712*.

[8] Simon Baker i Iain Matthews. „Lucas-Kanade 20 Years On: A Unifying Framework". W: *Int. J. Comput. Vision* 56.3 (lut. 2004), 221–255. ISSN: 0920-5691. DOI: *10.1023/B:VISI.0000011205.11775.fd*.

[9] Martin Ester i in. „A density-based algorithm for discovering clusters in large spatial databases with noise". W: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, 226–231.

[10] Andrea Fusiello. „Elements of Computer Vision : Multiple View Geometry". W: 2005.

[11] Nitin J. Sanket i in. „EVDodge: Embodied AI For High-Speed Dodging On A Quadrotor Using Event Cameras". W: *CoRR* abs/1906.02919 (2019). arXiv: *1906.02919*.

[12] Jawad Naveed Yasin i in. „Night vision obstacle detection and avoidance based on Bio-Inspired Vision Sensors". W: *CoRR* abs/2010.15509 (2020). arXiv: *2010.15509*.

[13] Sherif AS Mohamed i in. „DBA-Filter: A dynamic background activity noise filtering algorithm for event cameras". W: *Intelligent Computing: Proceedings of the 2021 Computing Conference, Volume 1*. Springer. 2022, s. 685–696.

[14] Christopher G. Harris i M. J. Stephens. „A Combined Corner and Edge Detector". W: *Alvey Vision Conference*. 1988.

[15] Etienne Perot i in. „Learning to Detect Objects with a 1 Megapixel Event Camera". W: *CoRR* abs/2009.13436 (2020). arXiv: *2009.13436*.

[16] Jianing Li i in. „Asynchronous Spatio-Temporal Memory Network for Continuous Event-Based Object Detection". W: *IEEE Transactions on Image Processing* 31 (2022), s. 2975–2987. DOI: *10.1109/TIP.2022.3162962*.

[17] Mathias Gehrig i Davide Scaramuzza. *Recurrent Vision Transformers for Object Detection with Event Cameras*. 2023. arXiv: *2212.05598* [cs.CV].

[18] Dongsheng Wang i in. *Dual Memory Aggregation Network for Event-Based Object Detection with Learnable Representation*. 2023. arXiv: *2303.09919* [cs.CV].

[19] Ting-Kang Yen i in. *Tracking-Assisted Object Detection with Event Cameras*. 2024. arXiv: *2403.18330* [cs.CV].

[20] Elias Mueggler i in. „The Event-Camera Dataset and Simulator: Event-based Data for Pose Estimation, Visual Odometry, and SLAM". W: *CoRR* abs/1610.08336 (2016). arXiv: *1610.08336*.

[21] Henri Rebecq, Daniel Gehrig i Davide Scaramuzza. „ESIM: an Open Event Camera Simulator". W: *Conf. on Robotics Learning (CoRL)* (paź. 2018).

[22] Yuhuang Hu, Shih-Chii Liu i Tobi Delbruck. *v2e: From Video Frames to Realistic DVS Events*. 2021. arXiv: *2006.07722* [cs.CV].

[23] Huaizu Jiang i in. *Super SloMo: High Quality Estimation of Multiple Intermediate Frames for Video Interpolation*. 2018. arXiv: *1712.00080* [cs.CV].

[24] Shasha Guo i Tobi Delbruck. „Low Cost and Latency Event Camera Background Activity Denoising". W: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.1 (2023), s. 785–795. DOI: *10.1109/TPAMI.2022.3152999*.

[25] Cedric Scheerlinck, Nick Barnes i Robert Mahony. *Continuous-time Intensity Estimation Using Event Cameras*. 2018. arXiv: *1811.00386* [cs.CV].

[26] L. Xu i E. Oja. „Randomized Hough Transform (RHT): Basic Mechanisms, Algorithms, and Computational Complexities". W: *CVGIP: Image Understanding* 57.2 (1993), s. 131–154. ISSN: 1049-9660. DOI: *https://doi.org/10.1006/ciun.1993.1009*.

[27] Alex Bewley i in. „Simple online and realtime tracking”. W: *2016 IEEE International Conference on Image Processing (ICIP)*. 2016, s. 3464–3468. DOI: *10.1109/ICIP.2016.7533003*.

[28] Henri Rebecq i in. „EMVS: Event-based Multi-View Stereo—3D Reconstruction with an Event Camera in Real-Time”. W: *Int. J. Comput. Vis.* 126 (12 grud. 2018), s. 1394–1414. DOI: *10.1007/s11263-017-1050-6*.

[29] Hanme Kim, Stefan Leutenegger i Andrew J. Davison. „Real-Time 3D Reconstruction and 6-DoF Tracking with an Event Camera”. W: *European Conference on Computer Vision*. 2016.

Kod projektu oraz inne pliki są dostępne w repozytorium:
https://github.com/romannowak9/DVS_Detekcja_przeszkod_Roman_Nowak