

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

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

KATEDRA AUTOMATYKI I INŻYNIERII BIOMEDYCZNEJ

Praca dyplomowa magisterska

*Biblioteka modułów sprzętowych do segmentacji obiektów
pierwszoplanowych*

Hardware modules library for foreground object segmentation

Autor:

Piotr Janus

Kierunek studiów:

Automatyka i Robotyka

Opiekun pracy:

dr inż. Tomasz Kryjak

Kraków, 2017

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

Spis treści

1. Wprowadzenie	9
1.1. Cel pracy	11
1.2. Zawartość pracy.....	12
2. Przegląd metod detekcji obiektów pierwszoplanowych	13
2.1. Wprowadzenie	13
2.2. Model tła bazujący na poprzednich ramkach	13
2.3. Statystyczne modele tła	14
2.4. Różne podejścia do aktualizacji modelu tła	15
2.5. Wykorzystanie map głębi obrazu	15
2.6. Mechanizm detekcji cieni.....	16
2.7. Algorytm detekcji wykorzystujący histogramy blokowe	16
2.8. Podsumowanie.....	17
3. Opis teoretyczny wybranych algorytmów	19
3.1. Proste metody segmentacji obiektów pierwszoplanowych	19
3.2. Visual Background Extractor.....	20
3.2.1. Opis algorytmu.....	20
3.2.2. Konwersja RGB – CIELab.....	22
3.2.3. Obsługa ruchomej kamery	23
3.2.4. Wyznaczanie rzadkiego przepływu optycznego	23
3.2.5. Detekcja narożników - metoda Harrisa-Stephensa	24
3.2.6. Uwagi	26
3.3. Pixel-Based Adaptive Segmenter	26
3.3.1. Opis algorytmu.....	26
3.3.2. Detekcja obiektów statycznych.....	28
3.3.3. Indeksacja obiektów	31
3.3.4. Uwagi	32
3.4. Gaussian Mixture Models.....	33

3.4.1. Opis Algorytmu.....	34
3.4.2. Uwagi	36
3.5. Algorytm Flux Tensor with Split Gaussian Models	36
3.5.1. Algorytm Flux Tensor	36
3.5.2. Algorytm Split Gaussian Models	38
3.6. Fuzja metody Flux Tensor i Split Gaussian Models.....	39
4. Implementacja sprzętowa wybranych algorytmów segmentacji obiektów pierwszoplanowych.....	41
4.1. Układy <i>FPGA</i> – wprowadzenie	41
4.2. Modele programowe.....	42
4.3. Wykorzystany układ oraz interfejs wizyjny	44
4.4. Trudności występujące w potokowym systemie wizyjnym	46
4.4.1. Kontekst poziomy i pionowy	46
4.4.2. Generowanie liczb losowych	47
4.4.3. Kontroler pamięci RAM	48
4.5. Implementacja algorytmu ViBE	50
4.6. Implementacja rozszerzonej wersji ViBE.....	53
4.7. Implementacja algorytmu PBAS	55
4.8. Implementacja rozszerzonej wersji metody PBAS.....	59
4.9. Implementacja GMM	62
4.10. Podsumowanie	65
5. Ewaluacja zaimplementowanych algorytmów	67
5.1. Metodologia przeprowadzonych testów	67
5.2. Szczegółowe testy	68
5.2.1. Sekwencje podstawowe.....	68
5.2.2. Dynamiczne tło i cienie.....	69
5.2.3. Drgania kamery	71
5.2.4. Obiekty zatrzymane	73
5.2.5. Kamera termowizyjna	75
5.2.6. Podsumowanie	76
6. Dalsze kierunki rozwoju	79
6.1. Poprawa algorytmów	79
6.2. Wzrost wydajności	79
6.3. Implementacja nowych rozwiązań	80

7. Zakończenie	81
A. Spis zawartości płyty DVD	87
B. Opis informatyczny	89

Abstrakt

Główym celem niniejszej pracy było zaimplementowanie biblioteki modułów sprzętowych, zawierającej różne algorytmy segmentacji obiektów pierwszoplanowych. Przedstawione metody opracowano na podstawie prac zrealizowanych dotychczas w Laboratorium Biocybernetyki AGH. Większość algorytmów służących do segmentacji obiektów ma jedną poważną wadę, którą jest duża złożoność obliczeniowa. Powoduje to, że nie mogą one zostać użyte w systemach wykorzystujących procesory ogólnego przeznaczenia. Zaprojektowane implementacje uruchomiono na karcie VC 707 z układem *FPGA* z rodziną *Virtex 7* firmy *Xilinx*. Zaproponowane metody porównano pod względem zarówno dokładności jak i zużycia energii. Przedstawione systemy przetwarzają obraz w rozdzielczości od $576x720@fps$ do $1920x1080@50fps$. Dodatkowo w pracy omówiono także bardziej zaawansowane zagadnienia związane z detekcją obiektów, takie jak porzucone obiekty czy dostosowanie algorytmu do ruchomej kamery *PTZ*.

Abstract

The main purpose of the following thesis was to implement the library of hardware implementations containing various foreground segmentation algorithms. Implemented methods are based on articles and thesis which have been developed in Biocybernetics laboratory at AGH University of Science and Technology in Kraków. Most of foreground segmentation algorithms has one major drwawback, which is high computational complexity. This fact implicates that they can not be applied in systems based on general purpose processors. Designed modules have been run on VC 707 development board with *Virtex 7 FPGA* device from *Xilinx*. Proposed algorithms have been compared in terms of both precission and power consumption. Presented solutions are able to handle video stream in resolution from $576x720@50fps$ up to $1920x1080@50fps$. Additionally, more adanvced topis such as detection of abandoned objects and adaptation to work with moving *PTZ* camera, have been also described.

1. Wprowadzenie

Zagadnienie segmentacji obiektów pierwszoplanowych definiujemy jako operację wyodrębniania tego typu obiektów na obrazie. Obiekty pierwszoplanowe, ze względu na ich charakterystykę, możemy podzielić na dynamiczne, czyli na przykład idący człowiek lub jadący samochód i statyczne. Przykładem obiektu statycznego może być natomiast zatrzymany na światłach samochód lub porzucony bagaż. Oczekiwany efektem końcowym, metody realizującej wspomnianą segmentację, jest maska binarna, na której piksele reprezentujące znalezione obiekty oznaczone są na biało, natomiast pozostałe na czarno.

Należy zwrócić szczególną uwagę na potencjalne problemy i trudności występujące przy projektowaniu tego typu algorytmów. Bardzo ważna jest poprawna detekcja, wspomnianych wcześniej, obiektów statycznych jak i dynamicznych. Z tym zagadnieniem powiązane jest także zjawisko tzw. „duchów” (ang. *ghost*). Można je zaobserwować w sytuacji, gdy obiekt zatrzymany przez dłuższy czas i błędnie zaklasyfikowany przez algorytm jako element tła, nagle zaczyna się poruszać. Rozróżnienie rzeczywistych obiektów zatrzymanych od „duchów” jest problematyczne, gdyż oba obiekty mają podobne właściwości. Zagadnienie to zostało szerzej opisane w rozdziale 2.4.

Kolejnym poważnym utrudnieniem jest występowanie dynamicznego tła (na przykład poruszające się w tle liście i gałęzie). Zagadnienie to jest zdecydowanie najbardziej złożone i do tej pory nie znaleziono rozwiązania, które eliminowałoby je w stu procentach. Oprócz tego, należy także mieć na uwadze zmienne oświetlenie, możliwość wystąpienie lekkich drgań kamery, różnego rodzaju szumy i zakłócenia. Do grupy istotnych i często występujących problemów, możemy zaliczyć również konieczność prawidłowej detekcji cieni i odróżniania ich od właściwych obiektów pierwszoplanowych.

Segmentacja obiektów pierwszoplanowych jest bardzo rozległą i cały czas dynamicznie rozwijającą się dziedziną. Pracownicy naukowi ze wszystkich uczelni technicznych publikują coraz to nowe rozwiązania, bazujące na istniejących już algorytmach lub proponują podejścia całkowicie nowe, odbiegające od wcześniej przyjętej metodyki. Celem tych badań, jest oczywiście rozwój opisywanego zagadnienia i dążenie do ideału, czyli algorytmu, dającego bezbłędne rezultaty nawet w rzeczywistych, najbardziej ekstremalnych warunkach. Na przestrzeni ostatnich kilku lat można zaobserwować spadek cen różnego rodzaju czujników, kamer, oraz układów scalonych, co w rezultacie przekłada się na większe zainteresowanie i zapotrzebowanie na inteligentne systemy przetwarzania i analizy obrazów. W związku z powyższym, algorytmy odpowiedzialne za detekcję obiektów pierwszoplanowych powoli stają się nieodłącznym elementem większych i bardziej zaawansowanych systemów wizyjnych.

Jednym z przykładów takiego systemu może być zautomatyzowany monitoring wizyjny. Tego typu narzędzie może zostać wykorzystane na przykład do obserwowania strefy zabronionej i wykrywania obecności niepożądanych tam osób. Kolejne zastosowanie algorytmów detekcji obiektów pierwszoplanowych to rozbudowany system nadzorowania ruchu drogowego. Jest to rozwiązańe umożliwiające gromadzenie danych archiwalnych oraz analizę w czasie rzeczywistym takich parametrów jak średnia prędkość pojazdów, czas przejazdu z punktu A do B, natężenie ruchu oraz aktualne utrudnienia w ruchu drogowym. Dzięki takiemu systemowi, możliwa jest między innymi optymalizacja cykli sygnalizacji świetlnej na wszystkich skrzyżowaniach, co w ostatczności skutkuje minimalizacją korków i czasu podróży.

Początkowo w dziedzinie segmentacji obiektów niezwykle istotny był czynnik ludzki, a rolę algorytmu wykrywającego i analizującego obiekty pierwszoplanowe pełnił wykwalifikowany operator. Taka sytuacja bardzo często występowała w przypadku systemów monitoringu wizyjnego. Niestety, takie rozwiązanie z biegiem czasu stało się nieopłacalne, na co miało wpływ kilka czynników. Po pierwsze, należy zaznaczyć, że zazwyczaj przez 99% czasu pracy operatora nie występują żadne angażujące go sytuacje. Człowiek nie jest w stanie przez dłuższy czas skupić się na analizie i nadzorować jednocześnie kilku obrazów. Z tego powodu, konieczne jest zastosowanie systemu działającego w czasie rzeczywistym, który daje możliwość przetwarzania określonej liczby ramek obrazu w danej rozdzielcości na sekundę.

W związku z powyższym, coraz częściej do realizacji systemów wizyjnych wykorzystuje się układy *FPGA* (ang. *Field-Programmable Gate Array* — bezpośrednio programowalna macierz bramek). Wraz z rozwojem technologicznym zaczęły one stanowić atrakcyjną alternatywę dla komputerów klasy *PC* (ang. *Personal Computer* – komputer osobisty) z procesorami typu *GPP* (ang. *General Purpose Processor – procesor ogólnego przeznaczenia*). Jedną z najważniejszych różnic pomiędzy tymi układami jest pełna rekonfigurowalność w przypadku *FPGA*. Podczas procesu produkcji procesorów typu *GPP* tranzystory są na stałe zatapiane w krzemie i nie ma możliwości późniejszej ingerencji w zaprojektowaną architekturę. Dostajemy możliwość modyfikacji jedynie programu, który na takim procesorze będzie wykonywany. Z kolei w układach *FPGA* dostajemy pełną możliwość przeprojektowania architektury i dostosowania jej do potrzeb danego zadania.

Ze względu na pełną rekonfigurowalność oraz możliwość zrównoleglenia obliczeń układy *FPGA* zyskują zdecydowaną przewagę nad procesorami *GPP* w kontekście realizacji operacji przetwarzania obrazów w czasie rzeczywistym. W przypadku standardowego procesora operacja przetworzenia jednej ramki obrazu wymaga odczytania jej w całości, następnie analizy po kolejno każdego piksela i na końcu przesłania finalnej ramki na wyjście. Systemy wizyjne w układach *FPGA* działają nieco inaczej, tutaj przetwarzanie obrazu odbywa się potokowo. W każdym taktie zegara na wejście modułu podawany jest jeden piksel z obrazu wejściowego oraz jeden piksel zostaje wystawiony na wyjściu. Dzięki temu latencja (opóźnienie) między obrazem wejściowym a wyjściowym trwa dokładnie tyle ile przeanalizowanie jednego piksela a nie całej ramki, jak ma to miejsce przy wykorzystaniu procesorów *GPP*. Jest to tzw. równoległość drobnoziarnista (ang. *fine-grained*). W przypadku standardowych procesorów *GPP* posiadających więcej niż jeden rdzeń, również istnieje możliwość wykonywania obliczeń równoległych.

Liczba dostępnych wątków, jest jednak zdecydowanie niższa niż w układzie *FPGA*, takie podejście nazywamy zrównolegleniem gruboziarnistym (ang. *coarse-grained*).

Implementowanie algorytmów do segmentacji obiektów pierwszoplanowych i uruchamianie ich na platformie sprzętowej jest procesem dość złożonym i wieloetapowym. Często stosowane jest podejście, w którym dany algorytm w pierwszej kolejności zostaje zaimplementowany właśnie na standardowej, ogólnodostępnej platformie takiej jak komputer *PC*. Taka implementacja jest zdecydowanie prostsza, gdyż mamy do dyspozycji wiele wysokopoziomowych języków np. *C++*, *Java*, *Python*, *Matlab*. Architektura w układach *FPGA* jest tworzona z wykorzystaniem języków do opisu sprzętu, takich jak *Verilog* i *VHDL* (*Very High Speed Integrated Circuits Hardware Description Language*). Niestety, nie wszystkie operacje i algorytmy mogą zostać bezproblemowo przeniesione z platformy *PC* na układ *FPGA* i zaimplementowane w systemie potokowym. Przykładami mogą być operacje zmienoprzecinkowe oraz skomplikowane obliczeniowo funkcje matematyczne (np. trygonometryczne, eksponenta, logarytmy). Mimo tych ograniczeń, ciężko wskazać alternatywne rozwiązanie, które spełniało by wymóg pracy w czasie rzeczywistym. Dodatkowo porównując do procesorów *GPP*, układy reprogramowalne są rozwiązaniem wydajniejszym i bardziej energooszczędnym.

Oprócz standardowych procesorów i układów *FPGA* istnieje jeszcze trzecia alternatywa, którą są procesory graficzne *GPU* (ang. *Graphics Processing Unit*). Takie rozwiązanie na pewno jest o wiele bardziej wydajne niż procesor *GPP* i posiada mniej ograniczeń niż układy *FPGA*. Zużycie energii jest jednak o wiele wyższe niż w przypadku dwóch omawianych wcześniej platform. Nie można również pomijać aspektu ekonomicznego, układy graficzne są dość kosztowne i dodatkowo wymagają współpracy ze standardowymi procesorami (technologie *OpenCL* i *OpenGL*). Reasumując, na chwilę obecną to właśnie układy *FPGA* wydają się być najrozsądzniejszą opcją, będącą jednocześnie rozwiązaniem wydajnym, energooszczędnym oraz stosunkowo tanim.

1.1. Cel pracy

Celem niniejszej pracy było stworzenie biblioteki modułów sprzętowych, które realizują różne algorytmy segmentacji obiektów pierwszoplanowych. W pracy położono główny nacisk na metody, które zostały już w pewnym stopniu zrealizowane w ramach innych prac inżynierskich, magisterskich oraz publikacji naukowych w Laboratorium Biocybernetyki Akademii Górnictwo-Hutniczej im. Stanisława Staszica w Krakowie. Zebrane algorytmy zostały przystosowane do uruchomienia na jednej platformie sprzętowej – karcie VC 707 z układem *FPGA* Virtex 7 firmy Xilinx. Oprócz implementacji sprzętowej, do każdej metody zapewniono odpowiedni model programowy, co dało możliwość przeprowadzenia miarodajnych testów efektywności każdego algorytmu.

Modele programowe przygotowano z wykorzystaniem biblioteki *OpenCV* [25]. W celu przetestowania każdego algorytmu użyto zbioru sekwencji testowych dostępnego w bazie *ChangeDetection* [3].

Jest to zbiór bardzo często wykorzystywany do testów nowych wersji algorytmów. Dzięki dużej popularności, istnieje możliwość porównania otrzymanych wyników z rezultatami uzyskanymi w innych publikacjach.

Najważniejszym etapem pracy była implementacja wszystkich metod na wspólnej platformie sprzętowej. Algorytmy porównano pod względem wykorzystania zasobów układu *FPGA*, ilości wymaganej pamięci *RAM*, wydajności obliczeniowej oraz akceleracji w stosunku do modelu programowego uruchamianego na komputerze PC. Dodatkowo, sprawdzono również możliwość przetwarzania obrazu w wyższych rozdzielczościach oraz dokonano pomiarów zużycia energii.

1.2. Zawartość pracy

W rozdziale 2 zamieszczono analizę publikacji dotyczących segmentacji obiektów pierwszoplanowych. Przedstawiono różne podejścia do omawianej tematyki, z wyraźnym zaznaczeniem, które algorytmy zostały zrealizowane również w Laboratorium Biocybernetyki AGH i zamieszczone w niniejszej pracy.

Rozdział 3 zawiera szczegółowe opisy wybranych algorytmów. Główny nacisk położono na wiadomości teoretyczne i dokładny zapis matematyczny. Przedstawiono również wady, zalety oraz domyślne wartości parametrów.

W rozdziale 4, opisano implementację wybranych algorytmów w układzie reprogramowalnym. Zamieszczono szczegółowe schematy przedstawiające zaimplementowaną architekturę dla każdego algorytmu oraz opisano trudności, które powstały podczas ich realizacji. Oprócz tego, porównano wszystkie algorytmy pod względem zużycia zasobów i energii.

Rozdział 5 zawiera szczegółowe testy poszczególnych algorytmów. Oprócz bezpośredniego porównania, otrzymane wyniki zestawiono także z innymi algorytmami dostępnymi w rankingu *ChangeDetection* [3].

W rozdziale 6 zamieszczono dalsze, możliwe kierunki rozwoju zaimplementowanych algorytmów oraz potencjalne możliwości ich poprawy. Natomiast rozdział 7 zawiera kompletne podsumowanie wykonanej pracy oraz konfrontację osiągniętych celów z przyjętymi na początku założeniami. Zamieszczono także, krótkie streszczenie wszystkich wniosków, które wyciągnięto na poszczególnych etapach realizacji niniejszej pracy.

2. Przegląd metod detekcji obiektów pierwszoplanowych

2.1. Wprowadzenie

Niniejszy rozdział ma na celu przedstawić dotychczasowe osiągnięcia w dziedzinie segmentacji obiektów pierwszoplanowych. Ponieważ jest to zagadnienie bardzo rozległe, położono nacisk głównie na najpopularniejsze podejścia, osiągające zadowalające rezultaty w różnego rodzaju testach weryfikacyjnych. Omówiono zarówno algorytmy opracowane w ramach prac dyplomowych oraz artykułów w Laboratorium Biocybernetyki AGH jak i inne metody opublikowane podczas konferencji naukowych na całym świecie.

Główny nacisk położono na pokazanie różnych podejść do tworzenia zaawansowanych modeli tła, wykorzystywanego następnie do klasyfikacji pikseli znajdujących się na kolejnych ramkach obrazu. Niezależnie od samego sposobu modelowania tła, istotnym zagadnieniem jest także sposób jego aktualizacji. Różne podejścia do tematu aktualizacji używanego modelu również zostały przedstawione w niniejszym rozdziale. Kolejnym elementem wartym zaznaczenia, są różnego rodzaju dodatkowe funkcjonalności wspierające podstawową wersję algorytmu. Może to być, na przykład moduł do wykrywania obiektów statycznych, bądź też mechanizm eliminujący drgania kamery.

2.2. Model tła bazujący na poprzednich ramkach

Pierwszym typem metod, są algorytmy bazujące na wartościach pikseli z poprzednich ramek obrazu. Niewątpliwie jedną z bardziej popularnych tego typu metod jest *ViBE* (*Visual Background Extractor*). Algorytm został opisany w wielu publikacjach, między innymi w [1, 4], doczekał się także wielu usprawnień. Oprócz zagranicznych publikacji metoda ta, była również obiektem badań w Laboratorium Biocybernetyki AGH, gdzie dokonano jej implementacji w układzie reprogramowalnym [23]. Oprócz standardowej wersji opracowano także rozszerzoną odmianę algorytmu, posiadającą dodatkowy mechanizm eliminacji efektów drgającej kamery [22]. Metoda ta została również zawarta w niniejszej pracy, jej szczegółowy opis przedstawiono w rozdziale 3.2.

Kolejnym algorytmem bazującym na poprzednich ramkach obrazu jest *PBAS* (*Pixel Based Adaptive Segmenter*). Metoda została zaprezentowana między innymi w [10]. W ramach badań w Laboratorium Biocybernetyki AGH opracowano implementację sprzętową tego algorytmu [18]. Udało się również dołączyć do podstawowej wersji algorytmu dodatkowy mechanizm zapewniający lepszą detekcję obiektów

statycznych [21]. Ulepszona wersja również została zaimplementowana w układzie *FPGA*. Omawiany algorytm również jest elementem badań niniejszej pracy, a jego dokładny opis teoretyczny zamieszczono w rozdziale 3.3.

2.3. Statystyczne modele tła

Druga grupą algorytmów, są metody bazujące w większym stopniu na modelach statystycznych. Jednym z najprostszych tego typu algorytmów jest tzw. „średnia krocząca”. Metoda ta została zrealizowana w niniejszej pracy, opis teoretyczny zamieszczono w rozdziale 3.1. Do tego typu algorytmów możemy zaliczyć także prostą metodę odejmowania ramek oraz porównywanie aktualnej ramki z modelem referencyjnym. Obie te metody zaprezentowano w przytoczonym wyżej rozdziale.

Spośród zaawansowanych metod, najprawdopodobniej najpopularniejszą z tego rodzaju jest algorytm *GMM* (*Gaussian Mixture Models*). Głównym założeniem tej metody jest reprezentacja modelu tła poprzez rozkłady Gaussa. Algorytm po raz pierwszy został opisany w 1999 roku w publikacji [29]. Powstały również publikacje przedstawiające implementację tej metody w układzie reprogramowalnym [6]. Algorytm, był także przedmiotem badań prac dyplomowych w Laboratorium Biocybernetyki AGH [13, 27]. Przygotowana w ramach pracy dyplomowej [27] implementacja sprzętowa została również przeanalizowana w niniejszej pracy. Szczegółowy opis algorytmu zamieszczono w rozdziale 3.4.

Kolejnym algorymem przedstawiającym nieco odmienne podejście jest metoda *Flux Tensor*. Jest to algorytm bazujący na wykrywaniu krawędzi obiektu i badaniu pochodnej obrazu po czasie w celu wykrycia obiektów ruchomych – niestety przy jej użyciu nie ma możliwości detekcji obiektów statycznych. Od strony teoretycznej, metoda została omówiona między innymi w publikacji [26], natomiast pierwsza implementacja sprzętowa została opracowana w Laboratorium Biocybernetyki AGH i przedstawiona na konferencji *ICCVG* (*International Conference on Computer Vision and Graphics*) [13, 14].

Bardzo ciekawe podejście do zagadnienia segmentacji obiektów zostało przedstawione w publikacji [31]. Pokazany algorytm *FTSG* (*Flux Tensor with Split Gaussian models*) jest metodą hybrydową wykorzystującą omówione wcześniej metody *GMM* i *Flux Tensor* oraz dodatkowy mechanizm detekcji dynamicznego tła i obiektów statycznych. Algorytm w momencie publikacji był najdokładniejszą metodą w rankingu *ChangeDetection* [3]. Metoda ta, została częściowo zaimplementowana w układzie reprogramowalnym w ramach jednej z pracy dyplomowych w Laboratorium Biocybernetyki AGH [13].

Do grupy metod statystycznych można zaliczyć także algorytm *KDE* (*nonparametric Kernel Density Estimation*). Metoda została po raz pierwszy opublikowana w 2002 roku [5]. Rezultaty, które można uzyskać w testach przy jej wykorzystaniu nadal są satysfakcyjne przez co często jest cytowana w różnego rodzaju publikacjach i służy jako punkt odniesienia do nowych algorytmów. Idea metody opiera się na funkcji gęstości prawdopodobieństwa, która jest używana do stworzenia modeli statystycznych tła i pierwszego planu. Podobnie jak w przypadku pozostałych algorytmów, ta metoda również doczekała się wielu rozszerzeń i usprawnień [32].

2.4. Różne podejścia do aktualizacji modelu tła

W przypadku większości algorytmów wykorzystujących zaawansowany model tła, pojawia się zagadnienie aktualizacji modelu. Temat ten został poruszony w większości publikacji przytoczonych w rozdziałach 2.2 i 2.3. Autorzy zazwyczaj rozróżniają dwa rodzaje podejść do aktualizacji modelu: liberalne (ang. *liberal approach*) oraz podejście konserwatywne (ang. *conservative approach*). Podejście liberalne zakłada aktualizację wszystkich pikseli, podczas gdy konserwatywne jedynie tych, które zostały sklasyfikowane jako tło. Obie metody mają oczywiście swoje wady i zalety.

Główną wadą podejścia liberalnego jest stosunkowo szybkie wtapianie się obiektów pierwszoplanowych do modelu tła. Zjawisko to, jest szczególnie widoczne w przypadku wolno poruszających się obiektów. Drugim niepożądanym efektem jest pozostawianie smugi za poruszającymi się obiekty. Polityka konserwatywnego aktualizowania modelu eliminuje ten problem, jednak ma inną poważną wadę. Podejście to, może prowadzić do omówionego już, wystąpienia tzw. „duchów”, czyli błędnej interpretacji odsłoniętego tła. Tego typu przypadek, może wystąpić w przypadku gdy obiekt pierwszoplanowy, początkowo statyczny (np. samochód stojący na światłach), zaczyna się poruszać. W takiej sytuacji algorytm może nadal interpretować obszar, w którym stał samochód, jako element pierwszego planu. Kolejnym problemem są również różnego rodzaju zakłócenia i szумy – raz błędnie sklasyfikowany obszar może pozostać już nienaprawiony.

Po analizie wspomnianych publikacji można zauważyć, że pomimo poważnych wad, częściej stosowanym podejściem jest polityka konserwatywna. Wykorzystywane są również dodatkowe mechanizmy pomagające eliminować wspomniane problemy. Przykładem może być niezależna aktualizacja pikseli sąsiadujących z aktualnie aktualizowanym jak ma to miejsce w algorytmach *ViBE* [23] lub *PBAS* [18]. Innym rozwiązaniem jest osobny moduł do całkowitej eliminacji zjawiska „duchów”, takie podejście zrealizowano między innymi w rozszerzonym algorytmie *PBAS* [21] oraz *FTSG* [31].

2.5. Wykorzystanie map głębi obrazu

Autorzy publikacji [9] przedstawili inne, bardzo ciekawe i niestandardowe podejście do segmentacji obiektów pierwszoplanowych. Zaprezentowany algorytm zakłada wykorzystanie sieci *CNN* (ang. *convolutional neural network* – konwolucyjne sieci neuronowe). Do uczenia sieci wykorzystywany jest, oprócz annotowanych danych uczących, także czujnik *RGB-D*, czyli urządzenie generujące obraz wraz z mapami głębi.

Oczywiście istnieje wiele metod segmentacji obiektów pierwszoplanowych, wykorzystujących sieci neuronowe, w tym przypadku autorzy skupili się na usprawnieniu procesu uczenia. Standardowo w tego typu algorytmach, sieć uczona jest na podstawie annotowanych obrazów w przestrzeni *RGB*. Zbiory testowe można podzielić na wiele kategorii, jednak uzyskany w ten sposób dokładność detekcji nie zawsze jest zadowalająca. W związku z tym autorzy przedstawili hybrydowy system, w którym równolegle

uczone są dwie sieci. Pierwsza z nich wykorzystuje standardowy zbiór obrazów zapisanych w przestrzeni *RGB*, natomiast w drugiej sieci wykorzystywana jest mapa głębi tego obrazu.

Głównym problemem w tego typu podejściu jest brak wystarczająco rozbudowanych zbiorów uczących, wykorzystujących mapy głębi obrazu. Ich liczba jest wyraźnie kilkukrotnie mniejsza niż standar-dowych zbiorów *RGB*. Autorzy zaproponowali podejście, w którym kluczową rolę odgrywa wymiana informacji pomiędzy obiema sieciami *CNN* w trakcie procesu uczenia. Dzięki takiemu rozwiążaniu można dużo efektywniej przeprowadzić proces uczenia obu sieci i następnie połączyć je w jedną. Testy wykonane przez autorów wykazał poprawę o około 21% w stosunku do standardowego podejścia.

2.6. Mechanizm detekcji cieni

W Laboratorium Biocybernetyki AGH, opracowano również algorytm zawierający dedykowany mechanizm detekcji cieni. Całość od strony teoretycznej jak i implementacji sprzętowej została opisana w publikacji [19]. Metoda została opracowana na bazie algorytmu klasteryzacji, którego opis można znaleźć między innymi w publikacji [2]. Przedstawiona podejście jest bardzo podobne do wspomnianego już algorytmu *GMM*. W tym przypadku model tła składa się z klastrów, inicjalizowanych wartościami piksela. Następnie przeprowadzany jest test dopasowania do poszczególny klastrów i na tej podstawie dokonywana jest ostateczna klasyfikacja oraz decyzja odnośnie aktualizacji modelu.

Najistotniejszym elementem metody jest mechanizm detekcji cieni. Jego działanie opiera się na założeniu, że cień nie zmienia ani koloru ani tekstury elementu na który pada. Jedyny parametr który ulega modyfikacji to jasność, co również nie zawsze jest prawdziwe, głównie w warunkach silnego oświetlenia. W algorytmie wykorzystano deskryptor teksturowy *SILTP* (ang. *Scale Invariant Local Ternary Pattern* – skalo-niezmiennej lokalny trzyelementowy wzorzec), opisany szerzej w publikacji [28]. Jego rolą jest w tym przypadku poprawa dokładności segmentacji obiektów ruchomych. Detekcja przeprowadzana jest w przestrzeni barw *YCbCr* oraz *CIELab*, ponieważ to właśnie z ich składowych najłatwiej wyodrębnić cechy charakteryzujące cienie.

Autorzy publikacji [19] opisali również implementację sprzętową przedstawionego rozwiązania. Deskryptora *SILTP* użyto między innymi z powodu jego niezbyt skomplikowanej logiki i łatwości zaimplementowania w układzie *FPGA*. Konwersja do przestrzeni *YCbCr* również nie jest operacją złożoną obliczeniowo, jedynym wyzwaniem była konwersja do *CIELab*, wymagająca wielu obliczeń. Warto zaznaczyć, że ta przestrzeń barw została wykorzystana w jednym z algorytmów zrealizowanych w ramach niniejszej pracy. Dokładny opis tej operacji i jej optymalna implementacja sprzętowa zostały zamieszczone w rozdziale 4.5.

2.7. Algorytm detekcji wykorzystujący histogramy blokowe

Innym ciekawym podejściem do detekcji obiektów jest algorytm wykorzystujący *HOG* (ang. *Histogram of Oriented Gradients* – histogram zorientowanych gradientów). W odróżnieniu od pozostałych

metod może on zostać zastosowany tylko w specyficznych warunkach, gdy celem jest segmentacja konkretnego typu obiektów. Implementacja sprzętowa została opracowana w Laboratorium Biocybernetyki AGH między innymi w ramach pracy dyplomowej [17]. W tym celu wykorzystano hybrydowy układ *Zynq* łączący w sobie procesor o architekturze *ARM* z *FPGA*. Zastosowaniem stworzonego systemu wizyjnego była detekcja sylwetek ludzkich. W niniejszym podrozdziale przedstawiono w skróconej wersji główne założenia i koncepcję tego algorytmu.

Metoda opiera się na połączeniu wspomnianych wcześniej histogramów zorientowanych gradientów i maszyny wektorów nośnych *SVM* (ang. *Support Vector Machine*). Pierwszym krokiem jest podzielenie obrazu na komórki (zazwyczaj o rozmiarze 8×8 pikseli). Histogram zostaje wyznaczony poprzez zliczanie gradientów wyliczonych w danej komórce. Komórki po raz kolejny grupowane są w większe bloki wewnętrz których następuje normalizacja. Taki zabieg ma na celu niwelację kontrastów pomiędzy blokami. Po znormalizowaniu wszystkich bloków tworzony jest ostateczny wektor cech.

Kolejny etap to klasyfikacja. Jest ona dokonywana na podstawie maszyny wektorów nośnych *SVM*. Sam proces polega na wyszukiwaniu hiperpłaszczyzny, która najlepiej oddziela punkty z różnych klas. Punte położone najbliżej wspomnianej hiperpłaszczyzny nazywane są wektorami nośnymi. Niezwykle istotnym elementem jest funkcja jądra, służąca do przekształcenia przestrzeni cech do przestrzeni o większej liczbie wymiarów. Do tego celu wykorzystywane są głównie funkcje wielomianowe jak i *RBF* (ang. *Radial Basis Function* – radialna funkcja bazowa). Sam proces klasyfikacji jest stosunkowo szybki, jednak dużo istotniejszy jest proces uczenia klasyfikatora.

Algorytm musi posiadać możliwość detekcji obiektów o różnych rozmiarach, ostatnim elementem jest zatem dodatkowy generator przeskakujący obraz wejściowy do różnych rozmiarów. Jego dokładna koncepcja jest dość złożona, w pierwszej kolejności wykonywana jest operacja rozmycia. Jest to standardowy zabieg w wielu metodach, mający na celu eliminację części zakłóceń i szumów. Następnie wykonywane jest właściwe skalowanie obrazu, piksel wynikowy wyznaczany jest zawsze na podstawie czterech sąsiadów (interpolacja dwuliniowa).

2.8. Podsumowanie

Powyższy rozdział, został napisany w celu zaprezentowania różnych podejść do tematu segmentacji obiektów pierwszoplanowych. Jak łatwo zauważyć, na przykładzie wymienionych metod, jest to dziedzina bardzo rozległa oraz stale się rozwijająca. Pokazane algorytmy dowodzą, że do tego zagadnienia można podejść na wiele sposobów. Prezentacja nowych algorytmów lub usprawnionych wersji metod już dostępnych zazwyczaj ma miejsce na różnego rodzaju konferencjach. Do najpopularniejszych możemy zaliczyć *AVSS (Advanced Video an Signal – Based Surveillance)*, *CVPR (Computer Vision and Pattern Recognition)* oraz *ICCVG (International Conference on Computer Vision and Graphics)*.

Ponieważ dokładny opis wszystkich algorytmów znajduje się we wskazanych artykułach, w tym rozdziale przedstawiono jedynie idee, oraz główne założenia poszczególnych metod. Warto zaznaczyć, że w niektórych przypadkach istnieje wiele podejść do realizacji tej samej metody. Takim przykładem, jest

między innymi metoda *GMM*. Drugim elementem wartym uwagi, jest sam dobór i kalibracja konkretnej metody do pracy w konkretnym środowisku, dyskusja na tym polu również może być bardzo rozległa.

3. Opis teoretyczny wybranych algorytmów

3.1. Proste metody segmentacji obiektów pierwszoplanowych

W tym rozdziale przedstawiono kilka prostych obliczeniowo metod segmentacji tła. Oczywiście, uzyskane z ich wykorzystaniem wyniki, znaczco odbiegają, od tego co oferują zaawansowane algorytmy. Szczególnie podczas testów w bardziej wymagającym środowisku, na przykład w przypadku drgań kamery lub z obecnością dynamicznego tła (np. ruszające się pod wpływem wiatru liście, fale na wodzie). Mimo to, często znajdują zastosowanie w mniej wymagających systemach, właśnie ze względu na swoją prostotę i w rezultacie niską złożoność obliczeniową.

Pierwszym omawianym podejściem jest **metoda naiwna**. W tym algorytmie, przyjmujemy założenie, że pierwsza ramka jest modelem referencyjnym tła i nie zawiera żadnych obiektów pierwszoplanowych. Proces segmentacji tła polega na obliczeniu różnicy między aktualną ramką, a zapisanym modelem. Jeżeli dla konkretnego piksela otrzymana różnica jest większa od przyjętej wartości progowej zostaje on uznany za element pierwszego planu. Przedstawiony algorytm oczywiście nadaje się jedynie do prostej segmentacji i nie zapewnia akceptowalnych rezultatów podczas pracy w bardziej wymagającym środowisku.

Kolejnym prostym algorymem jest technika **odejmowania ramek**, w tym przypadku również wykonywane jest odejmowanie modelu referencyjnego i aktualnej ramki. Jako model referencyjny wykorzystywana jest poprzednia ramka obrazu. Metoda ta nadaje się niestety jedynie do detekcji obiektów ruchomych. W niniejszej pracy algorytm ten został również wykorzystany jako element mechanizmu detekcji tzw. „duchów”, całość została szczegółowo przedstawiona w rozdziale 3.3.2.

Ostatnia spośród prostych metod to tzw. **adaptacyjne odejmowanie tła**, technika ta jest w pewnym stopniu rozszerzeniem opisanej powyżej metody naiwnej. W tym przypadku także wykonywane jest odejmowanie modelu tła i aktualnej ramki obrazu, a następnie dokonywanie klasyfikacji poprzez progowanie. Nieco bardziej złożona jest procedura aktualizacji modelu tła. Jest ona wykonywana po każdej ramce obrazu i została zapisana równaniem (3.1).

$$B(t) = \alpha I(t) + (1 - \alpha)B(t - 1) \quad (3.1)$$

gdzie:

$B(t)$ – model tła w chwili t

$I(t)$ – obraz wejściowy w chwili t

α – parametr z zakresu od 0 do 1

Warto zaznaczyć, że wszystkie trzy, wymienione algorytmy mogą przetwarzać obraz w różnych przestrzeniach barw, przy czym najpopularniejsza jest oczywiście standardowa przestrzeń kolorów *RGB*. W tym przypadku do obliczenia różnicy, koniecznej do segmentacji, wykorzystuje się metrykę euklidesową. Ostateczny wzór na klasyfikację piksela przedstawiono równaniem 3.2. Druga odmianą może być obraz w skali szarości, w takiej sytuacji obliczenia są uproszczone, gdyż konieczne jest wyznaczenie jedynie modułu z różnicą.

$$F(x, y) = \begin{cases} 1, & \text{gdy } \sum_{C \in \{R, G, B\}} |I(x, y, C) - B(x, y, C)| > T \\ 0, & \text{w pozostałych przypadkach} \end{cases} \quad (3.2)$$

gdzie:

$F(x, y)$ – wyjściowa maska binarna piksela o współrzędnych x, y

$I(x, y, C)$ – pojedyncza składowa piksela wejściowego o współrzędnych x, y

$B(x, y, C)$ – pojedyncza składowa piksela z modelu tła o współrzędnych x, y

T – próg klasyfikacji (dla przestrzeni *RGB* domyślnie 50)

3.2. Visual Background Extractor

Algorytm nazywany w skrócie *ViBE* (*Visual Background Extractor*), jak zostało wspomniane w rozdziale 2.2, jest metodą bazującą na modelu opartym o wartości pikseli obserwowane w poprzednich ramkach obrazu. Warto ponownie podkreślić, że zapamiętane wartości pikseli z poprzednich ramek obrazu to jedyna informacja wykorzystywana przez ten algorytm. Podobnie jak większość innych metod, *ViBE* może zostać dostosowane do pracy z różnymi przestrzeniami barw (np. *RGB*, skala szarości, *CIELab*).

Opisana w niniejszej pracy wersja algorytmu została przedstawiona w publikacji [22]. Standardową wersję metody dostosowano do pracy z poruszającą się kamerą. Realizowane jest to, poprzez mechanizm przesunięcia modelu tła. W celu wyznaczenia wektora przesunięcia wykorzystywane jest obliczenie rzadkiego przepływu optycznego, dokładne szczegóły takiego rozwiązania, zostały zaprezentowane w podrozdziale 3.2.4.

3.2.1. Opis algorytmu

Model tła zdefiniowany jest osobno dla każdego piksela i składa się z N próbek (tj. wartości pikseli). Proces inicjalizacji jest bardzo prosty i trwa tylko jedną ramkę obrazu. Każda z N próbek inicjalizowana

jest losowo, wartością odpowiadającą jej piksela lub któregoś z jego sąsiadów. W tym przypadku przez sąsiedztwo rozumiemy przestrzenny kontekst 3×3 , czyli jedynie najbliższe otaczające piksele.

Zdefiniujmy wartość piksela (o współrzędnych x, y) w wybranej przestrzeni barw jako $v(x, y)$, z kolei i-tą próbki z modelu tła oznaczmy jako v_i . Model tła dla konkretnego piksela, możemy wtedy zapisać za pomocą wzoru (3.3).

$$M(x, y) = \{v_1, v_2, \dots, v_N\} \quad (3.3)$$

W celu przeprowadzenia klasyfikacji nowego piksela i przypisania mu etykiety obiektu pierwszoplanowego lub tła, należy zdefiniować okrąg $S(v(x, y))$ o środku w punkcie $v(x, y)$ i promieniu R . Promień ten jest wartością stałą, identyczną dla każdego modelu. Punkt uznawany jest za pierwszoplanowy jeżeli przynajmniej $\#_{min}$ próbek z modelu tła zawiera się wewnątrz takiego okręgu. Oznaczmy przez F maskę reprezentującą obiekty pierwszoplanowe (1 – piksel pierwszoplanowy, 0 – tło), wówczas opisany warunek przedstawia równanie (3.4).

$$F(x, y) = \begin{cases} 1, & \text{gdy } \sum_{k=0}^N \{d(v(x, y), v_k(x, y)) < R\} < \#_{min} \\ 0, & \text{w pozostałych przypadkach} \end{cases} \quad (3.4)$$

gdzie d to funkcja odległości pomiędzy próbką z modelu tła, a aktualnym pikselem.

Sposób obliczania odległości zależy od przyjętej przestrzeni barw. Można wykorzystać obraz w skali szarości, standardową przestrzeń RGB lub $CIELab$. W przypadku dwóch pierwszych wykorzystywana jest prosta metryka miejska, czyli inaczej suma modułów różnicy każdej składowej. Alternatywnym rozwiązaniem, aczkolwiek bardziej złożonym obliczeniowo jest metryka euklidesowa. Autorzy artykułu [22, 23] zdecydowali się na wykorzystanie przestrzeni $CIELab$. Algorytm konwersji z przestrzeni RGB został opisany w rozdziale 3.2.2, natomiast odległość między próbками jest przedstawiona równaniem (3.5).

$$d_{CIELab} = \alpha \cdot |L_I - L_B| + \beta \cdot (|a_I - a_B| + |b_I - b_B|) \quad (3.5)$$

Gdzie:

L_I, a_I, b_I – składowe piksela wejściowego

L_B, a_B, b_B – składowe zapisane w modelu tła

α, β – wagi wynoszące odpowiednio 1 i 1,5

Składowa jasności L ma mniejsze znaczenie, w kontekście rozróżniania obiektów, od składowych chrominancji a i b . W związku z tym, eksperymentalnie zastosowano w jej przypadku minimalnie niższą wagę.

W omawianej metodzie wykorzystano konserwatywną politykę aktualizowania modelu tła – szcześliwe różnice pomiędzy podejściem liberalnym a konserwatywnym zaprezentowano w rozdziale 2.4.

Aktualizacji podlegają zatem jedynie modele reprezentujące obszar zaklasyfikowany jako tło. W trakcie aktualizacji, wprowadzony został element losowy. Mianowicie dla każdego modelu tła podejmowana jest decyzja (w sposób całkowicie losowy), czy należy dokonywać aktualizacji. Przyjęto, że prawdopodobieństwo wykonania aktualnienia jest stałe i reprezentowane przez parametr T .

Sam proces jest bardzo prosty, kiedy już dany model zostanie zakwalifikowany do aktualizowania wybierana jest losowa próbka i jej wartość zostaje nadpisana przez aktualną wartość piksela $v(x, y)$. Dodatkowo wybierany jest losowo jeden model z najbliższego sąsiedztwa (kontekst przestrzenny o rozmiarze 3×3) aktualnie analizowanego piksela i w nim także jedna losowo wybrana próbka zostaje nadpisana wartością $v(x, y)$. Takie działanie ma na celu, częściową eliminację negatywnych skutków konserwatywnego podejścia do aktualizacji modelu, czyli zjawiska „duchów”.

Warto podkreślić, że algorytm *ViBE* w omówionej tutaj, podstawowej wersji wymaga bardzo niewielu liczb parametrów. Należy zdefiniować jedynie liczbę próbek N zapisanych w każdym modelu, promień R okręgu wykorzystywanego w teście dopasowania, wymaganą minimalną liczbę próbek $\#_{min}$, leżącą wewnątrz wspomnianego okręgu oraz prawdopodobieństwo wykonania aktualizacji T .

3.2.2. Konwersja RGB – CIELab

Konwersja z przestrzeni *RGB* do *CIELab* jest stosunkowo złożona pod względem matematycznym, jednak jak zauważono, między innymi w publikacjach [23, 22], wykorzystanie tej przestrzeni kolorów daje znacznie dokładniejsze wyniki segmentacji. W związku z powyższym, omawiany algorytm w finalnej implementacji sprzętowej będzie zrealizowany z wykorzystaniem właśnie tej przestrzeni barw.

Proces konwersji składa się z dwóch etapów. Pierwszym krokiem jest przekształcenie składowych *RGB* do przestrzeni *CIE XYZ*, jest to operacja liniowa zapisana równaniem (3.6).

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.41245 & 0.35758 & 0.18042 \\ 0.21267 & 0.71516 & 0.07217 \\ 0.01933 & 0.11919 & 0.95923 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (3.6)$$

Następna operacja to przekształcenie z właśnie otrzymanej przestrzeni *CIE XYZ* do docelowej *CIELab*. W tym celu wykorzystywana jest funkcja $f(t)$ opisana równaniem (3.7). Ostatecznie wyznaczenie składowej jasności – L i dwóch składowych chrominacji – a i b przedstawia wzór 3.8. Wykorzystane współczynniki wynoszą odpowiednio $X_n = 0.950465$, $Y_n = 1$, $Z_n = 1.088754$

$$f(t) = \begin{cases} t^{-3}, & \text{dla } t > \left(\frac{6}{29}\right)^3 \\ \frac{1}{3} \left(\frac{29}{6}\right)^2 t + \frac{4}{29}, & \text{w przeciwnym razie} \end{cases} \quad (3.7)$$

$$\begin{bmatrix} L \\ a \\ b \end{bmatrix} = \begin{bmatrix} 116 f\left(\frac{Y}{Y_n}\right) - 16 \\ 500 \left[f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right) \right] \\ 200 \left[f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right) \right] \end{bmatrix} \quad (3.8)$$

Z równań (3.7) i (3.8) można wywnioskować, że składowa L przyjmuje wartości z zakresu 0 – 100, natomiast składowe chrominacji a i b mieszczą się w przedziale od –128 do 127. Warto zwrócić uwagę, że ze względu na skomplikowane obliczenia matematyczne (funkcja $f(t)$), implementacja sprzętowa takiej konwersji jest bardzo utrudniona, problem ten oraz jego rozwiązanie zostało szczegółowo opisane w rozdziale 4.5.

3.2.3. Obsługa ruchomej kamery

Kompletny algorytm segmentacji obiektów pierwszoplanowych musi między innymi poprawnie funkcjonować w przypadku lekkich drgań lub delikatnego przemieszczania się kamery. Rozwiązanie zapewniające taką funkcjonalność zostało zaprezentowane między innymi w publikacji [22] i polega na oszacowaniu przesunięcia kamery na podstawie badania przepływu optycznego.

Oznaczmy oszacowane przesunięcie kamery pomiędzy dwiema kolejnymi klatkami jako wektor $[dx, dy]$. Metoda wyznaczania tego przesunięcia została przedstawiona w rozdziale 3.2.4. W celu kompensacji ruchu kamery stosuje się przesunięcie modelu tła zgodnie z równaniem (3.9).

$$M_t(x, y) = M_{t-1}(x + dx, y + dy) \quad (3.9)$$

gdzie przez $M_t(x, y)$ rozumiemy model tła dla piksela o współrzędnych x, y w chwili t .

Jak łatwo zauważyc, po takiej operacji część modeli tła, związana ze skrajnymi pikselami została usunięta i wymaga ponownej inicjalizacji. W tym przypadku, zamiast losowego inicjalizowania, opartego na wartości sąsiednich pikseli, do wszystkich próbek w danym modelu przypisana zostaje aktualna wartość piksela. Takie działanie jest konieczne, ponieważ nie ma możliwości wyznaczenie sąsiedztwa dla skrajnych pikseli.

3.2.4. Wyznaczanie rzadkiego przepływu optycznego

Wspomniany w poprzednim podrozdziale wektor przesunięcia, może zostać oszacowany poprzez wyznaczenie przepływu optycznego. W przypadku opisywanej metody, należy przyjąć założenie, że przesunięcie jest identyczne na obszarze całego obrazu. Obliczanie przepływu nie jest wykonywane na całej ramce obrazu, a jedynie na reprezentatywnej grupie pikseli (tzw. rzadki przepływ optyczny). Wybierane są takie fragmenty, na których najprościej śledzić przesunięcie, w tym przypadku zdecydowano się na analizę pikseli reprezentujących narożniki obiektów. Do ich detekcji wykorzystany został algorytm Harrisza-Stephensa, dokładny opis tej metody zamieszczono w podrozdziale 3.2.5.

Obraz podzielono na takie same kwadratowe bloki o rozmiarze 32×32 piksele każdy. Dla kolejnych pikseli obliczana jest odpowiedź H_R detektora narożników (metoda Harrisza-Stephensa). W każdym z bloków wyznaczany jest piksel dla którego zwrocona wartość, jest najwyższa. Wyższa wartość H_R oznacza większe prawdopodobieństwo, że dany piksel jest właściwie narożnikiem jakiegoś obiektu.

Jeżeli otrzymana wartość H_R przekracza ustalony z góry próg H_{TH} , to dany piksel wraz z współrzędnymi w odpowiadającym mu bloku oraz z wartościami sąsiadujących pikseli (ponownie za sąsiedztwo uznajemy kontekst $3x3$) zostaje zapisany.

Kolejnym krokiem jest porównanie, zapamiętanych z poprzedniej ramki, pikseli reprezentujących narożniki obiektów z pikselami z aktualnego obrazu. Porównania przeprowadzane są w obszarze wyznaczonych wcześniej bloków o rozmiarze $32x32$. Kolejne piksele w danym bloku, wraz z otaczającym kontekstem $3x3$, są porównywane z zapamiętanym kontekstem piksela, będącego narożnikiem. Jeżeli dla danego bloku, nie wyznaczono w poprzedniej ramce takiego piksela, to cały blok zostaje wyłączony z analizy. Porównanie dwóch kontekstów opiera się na obliczeniu *SAD* (ang. *Sum of Absolute Difference* – suma modułów różnicy). Wartość odpowiadających sobie pikseli w obu kontekstach są od siebie odejmowane, następnie obliczana jest suma wartości bezwzględnych z tych różnic. Dla każdego bloku, poszukiwany jest piksel, dla którego obliczony *SAD* jest najmniejszy. Jego odległość od piksela będącego narożnikiem, to tzw. wektor przepływu optycznego (ang. *optical flow vector*). Jest ona obliczana osobno wzdłuż osi x i y , ze względu na przyjęty rozmiar każdego bloku może przyjmować wartości od -32 do 32 dla obu osi.

Ostatnią operacją jest wyznaczenie wektora przesunięcia, jest on definiowany jako mediana wektorów przepływu optycznego otrzymanych dla każdego z bloków. Przykład całej procedury został pokazany na rysunku 3.1. Przedstawia on dwie kolejne ramki obrazu (rysunki *a* i *b*), wykryte narożniki w każdym z bloków (czerwone piksele na rysunku *c*), natomiast zielonym kolorem na rysunku *d* zostały oznaczone piksele, dla których *SAD* między ich kontekstem a kontekstem piksela czerwonego w danym bloku był najmniejszy. Ostateczny wektor przesunięcia jest obliczany jako mediana wektorów przesunięcia między pikselem zielonym a czerwonym w każdym bloku i w tym przypadku wynosi $[0, -3]$.

3.2.5. Detekcja narożników - metoda Harrisza-Stephensa

Algorytm detekcji narożników został opisany w publikacji [8]. Z punktu widzenia obliczeń opiera się on głównie na operacjach konwolucji. W pierwszej kolejności należy określić tzw. macierz Harrisza. Jej definicję przedstawia równanie (3.10).

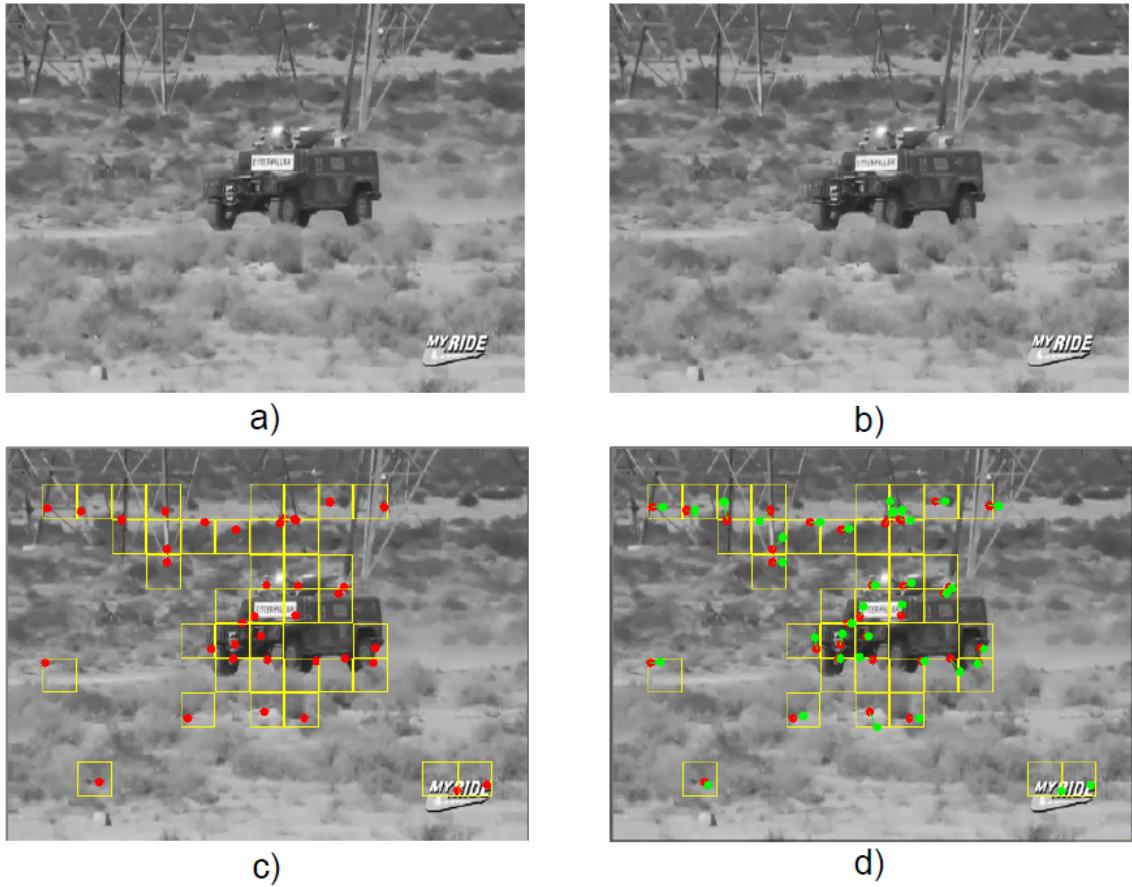
$$H = \begin{bmatrix} I_x^2 \otimes G & I_x I_y \otimes G \\ I_x I_y \otimes G & I_y^2 \otimes G \end{bmatrix} \quad (3.10)$$

gdzie:

I_x, I_y – pierwsza pochodna obrazu wejściowego (pozioma i pionowa)

$\otimes G$ – operacja konwolucji – wygładzenie obraz z wykorzystaniem filtra Gaussa

Pochodne pierwszego rzędu obrazu wejściowego są obliczane z wykorzystaniem filtru Prewitta, maska pozioma P_x i pionowa P_y zostały przedstawione równaniem (3.11). Z kolei przykładowa maska filtru Gaussa (dla $\sigma = 3$) została opisana równaniem (3.12).



Rys. 3.1. Przykładowe wyznaczanie wektora przesunięcia – źródło [22]

$$P_x = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix}, \quad P_y = \begin{bmatrix} +1 & +1 & +1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \quad (3.11)$$

$$G(\sigma = 3) = \begin{bmatrix} 0.1070 & 0.1131 & 0.1070 \\ 0.1131 & 0.1196 & 0.1131 \\ 0.1070 & 0.1131 & 0.1070 \end{bmatrix} \quad (3.12)$$

Ostateczna odpowiedź detektora krawędzi wymaga obliczenia wyznacznika i śladu macierzy H . Finalna wartość może zostać zapisana równaniem (3.13).

$$H_R = \det(H) - k \cdot \text{trace}^2(H) \quad (3.13)$$

gdzie: k – współczynnik skalowania (przyjmuje wartości 0,02 – 0,2)

Po uwzględnieniu wzorów na wskaźnik i ślad macierzy o rozmiarze 2×2 , wzór (3.13) można przekształcić do postaci (3.14). Otrzymana wartość H_R jest porównywana z progiem. Jeżeli wartość H_{TH} zostanie przekroczona dany piksel uznawany jest za narożnik obiektu.

$$H_R = I_x^2 \otimes G \cdot I_y^2 \otimes G - (I_x I_y \otimes G)^2 - k \cdot (I_x^2 \otimes G + I_y^2 \otimes G)^2 \quad (3.14)$$

3.2.6. Uwagi

Algorytm *ViBE* jest metodą zawierającą niezbyt złożony model tła oraz charakteryzującą się niedużą złożonością obliczeniową. Warto jeszcze raz podkreślić niewielką liczbę parametrów – ostateczna lista prezentują się następująco (w nawiasach podano wartości domyślne):

N – liczba próbek w modelu tła (20)

$\#_{min}$ – minimalna liczba próbek wymagana w teście dopasowania (2)

R – próg dopasowania próbki do modelu (15)

T – prawdopodobieństwo wykonania aktualizacji ($\frac{1}{16}$)

W przypadku rozszerzonej wersji algorytmu, należy uwzględnić jeszcze jeden dodatkowy parametr, mianowicie próg H_{TH} (domyślnie 10), wykorzystywany podczas detekcji narożników metodą Harrisa-Stephensa.

3.3. Pixel-Based Adaptive Segmenter

Omawiany algorytm jest rozszerzeniem opisanej rozdziale 3.2.1 metody *ViBE*. *PBAS* (ang. *Pixel Based Adaptive Segmenter*, podobnie jak opisana już wcześniej metoda, także posiada model, który składa się między innymi z próbek pikseli pochodzących z poprzednich ramek obrazu. Dodatkowo, próg dopasowania oraz prawdopodobieństwo dokonania aktualizacji są niezależne i na bieżąco uaktualniane dla każdego modelu. Opisywany algorytm może zostać zaimplementowany zarówno w wersji *RGB* jak i w skali szarości.

Algorytm został zaprezentowany w artykule [21] i to właśnie na jego podstawie powstał niniejszy opis. Oprócz standardowej wersji metody *PBAS* autorzy zaproponowali także dodatkowy mechanizm eliminacji tzw. „duchów” (opis tego zjawiska został dokładnie opisany w rozdziale 3.3.2). Zaproponowane rozwiązanie opiera się na algorytmie etykietowania obiektów i następnie porównywania krawędzi na obrazie wejściowym oraz masce wyjściowej w obrębie każdego z nich.

3.3.1. Opis algorytmu

Model tła jest nieco bardziej złożony niż ten przedstawiony w algorytmie *ViBE*. Jego pierwsza część, podobnie jak w poprzedniej metodzie, składa się z N zapamiętywanych próbek. W celu uproszczenia dalszego zapisu matematycznego, zdefiniujmy ten zbiór jako $B(x_i)$, gdzie x_i to aktualnie przetwarzany piksel obrazu, całość została opisana równaniem (3.15).

$$B(x_i) = \{B_1(x_i), B_2(x_i), \dots, B_N(x_i)\} \quad (3.15)$$

Test dopasowania, jest również bardzo podobny do tego, który występuje w metodzie *ViBE*. Jedyną różnicą jest niezależny dla każdego modelu próg przynależności do modelu $R(x_i)$, całość została przedstawiona równaniem (3.16).

$$F(x_i) = \begin{cases} 1, & \text{gdy } \sum_{k=0}^N \{d(I(x_i), B_k(x_i)) < R(x_i)\} < \#_{min} \\ 0, & \text{w pozostałych przypadkach} \end{cases} \quad (3.16)$$

gdzie d to funkcja odległości pomiędzy próbką z modelem tła, a aktualnym pikselem.

W przypadku algorytmu w wersji *RGB* każdy kanał przetwarzany jest osobno z wykorzystaniem niezależnego modelu tła. Finalna maska jest alternatywną logiczną wyników z poszczególnych kanałów. Oznaczając poszczególne maski jako F_R , F_G , F_B ostateczną klasyfikację możemy zapisać równaniem (3.17).

$$F_{RGB} = F_R \vee \vee F_G \vee F_B \quad (3.17)$$

Ponieważ każdy kanał analizowany jest osobno, funkcję odległości pomiędzy próbками można zapisać bardzo prosto równaniem (3.18). Jest to po prostu moduł różnicy.

$$d(I(x_i), B_k(x_i)) = |I(x_i) - B_k(x_i)| \quad (3.18)$$

Kolejnym krokiem po przeprowadzeniu testu dopasowania i klasyfikacji piksela jest aktualizacja modelu tła. Zastosowano konserwatywne podejście, czyli aktualizowane są tylko piksele sklasyfikowane jako tło. Analogicznie jak w algorytmie *ViBE* decyzja o aktualizacji podejmowana jest losowo. Prawdopodobieństwo jej wykonania wynosi $p = 1/T(x_i)$, gdzie parametr $T(x_i)$ jest dynamicznie aktualizowany i niezależny dla każdego piksela. Sama aktualizacja, polega na nadpisaniu, losowo wybranej próbki $B_k(x_i)$ z modelu aktualną wartością piksela $I(x_i)$. Dodatkowo, wybierany jest losowy piksel z otoczenia $3x3$ i losowo wybrana próbka z modelu mu odpowiadającego, jest nadpisywana wartością tego piksela. Wprowadzenie czynnika losowego, pozwala w pewnym stopniu wyeliminować negatywne skutki konserwatywnego podejścia do aktualizacji modelu, ma także pozytywny wpływ na zjawisko występowania „duchów”.

Niezależnie od aktualizacji części modelu zawierającej zapamiętane próbki dokonywana jest zmiana parametrów $R(x_i)$ i $T(x_i)$. W tym celu konieczne jest zdefiniowanie kolejnego elementu modelu tła, który zawiera zbiór minimalnych odległości pomiędzy próbką z modelem a aktualną wartością piksela. Zbiór ten został opisany równaniem (3.19).

$$D(x_i) = \{D_1(x_i), D_2(x_i) \dots, D_N(x_i)\} \quad (3.19)$$

Przedstawiony zbiór $D(x_i)$ aktualizowany jest razem ze zbiorem próbek. Nadpisywany jest jedynie element o indeksie k dla którego dystans pomiędzy próbką i aktualnym pikselem jest najmniejsza. Zostało to przedstawione równaniem (3.20).

$$d_{min}(x_i) = \min_k d(I(x_i), B_k(x_i)) \quad (3.20)$$

Do aktualizacji progu dopasowania, czyli parametru $R(x_i)$ konieczne jest wyznaczenie tzw. miary dynamiki tła, czyli inaczej wartości średniej ze zbioru $D(x_i)$. Finalny wzór na nową wartość progu

przedstawia równanie (3.21). Warto dodać, że przyjęto także dolne ograniczenie wartości parametru, wynoszące $R_{low} = 18$.

$$R(x_i) = \begin{cases} R(x_i)(1 - R_{inc/dec}), & \text{jeżeli } R(x_i) > \bar{d}_{min}(x_i)R_{sc} \\ R(x_i)(1 + R_{inc/dec}) & \text{w przeciwnym razie} \end{cases} \quad (3.21)$$

gdzie:

$R_{inc/dec}$ – stały współczynnik aktualizacji (domyślnie 0.05)

$\bar{d}_{min}(x_i)$ – wartość średnia zbioru $D(x_i)$

R_{sc} – współczynnik skalowania (domyślnie 5)

Ostatni etap to aktualizacja parametru opisującego prawdopodobieństwo dokonania aktualizacji, czyli $T(x_i)$. Nowa wartość zależy od wyniku klasyfikacji piksela i została opisana równaniem (3.22). Przyjęto założenie, że parametr ten posiada także ograniczenie dolne jak i górne wynoszące odpowiednio $T_{low} = 2$ i $T_{up} = 200$.

$$T(x_i) = \begin{cases} T(x_i) + \frac{T_{inc}}{\bar{d}_{min}(x_i)}, & \text{jeżeli } F(x_i) = 1 \\ T(x_i) - \frac{T_{dec}}{\bar{d}_{min}(x_i)} & \text{w przeciwnym razie} \end{cases} \quad (3.22)$$

Wartość $\bar{d}_{min}(x_i)$ określa dynamikę tła. W przypadku statycznego tła wartość ta będzie równa 0, natomiast jeżeli tło zawiera więcej dynamicznych elementów to będzie odpowiednio wyższa. Parametr ten jest wykorzystywany podczas dynamicznej aktualizacji parametru R , opisanej wzorem (3.21). Jak łatwo zauważyć, wzrost dynamiki tła, powoduje również delikatny wzrost progu R . W przypadku dynamicznego tła często dokonywana jest błędna klasyfikacja piksela jako pierwszoplanowego. Z tego powodu zastosowano dynamiczną aktualizację parametru T opisaną równaniem (3.22). Wartość jest podwyższana właśnie w sytuacji występowania dynamicznego tła w celu lepszego dostosowania modelu.

3.3.2. Detekcja obiektów statycznych

Autorzy publikacji [21] zaproponowali dodatkowy mechanizm eliminacji tzw. „duchów”. Przedstawiona metoda opiera się na porównaniu krawędzi na obrazie wejściowym i masce końcowej. Jeżeli układ krawędzi, na obu obrazach jest identyczny, to znaczy, że obiekt statyczny rzeczywiście istnieje, natomiast w przeciwnym wypadku jest on duchem i zostaje usunięty z finalnej maski.

Pierwszym krokiem, który należy wykonać w celu identyfikacji potencjalnych „duchów” jest operacja odejmowania dwóch kolejnych ramek. W przestrzeni RGB taka operacja jest opisana przez równanie (3.23).

$$dF(x_i) = \sum_{C \in \{R, G, B\}} |I(x_i)_K^C - I(x_i)_{K-1}^C| \quad (3.23)$$

gdzie przez $I(x_i)_K^C$ rozumiemy wartość jednej ze składowych R, G, B piksela o położeniu x_i w K -tej ramce obrazu.

Kolejny etap to obliczenie tzw. współczynnika stabilności. Operację tą opisuje równanie (3.24). Jest ona wykonywana niezależnie dla każdego zidentyfikowanego obiektu na masce wyjściowej. Do tego celu konieczna jest także operacja indeksacji obiektów, ponieważ jest to zagadnienie dość złożone zostało ono opisane oddzielne w rozdziale 3.3.3. Sam współczynnik definiuje się jako stosunek liczby pikseli, których różnica opisana równaniem (3.23) przekracza ustalony próg θ do całkowitej powierzchni obiektu.

$$S_{O_k} = \frac{\sum_{x_i \in O_k} dF(x_i) > \theta}{\sum_{x_i \in O_k} F(x_i)} \quad (3.24)$$

gdzie:

O_k – k -ty obiekt na masce wyjściowej, wyznaczony algorytmem indeksacji

$F(x_i)$ – maska wyjściowa ze standardowego algorytmu PBAS

Obiekt jest uznawany za statyczny (czyli potencjalnego ducha), jeżeli wartość współczynnika S_{O_k} przekracza ustalony próg S_{TH} (domyślnie równy 0.1). Takie działanie ma na celu eliminację błędnej klasyfikacji wynikającej z szumów i zakłóceń obrazu.

W celu weryfikacji, czy dany obiekt istnieje zarówno na obrazie wejściowym jak i masce wyjściowej, użyty został mechanizm porównywania krawędzi. Sama detekcja krawędzi została zrealizowana z wykorzystaniem filtra Sobela. Przykładowe maski dla osi X i Y przedstawiono w równaniu (3.25). Operacja konwolucji z maską Sobela realizowana jest osobno dla każdego kanału RGB , następnie obrazy wynikowe dla obu osi są sumowane i binaryzowane ze stałym progiem T_S . Maska finalna stanowi alternatywę logiczną wyników z wszystkich kanałów.

$$S_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \quad S_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (3.25)$$

Po wyznaczeniu krawędzi obrazu należy dla każdego obiektu obliczyć tzw. współczynnik podobieństwa krawędzi EC_{O_k} , został on przedstawiony za pomocą równania (3.26). Jeżeli otrzymana wartość przekracza próg 0.5 oznacza to, że obiekt istnieje, czyli występuje zarówno na obrazie wejściowym jak i masce wyjściowej.

$$EC_{O_k} = \frac{\sum_{x_i \in O_k} F_E(x_i) == 1 \wedge I_E(x_i) == 1}{\sum_{x_i \in O_k} F_E(x_i) == 1} \quad (3.26)$$

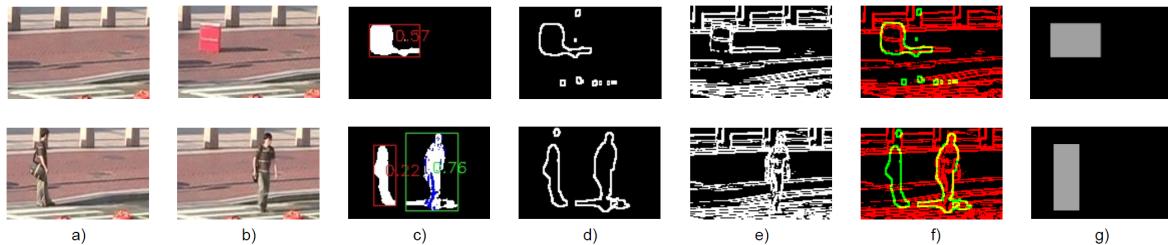
gdzie:

F_E – krawędzie na masce wyjściowej

I_E – maska zawierająca krawędzie na obrazie wejściowym

O_k – k -ty obiekt zidentyfikowany na masce wyjściowej

Opisany proces został przedstawiony na rysunku 3.2, pokazane zostały dwa scenariusze, w górnym rzędzie obiekt zatrzymany, natomiast w dolnym tzw. „duch”. Rysunek *a* przedstawia ramkę użytą do inicjalizacji modelu tła, *b* – aktualną ramkę, *c* – efekt końcowy analizy, *d* – krawędzie na masce pierwszoplanowej, *e* – krawędzie na aktualnej masce, *f* – połączone obrazy krawędzi, *g* – znalezione obiekty.



Rys. 3.2. Przykład detekcji tzw. „duchów”

Schemat blokowy rozszerzonej wersji metody *PBAS* został przedstawiony na rysunku 4.15. Oprócz modułu obliczającego maskę pierwszoplanową z użyciem standardowego algorytmu *PBAS* zawiera on dodatkowe bloki odpowiedzialne za wszystkie operacje opisane w niniejszym rozdziale. Moduł *EDGE* służy do wyznaczania krawędzi na aktualnym obrazie wejściowym (oznaczonym jako I_N). Blok *CFD* (ang. *consecutive frame differencing*) oblicza różnicę dwóch ramek kolejnych ramek (I_N i I_{N-1}) opisaną równaniem (3.23). Najbardziej złożonym elementem jest moduł *CCA* (ang. *Connected Component Analysis*). Jest w nim wykonywana indeksacja obiektów (opisana szczegółowo w rozdziale 3.3.3). Wartością końcową tej operacji są parametry prostokąta otaczającego zidentyfikowany obiekt (ang. *bounding box*). Oprócz tego są wyznaczane wartości S_{O_k} i EC_{O_k} – równania (3.24) i (3.26) oraz finalna maska wyjściowa. Obliczone współczynniki stabilności i podobieństwa krawędzi są przekazywane jako sprzężenie zwrotne do standardowego algorytmu *PBAS*.

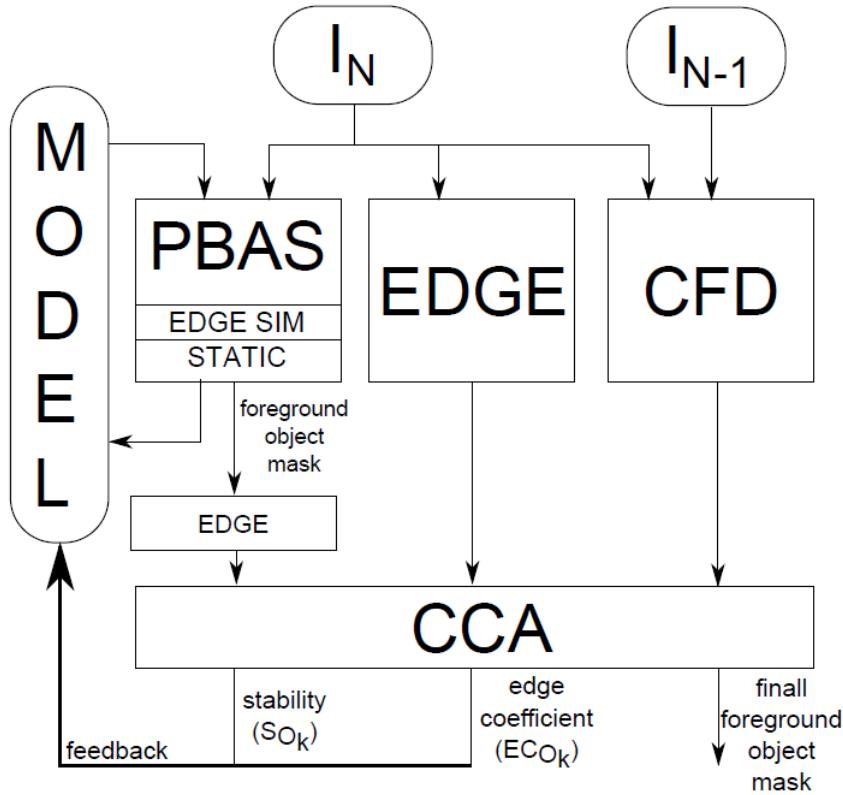
Ostatnim krokiem jest obsługa sprzężenia zwrotnego, w tym celu do modelu tła zostały dodane dwa kolejne elementy. Pierwszym z nich jest licznik $S(x_i)_{cnt}$ określający przez ile ramek dany piksel był rozpoznawany jako obiekt statyczny. Sposób jego aktualizacji został opisany równaniem (3.27).

$$S(x_i)_{cnt} = \begin{cases} S(x_i)_{cnt} + 1 & \text{jeżeli } S_{O_k} \geq S_{TH} \\ 0 & \text{jeżeli } S_{O_k} < S_{TH} \\ S(x_i)_{cnt} - 1 & \text{w pozostałych przypadkach} \end{cases} \quad (3.27)$$

Operacja ta jest przeprowadzana dla każdego piksela wewnętrz prostokąta otaczającego dany obiekt. Licznik jest inkrementowany dla obiektów zaklasyfikowanych jako statyczne, dla poruszających się zostaje wyzerowany, natomiast dla elementów tła jest dekrementowany.

Drugim dodatkowym elementem modelu jest średnia krocząca współczynnika podobieństwa krawędzi $EC(x_i)_{mean}$, jej wartość jest aktualizowana zgodnie z równaniem (3.28).

$$EC(x_i)_{mean} = \begin{cases} 0.5EC(x_i) + 0.5EC(x_i)_{mean} & \text{jeżeli } S_{O_k} \geq S_{TH} \wedge S(x_i)_{cnt} > S_{cnt_{TH}} \\ 1 & \text{w przeciwnym razie} \end{cases} \quad (3.28)$$



Rys. 3.3. Schemat blokowy rozszerzonej metody PBAS – źródło [21]

gdzie poprzez $S_{cnt_{TH}}$ oznaczamy minimalną liczbę ramek przez którą obiekt musi pozostać statyczny. Opisane parametry są wykorzystywane w procesie aktualizacji modelu, niezależnie od decyzji podjętej według standardowej procedury, model zostaje zaktualizowany jeżeli spełniony jest warunek logiczny opisany równaniem (3.29). W praktyce oznacza to, że aktualizowane zostają piksele potencjalnie reprezentujące duchy, czyli obiekty, które nie znajdują się na rzeczywistym obrazie.

$$S_{O_k} \geq S_{TH} \wedge S(x_i)_{cnt} > S_{cnt_{TH}} \wedge EC(x_i)_{mean} < 0.5 \quad (3.29)$$

3.3.3. Indeksacja obiektów

Opisany w rozdziale 3.3.2 mechanizm detekcji tzw. „duchów” wymaga rozróżnienia poszczególnych obiektów na wynikowej masce binarnej. W celu osiągnięcia takiego rezultatu, konieczne jest zastosowanie algorytmu indeksacji. Istnieje kilka podejść do realizacji tego zadania. W niniejszej pracy zdecydowano się zastosować metodę jednoprzebiegową, ze względu na jej stosunkowo prostą implementację w układach *FPGA*. Sam algorytm został dokładnie przedstawiony w publikacji [21], natomiast problem realizacji na platformie sprzętowej został szerzej rozwinięty w rozdziale 4.8.

Działanie algorytmu polega na nadawaniu poszczególnym pikselom etykiet reprezentujących kolejne obiekty. Piksele z taką samą etykietą reprezentują ten sam obiekt. Oczekiwany efektem końcowym, wymaganym w omawianym algorytmie jest pole oraz parametry prostokąta otaczającego każdy obiekt

w danej ramce obrazu. Prostokąt otaczający jest reprezentowany przez cztery liczby $[x_1, y_1, x_2, y_2]$, gdzie współrzędne z indeksem 1 oznaczają lewy górny róg prostokąta, natomiast te z indeksem 2 prawy dolny róg.

Pierwszym krokiem w procesie indeksacji jest wygenerowanie sąsiedztwa na podstawie którego aktualnie analizowany piksel będzie klasyfikowany. Jako sąsiedztwo, rozumie się najbliższe otaczające piksele. Ze względu na potokowe przetwarzanie obrazu wystarczy rozpatrzyć trzy piksele, znajdujące się w wyższym rzędzie oraz sąsiada po lewej stronie w rzędzie aktualnie analizowanym. Po wyznaczeniu sąsiedztwa, można przystąpić do klasyfikacji piksela. Jeżeli piksel ma wartość 1 (jest elementem pierwszego planu) należy mu przypisać etykietę.

Podczas operacji przypisania etykiety kolejnemu pikselowi mogą wystąpić trzy przypadki. Najprostszym z nich jest wtedy, gdy żaden z sąsiadujących pikseli nie ma przypisanej żadnej etykiety. W takiej sytuacji aktualny piksel traktowany jest jako fragment nowego obiektu i dostaje nową etykietę. Drugi przypadek to taki, w którym co najmniej jeden piksel posiada już przypisaną etykietą. Jeżeli pośród sąsiednich pikseli pojawia się tylko jedna etykieta oznacza to, że aktualny piksel także jest fragmentem obiektu oznaczonego właśnie tą etykietą, więc zostaje mu ona przypisana. Ostatnim przypadkiem jest tzw. konflikt. Występuje on wtedy, gdy pośród sąsiednich pikseli występują różne etykiety. Taka sytuacja może się przytrafić na przykład w sytuacji gdy analizowany jest obiekt w kształcie litery V. Warto zauważyc, że w obszarze tego samego sąsiedztwa mogą istnieć tylko dwie różne etykiety. W przypadku wystąpienia kolizji oba obiekty łączone są w jeden.

Oprócz nadawania etykiet dla poszczególnych pikseli, na bieżąco aktualizowane jest także pole oraz parametry prostokąta otaczającego każdy do tej pory zidentyfikowany obiekt. W przypadku nowego obiektu, pole inicjalizowane jest oczywiście wartością 1 natomiast współrzędne wierzchołków prostokąta otaczającego przyjmują współrzędne piksela. Dalsza aktualizacja pola obiektu wymaga jedynie inkrementacji wartości wraz z kolejnymi pikselami przypisanymi do danego obiektu. Prostokąt otaczający jest natomiast aktualizowany na podstawie współrzędnych x i y aktualnego piksela, zgodnie z równaniem (3.30).

$$(x_1, y_1, x_2, y_2) = (\min(x_1, x), \min(y_1, y), \max(x_1, x), \max(y_1, y)) \quad (3.30)$$

W przypadku wystąpienia konfliktu, pola obu obiektów zostają zsumowane. Parametry nowego prostokąta otaczającego dobierane są tak, aby pokryły ona oba wcześniejsze prostokąty. Spośród współrzędnych lewych górnych rogów obu łączonych obiektów, wybierane są wartości mniejsze i zostają ustawione jako współrzędne lewego rogu nowego prostokąta. W przypadku prawego dolnego rogu postępowanie jest analogiczne, tylko wybierane są wartości większe.

3.3.4. Uwagi

Przedstawiony algorytm jest rozszerzoną wersją omówionej w rozdziale 3.2 metody ViBE. W tym przypadku wykorzystywany jest bardziej rozbudowany model tła, a sam algorytm charakteryzuje się

większą złożonością obliczeniową. W stosunku do wcześniej omawianej metody, zwiększyła się także lista parametrów, kompletna lista została zamieszczona poniżej (w nawiasach umieszczone wartości domyślne):

N – liczba próbek w modelu tła (19)

$\#_{min}$ – minimalna liczba próbek wymagana w teście dopasowania (2)

R_{init} – początkowa wartość progu dopasowania próbki do modelu (30)

T_{init} – początkowe prawdopodobieństwo wykonania aktualizacji ($\frac{1}{16}$)

R_{low} – dolne ograniczenie progu dopasowania próbki do modelu (18)

T_{low}, T_{up} – dolne i górne ograniczenie progu dopasowania próbki do modelu (2 i 200)

$R_{inc/dec}$ – współczynnik wykorzystywany przy aktualizacji parametru R (0,05)

R_{SC} – współczynnik skalowania wykorzystywany przy aktualizacji parametru R (5)

W rozszerzonej wersji, zawierającej mechanizm detekcji duchów, należy uwzględnić jeszcze następujące parametry:

θ – próg wykorzystywany w (3.24), określający ruchome obiekty (50)

S_{TH} – wartość progowa współczynnika stabilności (0,01)

$S_{cnt_{TH}}$ – minimalna liczba ramek przez którą obiekt musi pozostać statyczny (5)

T_S – próg binaryzacji wykorzystywany w detekcji krawędzi metodą Sobela (50)

Kompletny model tła dla pojedynczego piksela składa się z N zapamiętanych próbek (3.15), N -elementowego zbioru minimalnych odległości pomiędzy próbką a pikselem (3.19) oraz parametrów R i T . W przypadku rozszerzonej wersji algorytmu należy jeszcze uwzględnić dodatkowy licznik $S(x_i)_{cnt}$ (3.27), współczynnik podobieństwa krawędzi $EC(x_i)_{mean}$ (3.28) oraz poprzednią wartość piksela konieczną do obliczenia różnicy dwóch kolejnych ramek.

3.4. Gaussian Mixture Models

Metoda *GMM* (ang. *Gaussian Mixture Models*) jest algorymem wykorzystującym statystyczny model tła. Autorzy niektórych publikacji wykorzystują również alternatywną nazwę *MOG* (ang. *Mixture of Gaussian*). W przeciwieństwie do algorytmów opisywanych w poprzednich rozdziałach, tutaj model tła nie składa się z zapamiętanych próbek. Podobnie jak inne metody, ta także może przetwarzać obraz w różnych przestrzeniach barw.

Jako, że jest to algorytm opublikowany po raz pierwszy w 1999 roku [29], to od tamtej pory doczekał się wielu różnych odmian i udoskonaleń. Ponieważ tematem niniejszej pracy jest analiza i unifikacja algorytmów opracowanych w Laboratorium Biocybernetyki AGH w tym rozdziale zostanie zamieszczony opis powstały w oparciu o prace inżynierskie [13, 27]. W wyżej wymienionych publikacjach zamieszczono wyczerpujący opis metody, zarówno od strony teoretycznej jak i implementacyjnej. Niniejsza praca zawiera jedynie podstawowe założenia i główne kroki algorytmu.

3.4.1. Opis Algorytmu

Opisywany algorytm opiera się na modelu tła składającym się z K rozkładów Gaussa. Są one podzielone na dwie grupy – część z nich reprezentuje tło, reszta obiekty pierwszoplanowe. Model skonstruowany jest tak, że dla każdego piksela zdefiniowany jest osobny zestaw rozkładów Gaussa, który następnie jest niezależnie aktualizowany. Niewątpliwą zaletą w kontekście implementacji w układzie *FPGA* jest brak jakichkolwiek operacji wykorzystujących otoczenie piksela. Pozwala to zaoszczędzić dużą część zasobów. Szczegóły implementacji sprzętowej przedstawiono w rozdziale 4.9.

Każdy rozkład Gaussa składa się z trzech elementów: wagi (ω), wariancji (σ) i wartości średniej (μ). W przypadku przetwarzania obrazu w przestrzeni *RGB*, w celu uproszczenie obliczeń, zakłada się, że waga i wariancja jest identyczna dla wszystkich trzech kanałów. Opierając się na publikacjach [29, 27], liczbę rozkładów (parametr K) najlepiej przyjąć z zakresu 3 – 5.

Pierwszym krokiem jest posortowanie rozkładów Gaussa według współczynnika $r_i = \frac{\omega_i}{\sigma_i}$ w kolejności rosnącej (przez i rozumiemy numer rozkładu, natomiast $\omega_{i,t}$ oznacza wagę i -tego rozkładu w czasie t). Ze względu na przyjęte uproszczenie, że wariancja jest identyczna dla wszystkich kanałów RGB, macierz kowariancji i -tego rozkładu w chwili t może zostać zapisana równaniem (3.31).

$$\Sigma_{i,t} = \sigma_{i,t}^2 I \quad (3.31)$$

Zastosowanie sortowania według współczynnika r_i powoduje umieszczenie na początku listy tych rozkładów, które najprawdopodobniej reprezentują tło (czyli tych o najwyższej wagie oraz najniższej wariancji). Przyjęto, że pierwsze B rozkładów z listy uznaje się za rozkładu reprezentujące tło, kolejne to już obiekty pierwszoplanowe. Jak łatwo zauważać, rozkłady przedstawiające tło, mają zdecydowanie wyższą wagę oraz niewielką wariancję. Równanie (3.32) określa wartość B , jest to liczba rozkładów, dla których suma wag przekracza próg T .

$$B = \arg \min_b \left(\sum_{i=1}^b \omega_{i,t} > T \right) \quad (3.32)$$

Kolejnym etapem to test dopasowania nowego piksela do modelu. Przeprowadzany jest on dla każdego rozkładu Gaussa z posortowanej listy i przerywany w momencie uzyskania pozytywnego wyniku (piksel wejściowy pasuje do modelu). Jeżeli przyjmiemy, że nowy piksel pojawia się w chwili $t + 1$ to test dopasowania przeprowadzany jest z modelem z chwili t . Pozytywny wynik testu określa zależność (3.33).

$$\sqrt{\left((X_{t+1} - \mu_{i,t})^T \cdot \Sigma_{i,t}^{-1} \cdot (X_{t+1} - \mu_{i,t}) \right)} < k\sigma_{i,t} \quad (3.33)$$

gdzie próg k domyślnie przyjmuje wartość 2,5

Na tym etapie algorytmu, należy rozważyć dwa przypadki. Jeżeli piksel wejściowy został dopasowany do jednego z K rozkładów Gaussa, to zostaje on klasyfikowany na podstawie wcześniejszej przynależności danego rozkładu, zgodnie z (3.32). Podczas aktualizacji modelu, zostanie wykorzystany

parametr α określający stałą uczenia. Parametry rozkładu Gaussa, który przeszedł test dopasowania określony równaniem (3.33), zostaje zaktualizowany według zależności (3.34), (3.35) i (3.36).

$$\omega_{i,t+1} = (1 - \alpha)\omega_{i,t} + \alpha \quad (3.34)$$

$$\mu_{i,t+1} = (1 - \rho)\mu_{i,t} + \rho X_{t+1} \quad (3.35)$$

$$\sigma_{i,t+1}^2 = (1 - \rho)\sigma_{i,t}^2 + \rho(X_{t+1} - \mu_i, t + 1)(X_{t+1} - \mu_i, t + 1)^T \quad (3.36)$$

Do przedstawienia funkcji ρ konieczna jest także definicja funkcji gęstości prawdopodobieństwa, która jest opisana równaniem (3.37). Z kolei samą funkcję ρ przedstawia równanie (3.38).

$$\eta(X_t, \mu, \Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} e^{-\frac{1}{2}(X_t - \mu)\Sigma^{-1}(X_t - \mu)} \quad (3.37)$$

gdzie:

X_t – wartość piksela w chwili t

μ – wektor wartości średnich rozkładu Gaussa

Σ – macierz kowariancji – wzór (3.31)

$$\rho = \alpha\eta(X_{t+1}, \mu_i, \Sigma_i) \quad (3.38)$$

Reszta rozkładów, która nie przeszła testu dopasowania, zostaje zaktualizowana według równania (3.39), nadpisywana jest tylko waga, wartość średnia i wariancja pozostają niezmienione.

$$\omega_{j,t+1} = (1 - \alpha)\omega_{j,t} \quad (3.39)$$

Drugim przypadek to piksel niepasujący do żadnego rozkładu. W takiej sytuacji jest on automatycznie klasyfikowany jako element pierwszoplanowy. Aktualizacja modelu w tym przypadku polega na zastąpieniu rozkładu o najwyższej wadze nowym. Sama waga pozostaje niezmieniona, natomiast aktualna wartość piksela zapisywana jest jako wartość średnia. Przypisywana jest także duża wariancja inicjalizująca, która jest określona globalnie jako parametr całego algorytmu.

Warto jeszcze raz nadmienić, że powyższe operacje wykonywane są cyklicznie dla każdego piksela z wykorzystaniem niezależnego modelu tła. Lista wszystkich parametrów algorytmu *GMM* przedstawia się następująco:

K – liczba rozkładów Gaussa

T – próg z przedziału 0–1 pozwalający rozgraniczyć rozkłady tła i pierwszoplanowe

k – współczynnik używany w teście dopasowania (3.33)

α – współczynnik uczenia z przedziału 0–1

σ_{init} – wariancja, którą inicjalizowany jest nowy gaussian

3.4.2. Uwagi

Algorytm opisany w niniejszym rozdziale, przedstawia nieco odmienne podejście do segmentacji obiektów pierwszoplanowych w stosunku do metod opisanych wcześniej. Jak już zostało wspomniane w rozdziale 2 algorytm *GMM* jest bardzo rozległym zagadnieniem, będącym przedmiotem badań wielu publikacji [29, 6, 27, 31]. Ze względu na skomplikowane operacje matematyczne z punktu widzenia układów *FPGA*, sama implementacja również jest dość złożonym problemem.

Jak słusznie zauważali autorzy przytoczonych publikacji, algorytm *GMM* zapewnia zadowalające rezultaty, aczkolwiek jest bardzo podatny na różnego rodzaju zakłócenia, szумy i zmiany oświetlenia. Mimo to w większość przypadków, jest to metoda, mogąca z powodzeniem zostać wykorzystana jako samodzielny system segmentacji tła.

3.5. Algorytm Flux Tensor with Split Gaussian Models

Metoda *FTSG*, jak zostało wspomniane w rozdziale 2.3, po raz pierwszy opisano w publikacji [31]. Jej niepełną wersję zaimplementowano także w układzie *FPGA* w ramach pracy inżynierskiej [13]. Jest to algorytm hybrydowy, łączący metodę *Flux Tensor* (opisaną w podrozdziale 3.5.1) oraz zmodyfikowaną metodę *GMM*, nazwaną *Split Gaussian Models* (podrozdział 3.5.2). Zamieszczony w niniejszym rozdziale opis powstał w oparciu o wspomnianą pracę dyplomową, również zrealizowaną w Laboratorium Biocybernetyki AGH.

Oba wykorzystane algorytmy zawierają pewne wady, jednak w teorii powinny doskonale się uzupełnić, co skłoniło autorów publikacji do ich jednoczesnego wykorzystania. Metoda *Flux Tensor* umożliwia jedynie wykrywanie obiektów ruchomych, dodatkowo jednak zapewnia bardzo dobrą eliminację zakłóceń spowodowanych zmianami oświetlenia. Przeciwieństwem takiego podejścia jest druga wykorzystana metoda, która będąc bardziej podatną na zakłócenia, umożliwia również detekcję obiektów statycznych.

3.5.1. Algorytm Flux Tensor

Metoda *Flux Tensor* jest uniwersalnym algorytmem, służącym do wykrywania ruchomych obiektów na obrazie ze statycznie umiejscowionej kamery. Oprócz zastosowania w algorytmie *FTSG*, metoda ta oraz jej implementacja sprzętowa zostały opisane między innymi w publikacji [14]. Idea tego podejścia opiera się na detekcji krawędzi na obrazie wejściowym oraz wyznaczaniu pochodnej tego obrazu po czasie. Następnie, na podstawie tych dwóch informacji dokonywana jest identyfikacja obiektów ruchomych.

W opisywanym wariantie algorytm operuje jedynie na obrazie w skali szarości. Przyjmijmy następujące oznaczenia: x, y – współrzędne piksela na obrazie, t – czas, wektor w przestrzeni (x, y, t) zapiszmy jako v , natomiast wartość piksela wejściowego jako $I(v)$. Kolejnym krokiem jest zdefiniowanie macierzy *Flux Tensor*, opisującej zmiany wartości piksela wejściowego w lokalnej czasoprzestrzeni. Oznaczmy przez Ω otoczenie w przestrzeni (x, y, t) , które jest rozważane przy analizie danego piksela. Macierz *Flux Tensor* została opisana równaniem (3.40).

$$J_F(i) = \begin{pmatrix} \int_{\Omega} \left(\frac{\partial^2 I(v)}{\partial x \partial t} \right)^2 dv & \int_{\Omega} \frac{\partial^2 I(v)}{\partial x \partial t} \frac{\partial^2 I(v)}{\partial y \partial t} dv & \int_{\Omega} \frac{\partial^2 I(v)}{\partial x \partial t} \frac{\partial^2 I(v)}{\partial^2 t} dv \\ \int_{\Omega} \frac{\partial^2 I(v)}{\partial y \partial t} \frac{\partial^2 I(v)}{\partial x \partial t} dv & \int_{\Omega} \left(\frac{\partial^2 I(v)}{\partial y \partial t} \right)^2 dv & \int_{\Omega} \frac{\partial^2 I(v)}{\partial y \partial t} \frac{\partial^2 I(v)}{\partial^2 t} dv \\ \int_{\Omega} \frac{\partial^2 I(v)}{\partial^2 t} \frac{\partial^2 I(v)}{\partial x \partial t} dv & \int_{\Omega} \frac{\partial^2 I(v)}{\partial^2 t} \frac{\partial^2 I(v)}{\partial y \partial t} dv & \int_{\Omega} \left(\frac{\partial^2 I(v)}{\partial^2 t} \right)^2 dv \end{pmatrix} \quad (3.40)$$

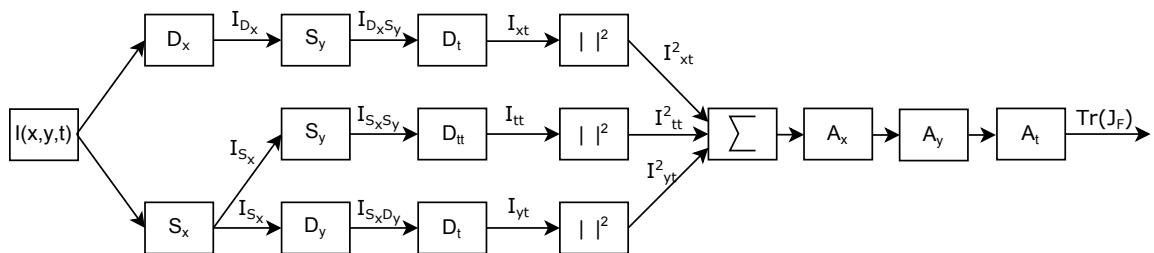
Następnym etapem jest wyznaczenie śladu (sumy elementów na przekątnej) opisanej macierzy. Na tej podstawie dokonywana jest ostateczna klasyfikacja piksela. Jeżeli otrzymana wartość jest większa od wartości progowej T , to piksel zostaje uznany za ruchomy. W przeciwnym wypadku zakładamy, że jest to element statyczny. Przyjmijmy uproszczone oznaczenia:

$$I_{xt} = \frac{\partial^2 I(x, y, t)}{\partial x \partial t}, \quad I_{yt} = \frac{\partial^2 I(x, y, t)}{\partial y \partial t}, \quad I_{tt} = \frac{\partial^2 I(x, y, t)}{\partial^2 t} \quad (3.41)$$

Wykorzystując zależności opisane w (3.41), ślad macierzy można zapisać równaniem:

$$\text{trace}(J_F) = \int_{\Omega} (I_{xt}^2(i) + I_{yt}^2(i) + I_{tt}^2(i)) dv \quad (3.42)$$

Wszystkie elementy konieczne do wyliczenia wartości danej równaniem (3.42) są wyznaczane przy użyciu operacji konwolucji. Do obliczania składowej I_{xt} wykorzystuje się filtry różniczkujące dla składowych x i t oraz filtr wygładzający dla składowej y . Analogicznie, w przypadku I_{yt} używany jest filtr wygładzający dla składowej x . Ostatnim elementem jest składowa I_{tt} . Tutaj wykorzystuje się filtr wygładzający dla składowych x i y . W operacji całkowania wykorzystuje się natomiast jednowymiarowe filtry uśredniające. Schemat ilustrujący wszystkie etapy został przedstawiony na rysunku 3.4.



Rys. 3.4. Schemat operacji wyznaczających ślad macierzy J_F – źródło [13]

Przedstawione na rysunku 3.4 operacje oznaczają odpowiednio:

D_x, D_y – filtry różniczkujące odpowiednio dla składowych x i y

S_x, S_y – filtry wygładzające dla składowych x i y

Dt, Dtt – odpowiednio pierwsza i druga pochodna z wartości piksela po czasie

Ax, Ay – filtry uśredniające przestrzenne

At – filtr uśredniający czasowy

Wygładzanie obrazu został zrealizowane poprzez filtr Gaussa ($\sigma = 3$). Różniczkę, w zależności od dobranej szerokości filtra, wyznacza się z wykorzystaniem następujących masek:

$$n = 3 \quad [-\frac{1}{2}, 0, \frac{1}{2}]$$

$$n = 5 \quad [\frac{1}{12}, -\frac{2}{3}, 0, \frac{2}{3}, -\frac{1}{12}]$$

$$n = 7 \quad [-\frac{1}{60}, \frac{3}{20}, -\frac{3}{4}, 0, \frac{3}{4}, -\frac{3}{20}, \frac{1}{60}]$$

Po dokonaniu pewnych uproszczeń i założenia, że maski dla osi x i y mają identyczne rozmiary, można przedstawić finalną listę parametrów algorytmu:

T – wartość progowa

nDs – rozmiar filtrów przestrzennych (różniczkujących i wygładzających)

nDt – liczba ramek używana do różniczkowania po czasie

nAs – rozmiar masek uśredniających przestrzennych

nAt – liczba ramek używana do uśredniania po czasie

3.5.2. Algorytm Split Gaussian Models

Metoda *Split Gaussian Models* opisana w [13] to zmodyfikowana wersja algorytmu *GMM*. Zamiast jednego zestawu rozkładów Gaussa, składającego się z K elementów, zdecydowano się wykorzystać dwa osobne modele – jeden reprezentujący tło i drugi obiekty pierwszoplanowe. Pierwszy z nich składa się ze zmiennej liczby rozkładów, natomiast drugi zawiera tylko jeden rozkład Gaussa. Test dopasowania do modelu tła uznawany jest za pozytywny, jeżeli piksel pasuje do chociaż jednego z rozkładów znajdujących się w modelu. W przypadku modelu pierwszoplanowego, piksel uznawany jest za element pierwszego planu jeżeli pasuje do rozkładu, w przeciwnym razie jest sklasyfikowany jako element tła. Sam test dopasowania przeprowadzany jest identycznie jak w przypadku standardowej metody *GMM*, przy użyciu wzoru (3.33).

Oprócz definicji rozkładów Gaussa zmieniony został również proces aktualizacji modeli. Jest ona wykonywana tylko wtedy, gdy piksel został sklasyfikowany jako tło. Analogicznie warunkiem aktualizacji modelu pierwszego planu jest sklasyfikowanie piksela jako pierwszoplanowego. Ostateczna decyzja o klasyfikacji piksela podejmowana jest po dokonaniu fuzji obu algorytmów. Procedura ta została

szerzej opisana w rozdziale 3.6. Dodatkowo wprowadzono także próg T_l , rozkłady których waga osiągnie wartość niższą od tego progu zostają usunięte.

Dwa równolegle działające modele dają także możliwość wykorzystania dwóch zestawów parametrów. W tym przypadku najistotniejsza jest stała uczenia α . W celu lepszej detekcji obiektów statycznych, jej wartość dla modelu tła powinna być jak najmniejsza. Odwrotna sytuacja występuje w przypadku modelu pierwszego planu, tutaj wskazana jest wyższa wartość, gdyż pozwala to częściowo zniwelować efekt duchów. Dodatkowo można także manipulować stałą k wykorzystywaną w teście dopasowania (3.33). Dla modelu tła przyjmuje się zazwyczaj $k = 3$, natomiast w modelu pierwszoplanowym, optymalna wartość to $k = 20$.

3.6. Fuzja metody Flux Tensor i Split Gaussian Models

Schemat przedstawiający sposób połączenia obu algorytmów został zaprezentowany na rysunku 3.5. Do analizy obrazu wykorzystywane są jednocześnie trzy modele: *Flux Tensor*, model tła ze zmienną liczbą rozkładów Gaussa oraz model pierwszoplanowy zawierający jeden rozkład Gaussa. W pierwszym etapie analizowany jest wynik jedynie z dwóch pierwszych modułów. Finalna maska binarna została zapisana równaniem (3.43).

$$F = \begin{cases} 1, & \text{gdy } F_F = 1 \wedge F_B = 1 \\ 0, & \text{gdy } F_B = 0 \\ F_S, & \text{gdy } F_F = 0 \wedge F_B = 1 \end{cases} \quad (3.43)$$

gdzie:

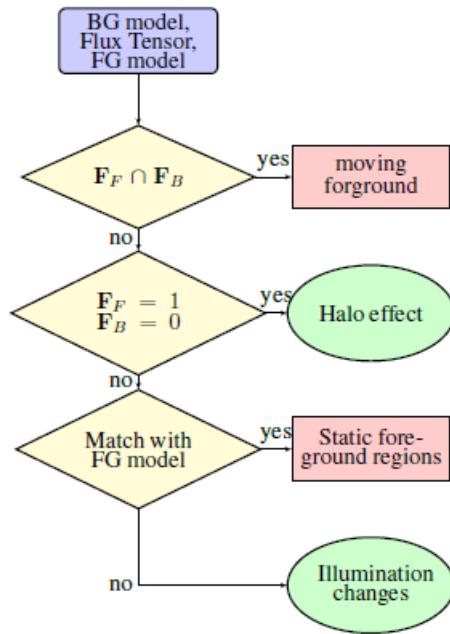
F_F – maska otrzymana z modelu *Flux Tensor*

F_B – maska otrzymana z modelu tła

F_S – maska definiująca obszar statyczny

W sytuacji, gdy wynik klasyfikacji dla obu metod jest identyczny, sprawa ostatecznej przynależności jest oczywista. Piksel zostaje uznany za obiekt jeżeli $F_F = 1 \wedge F_B = 1$ lub za statyczne tło ($F_F = 0 \wedge F_B = 0$). Jeżeli wyniki obu modeli są sprzeczne konieczna jest dodatkowa analiza. Prostszego przypadku występuje w sytuacji, gdy metoda *Flux Tensor* oznaczy piksel jako element pierwszego planu i jednocześnie pasuje on do modelu tła. Ostatecznie piksel klasyfikowany jest wtedy jako tło. Taka decyzja jest podejmowana, ponieważ algorytm *Flux Tensor* zawsze zaznacza obiekty o trochę większym obszarze niż są one w rzeczywistości (tzw. efekt halo). Z uwagi na fakt, że metoda wykorzystująca rozkłady Gaussa wykrywa zarówno obiekty statyczne jak i ruchome, nie występuje ryzyko utraty danych.

Bardziej skomplikowana jest sytuacja odwrotna, czyli $F_F = 0 \wedge F_B = 1$. Tutaj konieczne jest wykorzystanie modelu pierwszoplanowego i na jego podstawie podejmowana jest ostateczna decyzja. Jeżeli



Rys. 3.5. Schemat operacji algorytmu *FTSG* – źródło [31]

piksel pasuje do modelu to zostaje zaklasyfikowany jako statyczny obiekt pierwszoplanowy, w przeciwnym przypadku jako tło. Takie podejście pozwala zredukować znaczą liczbę szumów powstających na masce wyjściowej w skutek zmian oświetlenie i innych zakłóceń.

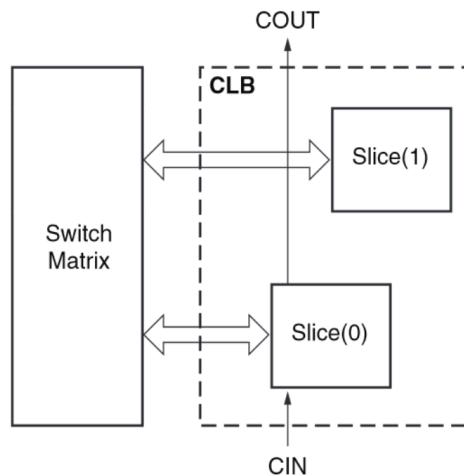
W oryginalnej publikacji przedstawiającej omawianą metodę [31] autorzy zaproponowali jeszcze dodatkowy mechanizm detekcji obiektów statycznych, bardzo podobny do tego zrealizowanego w algorytmie *PBAS* (rozdział 3.3.2). Implementacja sprzętowa opracowana w Laboratorium Biocybernetyki AGH w ramach pracy dyplomowej [13] zawiera pewne uproszczenia w stosunku do oryginału. Jedną z najistotniejszych modyfikacji było zrezygnowanie ze zmiennej liczby rozkładów Gaussa w modelu tła, zamiast tego wykorzystano podejście podobne do standardowego algorytmu *GMM*.

4. Implementacja sprzętowa wybranych algorytmów segmentacji obiektów pierwszoplanowych

4.1. Układy *FPGA* – wprowadzenie

W przypadku układów *FPGA*, do dobrego projektowania logiki programowalnej, konieczne jest poznanie podstaw ich budowy. W niniejszym rozdziale zostaną przedstawione podstawowe elementy wykorzystanego układu *FPGA*. Szczegółowy opis wykorzystanej platformy (seria *Virtex-7* firmy *Xilinx*) można znaleźć w dokumentacji producenta [12].

Podstawowym elementem, znajdującym się w każdym układzie *FPGA*, jest blok *CLB* (ang. *Configurable Logic Block* – konfigurowalny blok logiczny). Schemat jego budowy przedstawiono na rysunku 4.1. Składa się on z dwóch elementów *Slice*, które są połączone bezpośrednio z matrycą przełączników (ang. *Switch Matrix*). Przy operacjach arytmetycznych wykorzystywana jest szybka logika przeniesienia, na omawianym rysunku są to linie *CIN* i *COUT*.



Rys. 4.1. Schemat bloku *CLB* – źródło [12]

W wykorzystanym układzie z rodziną *Virtex 7* występują dwa typy bloków *Slice*: *SLICEL* oraz *SLICEC*. Standardowy *SLICEL* składa się z następujących elementów:

- czterech generatorów funkcyjnych, zrealizowanych jako *LUT* (ang. *Look-up Table*)
- przerzutników typu *D* (8 sztuk)

- multiplekserów wykorzystywanych do łączenia elementów *LUT*
- szybkiej logiki przeniesienia wykorzystywanej w operacjach arytmetycznych

Dodatkowo, bardziej rozbudowany typ *SLICEM* posiada jeszcze dwie dodatkowe funkcje. Pierwszą z nich jest możliwość przechowywania danych w rozproszonej pamięci *RAM* (ang. *Distributed RAM*). Istnieje możliwość konfiguracji zarówno rozmiaru jak i liczby portów, dostępne warianty to: 256×1 jednoportowa, 128×1 dwuportowa i 64×1 czteroportowa. Drugą dodatkową funkcją jest 32-bitowy rejestr przesuwny, który może zostać wykorzystany do tworzenia linii opóźniających.

Wspomniane bloki *LUT* posiadają 6 wejść i 2 niezależne wyjścia. Można zatem je skonfigurować jako 6-wejściową funkcję logiczną, dwie 5-wejściowe funkcje (posiadające wspólne wejście) lub dwie funkcje z 3 i 2 wejściami. Dodatkowo dzięki zastosowaniu multiplekserów, obecnych w każdym bloku *Slice*, istnieje możliwość implementacji funkcji 7 i 8-wejściowych poprzez połączenie bloków *LUT*.

Oprócz wymienionych wyżej, każdy układ *FPGA* posiada również wiele innych zasobów. Do najważniejszych z pewnością należy zaliczyć (szczegółowy opis można znaleźć w dokumentacji technicznej producenta [11]):

- *CMT* (ang. *Clock Management Tiles*) – blok służący do generacji różnych częstotliwości zegara oraz jego równomiernego rozpropagowania i tłumienia zakłóceń fazy.
- *Block RAM (BRAM)* – dwuportowa pamięć *RAM* o rozmiarze 36Kb, istnieje możliwość skonfigurowania jej jako kolejki *FIFO* (ang. *First In First Out* – pierwszy na wejściu, pierwszy na wyjściu).
- *DSP Slice* (ang. *Digital Signal Processing*) – moduły z mnożarką 25×18 oraz 48-bitowym akumulatorem.
- *Select I/O* – zasoby wejścia/wyjścia, wspierające wiele standardów, między innymi: *LVCMOS*, *LVTTL*, *HSTL*, *PCI*, *SSTL*, *LVDS*.
- *Low-Power Gigabit Transceivers* – moduły umożliwiające szybką transmisję szeregową, maksymalne prędkości przesyłu danych zależą od typu transceivera i wynoszą odpowiednio: 6,6 Gb/s (*GTP*), 12,5 Gb/s (*GTX*), 13,1 Gb/s (*GTH*) lub 28,05 Gb/s (*GTZ*).
- *Moduł PCI Express* – wspiera transmisję w standardzie *PCI Express 3.0* z przepustowością do 8 Gb/s.
- Sterowniki zewnętrznej pamięci *RAM* – wsparcie dla pamięci *DDR3*, obsługa transmisji do 1866 Mb/s.

4.2. Modele programowe

Przed przystąpieniem do implementacji sprzętowej, dla każdego algorytmu przygotowano model programowy. Został on zaimplementowany w języku *C++* z wykorzystaniem liczb zmienoprzecinkowych oraz biblioteki *OpenCV* [25]. Całość przygotowano w środowisku programistycznym *Microsoft*

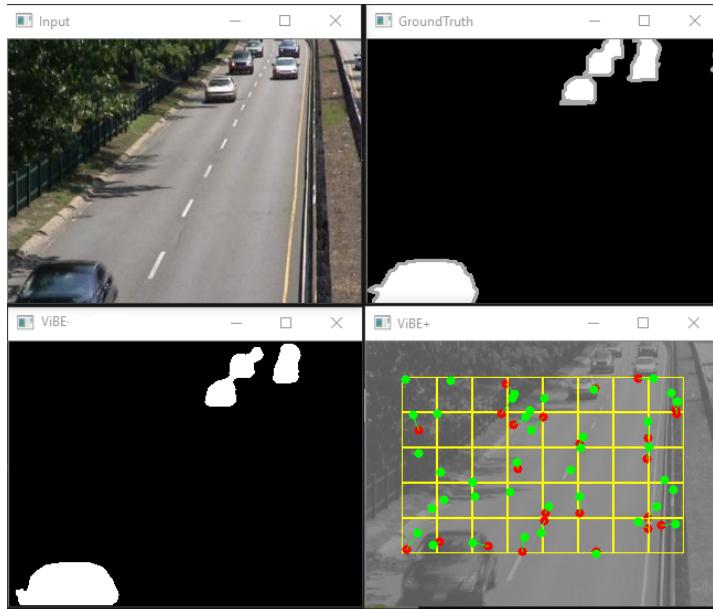
Visual Studio 2015. Głównym celem stworzonych implementacji było przeprowadzenie szczegółowych testów poszczególnych algorytmów. Tematyka ta została szczerzej opisana w rozdziale 5, natomiast architekturę oraz hierarchię zaimplementowanych funkcji i klas realizujących poszczególne algorytmy zaprezentowano w dodatku B. W niniejszej pracy wykorzystano częściowo kod źródłowy udostępniony przez autorów artykułów z których pochodzą omawiane metody. Poszczególne implementacje zostały, na potrzeby pracy, zmodyfikowane i zintegrowane w jeden projekt, aby było możliwe ich kompleksowe przetestowanie. Warto zaznaczyć, że we wszystkich przypadkach zastosowano obliczenia na liczbach zmiennoprzecinkowych.

Na rysunku 4.2 przedstawiono zrzut ekranu prezentujący działanie modelu programowego metody *PBAS*. Aplikacja wyświetla cztery okna, na których przedstawione są kolejno: obraz wejściowy w przestrzeni RGB, wzorcowa maska wyjściowa (ang. *ground truth*), efekt końcowy algorytmu *PBAS*, wizualizacja rozszerzonej wersji tej metody. Rysunek 4.3 przedstawia z kolei działający algorytm *ViBE*. Podobnie jak w poprzednim przypadku wyświetlane są cztery okna: wyjście, wzorzec, maska wyjściowa oraz wizualizacja przepływu optycznego.



Rys. 4.2. Model programowy metody *PBAS*

Wykonane implementacje oczywiście nie są w stanie zapewnić przetwarzania obrazu w czasie rzeczywistym. Rozdzielcość obrazu w wykorzystanych sekwencjach testowych zawiera się w przedziale od 320×240 do 720×576 pikseli. Nawet w przypadku tak niedużego obrazu otrzymana prędkość przetwarzania, w zależności od algorytmu, wahała się od kilku do co najwyżej kilkunastu klatek na sekundę. Wszystkie testy przeprowadzono na laptopie wyposażonym w procesor *Intel Core i7 3630QM@3,4GHz* i $8Gb$ pamięci RAM DDR3.



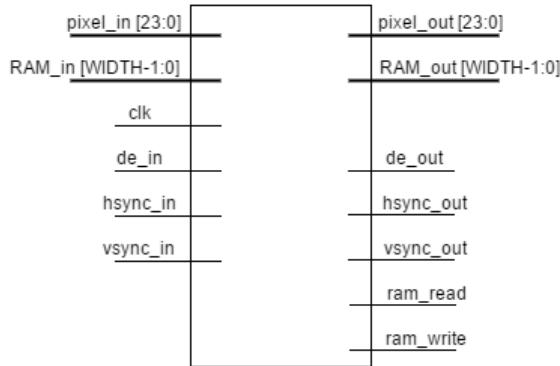
Rys. 4.3. Model programowy metody *ViBE*

4.3. Wykorzystany układ oraz interfejs wizyjny

Do implementacji algorytmów wykorzystano środowisko *Vivado*. Użyto najnowszej wersji, na moment tworzenia niniejszej pracy (czerwiec 2017), oznaczonej numerem *2017.1*. Weryfikację modelu programowego z implementacją sprzętową, przeprowadzono z wykorzystaniem, dostępnego w *Vivado*, środowiska symulacyjnego. Większość modułów zaimplementowano w języku *Verilog*. W pojedynczych przypadkach wykorzystano również język *VHDL*. Wszystkie metody uruchomiono i sprawdzono na karcie *VC707* z układem *Virtex-7 (XC7VX485T-2FFG176I)*. Platformy z rodziny *Virtex-7* należą do najwydajniejszych układów firmy *Xilinx*. Dodatkowo w celu podłączenia kamery i transmisji obrazu wykorzystano kartę *Avnet DVI I/O FMC*. Przyjęto założenie, że zaimplementowane algorytmy mają pracować w rozdzielcość *720x576* w 50 klatkach na sekundę. Opcjonalnie obsługiwana może być również rozdzielcość *1280x720* i *1920x1080*, również z prędkością *50 fps* (ang. *Frames Per Second* – klatki na sekundę). Ograniczeniem w przypadku wyższych rozdzielcości są głównie przepustowość pamięci *RAM* oraz wymagana częstotliwość zegara. Dla obrazu *576p@50fps* wymagana jest praca układu z częstotliwością *27MHz*. W przypadku rozdzielcości *720p@50fps* oraz *1080p@50fps* minimalna częstotliwość zegara musi wynosić już odpowiednio *74,25MHz* i *148,5MHz*. Ograniczona przepustowość pamięci *RAM* przekłada się natomiast bezpośrednio na maksymalny rozmiar modelu tła dla pojedynczego piksela.

W przypadku implementacji sprzętowej w układzie *FPGA* został wykorzystany uniwersalny interfejs wizyjny. Jego układ, przedstawiający podstawowe sygnały wejściowe i wyjściowe, został pokazany na rys. 4.4. Przedstawiony interfejs został wykorzystany jako szablon modułu segmentacji obiektów pierwszoplanowych i występuje we wszystkich zaimplementowanych algorytmach. Oprócz sygnałów wejściowych/wyjściowych z kamery, konieczne jest także zapewnienie niezależnego obszar pamięci *RAM* dla

każdego piksela. Maksymalnie do wykorzystania są 1024 bity na piksel w przypadku obrazu o rozdzielcości $720x576$ oraz 256 bitów dla wyższych rozdzielcości ($720p$ i $1080p$). Sygnały pochodzące z kamery zostały szerzej opisane w rozdziale 4.4.1. Należy zwrócić uwagę, że przedstawiony tutaj interfejs, zawiera jedynie niezbędne sygnały do funkcjonowania systemu. W przypadku niektórych algorytmów został on rozszerzony o dodatkowe wejścia/wyjścia.



Rys. 4.4. Szablon interfejsu segmentacji obiektów pierwszoplanowych

Moduł przedstawiony na rys. 4.4 posiada następujące sygnały wejściowe:

pixel_in – wartość piksela wejściowego w formacie RGB (24 bity)

bg_model_in – odczytana pamięć RAM o długości *WIDTH* (maksymalnie 1024 bity)

clk – zegar piksela

de_in – flaga poprawności piksela wejściowego

hs_in – flaga synchronizacji poziomej

vs_in – flaga synchronizacji pionowej

Natomiast sygnałami wyjściowymi są:

pixel_out – wartość piksela wyjściowego w formacie RGB (24 bity)

bg_model_out – blok o długości *WIDTH* do zapisania w pamięci RAM (maksymalnie 1024 bity)

de_out – opóźniona flaga poprawności piksela wejściowego

hs_out – opóźniona flaga synchronizacji poziomej

vs_out – opóźniona flaga synchronizacji pionowej

ram_read – sygnał odczytu pamięci RAM

ram_write – sygnał zapisu do pamięci RAM

Istotną kwestią jest również użycie zasobów oraz pobór energii przez układ *FPGA*. Jak łatwo zauważyć, sam interfejs wizyjny w połączeniu z kontrolerem pamięci *RAM* jest zagadnieniem dość rozbudowanym. W tabeli 4.1 zamieszczono wykorzystanie zasobów przez sam tor wizyjny (przepisanie sygnału wejściowego na wyjście), kontroler do zewnętrznej pamięci *RAM* oraz prosty algorytm odejmowania

ramek *FD* (ang. *Frame Difference*). Oszacowany pobór mocy takiej implementacji wynosi 2,804W. Pokazane dane dotyczą implementacji działającej w podstawowej rozdzielczości $576p@50fps$. W przypadku wyższych rozdzielczości, wykorzystywany jest mniejszy model tła, a co za tym idzie krótsze kolejki w kontrolerze pamięci. Ostatecznie, zużycie pamięci *BRAM* w takiej sytuacji jest o połowę niższe w stosunku do standardowej rozdzielczości. Odejmowanie ramek jest najprostszą metodą zamieszczoną w niniejszej pracy, jej szczegółowy opis znajduje się w rozdziale 3.1. Zaprezentowane tutaj zestawienie będzie stanowić punkt odniesienia dla bardziej zaawansowanych implementacji.

Tabela 4.1. Wykorzystanie zasobów przez podstawowe moduły (*Virtex 7*)

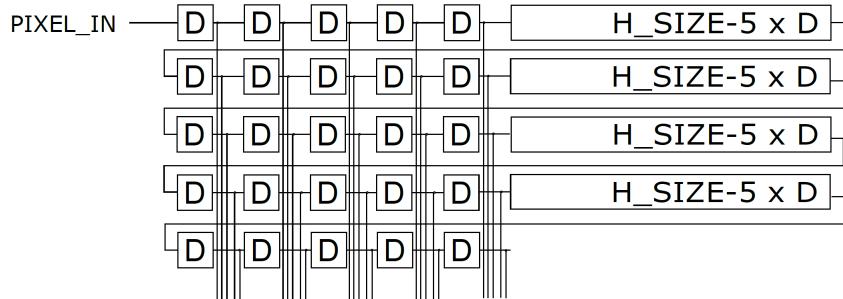
Zasoby	Wykorzystane				Dostępne	Użycie w %
	HDMI	RAM	FD	razem		
LUT	106	9372	36	9514	303600	3,13
LUTRAM	21	1397	0	1418	130800	1,08
FF	133	8906	24	9063	607200	1,49
BRAM	6	65	0	71	1030	6,89
DSP	0	0	0	0	2800	0,0

4.4. Trudności występujące w potokowym systemie wizyjnym

4.4.1. Kontekst poziomy i pionowy

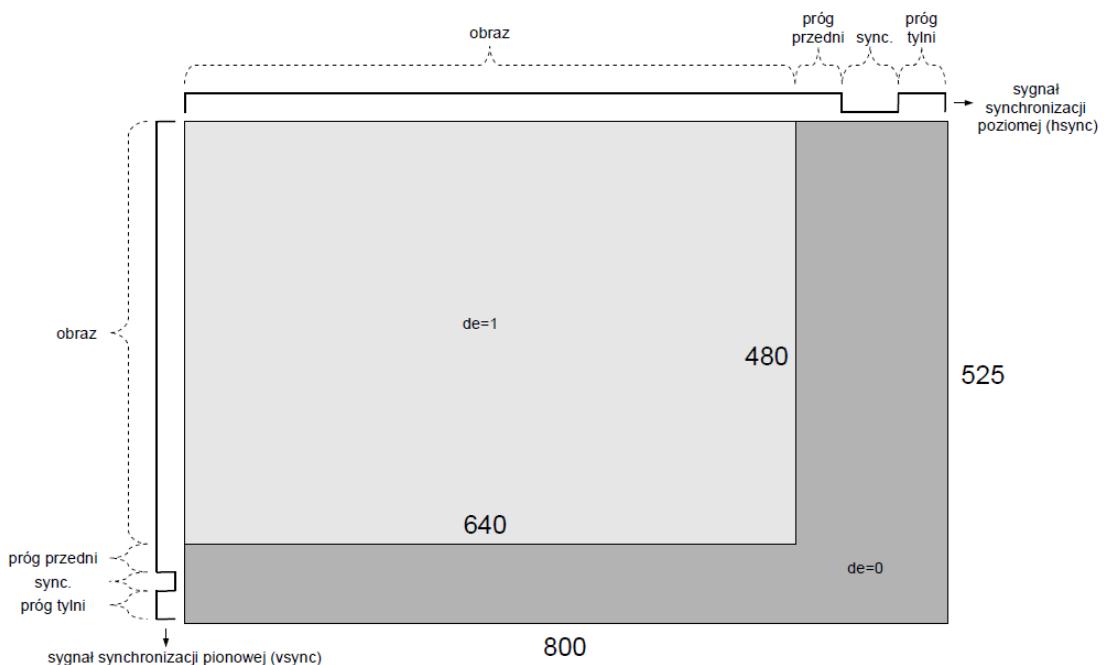
Wszystkie zrealizowane w niniejszej pracy algorytmy są realizowane w systemie potokowym. Oznacza to, że ramka obrazu przetwarzana jest piksel po pikselu. W związku z tym, w celu wykonania operacji kontekstowej konieczne jest zastosowanie odpowiednich opóźnień. Dla kontekstu poziomego nie jest to duży problem i może zostać rozwiązany z wykorzystaniem zwykłych rejestrów. Natomiast w przypadku kontekstu pionowego konieczne jest zastosowanie opóźnienia o całą linie obrazu. Do tego celu zwykle wykorzystuje się dostępną w układach FPGA pamięć blokową (*BRAM*). Przykładowy system linii opóźniających, służących do wyznaczenia kontekstu o rozmiarze $5x5$ został przedstawiony na rysunku 4.5. Poprzez pojedynczy blok **D** rozumiemy opóźnienie o jeden takt zegara, z kolei stała *H_SIZE* definiuje liczbę pikseli w jednej linii obrazu.

Podczas ustalania parametru *H_SIZE* należy zwrócić uwagę na tzw. obszar synchronizacji. W rzeczywistości liczba pikseli przesyłanych w jednej ramce obrazu jest większa niż sugerowałaby rozdzielcość obrazu. Przykładowo, dla obrazu o rozdzielczości $640x480$ pikseli, rzeczywista liczba pikseli, po uwzględnieniu obszaru synchronizacji wynosi $800x600$. Zostało to przedstawione na rysunku 4.6. Wielkość obszaru synchronizacji zależy zarówno od rozdzielczości obrazu jak i liczby klatek na sekundę. Przy wyznaczaniu odpowiedniej długości wymaganej linii opóźniającej pomocna może okazać się baza danych *Modeline Database* [24]. W niniejszej pracy wykorzystano trzy rozdzielczości: $720x576$,



Rys. 4.5. Schemat długiej linii opóźniającej – źródło [16]

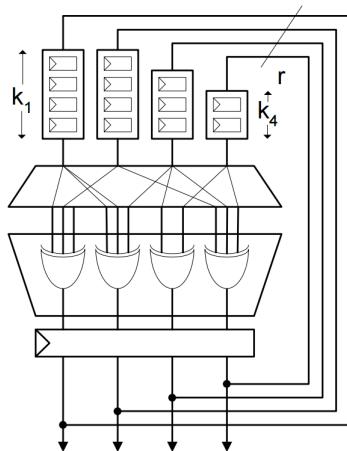
1280×720 , 1920×1080 przy prędkości 50fps . Całkowita długość linii poziomych, dla takich konfiguracji, wynosi odpowiednio: 864 , 1680 , 2640 .

Rys. 4.6. Synchronizacja dla obrazu o rozdzielcości 640×480 – źródło [16]

4.4.2. Generowanie liczb losowych

Moduł odpowiedzialny za generację liczb pseudolosowych został opracowany na podstawie publikacji [30]. Wykorzystano go także w wielu innych badaniach przeprowadzanych w Laboratorium Biocybernetyki AGH, między innymi [21, 22]. W przeciwieństwie do pozostałych modułów i algorytmów, generator został w całości zaimplementowany w języku VHDL. Wykorzystany kod źródłowy został udostępniony przez autorów wymienionych publikacji.

Zagadnienie generacji liczb pseudolosowych w układach reprogramowalnych jest tematem bardzo rozległym i zostało dokładnie opisane w przytoczonym artykule. W niniejszym rozdziale zostanie przedstawiona jedynie skrótowa idea działania użytego generatora. Cała koncepcja opiera się na wykorzystaniu rejestrów przesuwnych (ang. *shift register*) i modułów *LUT*. Uproszczony schemat tego typu generatora zamieszczono na rysunku 4.7. Wartości wyjściowe są zapisywane w rejestrach przesuwnych o różnych długościach. Następnie różne kombinacje zapamiętanych bitów są podawane na wejścia bramek *XOR*, które generują pseudolosowy sygnał wyjściowy.



Rys. 4.7. Schemat generatora liczb pseudolosowych – źródło [30]

Architekturę generatora można opisać za pomocą zestawu 4 parametrów (n, r, t, k) , gdzie:

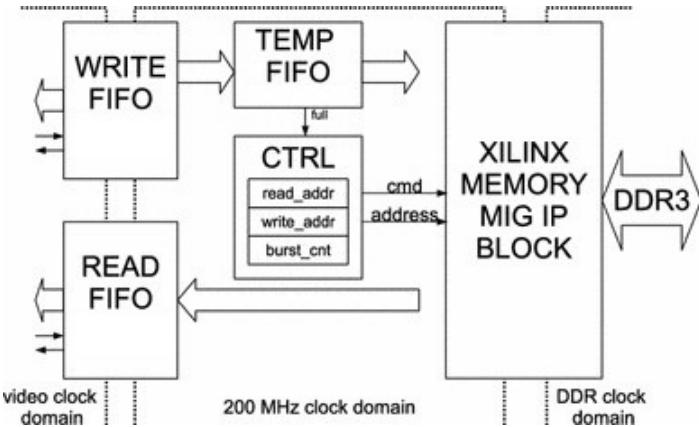
- n – liczba stanów generatora (okres możemy zapisać jako $2^n - 1$)
- r – liczba bitów generowanych w każdym cyklu
- t – liczba wejść każdej bramki *XOR*
- k – maksymalna długość rejestru przesuwnego

Wykorzystany w niniejszej pracy generator posiada następujące parametry: $n = 3900$, $r = 128$, $t = 5$, $k = 32$. Istnieje jeszcze jeden dodatkowy parametr s , który określa sposób połączeń wyjść rejestrów przesuwnych z bramkami *XOR*. Dokładny algorytm, opisujący sposób doboru optymalnych połączeń, jest bardzo zaawansowanym zagadnieniem, wybiegającym poza tematykę niniejszej pracy. Szczegółowy opis i analiza tego podejścia została przedstawiona w przytoczonej publikacji.

4.4.3. Kontroler pamięci RAM

Kontroler pamięci *RAM DDR3* jest niezbędnym elementem, koniecznym do prawidłowego działania każdego rozbudowanego systemu wizyjnego. Zewnętrzna pamięć, jest bowiem konieczna do przechowywania modelu tła dla poszczególnych pikseli. Układy *FPGA* posiadają oczywiście również bloki *BRAM*, jednak dostępna w nich pamięć jest zbyt mała, aby przechować choć jedną ramkę obrazu. Przedstawiony kontroler, został opracowany w Laboratorium Biocybernetyki AGH w ramach publikacji [20], autorzy

udostępnili, na potrzeby niniejszej pracy, kod źródłowy w języku *Verilog*. Podobnie jak w przypadku generatora liczb pseudolosowych, rozdział ten ma na celu jedynie przedstawienie uproszczonej koncepcji działania modułu. Dokładny opis i dyskusja zostały zawarte w przytoczonej publikacji. Firma *Xilinx* zapewnia dedykowany moduł *MIG* (*Memory Interface Generator*) do komunikacji z zewnętrzną pamięcią DDR3 *RAM*. Schemat kontrolera pamięci wykorzystującego *MIG* został przedstawiony na rysunku 4.8.



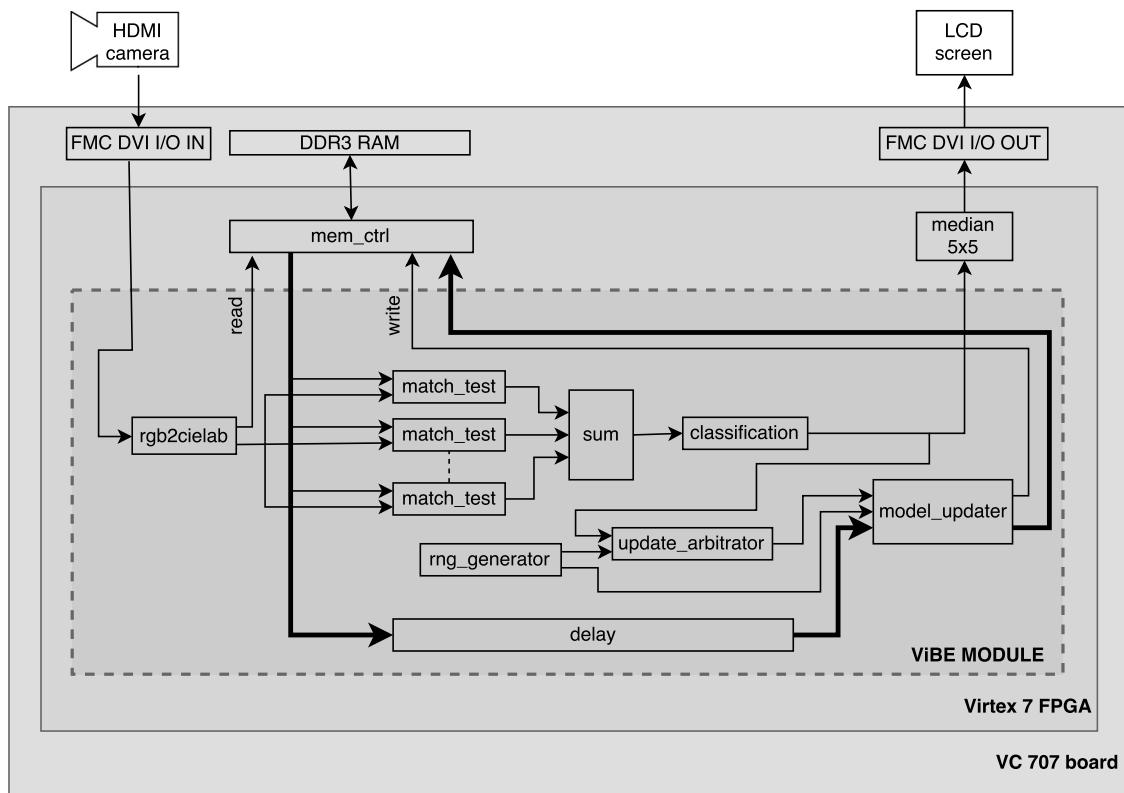
Rys. 4.8. Schemat kontrolera pamięci *RAM* – źródło [20]

Pamięć *RAM* synchronizowana jest innym zegarem niż sygnał z kamery, w związku z tym konieczne jest zastosowanie buforów w postaci dodatkowych kolejek *FIFO*. Omawiany kontroler (moduł *CTRL* na zamieszczonym rysunku) został zaimplementowany jako maszyna stanów. Podczas etapu inicjalizacji zapełniona zostaje w całości kolejka *READ FIFO*. Następnie, w momencie gdy pojawi się na wejściu nowa ramka obrazu, modele tła z kolejki *READ FIFO* zostają odczytywane i przekazywane do modułu realizującego algorytm segmentacji tła. Zaktualizowany model tła otrzymywany na wyjściu jest umieszczany w kolejce *WRITE FIFO*. Kolejnym krokiem jest przeniesienie danych z kolejki *WRITE FIFO* do dużo krótszej *TEMP FIFO*. W momencie, gdy kolejka ta jest zapełniona, uruchamiany jest tryb *burst*, zapisujący wszystkie modele do pamięci *RAM*. Po dokonaniu zapisu, dokładnie taka sama liczba modeli, zostaje odczytana z pamięci *RAM* i ponownie umieszczona w kolejce *READ FIFO*. Następnie kontroler przechodzi do stanu oczekiwania, aż do momentu ponownego zapełnienia się kolejki *TEMP FIFO*.

W zależności od obsługiwanej rozdzielczości zastosowano różne rozmiary opisywanych kolejek. Dane z pamięci *RAM* zawsze odczytywane są w blokach po 512 bitów każdy. Dla standardowej rozdzielczości *576p* pojedynczy model tła ma rozmiar 1024 bity, kolejka *READ FIFO* w takiej konfiguracji posiada szerokość 512 bitów na wejściu oraz 1024 na wyjściu i może pomieścić 8192 elementy. Druga z kolejek (*WRITE FIFO*) ma rozmiar 2048 oraz szerokość 1024 bitów na wejściu i 512 na wyjściu. W konfiguracji z wyższą rozdzielczością i modelem tła o rozmiarze 256 bitów, pierwsza kolejka musi mieć oczywiście szerokość 512 na wejściu i 256 na wyjściu. Druga, analogicznie 256 na wejściu i 512 na wyjściu. W tym przypadku rozmiar obu buforów jest taki sam i wynosi 4096. Ostatni z opisanych elementów, czyli kolejka *TEMP FIFO* we wszystkich konfiguracjach ma takie same parametry, szerokość 512 bitów i długość 256.

4.5. Implementacja algorytmu ViBE

Przygotowana implementacja sprzętowa powstała na podstawie opisu teoretycznego metody – przedstawionego w rozdziale 3.2. Wykorzystano wersję operującą w przestrzeni *CIELab*. Autorzy publikacji [22] udostępnili kod źródłowy w języku *VHDL*, jednak przedstawiona w niniejszym rozdziale wersja została zaimplementowana od podstaw, przy użyciu języka *Verilog*. Wysokopoziomowy schemat implementacji, zawierający główny moduł realizujący algorytm, wejściowy i wyjściowy sygnał z kamery oraz połączenie z pamięcią RAM, zostało przedstawiony na rysunku 4.9. Dla zachowania spójności, w dalszej części opisu, przyjęto oznaczenia parametrów algorytmu identyczne jak te zestawione w podsumowaniu rozdziału teoretycznego (podrozdział 3.2.6).



Rys. 4.9. Wysokopoziomowy schemat implementacji algorytmu *ViBE*

W podstawowej wersji algorytmu wykorzystany został standardowy interfejs wizyjny przedstawiony na rysunku 4.4 – bez żadnych dodatkowych sygnałów. Pierwszą operacją jest oczywiście konwersja z przestrzeni *RGB* do *CIELab* (moduł *rgb2cielab*). Operacja przekształcenia do macierzy *XYZ*, opisana równaniem (3.6), jest prosta do zaimplementowania w logice programowej i została zrealizowana z wykorzystaniem mnożarek sprzętowych operujących na liczbach stałoprzecinkowych. Obliczanie wartości funkcji $f(t)$ zdefiniowanej równaniem (3.7) zostało zaimplementowane z użyciem operacji *LUT*. Zastosowano trzy moduły, przechowujące stablicowane wartości funkcji $f(\frac{X}{X_n})$, $f(\frac{Y}{Y_n})$, $f(\frac{Z}{Z_n})$ wymagane do obliczenia składowych a i b danych równaniem (3.8) oraz jeden moduł zawierający wartości składowej L . Jak zostało już wspomniane w rozdziale 3.2.2 składowe a i b mieszczą się w przedziale –128 do 127.

Natomiast składowa L w zakresie 0 – 100. W związku z tym rozmiar piksela to **23 bity** zamiast 24, jak miało to miejsce w przypadku przestrzeni *RGB*.

Dla poszczególnych próbek modelu obliczany jest dystans do aktualnego piksela zgodnie z równaniem (3.5). Otrzymana wartość zostaje porównywana z progiem R . Operacje te wykonywane są równolegle dla każdej próbki z wykorzystaniem bloczków *match_test*. Następnie w module *sum* zliczane są próbki dla których test dopasowania przeszedł pozytywnie. Ostatnim etapem jest blok *classification* dokonujący ostatecznej klasyfikacji piksela zgodnie z równaniem (3.4). Dodatkowo, w celu eliminacji szumów, maska wyjściowa jest filtrowana, wykorzystano filtr medianowy o rozmiarze 5×5 .

W procesie aktualizacji modelu wykorzystywany jest 128 bitowy losowy sygnał otrzymyany z bloku *rng_generator*. Idea działania generatora liczb losowych została szerzej opisana w rozdziale 4.4.2. Modułu *update_arbitrator* jest wykorzystywany do podejmowania decyzji odnośnie aktualizacji. Sama aktualizacja wykonywana jest natomiast w bloku *model_updater*. W tym celu, generowany jest kontekst o rozmiarze 3×3 , zawierający modele tła sąsiadujących pikseli. Na podstawie informacji, otrzymanych z modułu *update_arbitrator*, aktualizowany jest model piksela oraz losowo wybranego sąsiada. Następnie nowy model zostaje zapisany do pamięci *RAM*. Warto zwrócić uwagę, że model jest odczytywany z pamięci po konwersji do przestrzeni *CIELab*, aby wykorzystać go w procesie aktualizacji, konieczne jest zastosowanie linii opóźniającej o długości równej latencji procesu klasyfikacji piksela. Operacja ta została zrealizowana przez moduł *delay*.



Rys. 4.10. Działający algorytm *ViBE*

Przedstawioną implementację udało się uruchomić w rozdzielczościach: *576p*, *720p*, *1080p* w 50 klatkach na sekundę. Na rysunku 4.10 przedstawiono zdjęcie działającego systemu. W przypadku najwyższej rozdzielczości przyjęto model składający się z $N = 20$ próbek (rozmiar modelu wynosi w tym przypadku $20 \cdot 23 = 460$ bitów). Dla wyższych rozdzielczości, ze względu na ograniczenia pamięci *RAM*, musiał on zostać ograniczony do 10 (rozmiar modelu równy $10 \cdot 23 = 230$ bity). Pozostałym parametrom przypisano wartości domyślne, zostały one zapisane jako liczny całkowite. Wyjątkiem jest

parametr T , opisujący prawdopodobieństwo wykonania aktualizacji, w tym przypadku została wykorzystana 16 bitowa liczba stałoprzecinkowa (ang. *fixed point*) bez znaku o oznaczeniu $8z8u$, czyli po 8 bitów przeznaczonych na część całkowitą i ułamkową. Tak dobrana reprezentacja zapewnia dobrą dokładność przy stosunkowo niewielkim rozmiarze, co jest bardzo istotne ze względu na bardzo ograniczony rozmiar modelu tła.

Użycie zasobów logicznych układu *FPGA* zamieszczono w tabelach 4.2 i 4.3, odpowiednio dla rozdzielczości $576p$ oraz $1080p$. Oszacowany w obu przypadkach pobór mocy wynosi $3,957W$ i $3,370W$. W celu uproszczenia implementacji, dla rozdzielczości $720p$ i $1080p$ zastosowano te same rozmiary kolejek i linii opóźniających (zmieniono jedynie stałą H_SIZE), w związku z tym zarówno pobór mocy jak i wykorzystanie zasobów w obu przypadkach jest identyczne. Można zauważyć wyraźny wzrost wykorzystania dostępnych zasobów w stosunku do algorytmu odejmowania ramek (tabela 4.1), wynika to oczywiście ze zdecydowanie bardziej rozbudowanej logiki algorytmu. Na wyraźnie większe użycie *LUT* miało wpływ między innymi zastosowanie konwersji *RGB–CIELab*, gdzie konieczne było tablicowanie wartości niektórych funkcji. Pamięć *BRAM* została z kolei wykorzystana w długich liniach opóźniających piksele jak i cały model tła. Warto jeszcze raz zaznaczyć, że w przypadku wyższej rozdzielczości konieczne było ograniczenie rozmiaru modelu (ze względu na ograniczenia zewnętrznej pamięci *RAM*). W związku z tym, ostateczne zużycie zasobów w systemie pracującym w $1080p@50fps$ jest niższe niż dla $576p@50fps$.

Tabela 4.2. *ViBE 576p@50fps* - wykorzystanie zasobów (*Virtex 7*)

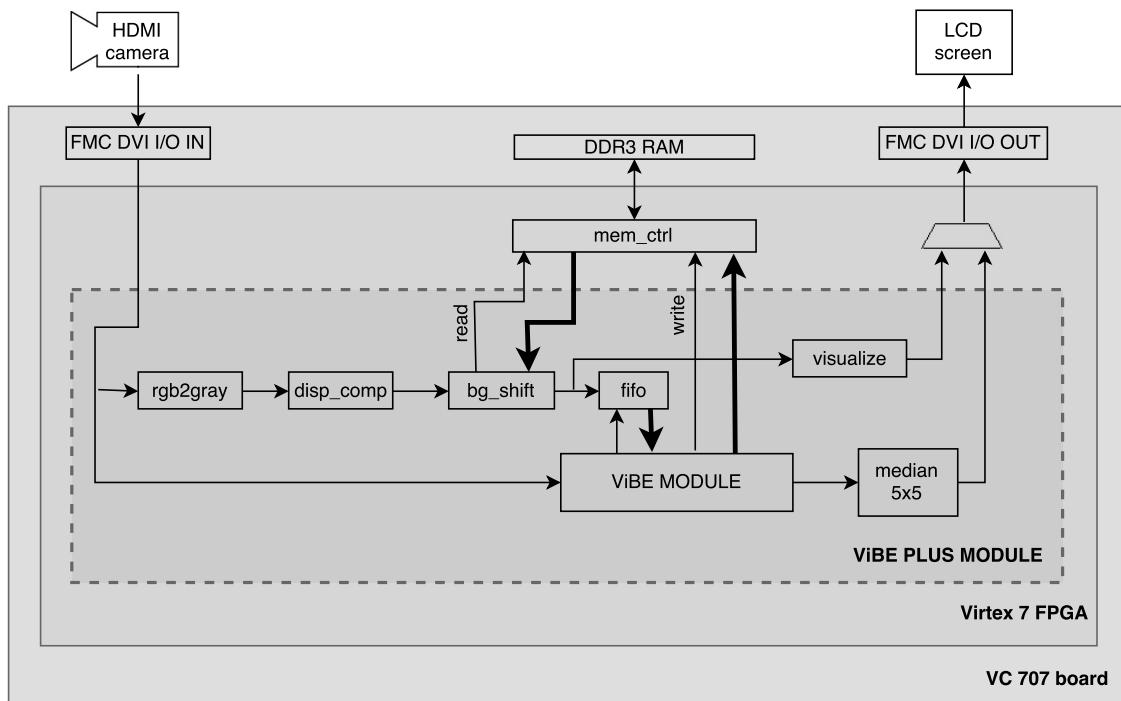
Zasoby	Wykorzystane	Dostępne	Użycie w %
LUT	19774	303600	6,51
LUTRAM	2083	130800	1,59
FF	20936	607200	3,45
BRAM	207	1030	20,10
DSP	54	2800	1,93

Tabela 4.3. *ViBE 1080p@50fps* - wykorzystanie zasobów (*Virtex 7*)

Zasoby	Wykorzystane	Dostępne	Użycie w %
LUT	16224	303600	5,34
LUTRAM	2107	130800	1,61
FF	16236	607200	2,67
BRAM	163	1030	15,83
DSP	29	2800	1,04

4.6. Implementacja rozszerzonej wersji ViBE

W rozdziale 3.2.3 opisany został dodatkowy mechanizm usprawniający pracę algorytmu, w przypadku występowania drgań kamery (ang. *camera jitter*). Schemat implementacji rozszerzonej wersji algorytmu, zawierającej dodatkowy moduł, został przedstawiony na rysunku 4.11. Implementacja odpowiada opisowi teoretycznemu, który został przedstawiony w podrozdziale 3.2.3. Oprócz bloku realizującego standardowy algorytm *ViBE*, opisanego szczegółowo w poprzednim rozdziale, ta wersja algorytmu zawiera także szereg modułów zapewniających prawidłowe przesunięcie modelu tła na podstawie wyznaczonego przepływu optycznego. Moduł, podobnie jak poprzednia wersja, wykorzystuje standardowy interfejs wizyjny, schemat modelu tła jest również identyczny jak w podstawowej wersji algorytmu. W tym przypadku wykorzystany został w całości kod źródłowy w języku *Verilog*, udostępniony przez autorów. Dodatkowy moduł został zintegrowany z implementacją opisaną w poprzednim rozdziale.

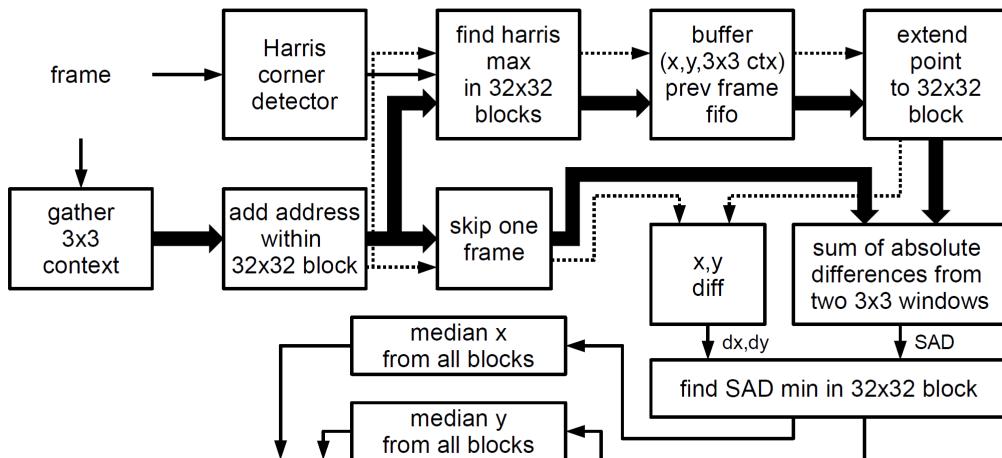


Rys. 4.11. Wysokopoziomowy schemat implementacji rozszerzonego algorytmu *ViBE*

Pierwszym krokiem, podczas obliczania przepływu optycznego, jest konwersja obrazu z przestrzeni *RGB* do skali szarości. Jest to wykonywane poprzez moduł *rgb2gray*. Przekształcony sygnał z kamery jest następnie podawany na wejście bloku *disp_comp* realizującego operację wyznaczania przesunięcia modelu na podstawie wyliczonego przepływu optycznego. Jest to najbardziej złożony moduł w całej implementacji. Jego szczegółowy schemat został przedstawiony na rysunku 4.12. Na podstawie otrzymanego wektora przesunięcia blok *bg_shift* zapewnia prawidłową synchronizację i odczyt kolejnych modeli tła z pamięci *RAM*. Ze względu na przesunięcie operacja odczytu musi zostać wykonana z odpowiednim wyprzedzeniem lub opóźnieniem. Dane z pamięci *RAM* są umieszczane w kolejce *FIFO*. Skąd następnie

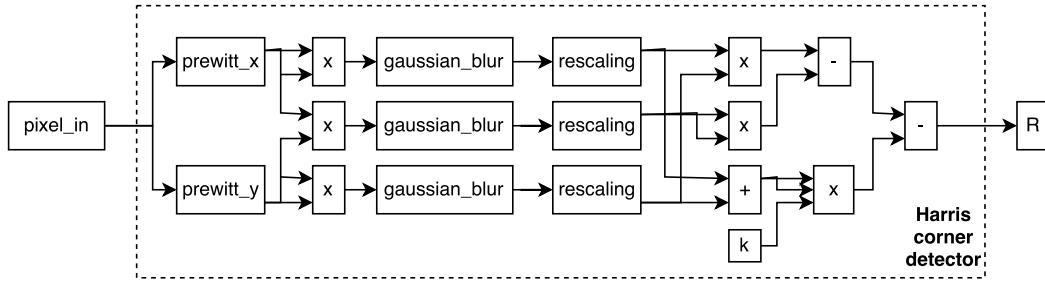
są pobierane przez algorytm *ViBE*. Kolejka *FIFO*, podobnie jak linie opóźniające, została zaimplementowana z wykorzystaniem pamięci blokowej (*BRAM*). Dzięki zastosowaniu uniwersalnego interfejsu z niezależnymi sygnałami odczytu i zapisu do pamięci *RAM*, nie było konieczności modyfikowania oryginalnej implementacji metody *ViBE*. Na maskę wyjściową, nakładany jest filtr medianowy o rozmiarze 5×5 , dodatkowo został zaimplementowany moduł wizualizujący operację obliczania przepływu optycznego.

Z uwagi na dodatkowy mechanizm obsługujący ruch kamery, rozszerzona wersja algorytmu operuje jedynie na rozdzielczości 720×576 . Zastosowano identyczne parametry jak w podstawowej wersji metody, czyli model tła składający się z 20 próbek w przestrzeni *CIELab*. Składowa wektora przesunięcia może przyjąć maksymalnie wartość 32, zatem kolejka *FIFO* gromadząca modele tła musi mieć długość $32 \cdot H_SIZE + 32$, gdzie każdy element ma szerokość równą modelowi tła. Dla obsługiwanej rozdzielczości parametr *H_SIZE* wynosi 864, natomiast model tła ma rozmiar 460 bitów. Ostatecznie ustwiono szerokość pojedynczego elementu na 512 bitów, natomiast głębokość kolejki wynosi 32768.



Rys. 4.12. Implementacja modułu wyznaczającego wektor przesunięcia – źródło [22]

Opis teoretyczny tej metody został przedstawiony w podrozdziale 3.2.4. Pierwszym krokiem algorytmu jest detekcja narożników metodą Harrisza-Stephensa. Schemat przedstawiający implementację tej funkcji został pokazany na rysunku 4.13. Wykonane tam operacje odpowiadają opisowi teoretycznemu zamieszczonemu w podrozdziale 3.2.5. Przy implementacji uwzględniono w maksymalnym stopniu możliwości układu FPGA w zakresie zrównoleglania obliczeń. Moduły *prewitt_x* i *prewitt_y* realizują detekcję krawędzi pionowych i poziomych. Następnie wszystkie trzy elementy macierzy *H* są obliczane równolegle. Po dokonaniu operacji wygładzania filtrem Gaussa (blok *gaussian_blur*) w bloku *rescaling* następuje przeskalowanie (dzielenie przez 2^{10}) otrzymanych wartości. Finalnie wyznaczany jest wyznacznik macierzy *H* – zgodnie z równaniem (3.14). Całość została zrealizowana z wykorzystaniem sprzętowych mnożarek i sumatorów. Konieczne jest wykonanie 6 operacji mnożenia, jednego dodawania i dwóch odejmowania.



Rys. 4.13. Moduł realizujący detekcję narożników metodą Harrisza-Stephensa

Równolegle do detekcji krawędzi wyznaczany jest kontekst aktualnego piksela o rozmiarze 3×3 . Określana jest także jego pozycja w aktualnie przetwarzanym bloku o rozmiarze 32×32 . Piksel dla którego otrzymano największą wartość współczynnika R w obszarze każdego bloku zostaje zapisany razem z kontekstem 3×3 w kolejce FIFO (w przypadku obrazu o rozdzielcości 720×576 konieczne jest zdefiniowanie kolejki o rozmiarze 396, gdyż tyle jest bloków 32×32 w jednej ramce). Kolejne elementy z kolejki, są pobierane w trakcie przetwarzania następnej ramki obrazu. W każdym z bloków znajdowana jest minimalna wartość sumy różnicy modułów pomiędzy pikselem pobranym z kolejki i pikselami z aktualnej ramki obrazu. Znaleziona wartość, wraz z wektorem przesunięcia, zostaje po raz kolejny zapamiętana w pamięci blokowej. Operacja ta wykonywana jest w bloku *find SAD min in 32×32 block*.

Ostatnim krokiem jest obliczenie mediany przesunięcia dx i dy wśród znalezionych wartości minimalnych. Operacja ta składa się z dwóch etapów wykonywanych w momencie przerwy pomiędzy kolejnymi ramkami obrazu (wartość sygnału *vsync* wynosi wtedy 0). Najpierw wyliczany jest histogram wartości dx i dy , równolegle zliczana jest także liczba próbek. Drugim krokiem jest sumowanie wartości histogramu. Poprzez medianę rozumiemy wartość dla której aktualna suma wartości histogramu jest większa niż połowa liczby wszystkich próbek. Do obliczenia histogramu ponownie został wykorzystany moduł pamięci blokowej (*BRAM*). Na rysunku 4.14 przedstawiono wizualizację wyznaczanego przepływu optycznego, analogiczną do tej pokazanej na rysunku 3.1.

W tabeli 4.4 zamieszczono użycie zasobów w układzie *FPGA*, oszacowany pobór mocy wynosi 4,011W. W porównaniu do standardowej wersji algorytmu (tabela 4.2) można zauważycy znaczący wzrost wykorzystania pamięci *BRAM*. Algorytm obliczania przepływu optycznego wymaga dodatkowej pamięci, która wykorzystywana jest do przechowywania informacji z każdego bloku oraz do obliczania mediany. Wzrost zużycia pozostałych zasobów jest minimalny.

4.7. Implementacja algorytmu PBAS

Szczegółowy opis teoretyczny algorytmu *PBAS* został zamieszczony w podrozdziale 3.3. Zaimplementowano wariant operujący w przestrzeni kolorów *RGB* w rozdzielcości 720×576 pikseli oraz wersję przetwarzającą obraz w skali szarości w przypadku wyższych rozdzielcości ($720p$ i $1080p$). Wysokopoziomowy schemat całego algorytmu został przedstawiony na rysunku 4.15. Jak zostało już wspomniane



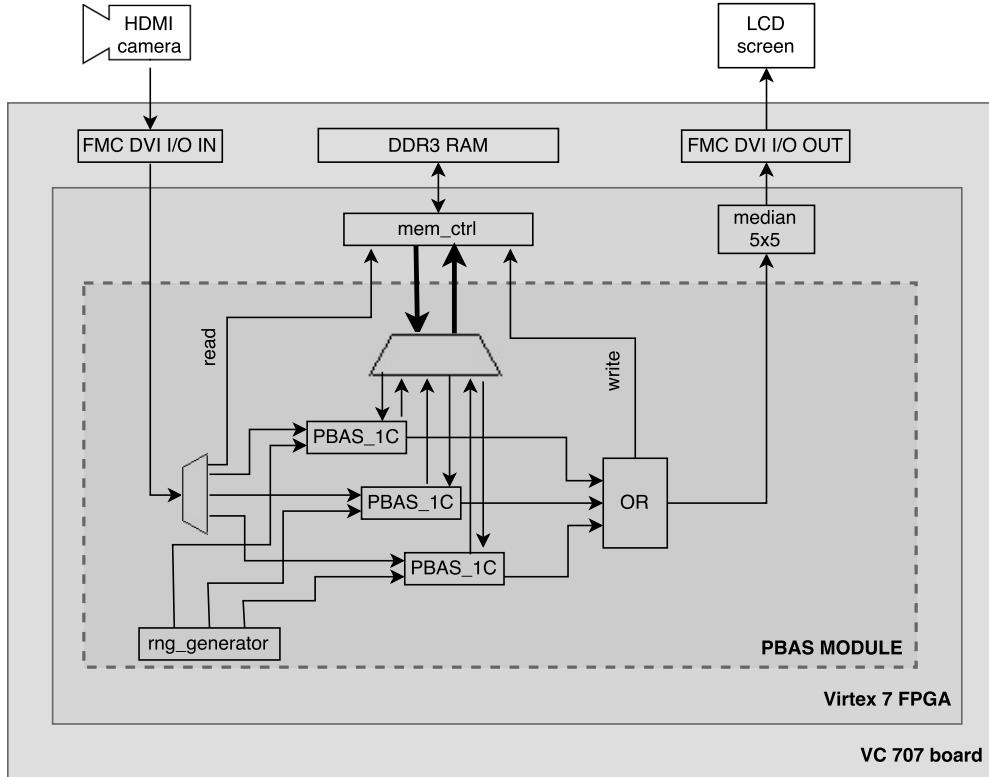
Rys. 4.14. Wizualizacja przepływu optycznego

Tabela 4.4. Rozszerzony algorytm *ViBE* - wykorzystanie zasobów (*Virtex 7*)

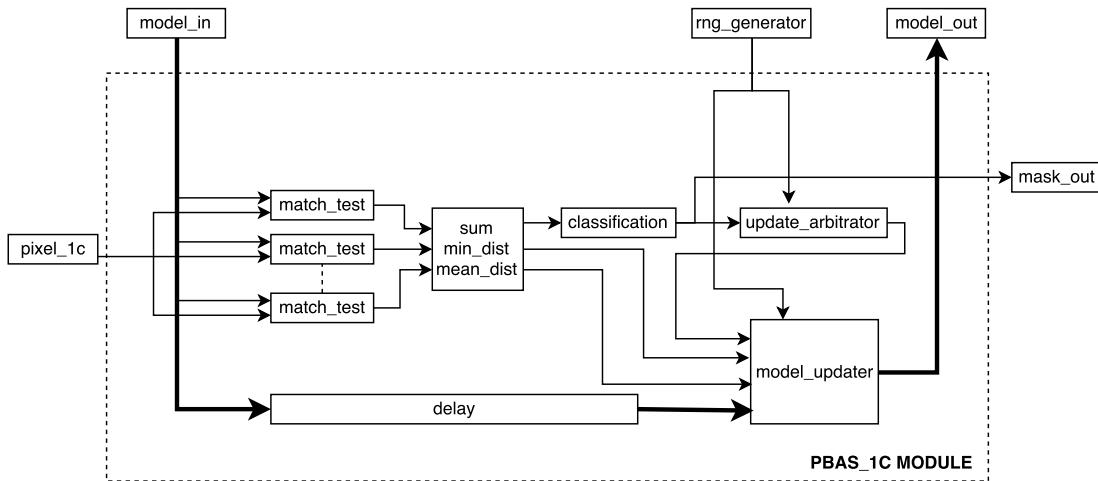
Zasoby	Wykorzystane	Dostępne	Użycie w %
LUT	22626	303600	7,45
LUTRAM	2253	130800	1,80
FF	25004	607200	4,12
BRAM	354	1030	34,37
DSP	49	2800	1,75

w części teoretycznej, wersja algorytmu działająca w przestrzeni *RGB*, przetwarza niezależnie poszczególne składowe, tworząc dla każdej z nich osobny model tła. Podobnie jak w przypadku algorytmu *ViBE* (rozdział 4.5), ta podstawowa wersja metody została zaimplementowana od podstaw z wykorzystaniem języka *Verilog*.

Pierwszą operacją, jest rozbicie sygnału wejściowego na składowe *RGB*. Następnie każda z nich jest analizowana niezależnie poprzez algorytm *PBAS* (blok *PBAS_IC*). Szczegółowy schemat implementacji modułu realizującego algorytm dla jednej składowej przedstawiono na rysunku 4.16. Maska finalna stanowi alternatywę logiczną wyników otrzymanych z poszczególnych modułów. Podobnie jak w przypadku innych algorytmów, tutaj także, na koniec, nakładany jest filtr medianowy w celu wyeliminowania szumów i zakłóceń. Algorytm wykorzystuje także generator liczb losowych, zaimplementowany w module *rng_generator*. Jego opis został szerzej przedstawiony w podrozdziale 4.4.2. Opisywany moduł wykorzystuje oczywiście uniwersalny interfejs wizyjny przedstawiony w rozdziale 4.3.



Rys. 4.15. Implementacja algorytmu PBAS w wersji RGB



Rys. 4.16. Implementacja algorytmu PBAS dla jednej składowej RGB

Implementacja algorytmu PBAS została przedstawiona na rysunku 4.16. Przedstawiona wersja może operować na jednej składowej *RGB* bądź na obrazie w skali szarości. Jak łatwo zauważyć, przedstawiony schemat jest bardzo podobny do implementacji metody *ViBE* przedstawionej w rozdziale 4.5. Podobnie jak w tamtym przypadku, tutaj także, pierwszym krokiem jest równoległy test dopasowania do próbek zapisanych w pamięci *RAM*, zgodnie z równaniem (3.18). Następnie przeprowadzana jest klasyfikacja,

opisana równaniem (3.16). Dodatkowo obliczany jest minimalny dystans pomiędzy próbками z modelu, a aktualnym pikselem oraz średnia wartość odległości zapisanych w modelu tła. Tak samo jak w metodzie *ViBE*, decyzja o podjęciu aktualizacji realizowana jest przez moduł *update_arbitrator*. Blok dokonujący aktualizacji, jest rozszerzeniem tego co zostało zaprezentowane w implementacji metody *ViBE*. Dodatkowo, oprócz próbek, aktualizowany jest także zbiór minimalnych odległości oraz parametry R i T , zgodnie z równaniami (3.21) i (3.22).

Pierwsza część modelu składa się z N próbek, każda próbka zawiera wartość piksela – B_i oraz zapamiętany minimalny dystans – D_i pomiędzy aktualną próbką a aktualną wartością piksela. Na końcu znajdują się parametry R i T . Modele dla poszczególnych składowych *RGB* ustawione kolejno po sobie. Dla rozdzielczości $576p$ przyjęto model składający się z $N = 19$ próbek. Parametry R i T zapisano w postaci 16-bitowych liczb stałoprzecinkowych w formacie $8z8u$. Minimalny dystans, podobnie jak próbka, zapisana jest jako 8-bitowa liczba całkowita. Sumarycznie model tła dla jednej składowej *RGB* ma rozmiar $19 \cdot (8 + 8) + 16 + 16 = 336$ bitów, zatem cały model zajmuje $3 \cdot 336 = 1008$ bitów. Dla wyższych rozdzielczości wykorzystano algorytm operujący na obrazie w skali szarości, model w tym przypadku składa się z $N = 14$ próbek. Zatem cały model można zapisać za pomocą $14 \cdot (8 + 8) + 16 + 16 = 256$, jest to maksymalna szerokość modelu, która może zostać wykorzystana w przypadku obrazu w rozdzielczości $720p$ lub $1080p$. Wykorzystanie zasobów jest identyczne dla obu rozdzielczości, ze względu na uproszczenia w implementacji. Wykorzystano te same bufory zarówno w kontrolerze pamięci jak i w długich liniach opóźniających (dokonano jedynie modyfikacji parametry *H_SIZE*).

Użycie zasobów dla algorytmu *PBAS* w wersji $576p@50fps$ i $1080p@50fps$ zamieszczono w tabelach 4.5 i 4.6, pobór mocy w obu przypadkach wynosi odpowiednio 4,686W i 3,263W. Należy jeszcze raz podkreślić, że dla wyższej rozdzielczości zastosowano algorytm operujący na obrazie w skali szarości z dużo mniej dokładnym modelem tła. Takie działanie wymuszone było ograniczeniami zewnętrznej pamięci *RAM* i w rezultacie mniejszym rozmiarem modelu tła. Ostatecznie z tego powodu zużycie zasobów oraz pobór mocy dla rozdzielczości $1080p$ jest zdecydowanie niższy niż dla $576p$. W porównaniu do algorytmu *ViBE* (tabela 4.2) możemy zauważyć znaczący wzrost wykorzystywanych zasobów. Taki rezultat jest zgodny z oczekiwaniami, gdyż jak zostało już wcześniej powiedziane, omawiany algorytm jest rozszerzeniem metody *ViBE*.

Tabela 4.5. *PBAS 576p@50fps* - wykorzystanie zasobów (*Virtex 7*)

Zasoby	Wykorzystane	Dostępne	Użycie w %
LUT	28142	303600	9,27
LUTRAM	3492	130800	2,67
FF	33849	607200	5,57
BRAM	244	1030	23,69
DSP	24	2800	0,86

Tabela 4.6. PBAS 1080p@50fps - wykorzystanie zasobów (Virtex 7)

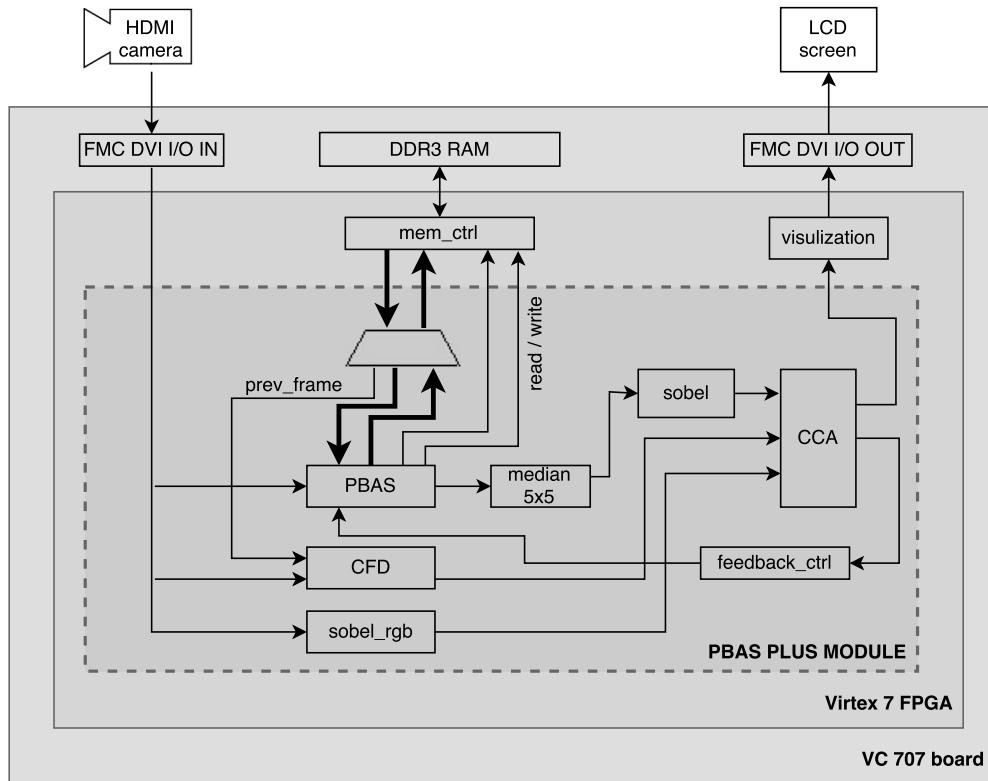
Zasoby	Wykorzystane	Dostępne	Użycie w %
LUT	16389	303600	5,40
LUTRAM	2816	130800	2,15
FF	16494	607200	2,72
BRAM	170	1030	16,50
DSP	11	2800	0,39

4.8. Implementacja rozszerzonej wersji metody PBAS

Przedstawiona implementacja, powstała na podstawie opisu teoretycznego, zamieszczonego w rozdziale 3.3.2. Opisana tutaj, rozszerzona wersja algorytmu *PBAS*, zawiera dodatkowy mechanizm rozróżniania obiektów statycznych od tzw. „duchów”. Koncepcja ta wymaga także zaimplementowania metody indeksacji obiektów co sprawia, że omawiany moduł jest najbardziej złożonym spośród wszystkich pokazanych w niniejszej pracy. Fragmenty przedstawionego tutaj rozwiązania pochodzą z publikacji [21], również powstałej w Laboratorium Biocybernetyki AGH. Autorzy udostępnili kod źródłowy modułu wykonującego indeksację oraz detekcję obiektów statycznych. Został on zintegrowany z podstawową wersją metody opisaną w poprzednim rozdziale. Ogólny schemat przedstawiający architekturę modułu pokazano na rysunku 4.17.

Pierwszym istotnym elementem jest realizacja podstawowego algorytmu *PBAS*, w stosunku do wersji przedstawionej w rozdziale 4.7 musiał on zostać zmodyfikowany. Konieczne było dodanie obsługi sprzężenia zwrotnego zgodnie z równaniami (3.24) i (3.26). Oprócz tego należało także zapewnić aktualizację dwóch nowych parametrów $S(x_i)_{cnt}$ i $E(x_i)_{mean}$, zgodnie z równaniami (3.27) i (3.28). Dodatkowo w modelu tła musi znaleźć się też poprzednia wartość piksela (24 bity) wykorzystywana podczas odejmowania ramek (blok *CFD*). W związku z tym, aby rozmiar modelu nie przekroczył maksymalnego rozmiaru wynoszącego 1024 bity, konieczne było zredukowanie liczby próbek w algorytmie *PBAS* do 18. Parametr $S(x_i)_{cnt}$ zapisany jest jako 8 bitowa liczba całkowita, z kolei wartość $E(x_i)_{mean}$ to 16 bitowa liczba stałoprzecinkowa w formacie $8\frac{1}{2}8u$. Ostatecznie rozmiar modelu tła w takiej konfiguracji wynosi: $3 \cdot (18 \cdot (8 + 8) + 16 + 16) + 8 + 16 + 24 = 1008$ bitów.

Równolegle do algorytmu *PBAS* wykonywane jest, wspomniane wcześniej, odejmowanie ramek oraz detekcja krawędzi metodą Sobela. Na maskę wyjściową podobnie jak w poprzednich metodach, nakładany jest filtr medianowy. Wyznaczane są także krawędzie na masce binarnej, gdyż jest to niezbędne podczas prowadzenia analizy poszczególnych obiektów. Schemat bloku *CCA* odpowiedzialnego za to zadanie został pokazany na rysunku 4.18. Ze względu ma fakt, że dane z modułu analizy obiektów dostępne są dopiero po przetworzeniu całej ramki obrazu, konieczne jest zapewnienie odpowiedniej synchronizacji sygnałów sprzężenia zwrotnego z modułem *PBAS*. Zadanie to zostało zrealizowane poprzez blok *feedback_ctrl*. Wyznaczone wartości współczynników stabilności i podobieństwa krawędzi oraz

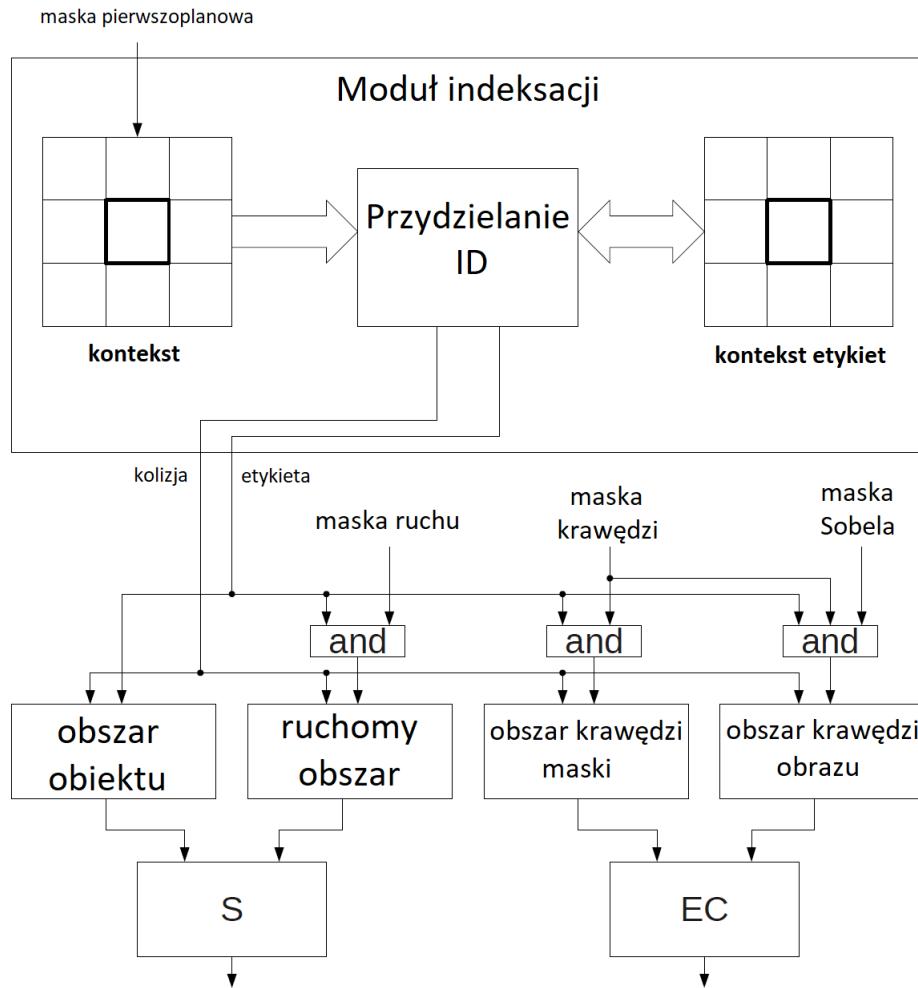


Rys. 4.17. Implementacja rozszerzonej wersji algorytmu PBAS

finalna maska są zapisywane w pamięci *BRAM*. Działający algorytm poprawnie identyfikujący obiekt pierwszoplanowy oraz prostokąt go otaczający został pokazany na rysunku 4.19.

Do zrealizowania zadania indeksacji obiektów wykorzystano moduł *CCA* opracowany w ramach publikacji [21]. Całość została przygotowana w oparciu o teorię na temat indeksowania obiektów, przedstawioną w rozdziale 3.3.3. Moduł składa się z dwóch części, pierwszą z nich jest fragment logiki programowej (*Labelling Block*), odpowiedzialny za przypisanie kolejnym pikselom odpowiednich etykiet. Jak zostało to opisane w części teoretycznej, etykieta jest dobierana na podstawie sąsiednich pikseli oraz etykiet im przypisanych. Konieczne jest zatem wygenerowanie kontekstu piksela wejściowego, zawierającego etykiety przypisane sąsiadom.

Informacja o przydzielonej etykiecie przekazywana jest do czterech bloków obliczających pole zidentyfikowanego obiektu. Pierwszy licznik (*segment area*) służy do wyznaczenia całkowitego pola obiektu. Drugi moduł (*movement area*) zlicza natomiast jedynie ruchome piksele. Wartości te są używane do wyznaczenia współczynnika S_{O_k} zgodnie z równaniem (3.24). Wykorzystano w tym celu dzielarkę sprzętową. Kolejny licznik (*edge mask area*) służy do zliczania pikseli na pierwszoplanowej masce krawędzi. Ostatni z modułów (*edge image area*) wyznacza liczbę pikseli znajdujących się na wspomnianej wyżej masce oraz na zbinaryzowanym obrazie krawędzi obrazu wejściowego. Te dwie wartości służą do wyznaczenia parametru EC_{O_k} , zgodnie z równaniem (3.26). Podobnie jak w poprzednim przypadku, tutaj także wykorzystano dzielarkę sprzętową.



Rys. 4.18. Architektura modułu CCA – źródło [21]

Oprócz wymienionych czterech istnieje jeszcze jeden blok, aktualizujący na bieżąco parametry prostokąta otaczającego poszczególne obiekty. Wszystkie liczniki zostały zaimplementowane z użyciem pamięci blokowej *BRAM*. Problem konfliktów rozwiązano poprzez sumowanie wartości zapamiętanych dla obu łączonych obiektów i zapisanie jej jako nowej wartości dla obiektu o niższej etykiecie. Wartość zapisana pod adresem wyższej etykiety zostaje natomiast oznaczona jako nieważna.

Wykorzystanie zasobów przez rozszerzoną wersję algorytmu *PBAS* przedstawiono w tabeli 4.7, oszacowany pobór mocy wynosi natomiast 5,087W. Ze względu na mechanizm indeksacji, jest to najbardziej złożony ze wszystkich omawianych algorytmów, stąd też najwyższe zużycie zasobów jak i pobór mocy. W stosunku do standardowej metody *PBAS* możemy zauważyć kilku procentowy wzrost dla większości typów zasobów.



Rys. 4.19. Działający algorytm PBAS wraz z indeksacją obiektów

Tabela 4.7. Rozszerzony algorytm PBAS - wykorzystanie zasobów (*Virtex 7*)

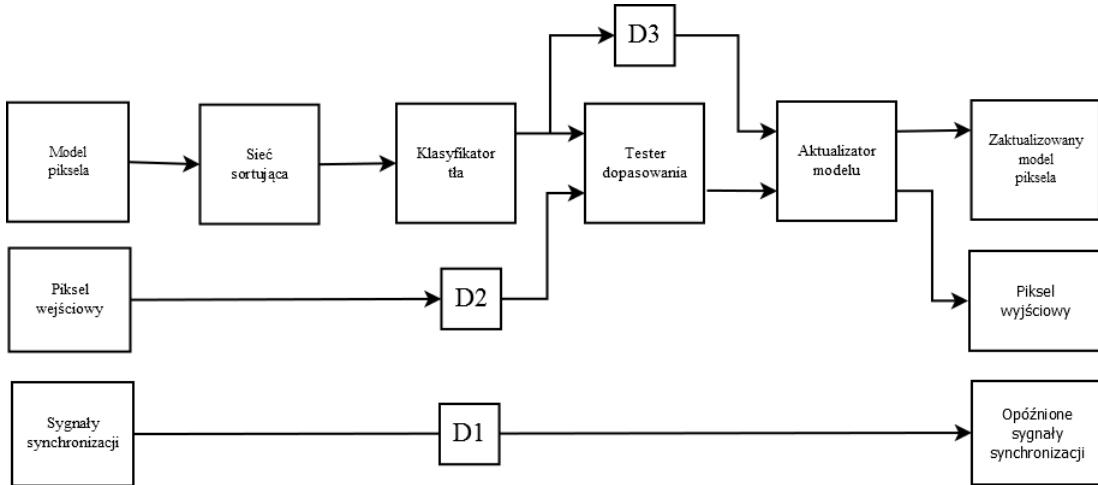
Zasoby	Wykorzystane	Dostępne	Użycie w %
LUT	40329	303600	13,28
LUTRAM	5549	130800	4,24
FF	40849	607200	6,73
BRAM	260	1030	25,19
DSP	24	2800	0,86

4.9. Implementacja GMM

Przedstawiona tutaj realizacja sprzętowa algorytmu *GMM* została przygotowana w ramach pracy inżynierskiej [27]. Sposób implementacji omawianej metody jest tematem niezwykle rozległym. W niniejszym rozdziale przedstawiono jedynie ogólną ideę przygotowanej architektury bez zagłębiania się w szczegóły realizacji poszczególnych operacji w układzie reprogramowalnym. Schemat przedstawiający implementację i wykorzystane moduły został pokazany na rysunku B.1.

Niewątpliwą zaletą *GMM*, znacząco ułatwiającą implementację w układzie reprogramowalnym, jest brak operacji kontekstowych. Dzięki temu możemy znaczowo uprościć logikę programową i zredukować użycie zasobów logicznych. Nawiązując do opisu teoretycznego zamieszczonego w rozdziale 3.4, pierwszym krokiem algorytmu jest sortowanie rozkładów Gaussa według współczynnika $r_i = \frac{\omega_i}{\sigma_i}$. W tym celu wykorzystana została specjalnie zaprojektowana sieć sortująca, jest to najbardziej złożony element przedstawianej implementacji.

Po dokonaniu sortowania, poszczególne rozkłady Gaussa są klasyfikowane, na podstawie równania 3.32. Następnie przeprowadzany jest test dopasowania i ostateczna klasyfikacja piksela, zgodnie z równaniem 3.33. Po dokonaniu klasyfikacji rozkłady Gaussa są aktualizowane i zapisywane w pamięci *RAM*.



Rys. 4.20. Implementacja algorytmu GMM – źródło [27]

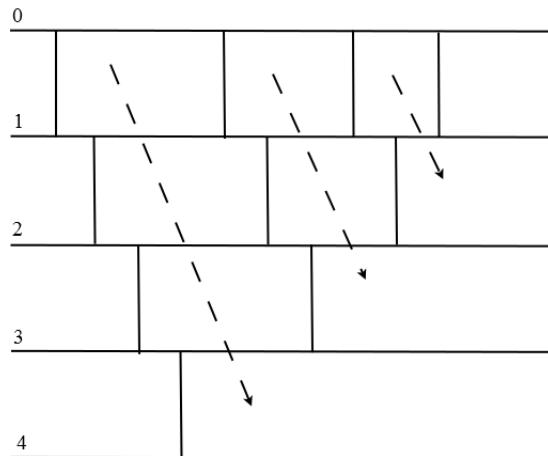
Przygotowana implementacja przetwarza obraz we wszystkich wymaganych rozdzielczościach, czyli $576p$, $720p$ i $1080p$ dla 50 klatek na sekundę. Dla obrazu 720×576 przyjęto $K = 5$ rozkładów Gaussa w każdym modelu, natomiast dla wyższych rozdzielczości, ze względu na ograniczenia pamięci RAM , liczba rozkładów wynosi $K = 3$. W modelu tła pojedynczy rozkład Gaussa ma rozmiar 68 bitów, dokładny opis i reprezentacja poszczególnych bitów modelu została przedstawiona w tabeli 4.8.

Tabela 4.8. Znaczenie kolejnych bitów w reprezentacji rozkładu Gaussa – źródło [27]

Zakres bitowy	Oznaczenie
0	Bit pierwszego planu. Jeśli jest równy 1 oznacza to, że rozkład Gaussa reprezentuje obiekty pierwszoplanowe, w przeciwnym razie reprezentuje on tło.
1 – 7	Bity zarezerwowane (w razie wprowadzenia nowych flag).
8 – 19	Waga rozkładu Gaussa ω jako liczba stałoprzecinkowa (zakres 0 – 1). Część całkowita zajmuje 0 bitów, część ułamkowa 12 bitów.
20 – 31	Wariancja (σ^2) jako liczba stałoprzecinkowa (zakres 0 – 256). Część całkowita zajmuje 8 bitów, część ułamkowa 4 bity.
32 – 43	Średnia wartość barwy czerwonej piksela jako liczba stałoprzecinkowa (zakres 0 – 256). Część całkowita zajmuje 8 bitów, część ułamkowa 4 bity.
44 – 55	Średnia wartość barwy zielonej piksela jako liczba stałoprzecinkowa (zakres 0 – 256). Część całkowita zajmuje 8 bitów, część ułamkowa 4 bity.
56 – 67	Średnia wartość barwy niebieskiej piksela jako liczba stałoprzecinkowa (zakres 0 – 256). Część całkowita zajmuje 8 bitów, część ułamkowa 4 bity.

Poglądowy schemat przedstawiający zasadę działania sieci sortującej dla $K = 5$ elementów, został pokazany na rysunku 4.21. Jest to sprzętowa realizacja algorytmu *BS* (ang. *Bubble Sort* – sortowanie

bąbelkowe). Pionowe linie oznaczają porównania między elementami. Przerywane linie z grotkiem określają z kolei kierunek przesuwania się najmniejszego elementu w danej iteracji. W pierwszym kroku spośród K elementów, wyłaniany jest najmniejszy poprzez wykonanie $K - 1$ porównań. Wyznaczony w ten sposób element jest przedstawiany na sam dół sieci. Następnie wśród pozostałych $K - 1$ elementów wyszukiwany jest kolejny najmniejszy, w tym przypadku należy wykonać $K - 2$ porównania itd. Ostatecznie do otrzymania posortowanego ciągu potrzeba $K - 1$ iteracji algorytmu.



Rys. 4.21. Sieć sortująca w układzie *FPGA* – źródło [27]

Tabela 4.9 przedstawia wykorzystanie zasobów w układzie *FPGA* przez algorytm *GMM*. Zaimplementowana metoda, w odróżnieniu od innych, nie zawiera operacji kontekstowych (poza filtrem medianowym nakładanym na maskę wyjściową). W związku z tym nie jest wykorzystywana pamięć *BRAM*. W przypadku pozostałych zasobów kluczowy jest parametr K reprezentujący liczbę rozkładów Gaussa. Dla $K = 5$, czyli domyślnej wartości wykorzystanie zasobów jest zdecydowanie niższe niż w przypadku algorytmów *ViBE* i *PBAS*, co niestety przekłada się również na dokładność metody. Testy poszczególnych algorytmów zostały szczegółowo przedstawione w rozdziale 5.

Tabela 4.9. *GMM* - wykorzystanie zasobów (*Virtex 7*)

Zasoby	Wykorzystane	Dostępne	Użycie w %
LUT	426	303600	0,14
LUTRAM	338	130800	0,26
FF	752	607200	0,12
BRAM	3	1030	0,002
DSP	0	2800	0,0

4.10. Podsumowanie

Implementacja opracowanych algorytmów w układzie reprogramowalnym zakończyła się sukcesem. Udało się uruchomić wszystkie przygotowane metody na jednej platformie sprzętowej. W przypadku mniej złożonych obliczeniowo algorytmów dodano także obsługę wyższych rozdzielczości. Zgodnie z oczekiwaniami uzyskano zdecydowanie wyższą wydajność niż w przypadku modeli programowych napisanych w *C++*.

Warto zwrócić uwagę na wykorzystanie zasobów przez poszczególne algorytmy. Zdecydowanie najbardziej złożonym i jednocześnie wykorzystującym najwięcej elementów logicznych jest metoda *PBAS* wraz z dodatkowymi mechanizmem detekcji obiektów statycznych. Drugi pod tym względem jest algorytm *ViBE* z modułem niwelującym drgania kamery. Stosunkowo niewielkie użycie zasobów uzyskano w przypadku algorytmu *GMM*, jest to spowodowane między innymi brakiem operacji kontekstowych i długich linii opóźniających.

Warto zwrócić uwagę na zależność wykorzystania zasobów od rozmiaru modelu tła. Można to zaobserwować w przypadku algorytmów *ViBE* i *PBAS* w wersjach działających w wyższej rozdzielczości. W obu przypadkach model został ograniczony poprzez zredukowanie liczby zapisanych próbek, dodatkowo w metodzie *PBAS* zdecydowano się zastosować uproszczenie w postaci przetwarzania obrazu w skali szarości. Wprowadzone zmiany spowodowały znaczące obniżenie liczby wymaganych elementów logicznych. Dzięki temu nawet po zwiększeniu rozdzielczości (co skutkowało większym zapotrzebowaniem na pamięć *BRAM*) ostateczne wykorzystanie zasobów okazało się niższe niż dla standardowych odmian tych algorytmów.

5. Ewaluacja zaimplementowanych algorytmów

5.1. Metodologia przeprowadzonych testów

Testy algorytmów zaimplementowanych w Laboratorium Biocybernetyki AGH zostały przeprowadzone zgodnie z metodologią opisaną w [7]. Sekwencje testowe pochodzą z bazy *ChangeDetection* [3]. Tego typu podejście do ewaluacji zaimplementowanych algorytmów zostało wykorzystane między innymi w [13, 21, 22]. Wszystkie metody zostały przetestowane z wykorzystaniem 31 sekwencji testowych podzielonych na 6 różnych kategorii. Zbiór testowy został tak dobrany, aby odwzorować jak największą liczbę sytuacji mogących wystąpić w rzeczywistych warunkach. Każda z kategorii została szczegółowo opisana w rozdziale 5.2. Wszystkie testy przeprowadzono z wykorzystaniem modeli programowych, opisanych w rozdziale 4.2.

Dla każdej sekwencji testowej zawartej w bazie, dostępny jest model wzorcowy tj. ręcznie anotowana maska obiektów (ang. *ground truth*). Wzorzec zapisany jest jako obraz w skali szarości, gdzie piksele przyjmują jedną z pięciu wartości:

0 – tło

50 – cienie

85 – obszar wyłączony z analizy

175 – obszar trudny do zidentyfikowania (np. kontur otaczający ruchomy obiekt)

255 – obiekt pierwszoplanowy

Porównując ramki wyjściowe testowanego algorytmu z odpowiadającymi im ramkami modelu wzorcowego, można wyznaczyć następujące współczynniki:

TP – liczba pikseli poprawnie zakwalifikowanych jako pierwszy plan (ang. *true positive*)

TN – liczba pikseli poprawnie zakwalifikowanych jako tło (ang. *true negative*)

FN – liczba pikseli błędnie zakwalifikowanych jako tło (ang. *false negative*)

FP – liczba pikseli błędnie zakwalifikowanych jako pierwszy plan (ang. *false positive*)

Na podstawie wyznaczonych współczynników oblicza się 7 wskaźników jakości określających dokładność metody:

1. <i>Recall (Re)</i> :	$TP/(TP + FN)$
2. <i>Specificity (Spec)</i> :	$TN/(TN + FP)$
3. <i>False Positive Rate (FPR)</i> :	$FP/(FP + TN)$
4. <i>False Negative Rate (FNR)</i> :	$FN/(FN + TP)$
5. <i>Percentage of Wrong Classifications (PWC)</i> :	$100(FN + FP)/(TP + FN + FP + TN)$
6. <i>Precision (Pr)</i> :	$TP/(TP + FP)$
7. <i>F-measure (F1)</i> :	$2 \frac{P_r * R_e}{P_r + R_r}$

Parametr *Re* definiuje jaki procent pikseli pierwszoplanowych został rozpoznany. Analogiczną wartość dla pikseli reprezentujących tło określa parametr *Se*. Parametry *FPR* i *FNR* są przeciwnieństwem wartości opisanych wyżej i wynoszą odpowiednio $1 - Se$ i $1 - Re$. *PWC* określa procent źle sklasyfikowanych pikseli, natomiast *Pr* informuje jaki procent spośród pikseli sklasyfikowanych jako pierwszoplanowe został rozpoznany prawidłowo. Ostatnim wskaźnikiem jest *F1 (F-measure)*, został wykorzystany, ponieważ według autorów przedstawionej metodologii, jego wartość najlepiej koreluje z wartością średnią wszystkich pozostałych wskaźników.

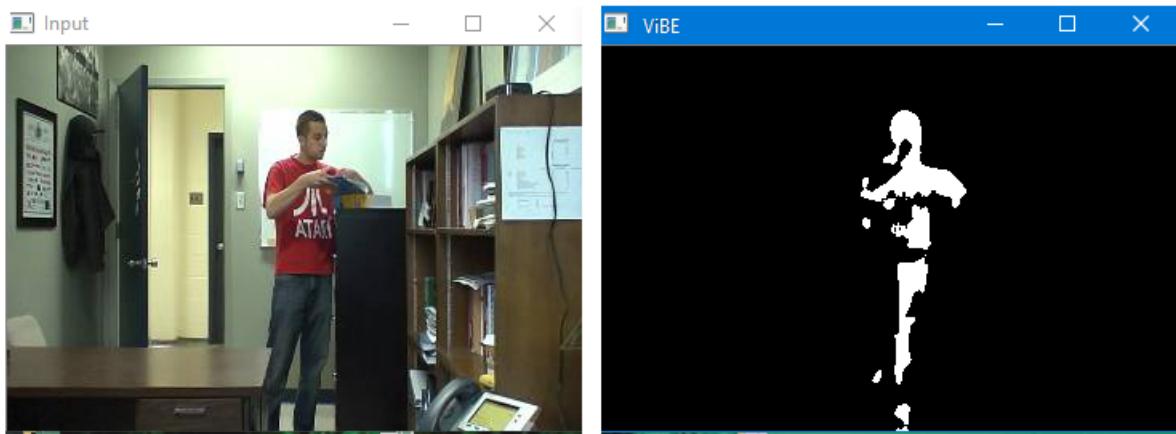
5.2. Szczegółowe testy

5.2.1. Sekwencje podstawowe

W pierwszej kolejności dokonano porównania algorytmów przy użyciu sekwencji testowych z kategorii *Baseline*. Wykorzystano 4 sekwencje, z czego dwie nagrano na otwartej przestrzeni i dwie w zamkniętym pomieszczeniu. Przykładowa sekwencja została pokazana na rysunku 5.1. Jest to zdecydowanie najprostsza spośród wszystkich kategorii i jednocześnie najbardziej zróżnicowana. Dzięki temu może stanowić dobry punkt odniesienia dla dalszych testów. Dwie spośród wykorzystanych sekwencji zawierają głównie obiekty ruchome, natomiast w dwóch kolejnych występują także obiekty zatrzymane. Oprócz tego miejscami pojawiają się cienie oraz dynamiczne tło. Szczegółowe wyniki wykonanych testów przedstawiono w tabeli 5.1.

Spośród prostych algorytmów, zdecydowanie najlepszy wynik uzyskano przy wykorzystaniu metody naiwnej. Odejmowanie ramek oraz średnia krocząca zwracają już zdecydowanie gorsze rezultaty. Jak zostało już wspomniane w części teoretycznej (rozdział 3.1), są to algorytmy nadające się głównie do detekcji obiektów ruchomych, stąd też zdecydowanie gorszy wynik. Dla większości testowanych algorytmów można zaobserwować stosunkowo wysoką wartość wskaźników *Spec* i *Prec* oraz niewielką wartość *PWC*, co oznacza bardzo dobrą eliminację szumów.

Wartym uwagi jest fakt, że dla standardowej wersji algorytmu *PBAS* otrzymano gorsze wartości wskaźników niż dla metody *ViBE*. W testowanych sekwencjach występowały również obiekty zatrzymane, które w przypadku obu wspomnianych metod często zostawały wtapiane w tło, świadczy o tym



Rys. 5.1. Sekwencja *office* z kategorii *Baseline*

Tabela 5.1. Średnie rezultaty uzyskane dla sekwencji z kategorii *Baseline*

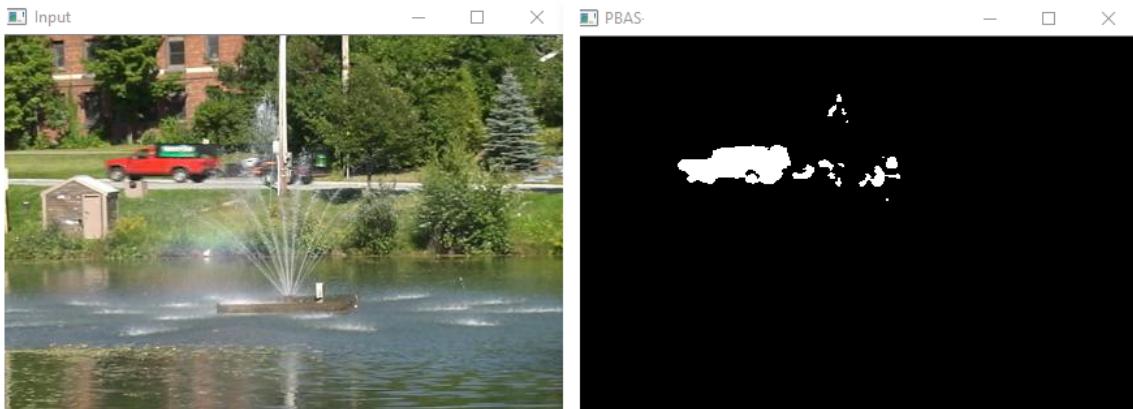
	<i>Recall</i>	<i>Spec</i>	<i>FPR</i>	<i>FNR</i>	<i>PWC</i>	<i>F1</i>	<i>Prec</i>
<i>metoda naiwna</i>	0,8630	0,9973	0,0027	0,1370	0,68	0,8918	0,9237
<i>odejmowanie ramek</i>	0,2567	0,9988	0,0012	0,7433	3,06	0,3521	0,8352
<i>średnia krocząca</i>	0,6594	0,9797	0,0203	0,3406	3,62	0,4828	0,4685
<i>ViBE</i>	0,8871	0,9977	0,0023	0,1129	0,64	0,9126	0,9403
<i>PBAS</i>	0,7457	0,9978	0,0022	0,2543	1,23	0,8084	0,9109
<i>PBAS+</i>	0,9075	0,9965	0,0035	0,0925	0,58	0,8998	0,8932
<i>GMM</i>	0,4711	0,9503	0,0497	0,5289	5,85	0,5266	0,5401

stosunkowo niska wartość wskaźnika *recall*. Problem ten nie został zaobserwowany dla rozszerzonej wersji algorytmu *PBAS* (oznaczonej jako *PBAS+*), w której wykorzystano dodatkowy mechanizm detekcji obiektów statycznych. Wskaźniki jakości otrzymane dla algorytmu *GMM* są z kolei niższe niż dla omówionych wyżej właśnie algorytmów, ale również wyraźnie lepsze niż dla najprostszej metody odejmowania ramek i średniej kroczacej.

5.2.2. Dynamiczne tło i cienie

Kategoria *Dynamic Background* składa się z 6 sekwencji zawierających dynamiczne tło. Występuje ono pod różnymi formami, takimi jak fale na wodzie, fontanna oraz gałęzie z liśćmi poruszające się pod wpływem wiatru. Przykładowy kadr z takiej sekwencji zamieszczono na rysunku 5.2. Druga z omawianych kategorii to *Shadows*. Również zawiera 6 sekwencji, w których głównym utrudnieniem dla algorytmów segmentacji tła są licznie występujące cienie. Otrzymane dla poszczególnych algorytmów rezultaty zamieszczone zostały tabelach 5.2 (kategoria *Dynamic Background*) oraz 5.3 (kategoria *Shadows*).

W przedstawionych algorytmach nie zastosowano żadnych dodatkowych mechanizmów wspomagających wykrywanie dynamicznego tła oraz cieni. Po raz kolejny stosunkowo dobre wyniki, jak na prostotę tego algorytmu, uzyskano dla metody naiwnej. Dla większości algorytmów, podczas testów



Rys. 5.2. Sekwencja *fountain02* z kategorii *Dynamic Background*

Tabela 5.2. Średnie rezultaty uzyskane dla sekwencji z kategorii *Dynamic Background*

	<i>Recall</i>	<i>Spec</i>	<i>FPR</i>	<i>FNR</i>	<i>PWC</i>	<i>F1</i>	<i>Prec</i>
<i>metoda naiwna</i>	0,7994	0,9603	0,0397	0,2006	4,0673	0,3819	0,2866
<i>odejmowanie ramek</i>	0,2841	0,9913	0,0087	0,7159	1,8137	0,2848	0,6162
<i>średnia krocząca</i>	0,6736	0,9605	0,0395	0,3264	4,3561	0,3308	0,2593
<i>ViBE</i>	0,8108	0,9877	0,0123	0,1892	1,3591	0,6652	0,6748
<i>PBAS</i>	0,4749	0,9980	0,0020	0,5251	0,9153	0,5325	0,7776
<i>PBAS+</i>	0,9396	0,9748	0,0252	0,0604	2,5478	0,6187	0,5567
<i>GMM</i>	0,4920	0,9318	0,0682	0,5080	6,5724	0,2459	0,2001

w kategorii *dynamic background* uzyskano wysoką wartość wskaźnika *recall*, co oznacza dokładną detekcję rzeczywistych obiektów pierwszoplanowych. Słaby rezultat, zgodnie z oczekiwaniemi, otrzymano w przypadku metody odejmowania ramek. Wyniki niższe od pozostałych uzyskano również dla algorytmów *GMM* i *PBAS*, z kolei najwyższy wskaźnik otrzymano dla rozszerzonej wersji metody *PBAS+*.

Niestety w większości przypadków uzyskano również zdecydowanie niższą wartość wskaźnika *Prec* i *PWC*, co oznacza, że dynamicznie poruszające się elementy w tle zostały mylnie zaklasyfikowane jako pierwszy plan. Szczególnie widoczne jest to dla metody naiwnej, natomiast najlepiej pod tym względem wypadły algorytmy *ViBE* oraz *PBAS*. Warto zaznaczyć, że rozszerzona wersja metody *PBAS* charakteryzuje się zdecydowanie słabszą zdolnością wyodrębniania dynamicznego tła od statycznego planu, jednak zapewnia lepszą dokładność w wykrywaniu prawdziwych obiektów pierwszoplanowych (wyższa wartość wskaźnika *recall*). Najprawdopodobniej, jest to spowodowane tym, że dynamiczne elementy tła czasami mylnie są identyfikowane jako statyczne obiekty pierwszoplanowe.

Testy w kategorii *shadows* pokazały, że mimo braku osobnego mechanizmu do detekcji i eliminacji cieni, podobnie jak w przypadku poprzedniego testu, wyniki są satysfakcyjne. Również w tym przypadku metoda naiwna zwraca bardzo dobre rezultaty, zarówno pod względem liczby wykrytych obiektów (parametr *recall*) jak i dokładności (parametry *PWC* i *prec*). Ponownie można zaobserwować

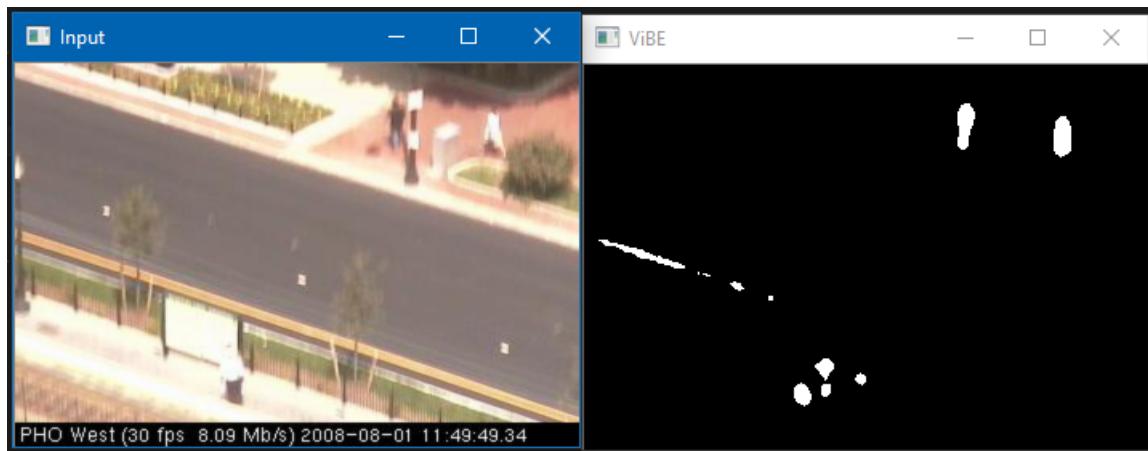
Tabela 5.3. Średnie rezultaty uzyskane dla sekwencji z kategorii *Shadows*

	<i>Recall</i>	<i>Spec</i>	<i>FPR</i>	<i>FNR</i>	<i>PWC</i>	<i>F1</i>	<i>Prec</i>
<i>metoda naiwna</i>	0,8253	0,9686	0,0314	0,1747	3,7359	0,6603	0,6129
<i>odejmowanie ramek</i>	0,1809	0,9970	0,0030	0,8191	3,7981	0,2700	0,7968
<i>średnia krocząca</i>	0,5992	0,9551	0,0449	0,4008	6,0793	0,4864	0,4436
<i>ViBE</i>	0,8738	0,9891	0,0109	0,1262	1,5202	0,8212	0,7884
<i>PBAS</i>	0,6152	0,9889	0,0111	0,3848	2,7448	0,6739	0,7951
<i>PBAS+</i>	0,8590	0,9881	0,0119	0,1410	1,8909	0,7903	0,7479
<i>GMM</i>	0,5261	0,9849	0,0151	0,4739	3,3636	0,6425	0,8312

wyraźną różnicę pomiędzy podstawową i rozszerzoną wersją algorytmu *PBAS* oraz bardzo dobry wynik metody *ViBE*. W przypadku tego testu algorytm *GMM* charakteryzuje się bardzo wysoką precyzją, jednak przy zdecydowanie za niskim poziomie rozpoznawalności obiektów.

5.2.3. Drgania kamery

W kategorii *Camera Jitter* zawierają się 4 sekwencje testowe, w których kamera nieustannie poddawana jest lekkim oraz silniejszym wibracjom. Poziom vibracji jest zróżnicowany dla poszczególnych sekwencji. Przykładową sekwencję zamieszczono na rysunku 5.3, natomiast uzyskane rezultaty przedstawiono w tabeli 5.4.

**Rys. 5.3.** Sekwencja *boulevard* z kategorii *Camera Jitter*

Oprócz testowanych wcześniej algorytmów, w tym przypadku ewaluacji poddano także rozszerzoną wersję metody *ViBE*, zawierającą dodatkowy mechanizm redukcji efektu ruchomej kamery. Niestety, zimplementowane rozwiązanie nie zdało egzaminu i dla wykorzystanych sekwencji testowych uzyskany wynik jest niezadowalający. Według autorów algorytm został przygotowany z myślą o zastosowaniu w kamerach typu *PTZ* (ang. *pan–tilt–zoom camera*). Wymagania systemu działającego z tego typu urządzeniem częściowo pokrywają się z tym czego wymagamy od algorytmu redukującego drgania kamery.

Tabela 5.4. Średnie rezultaty uzyskane dla sekwencji z kategorii *Camera Jitter*

	<i>Recall</i>	<i>Spec</i>	<i>FPR</i>	<i>FNR</i>	<i>PWC</i>	<i>F1</i>	<i>Prec</i>
<i>metoda naiwna</i>	0,6797	0,8546	0,1454	0,3203	15,0577	0,2906	0,1884
<i>odejmowanie ramek</i>	0,4640	0,8746	0,1254	0,5360	14,1708	0,2134	0,1408
<i>srednia krocząca</i>	0,7625	0,8487	0,1513	0,2375	15,2717	0,3205	0,2091
<i>ViBE</i>	0,7510	0,9371	0,0629	0,2490	6,9322	0,5119	0,3964
<i>ViBE+</i>	0,5510	0,8941	0,1059	0,4490	10,8422	0,4264	0,3182
<i>PBAS</i>	0,6523	0,9843	0,0157	0,3477	2,7535	0,6416	0,6382
<i>PBAS+</i>	0,8907	0,8889	0,1111	0,1093	11,0708	0,4072	0,2697
<i>GMM</i>	0,5348	0,9017	0,0983	0,4652	11,7679	0,3925	0,3107

W związku z tym zdecydowano się przetestować zaproponowane rozwiązanie właśnie w takim środowisku, uwzględniając jednak, że docelowe zastosowanie było inne, uzyskanego wyniku nie należy traktować jak niespełniającego wymagań. Pozostałe algorytmy uzyskały niskie wartości wskaźnika *prec* oraz wysoki procent źle sklasyfikowanych pikseli (*PWC*). Jest to naturalny efekt drgań kamery, ponieważ w takim przypadku niemal cały obraz interpretowany jest jako pierwszy plan.

Przygotowana implementacja algorytmu *ViBE*, została początkowo opracowana w Laboratorium Biocybernetyki AGH i po raz pierwszy opublikowana w artykule [22]. Autorzy tej publikacji zaproponowali alternatywny sposób przetestowania przygotowanego rozwiązania. Koncepcja ta zakłada testowanie metody w rzeczywistych warunkach, w trzech różnych sytuacjach oznaczanych dalej jako *C1*, *C2*, *C3*:

C1 – brak obiektów pierwszoplanowych, jedynie ruch kamery

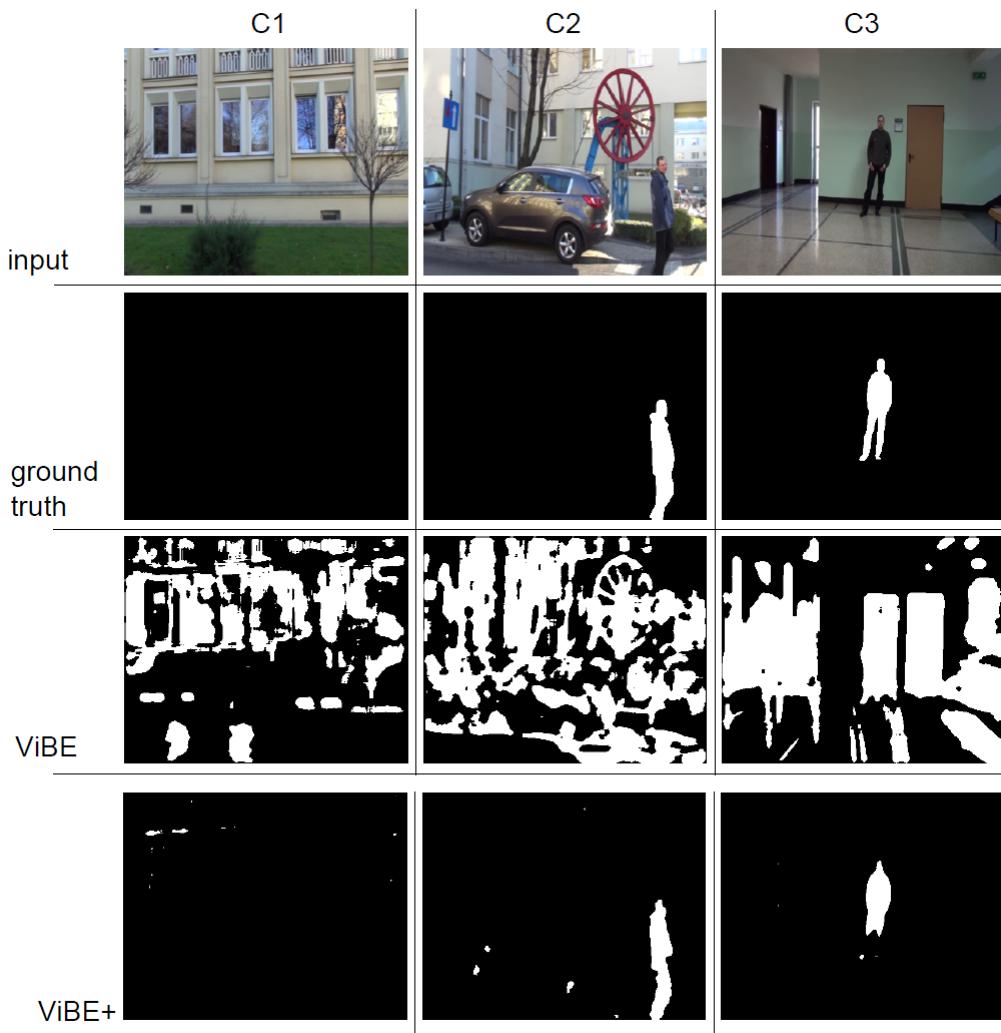
C2 – poruszające się obiekty pierwszoplanowe oraz ruchoma kamera

C3 – pojawia się obiekt zatrzymany, następnie kamera zaczyna się ruszać

Działanie algorytmu w takim środowisku zilustrowano na rysunku 5.4, natomiast otrzymane wyniki z przeprowadzonych w ten sposób testów zamieszczono w tabeli 5.5.

Tabela 5.5. Testy algorytmu *ViBE* w rzeczywistym środowisku – źródło [22]

<i>Kategoria</i>	<i>Algorytm</i>	<i>Recall</i>	<i>PWC</i>	<i>F1</i>	<i>Prec</i>
<i>C1</i>	<i>ViBE</i>	–	36,66	–	0,0
	<i>ViBE+</i>	–	0,07	–	0,0
<i>C2</i>	<i>ViBE</i>	0,73	32,83	0,06	0,03
	<i>ViBE+</i>	0,79	0,75	0,76	0,74
<i>C3</i>	<i>ViBE</i>	0,85	30,75	0,09	0,05
	<i>ViBE+</i>	0,58	0,83	0,72	0,95



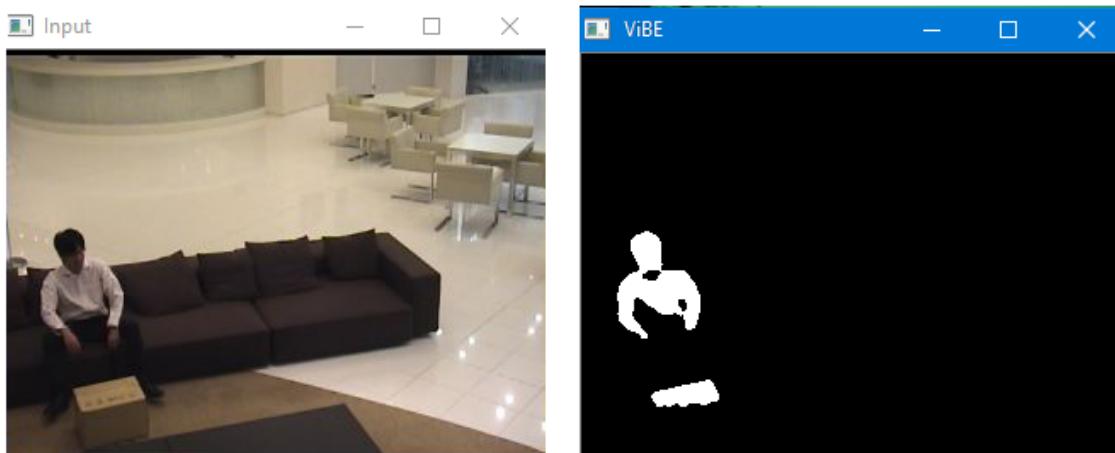
Rys. 5.4. Działanie algorytmu *ViBE* w rzeczywistym środowisku – źródło [22]

W przypadku kategorii *C1* nie występują piksele pierwszoplanowe (porusza się jedynie kamera), dlatego też nie ma możliwości wyznaczenia wskaźników *Recall* i *F1*. Na podstawie uzyskanych wyników można wywnioskować, że rozszerzona wersja algorytmu spełnia swoje zadanie. Świadczy o tym, między innymi, bardzo duży spadek wskaźnika *PWC* oraz wzrost wartości *Prec*. Oznacza to, że efekty drgającej kamery są dobrze niwelowane. Warto zauważyć, że w przypadku kategorii *C3*, spada procent wykrytych obiektów (parametr *recall*) kosztem znacznie lepszej obsługi drgającej kamery. Jest to potwierdzone także przykładowym kadrem z przeprowadzonego testu (rysunek 5.4), gdzie w kategorii *C3*, na masce wyjściowej algorytmu *ViBE+* widoczny jest tylko fragment obiektu.

5.2.4. Obiekty zatrzymane

Kategoria *Intermittent Object Motion* składa się z 6 sekwencji testowych, które zawierają wiele statycznych obiektów pierwszoplanowych. Głównym zadaniem tego testu jest sprawdzenie, czy algorytm eliminuje tzw. „duchy”. Zjawisko występuje między innymi w sytuacji gdy np. samochód zatrzyma się

na chwilę na światłach i następnie ruszy ponownie. Na wykorzystanych sekwencjach występują też momenty, gdy obiekt niespodziewanie zaczyna się poruszać, np. auto odjeżdżające z parkingu lub porzucone pudełko. Przykładowa sekwencja z tej kategorii została pokazana na rysunku 5.5, natomiast wyznaczone wskaźniki jakości zamieszczono w tabeli 5.6.



Rys. 5.5. Sekwencja sofa z kategorii *Intermittent Object Motion*

Tabela 5.6. Średnie rezultaty uzyskane dla sekwencji z kategorii *Intermittent Object Motion*

	Recall	Spec	FPR	FNR	PWC	F1	Prec
metoda naiwna	0,7292	0,8802	0,1198	0,2708	12,1482	0,5184	0,5065
odejmowanie ramek	0,0497	0,9986	0,0014	0,9503	6,2573	0,0869	0,8729
średnia krocząca	0,2402	0,9817	0,0183	0,7598	6,5975	0,2903	0,4795
ViBE	0,6338	0,9055	0,0945	0,3662	10,0766	0,4562	0,4463
PBAS	0,1471	0,9997	0,0003	0,8529	5,6189	0,2389	0,9293
PBAS+	0,8048	0,9868	0,0132	0,1952	2,1968	0,7778	0,7847
GMM	0,3113	0,9629	0,0371	0,6887	11,1263	0,4005	0,6182

Na podstawie uzyskanych wyników można zauważyć, że testowane algorytmy nie dają zbyt dobrych rezultatów. W większości przypadków występuje niska wartość wskaźnika *recall* oraz duża wartość *PWC*, co oznacza, że wiele spośród statycznych obiektów zostało wtopionych w model tła. Ponownie zaskakująco dużą dokładność uzyskano dla metody naiwnej. Otrzymane wskaźniki są na podobnym poziomie co te uzyskane dla algorytmu *ViBE*. Taki rezultat należy jednak traktować jako przypadek, spowodowany faktem, że scena w pierwszej ramce obrazu była pusta. W rzeczywistych warunkach algorytm ten nie zapewnia zadowalających wyników. Metody *PBAS* w wersji podstawowej oraz zawierającej dodatkowy mechanizm detekcji obiektów statycznych (*PBAS+*) zostały porównane niezależnie od reszty. Wyniki zawierające rezultaty obu algorytmów dla poszczególnych sekwencji z tej kategorii zamieszczono w tabeli 5.7.

Tabela 5.7. Średnie wyniki uzyskane dla sekwencji z kategorii *Intermittent Object Motion*

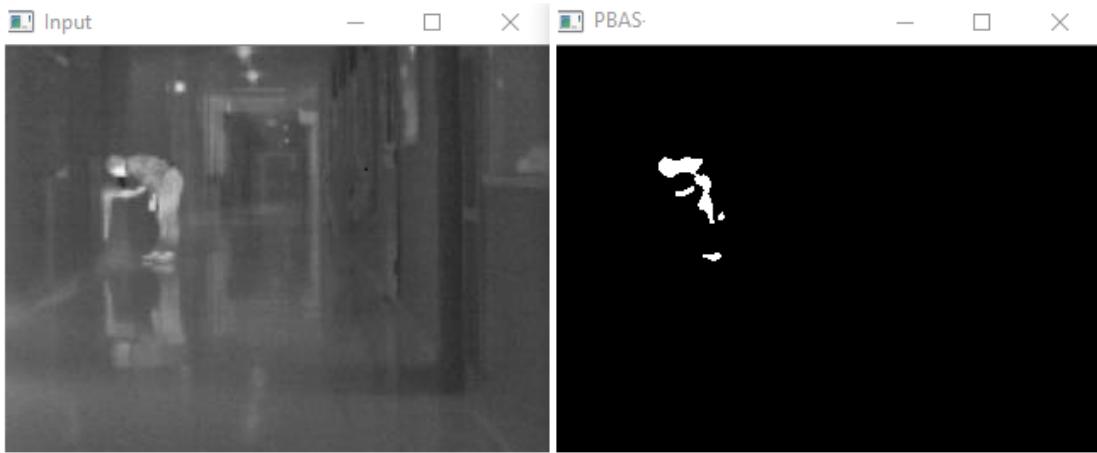
Kategoria	Algorytm	Recall	Spec	FPR	FNR	PWC	F1	Prec
<i>abandonedBox</i>	<i>PBAS</i>	0,2217	0,9991	0,0009	0,7783	3,8330	0,3574	0,9224
	<i>PBAS+</i>	0,9879	0,9739	0,0261	0,0121	2,5427	0,7889	0,6566
<i>parking</i>	<i>PBAS</i>	0,0032	1,0000	0,0000	0,9968	7,7201	0,0064	1,0000
	<i>PBAS+</i>	0,6120	0,9873	0,0127	0,3880	4,1770	0,6941	0,8017
<i>sofa</i>	<i>PBAS</i>	0,2760	0,9994	0,0006	0,7240	3,2187	0,4284	0,9562
	<i>PBAS+</i>	0,5898	0,9967	0,0033	0,4102	2,1043	0,7101	0,8921
<i>streetLight</i>	<i>PBAS</i>	0,0421	1,0000	0,0000	0,9579	4,6503	0,0808	0,9999
	<i>PBAS+</i>	0,9590	0,9999	0,0001	0,0410	0,2093	0,9780	0,9978
<i>tramstop</i>	<i>PBAS</i>	0,2442	0,9998	0,0002	0,7558	13,5829	0,3922	0,9966
	<i>PBAS+</i>	0,9558	0,9685	0,0315	0,0442	3,3774	0,9104	0,8691
<i>winterDriveway</i>	<i>PBAS</i>	0,0954	0,9997	0,0003	0,9046	0,7083	0,1679	0,7008
	<i>PBAS+</i>	0,7245	0,9943	0,0057	0,2755	0,7700	0,5851	0,4906
średnia	<i>PBAS</i>	0,1471	0,9997	0,0003	0,8529	5,6189	0,2389	0,9293
	<i>PBAS+</i>	0,8048	0,9868	0,0132	0,1952	2,1968	0,7778	0,7847

Po dokonaniu analizy otrzymanych wyników można śmiało stwierdzić, że zaimplementowany mechanizm spełnia swoje zadanie. Rezultaty osiągnięte przez algorytm *PBAS+* są zdecydowanie lepsze niż innych algorytmów, dodatkowo widać olbrzymią poprawę w stosunku do standardowej wersji tej metody. Należy zaznaczyć, że w niektórych przypadkach statyczny obiekt umieszczony był daleko od kamery, na drugim planie. Skutkowało to jego niewielkim rozmiarem co stanowiło dodatkowe utrudnienie dla algorytmu. Przykładem takiej sekwencji jest *parking*. Mimo takich utrudnień uzyskany wynik można uznać za zadowalający. Drugą bardzo skomplikowaną do analizy sekwencją jest *sofa*, gdzie za największą trudność należy uznać zbliżoną kolorystykę obiektów. W przypadku sekwencji *winterDriveway* wpływ na gorszy niż w pozostałych testach wynik, miały głównie bardzo trudne warunki pogodowe w których film ten został nagrany.

5.2.5. Kamera termowizyjna

W kategorii *Thermal* zawiera się 5 sekwencji, do których nagrania wykorzystano kamerę pracującą w paśmie podczerwieni. Przykładowy kadr z takiej sekwencji zamieszczono na rysunku 5.6. Głównymi trudnościami w tego typu obrazach jest słaby kontrast pomiędzy obiektem i tłem oraz różnego rodzaju odbicia cieplne np. oknie lub na podłodze. Wyniki przeprowadzonych testów dla poszczególnych algorytmów zamieszczone w tabeli 5.8.

Uzyskane wyniki pokazują, że w przypadku tej kategorii, większą trudnością jest identyfikacja obiektów. Rzadko występują przypadki, kiedy element tła zostanie błędnie zaklasyfikowany jako pierwszy



Rys. 5.6. Sekwencja *corridor* z kategorii *Thermal*

Tabela 5.8. Średnie rezultaty uzyskane dla sekwencji z kategorii *Thermal*

	Recall	Spec	FPR	FNR	PWC	F1	Prec
<i>metoda naiwna</i>	0,6396	0,9936	0,0064	0,3604	1,9062	0,6982	0,8189
<i>odejmowanie ramek</i>	0,0812	0,9997	0,0003	0,9188	6,7849	0,1283	0,9158
<i>średnia krocząca</i>	0,3447	0,9933	0,0067	0,6553	5,8723	0,4238	0,7338
<i>ViBE</i>	0,2772	0,9930	0,0070	0,7228	5,2324	0,3936	0,7555
<i>PBAS</i>	0,2737	0,9980	0,0020	0,7263	5,3672	0,4005	0,9489
<i>PBAS+</i>	0,4236	0,9976	0,0024	0,5764	4,8207	0,5507	0,8981
<i>GMM</i>	0,3353	0,9933	0,0077	0,6647	6,7093	0,5491	0,9134

plan. Świadczy o tym dość wysoka wartość wskaźników *spec* i *prec* dla wszystkich algorytmów. Niestety w większości przypadków nie udało się rozpoznać znacznej ilości obiektów, rezultatem tego są niskie wartości wskaźnika *recall*.

5.2.6. Podsumowanie

Tabela 5.9 zawiera zestawienie uśrednionych wyników dla poszczególnych algorytmów. Dla każdej metody możemy zaobserwować wysoką wartość wskaźnika *spec*, oznacza to dobrą dokładność przy rozpoznawaniu pikseli będących elementami tła. Taki wynik jest przewidywalny, gdyż pikseli reprezentujących tło jest w większości przypadków zdecydowanie więcej niż pierwszoplanowych. Niestety dużo częściej występującym zjawiskiem jest zakwalifikowanie piksela będącego elementem pierwszego planu jako tła, sytuacja odwrotna zdarza się zdecydowanie rzadziej. Dla każdego algorytmu wartość wskaźnika *prec* jest wyraźnie niższa niż wartość *spec*, taki rezultat potwierdza omówione zjawisko.

Tabela 5.9. Średnie wyniki uzyskane dla poszczególnych algorytmów

	<i>Recall</i>	<i>Spec</i>	<i>FPR</i>	<i>FNR</i>	<i>PWC</i>	<i>F1</i>	<i>Prec</i>
<i>metoda naiwna</i>	0,7560	0,9424	0,0576	0,2440	6,2653	0,5735	0,5562
<i>odejmowanie ramek</i>	0,2195	0,9766	0,0234	0,7805	5,9806	0,2226	0,6963
<i>średnia krocząca</i>	0,5466	0,9532	0,0468	0,4534	6,9666	0,3891	0,4323
<i>ViBE</i>	0,7056	0,9683	0,0317	0,2944	4,2938	0,6268	0,6670
<i>PBAS</i>	0,4848	0,9944	0,0056	0,5152	3,1042	0,5493	0,8333
<i>PBAS+</i>	0,8042	0,9721	0,0279	0,1958	3,8507	0,6741	0,6917
<i>GMM</i>	0,4693	0,9402	0,0597	0,5307	6,7255	0,4685	0,6166
<i>FTSG</i> [31]	0,8400	0,9940	0,0060	0,1620	1,1200	0,8400	0,8700
<i>GMM KaewTraKulPong</i> [15]	0,5100	0,9950	0,0500	0,4920	3,1100	0,5900	0,8200
<i>KDE Elgammal</i> [5]	0,7400	0,9760	0,0240	0,2550	3,4600	0,6700	0,6800
<i>KDE Nonaka et al.</i> [32]	0,6500	0,9930	0,0070	0,3490	2,8900	0,6400	0,7700

Jak zostało już wielokrotnie zaznaczone przy opisie poszczególnych kategorii sekwencji testowych, stosunkowo dobre wyniki uzyskano dla metody naiwnej. Otrzymane wartości wskaźniki jakości są zdecydowanie wyższe niż w przypadku pozostałych prostych algorytmów. Zdecydowanie najlepiej z sekwencjami testowymi poradził sobie algorytm *PBAS* z dodatkowym mechanizmem detekcji obiektów statycznych. Uzyskano wysoki procent rozpoznanych obiektów pierwszoplanowych (parametr *recall*), przy zachowaniu dobrej dokładności i niewielkim procencie źle rozpoznanych pikseli (parametry *prec* i *PWC*). Takie wyniki oznaczają, że mechanizm indeksacji, połączony z analizą krawędzi poprawia jakość detekcji nie tylko w przypadku obiektów statycznych.

Kolejną wartą uwagi obserwacją jest fakt, że standardowa wersja algorytmu *PBAS*, osiąga słabsze wyniki od metody *ViBE*, mimo że teoretycznie jest jej bardziej rozbudowaną wersją. Dodatkowe manipulowanie programem przynależności i prawdopodobieństwem aktualizacji nie zawsze wpływa pozytywnie na wynik końcowy. Rozczarowujące są również wyniki otrzymane dla algorytmu *GMM*, szczególnie widoczny jest bardzo wysoki procent źle sklasyfikowanych pikseli *PWC* oraz słaba rozpoznawalność obiektów pierwszoplanowych (parametr *recall*).

Niestety rozszerzona wersja algorytmu *ViBE*, zawierająca dodatkowy mechanizm redukcji ruchu kamery, nie sprawdziła się w testach z wykorzystaniem sekwencji testowych. Autorzy publikacji [22] z której pochodzi metoda zaproponowali alternatywny sposób przetestowania metody w rzeczywistym środowisku, jedynie z wykorzystaniem ruchomej, nie drgającej kamery. Uzyskane w przeprowadzonym przez nich badaniu wyniki dały bardzo obiecujące wyniki.

W porównaniu do metod zaprezentowanych w innych publikacjach, wyniki najlepszego z zaimplementowanych algorytmów (*PBAS+*) można uznać za zadowalające. Zaprezentowana w publikacji [31] metoda *FTSG* osiąga co prawda zdecydowanie lepsze wyniki, jednak należy pamiętać, że zawiera ona

wiele dodatkowych mechanizmów takich jak na przykład wykrywanie dynamicznego tła. Rezultaty wersji *GMM* przedstawionej w [15] są bardzo zbliżone do wyników implementacji, która została opisana w niniejszej pracy.

6. Dalsze kierunki rozwoju

6.1. Poprawa algorytmów

Oprócz standardowych wersji algorytmów, udało się także zaimplementować ich rozszerzone wersje zdecydowanie poprawiając efektywność w specyficznych warunkach. W przypadku algorytmu *ViBE* zaimplementowano moduł, który w znacznym stopniu eliminuje efekt ruchomej kamery. Metoda *PBAS* została natomiast rozszerzona o funkcjonalność detekcji obiektów statycznych. Kolejnym krokiem w rozwoju algorytmów i zwiększania ich efektywności mogłoby być zaimplementowanie metody zawierającej oba te udoskonalenia. Dobrym punktem wyjścia może być w tym przypadku algorytm *PBAS*, zawierający już moduł detekcji obiektów statycznych. Dodanie do takiej implementacji modułu wyliczającego przepływ optyczny powinno zostać wykonane bez dużych komplikacji.

W Laboratorium Biocybernetyki AGH, w ramach pracy inżynierskiej [13] został częściowo zaimplementowany algorytm *FTSG*. Jest to pierwsza implementacja tej metody w układzie reprogramowalnym. Nie udało się niestety opracować pełnej wersji, zaproponowanej przez autorów oryginalnej publikacji [31]. Zabrakło między innymi mechanizmu detekcji obiektów statycznych, który według założeń miał być bardzo podobny do tego wykorzystanego w metodzie *PBAS*. Celem przyszłych badań może być próba wykorzystania opracowanego dla algorytmu *PBAS* modułu analizy obiektów *CCA* i zintegrowania go z algorytmem *FTSG*.

6.2. Wzrost wydajności

Algorytmy przedstawione w niniejszej pracy starano się uruchomić w zarówno niskich (720×576) jak i wysokich (1280×720 , 1920×1080) rozdzielczościach. Zamierzony cel udało się osiągnąć w przypadku podstawowych wersji algorytmu, niezawierających dodatkowych modułów odpowiedzialnych między innymi za redukcję drgań kamery lub detekcję obszarów statycznych. W wielu przypadkach wiązało się to niestety z obniżeniem dokładności algorytmu – na przykład z powodu redukcji modelu tła, bądź konieczności przejścia z przestrzeni kolorów *RGB* do skali szarości.

W przypadku rozszerzonych odmian algorytmów *ViBE* oraz *PBAS*, udało się jedynie zapewnić przetwarzanie w najniższej rozdzielczości. Istotną kwestią jest zatem optymalizacja przygotowanych implementacji sprzętowych. Wraz z postępem technologicznym i pojawianiem się nowych, wydajniejszych układów *FPGA*, należy dążyć do obsługi wyższych rozdzielczości. Docelowo system wizyjny powinien

przetwarzać obraz w rozdzielczości 1920×1080 w 50 klatkach na sekundę, w dalszej przyszłości wymaganym standardem może stać się rozdzielcość $4K$ (3840×2160 pikseli).

Wzrost wydajności systemów wizyjnych jest ograniczony przez kilka czynników. Jak zostało już podkreślone jedną z barier jest aktualnie dostępny sprzęt. W niektórych przypadkach ujawnia się ograniczona liczba zasobów logicznych w układzie *FPGA* oraz maksymalna przepustowość pamięci *RAM*. Jednak, aby zapewnić wzrost wydajności nie należy jedynie oczekwać na postęp technologiczny. Każdy algorytm można w pewnym stopniu zoptymalizować, poprzez uproszczenie logiki i zredukowanie liczby operacji wykonywanych w jednym cyklu. Taki zabieg może co prawda zwiększyć sumaryczną latencję, ale jednocześnie zapewni wyższą maksymalną częstotliwość pracy zegara.

6.3. Implementacja nowych rozwiązań

Mimo stosunkowo zadowalających efektów końcowych przygotowanych implementacji, nadal istnieje wiele kierunków w których algorytmy powinny być rozwijane. W przyszłości należy zastanowić się nad przygotowaniem między innymi dodatkowego mechanizmu detekcji dynamicznego tła i eliminacji jego wpływu na pracę algorytmu. Kolejnym wartym uwagi zagadnieniem jest prawidłowa detekcja cieni i ich odróżnienie od rzeczywistych obiektów. Oba zagadnienia są tematami bardzo rozległymi, które były już częściowo poruszane w ramach badań w Laboratorium Biocybernetyki AGH, jednak nie zostały uwzględnione w niniejszej pracy.

Kolejną drogą rozwoju może być próba tworzenia implementacji sprzętowych istniejących już algorytmów. Autorzy publikacji na różnego rodzaju konferencjach poświęconych systemom wizyjnym prezentują nowatorskie rozwiązania, często jednak nowy algorytm przedstawiany jest jedynie od strony teoretycznej. Mimo, że model programowy przygotowany na komputerze klasy PC daje świetne rezultaty w testach, jego zastosowanie w rzeczywistym systemie wizyjnym jest niemożliwe dopóki nie powstanie dedykowana implementacja sprzętowa. Przykładem takiego działania może być algorytm *Flux Tensor*, którego pierwsza implementacja sprzętowa [14] została przygotowana właśnie w Laboratorium Biocybernetyki AGH.

7. Zakończenie

Wszystkie cele postawione w rozdziale 1.1 udało się zrealizować, zatem można stwierdzić, że badania wykonane w ramach niniejszej pracy zakończyły się sukcesem. Pierwszym krokiem było przygotowanie listy algorytmów, które miały zostać zaimplementowane na docelowej platformie. Zgodnie z założeniami pracy zdecydowano się wybrać metody, będące już częściowo opracowane w ramach badań prowadzonych w Laboratorium Biocybernetyki AGH. Wybrany zestaw zawiera algorytmy prezentujące różnorodne podejścia do zagadnienia detekcji obiektów pierwszoplanowych i segmentacji tła. W stworzonej bibliotece zawarto zarówno bardzo proste obliczeniowo metody jak i bardzo zaawansowane algorytmy, zawierające wiele dodatkowych mechanizmów usprawniających ich działanie.

Kolejnym etapem było przygotowanie modelu programowego dla każdego algorytmu. Wszystkie implementacje przygotowano z wykorzystaniem biblioteki *OpenCV* i uruchomiono na tym samym komputerze klasy PC. Do przetestowania zaimplementowanych rozwiązań wykorzystano bardzo często stosowaną metodologię [7] oraz zbiór sekwencji testowych *Change Detection* [3].

Otrzymane wyniki były momentami zaskakujące, okazało się między innymi, że w wielu przypadkach bardziej zaawansowane algorytmy dają wyraźnie gorsze rezultaty niż o wiele mniej skomplikowane rozwiązania. Niestety, nie każde z przedstawionych podejść okazało się być idealnym rozwiązaniem. Zaproponowany mechanizm niwelowania efektów ruchomej kamery, spróbowano zastosować w sekwencjach testowych z drgającą kamerą, jednak opracowany algorytm nie sprawdził się w takim środowisku. Efekty tego rozwiązania były jednak widoczne podczas działania algorytmu w rzeczywistym warunkach z ruchomą kamerą.

Następny i jednocześnie najbardziej zaawansowany etap pracy to przygotowanie implementacji sprzętowej każdego z algorytmów. Przygotowane modele programowe nie dają możliwości przetwarzania obrazu w czasie rzeczywistym. Maksymalna prędkość analizy obrazu zależy od stopnia złożoności algorytmu i waha się w przedziale od jednej do kilkunastu klatek na sekundę. W związku z powyższym, wszystkie algorytmy zaimplementowano na wspólnej platformie sprzętowej i porównano pod kątem wykorzystania dostępnych zasobów oraz mocy.

Podczas tworzenia implementacji wzorowano się i częściowo wykorzystano moduły stworzone w ramach wcześniejszych prac w Laboratorium Biocybernetyki AGH. Większość metod udało się uruchomić we wszystkich wymaganych rozdzielczościach, od $576p@50fps$ i $1080p@50fps$. W pojedynczych przypadkach ze względu stopień rozbudowania algorytmu, zaprzestano na podstawowej rozdzielczości.

Podsumowując, przygotowana biblioteka zawiera wiele ciekawych i efektywnych algorytmów, a uzyskane wyniki testów są zbliżone do rezultatów innych popularnych metod z rankingu *Change Detection* [3]. Przeniesienie implementacji na układ *FPGA* w niektórych przypadkach było kłopotliwe, między innymi ze względu na występujące ograniczenia sprzętowe. Ostatecznie jednak wszystkie metody udało się zaimplementować i pomyślnie uruchomić. Pomimo osiągniętego sukcesu nadal istnieje wiele możliwości rozwoju stworzonej biblioteki. Jak zostało wspomniane w rozdziale 6, można rozszerzyć algorytmy o dodatkową funkcjonalność jak na przykład, detekcja cieni i eliminacja dynamicznego tła. Kolejnym kierunkiem rozwoju jest optymalizacja i wzrost wydajności oraz dostosowanie wszystkich metod do przetwarzania obrazu w rozdzielcości *Full HD* lub wyższej. Zakres nowych możliwości i dalszych prac jest bardzo szeroki, jednak biorąc pod uwagę przyjęte na początku cele, uzyskane efekty można z pewnością uznać za zadowalające.

Bibliografia

- [1] O. Barnich i M. Van Droogenbroeck. „ViBe: A universal background subtraction algorithm for video sequences”. W: *Image Processing, IEEE Transactions on* 20.6 (2011), s. 9–14 (cyt. na s. 13).
- [2] D. Butler, S. Sridharan i VMJr. Bove. „Real-time Adaptive Background Segmentation. Acoustics, Speech, and Signal Processing”. W: *2003 IEEE Int. Conf.* (2003), s. 49–52 (cyt. na s. 16).
- [3] ChangeDetection. Strona internetowa: <http://www.changedetection.net/> (ostatni dostęp 01.06.2017). 2017 (cyt. na s. 11, 12, 14, 67, 81, 82).
- [4] M. V. Droogenbroeck i O. Paquot. „Background subtraction: Experiments and improvements for ViBe”. W: *Computer Vision and Pattern Recognition Workshops (CVPRW), 2012 IEEE Computer Society Conference on* (2012), s. 32–37 (cyt. na s. 13).
- [5] A. Elgammal i in. „Background and foreground modeling using nonparametric kernel density estimation for visual surveillance”. W: *Proceedings of the IEEE* 90.7 (2002), s. 1151–1163 (cyt. na s. 14, 77).
- [6] M. Genovese i E. Napoli. „FPGA-based architecture for real time segmentation and denoising of HD video”. W: *Journal of Real-Time Image Processing* 8.4 (2013), s. 389–401 (cyt. na s. 14, 36).
- [7] N. Goyette i in. „Changedetection.net: A new change detection benchmark dataset”. W: *Computer Vision and Pattern Recognition Workshops (CVPRW), 2012 IEEE Computer Society Conference on* (2012), s. 1–8 (cyt. na s. 67, 81).
- [8] C. Harris i M. Stephens. „A combined corner and edge detector”. W: *Proceedings of Fourth Alvey Vision Conference* (1988), s. 147–151 (cyt. na s. 24).
- [9] J. Hoffman i in. „Cross-modal adaptation for RGB-D detection”. W: *2016 IEEE International Conference on Robotics and Automation (ICRA)* (2016) (cyt. na s. 15).
- [10] M. Hofmann. „Background segmentation with feedback: The pixelbased adaptive segmenter”. W: *Computer Vision and Pattern Recognition Workshops (CVPRW), 2012 IEEE Computer Society Conference on* (2012), s. 38–43 (cyt. na s. 13).
- [11] Xilinx Inc. *7 Series FPGAs Data Sheet: Overview*. 2017 (cyt. na s. 42).
- [12] Xilinx Inc. *Virtex-7 FPGA Configurable Logic Block - User Guide*. 2016 (cyt. na s. 41).

- [13] P. Janus. *Implementacja algorytmu Flux Tensor with Split Gaussian Models w układzie reprogramowalnym*. Praca inżynierska – AGH Kraków. 2015 (cyt. na s. 14, 33, 36–38, 40, 67, 79).
- [14] P. Janus, K. Piszczeł i T. Kryjak. „FPGA Implementation of the Flux Tensor Moving Object Detection Method”. W: *International Conference on Computer Vision and Graphics* (2016), s. 486–497 (cyt. na s. 14, 36, 80).
- [15] P. KaewTraKulPong i R. Bowden. „An improved adaptive background mixture model for realtime tracking with shadow detection”. W: *European Workshop on Advanced Video Based Surveillance Systems* (2001) (cyt. na s. 77, 78).
- [16] M. Komorkiewicz i T. Kryjak. *Projektowanie struktury układów FPGA, skrypt do ćwiczeń laboratoryjnych*. Akademia Górnictwo-Hutnicza im. Stanisława Staszica w Krakowie. 2014 (cyt. na s. 47).
- [17] D. Kotarba. *Wbudowany system wizyjny do detekcji sylwetek ludzkich z wykorzystaniem podejścia HOG i SVM*. Praca inżynierska – AGH Kraków. 2015 (cyt. na s. 17).
- [18] T. Kryjak, M. Komorkiewicz i M. Gorgoń. „Hardware implementation of the PBAS foreground detection method in FPGA”. W: *Proceedings of the 20th International Conference Mixed Design of Integrated Circuits and Systems (MIXDES)* (2013), s. 479–484 (cyt. na s. 13, 15).
- [19] T. Kryjak, M. Komorkiewicz i M. Gorgoń. „Implementacja algorytmu generacji tła wraz z modelem segmentacji obiektów ruchomych i eliminacji cieni w układach FPGA serii Spartan 6 — Implementation of a background generation algorithm with moving object detection and shadow suppressing in Spartan 6 series FPGA devices”. W: *Automatyka : półrocznik Akademii Górnictwo-Hutniczej im. Stanisława Staszica w Krakowie* (2011), s. 197–217 (cyt. na s. 16).
- [20] T. Kryjak, M. Komorkiewicz i M. Gorgoń. „Real-time background generation and foreground object segmentation for high-definition colour video stream in FPGA device”. W: *Journal of Real-Time Image Processing* (2013), s. 61–77 (cyt. na s. 48, 49).
- [21] T. Kryjak, M. Komorkiewicz i M. Gorgoń. „Real-time Foreground Object Detection Combining the PBAS Background Modelling Algorithm and Feedback from Scene Analysis Module”. W: *International Journal of Electronics and Telecommunications* 60.1 (2014), s. 61–72 (cyt. na s. 14, 15, 26, 28, 31, 47, 59–61, 67).
- [22] T. Kryjak, M. Komorkiewicz i M. Gorgoń. „Real-time Implementation of Foreground Object Detection From a Moving Camera Using the ViBE Algorithm”. W: *Computer Science and Information Systems* 11.4 (2014), s. 1617–1637 (cyt. na s. 13, 20–23, 25, 47, 50, 54, 67, 72, 73, 77).
- [23] T. Kryjak, M. Komorkiewicz i M. Gorgoń. „Real-time implementation of the ViBe foreground object segmentation algorithm”. W: *Computer Science and Information Systems* (2013), s. 591–596 (cyt. na s. 13, 15, 21, 22).
- [24] ModelineDatabase. Strona internetowa: <https://www.mythtv.org/> (ostatni dostęp 01.06.2017). 2017 (cyt. na s. 46).

- [25] OpenCV. Strona internetowa: <http://opencv.org/> (ostatni dostęp 01.06.2017). 2017 (cyt. na s. 11, 42).
- [26] K. Palaniappan i in. „Parallel flux tensor analysis for efficient moving object detection”. W: *Information Fusion (FUSION), 2011 Proceedings of the 14th International Conference on* (2011), s. 1–8 (cyt. na s. 14).
- [27] K. Piszczek. *Implementacja algorytmu Gaussian Mixture Models w układzie reprogramowalnym*. Praca inżynierska – AGH Kraków. 2015 (cyt. na s. 14, 33, 34, 36, 62–64, 92).
- [28] R. Qin i in. „Moving Cast Shadow Removal ased on Local Descriptors”. W: *International Conference on Pater REcognition (ICPR)* (2010), s. 1377–1380 (cyt. na s. 16).
- [29] C. Stauffer i W. Grimson. „Adaptive background mixture models for real-time tracking”. W: *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on* 2 (1999), s. 246–252 (cyt. na s. 14, 33, 34, 36).
- [30] D. Thomas i W. Luk. „FPGA–Optimised Uniform Random Number Generators Using LUTs and Shif Registers”. W: *International Conference on Field Programmable Logic and Applications (FPL)* (2010), s. 77–82 (cyt. na s. 47, 48).
- [31] R. Wang i in. „Static and Moving Object Detection Using Flux Tensor with Split Gaussian Models”. W: *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on* (2014), s. 420–424 (cyt. na s. 14, 15, 36, 40, 77, 79).
- [32] H. Nagahara Y. Nonaka A. Shimada i R. Taniguchi. „Evaluation report of integrated background modeling based on spatio-temporal features”. W: *Computer Vision and Pattern Recognition Workshops (CVPRW), 2012 IEEE Computer Society Conference on* (2012), s. 9–14 (cyt. na s. 14, 77).

A. Spis zawartości płyty DVD

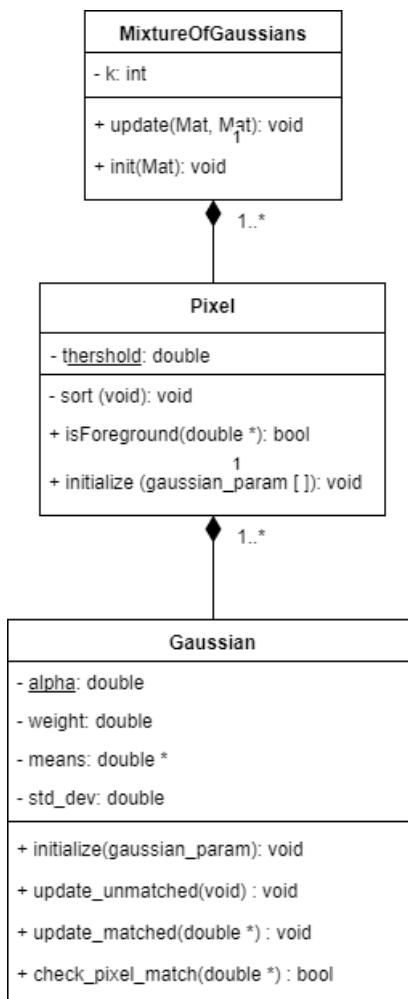
1. Praca zapisana w formacie *PDF*.
2. Praca zapisana w systemie *LATEX*.
3. Wszystkie rysunki użyte w pracy, zapisane w formacie *PNG* lub *PDF*.
4. Sekwencje testowe pobrane ze strony <http://www.changedetection.net>:
 - *Baseline*
 - *Dynamic Background*
 - *Camera Jitter*
 - *Intermittent Object Motion*
 - *Shadows*
 - *Thermal*
5. Projekt w środowisku *Microsoft Visual Studio 2015* zawierający implementację:
 - prostej *metody naiwnej, odejmowania ramek* oraz *średniej kroczącej*
 - algorytmu *GMM*
 - algorytmu *ViBE* wraz z dodatkowym modułem niwelującym ruch kamery
 - algorytmu *PBAS* wraz z dodatkowych mechanizmem wykrywania obiektów statycznych
 - środowiska do przeprowadzania testów i wyznaczania współczynników jakości
6. Projekty w środowisku *Vivado 2017.1* zawierający implementację sprzętową następujących algorytmów (w nawiasach podano nazwę projektu):
 - *metoda naiwnej, odejmowania ramek* oraz *średnia krocząca* (*hdmi_vc707*)
 - *GMM* (*gmm_vc707*)
 - *ViBE* (*vibe_vc707*)
 - *ViBE* w wysokiej rozdzielczości (*vibe_high_res_vc707*)
 - rozszerzona wersja *ViBE* (*vibe_plus_vc707*)
 - *PBAS* (*pbas_vc707*)

- *PBAS* w wysokiej rozdzielczości (*pbas_high_res_vc707*)
- rozszerzona wersja *PBAS* (*pbas_plus_vc707*)

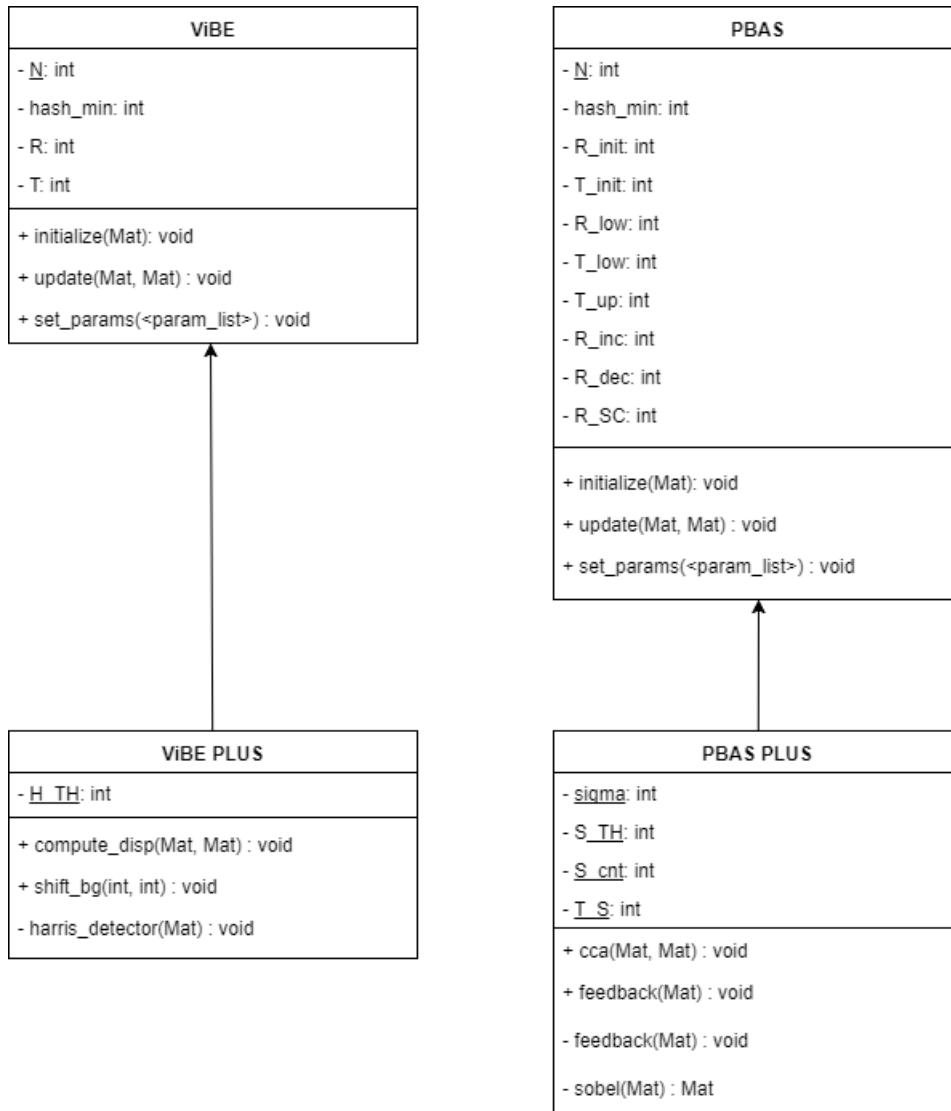
B. Opis informatyczny

Model programowy

Na rysunku B.1 przedstawiono zależności pomiędzy poszczególnymi klasami w implementacji algorytmu *GMM*. Z kolei rysunek B.2 pokazuje architekturę i funkcje zaimplementowane w algorytmach *ViBE* oraz *PBAS*. Wyszczególniono najistotniejsze komponenty, zarówno te o dostępie publicznym (poprzedzone znakiem +), jak i prywatnym (poprzedzone znakiem -).



Rys. B.1. Diagram zależności pomiędzy klasami – algorytm *GMM*



Rys. B.2. Diagram zależności pomiędzy klasami – algorytmy *ViBE* i *PBAS*

Projekty w środowisku Vivado

Moduły użyte w algorytmie *ViBE*

- harrisEdge.v* – moduł realizujący detekcję narożników metodą Harrisza-Stephensa
- medianHistogram.v* – moduł obliczający medianę z zebranych próbek
- model_3d_update.v* – moduł aktualizujący sąsiednie modele tła
- model_update.v* – moduł aktualizujący model tła
- pad_crop.v* – moduł realizujący przesunięcie modelu tła
- pixel_match_test_cielab.v* – moduł realizujący test dopasowania
- rgb2cielab.v* – konwersja do przestrzeni *CIELab*

- update_arbitrator.v*** – moduł decydujący o dokonaniu aktualizacji
vibe_cielab.v – moduł realizujący algorytm *ViBE* w przestrzeni *CIELab*
vibe_cielab_init.v – moduł inicjalizujący model tła
vibe_plus.v – moduł realizujący rozszerzony algorytm *ViBE*
visualize.v – moduł wizualizujący przepływ optyczny

Moduły użyte w algorytmie *PBAS*

- cca.v*** – moduł analizujący obiekty statyczne
cfd.v – moduł realizujący odejmowanie dwóch kolejnych ramek
decision_thr.v – moduł aktualizujący próg decyzji
feedback_ctrl.v – moduł realizujący pętlę sprzężenia zwrotnego
labeller.v – moduł realizujący indeksację jednoprzebiegową
learning_rate.v – moduł aktualizujący współczynnik uczenia
model_3d_update.v – moduł aktualizujący sąsiednie modele tła
model_update.v – moduł aktualizujący model tła
pbas_1c.v – moduł realizujący algorytm dla jednej składowej
pbas_rand_init.v – moduł inicjalizujący model tła
pbas_rgbs.v – moduł realizujący algorytm *PBAS* w przestrzeni *RGB*
pbas_plus_rgbs.v – moduł realizujący rozszerzony algorytm *PBAS*
pbas_gray.v – moduł realizujący algorytm *PBAS* w skali szarości
pixel_match_test_1c.v – moduł realizujący test dopasowania dla jednej składowej
update_arbitrator.v – moduł decydujący o dokonaniu aktualizacji
updateEdgeSimMeasure.v – moduł aktualizujący parametr S_{O_k}
updateStaticObjectHistory.v – moduł aktualizujący parametr EC_{O_k}

Pozostałe moduły

- context_3x3.v*** – moduł generujący kontekst o rozmiarze 3x3
delay_line_bram_wp.v – linia opóźniająca zrealizowana z wykorzystaniem pamięci *BRAM*
delay_x_x.v – linia opóźniająca
median_9x9.v – filtr medianowy 9x9
mem_ctrl.v – kontroler pamięci RAM

- moving_avg.v*** – moduł realizujący algorytm średniej kroczącej
- naive_method.v*** – moduł realizujący metodę naiwną
- rgb2gray.v*** – moduł realizujący konwersję do skali szarości
- rng_128.vhd*** – generator liczb pseudolosowych
- subtract.v*** – moduł realizujący odejmowanie ramek

Lista plików realizujących algorytm *GMM* – źródło [27]

- *background_classifier.v*
- *comparator.v*
- *gaussian_updater.v*
- *match_condition.v*
- *match_found_updater.v*
- *match_tester.v*
- *model_updater.v*
- *no_match_found_updater.v*
- *sorter.v*