

ROZPOZNAWANIE I PRZETWARZANIE OBRAZÓW

ŚLEDZENIE STANU STOŁU BILARDOWEGO

Damian Koper, 241292

Mateusz Gurski, 242089

16 maja 2020

Spis treści

1 Cel projektu	4
1.1 Cele dodatkowe	4
2 Stół bilardowy	4
3 Przepływ danych i architektura	4
3.1 Architektura	6
3.2 Komponent VideoProcessor	6
3.3 Komponent PoolVD	9
4 Detekcja i klasyfikacja	11
4.1 Proces inicjalizacji	11
4.2 Produkty przetwarzania wstępnego	13
4.3 Bile	15
4.4 Kij	21
5 Interfejs użytkownika	23
6 Uruchomienia systemu	26
7 Podsumowanie	27
7.1 Zrealizowane założenia	27
7.2 Możliwości rozwoju	27

Spis rysunków

1 Przepływ danych.	5
2 Diagram klas komponentu VideoProcessor.	7
3 Struktura PoolState.	8
4 Proces przesyłania stanu stołu do komponentu PoolVD	9
5 Obsługa zdarzenia wywołanego przez PoolVD	10
6 Oryginalna klatka pozyskiwana w VideoProcessor	11
7 Uzyskiwana w procesie inicjalizacji maska.	12

8	Wykryty stół	13
9	Uśredniona i wycięta klatka.	14
10	Odpowiednio wycięta aktualna klatka obrazu.	14
11	Różnica aktualnej klatki z klatką uśredzoną w skali szarości.	15
12	Sprobowany obraz przed przeprowadzeniem operacji morfologicznych.	16
13	Sprobowany obraz po przeprowadzeniu operacji morfologicznych.	17
14	Bile znajdujące się daleko od siebie - znalezione przy użyciu FindContours	17
15	Sklejenie konturu gracza i bili znajdującej się obok niego.	18
16	Uzyskana po pierwszym etapie maska z niewykrytą jedną bilą.	19
17	Nałożenie maski na różnice tła i klatki aktualnej.	19
18	Wykryte bile po obu etapach detekcji.	20
19	Fragment utworzonego datasetu.	20
20	Ostateczny produkt uzyskiwany w BallProcesor - wykryte i sklasyfikowane bile. .	21
21	Sprobowany obraz po przeprowadzeniu operacji morfologicznych dla CueProcessor . .	22
22	Wykryty kij.	22
23	Widok stołu w komponencie PoolVD	23
24	Panel opcji w komponencie PoolVD	24
25	Interfejs komponentu PoolVD z testowymi danymi.	25

1 Cel projektu

Celem projektu było stworzenie systemu umożliwiającego śledzenie stanu stołu podczas gry w bilard. Stan stołu obejmuje pozycję i numer bil obecnych na stole, jak i tych znajdujących się w luzach. Obejmuje on również pozycje kija gracza.

1.1 Cele dodatkowe

Dodatkowym celem było rozpoznanie gracza aktualnie wykonującego ruch oraz zliczanie punktów i sygnalizowanie błędów i fauli.

2 Stół bilardowy

Kluczowym elementem projektu jest stół bilardowy. Na etapie projektowania systemu przyjęto następujące założenia dotyczące kamery i stołu:

1. Kamera jest umieszczona nad stołem, prostopadle do płaszczyzny stołu.
2. Stół musi w całości zawierać się w kadrze zgodnie z orientacją kadru. Nie ma znaczenia jednak przesunięcie i obrót obrazu ($\pm 45^\circ$).
3. Stół musi być dobrze i jednolicie oświetlony.
4. Przed rozpoczęciem działania systemu, po zmianie pozycji kamery, system musi wykonać proces inicjalizacji, który wymaga usunięcia wszystkich elementów ruchomych ze stołu.

Ze względu na sytuację epidemiczną na terenie Polski w czasie prowadzenia prac nad projektem, nie było możliwości uzyskania dostępu do żadnego stołu, przeprowadzania i nagrania rozgrywki. Posiłkowano się nagraniami rozgrywek dostępnymi w serwisie YouTube z ustawieniami kamery spełniającymi wymagania[1].

3 Przepływ danych i architektura

Stworzony system składa się z dwóch głównych komponentów:

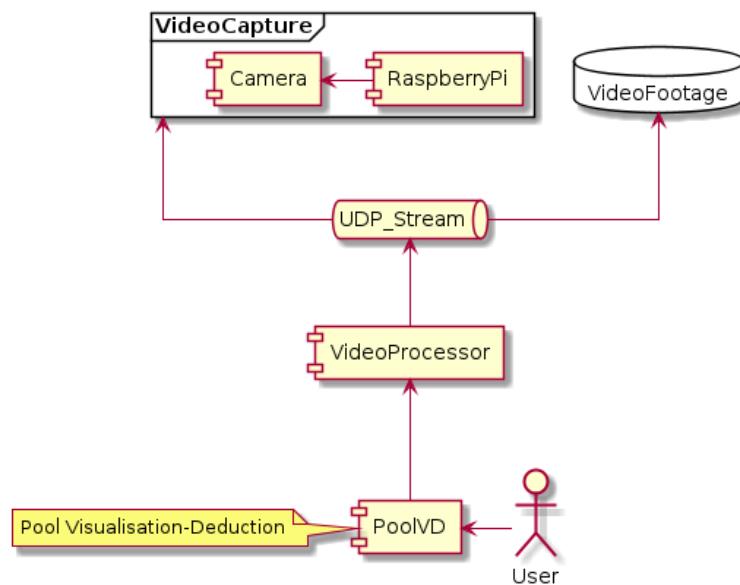
- VideoProcessor
- PoolVD

Komponent *VideoProcessor* zajmuje się przetwarzaniem wideo i komunikacją z komponentem *PoolVD*. Pozyskuje on obraz dostarczany strumieniem używającym protokołu UDP. Nie użyto tutaj protokołu TCP, ponieważ analiza stołu powinna odbywać się w czasie rzeczywistym i rozwiązanie to wprowadziłoby dodatkowy element synchronizacji, który mógłby mieć wpływ na dostarczany obraz.

Źródłem obrazu może być klip wideo wysyłany z użyciem narzędzia *ffmpeg*, albo obraz na żywo z kamery RaspberryPI. Oba te rozwiązania zostały przetestowane. Wykorzystanie strumienia sieciowego jako źródła obrazu jest rozwiązaniem uniwersalnym, ponieważ oferuje stabilny interfejs po stronie komponentu *VideoProcessor*.

Komponent *VideoProcessor* jest równocześnie serwerem obsługującym połączenia zgodne z protokołem WebSocket. Umożliwia to szybkie dostarczanie danych wyjściowych do dowolnej liczby klientów. Umożliwia to również na opartą na zdarzeniach komunikację pomiędzy klientami wyświetlającymi dane, omówioną w dalszej części sprawozdania. Protokół WebSocket, w przeciwieństwie do HTTP do dwukierunkowej komunikacji wykorzystuje zawsze jeden strumień.

Przepływ danych od materiału wideo do użytkownika zobrazowany został na diagramie z rysunku 1.



Rysunek 1: Przepływ danych.

3.1 Architektura

Detekcja, klasyfikacja, wyświetlanie i dedukcja są złożonymi procesami używającymi wiele zasobów sprzętowych. Jednym z filarów sprawnego działania systemu jest sprawny przepływ danych i wielokrotne używanie produktów pośrednich tego przetwarzania. Aby uzyskać płynność przetwarzanego obrazu i optymalne wykorzystanie zasobów wykorzystano podejście wielowątkowe połączone z asynchronicznością.

3.2 Komponent VideoProcessor

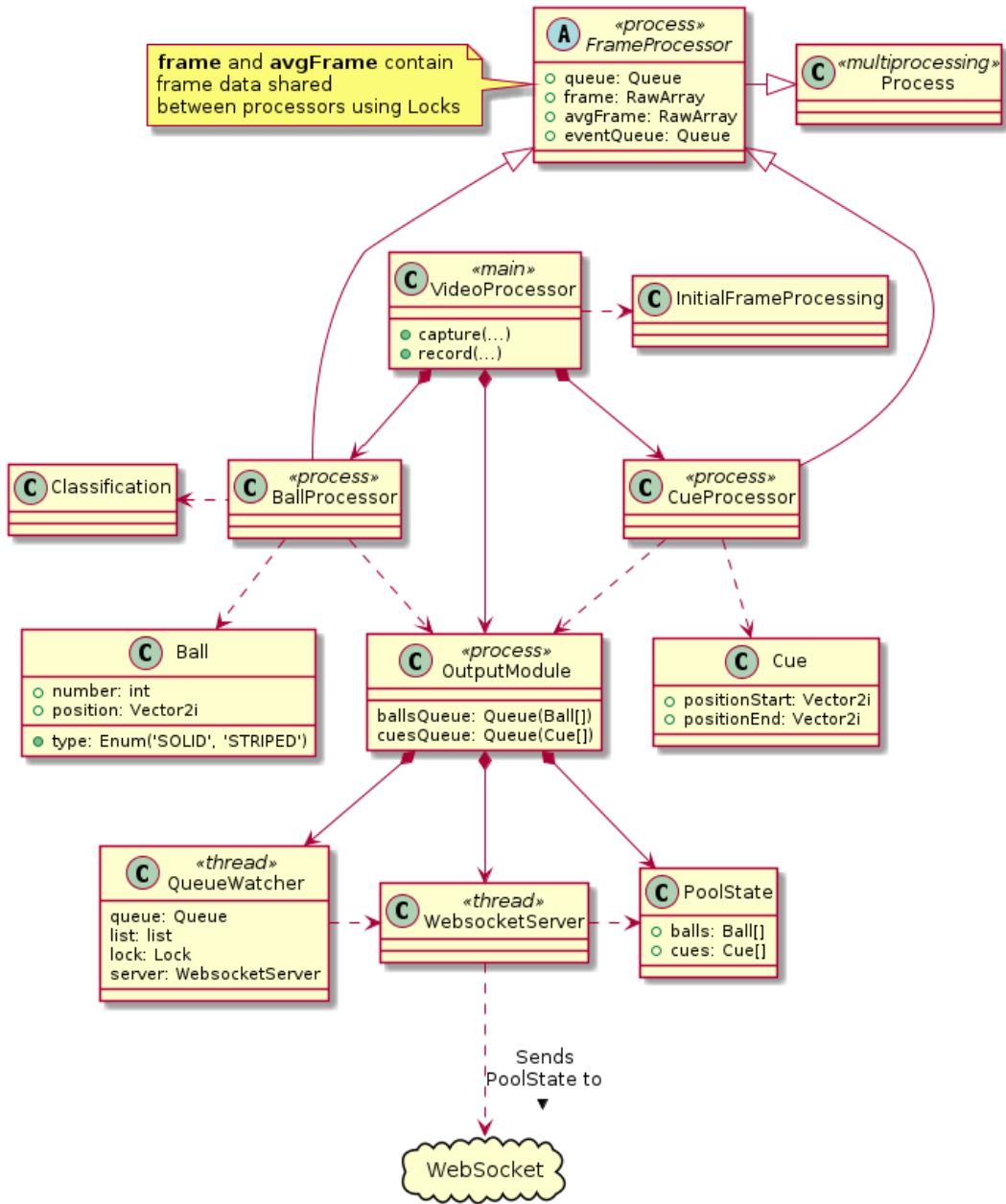
Punktem wejścia programu jest moduł `main`. Przyjmuje on argumenty określające jego działanie - parametry strumienia wejściowego oraz tryb pracy. Może on zostać uruchomiony w trybie domyślnej analizy obrazu lub w celu jego nagrania. Dalszy opis tyczy się trybu analizy obrazu.

Działanie przetwarzania klatki bazuje na działaniu trzech głównych klas - procesów systemowych, nazwanych procesorami. Każdy z nich posiada swój rozbudowany obiekt konfiguracji. Ogólną strukturę klas komponentu przedstawia diagram na rysunku 2.

1. **VideoProcessor**. Nazwa taka sama jak nazwa całego komponentu. Klasa odpowiedzialna za odebranie klatki i oddanie sterowania do klasy `InitialFrameProcessing` w celu wykonania procesu inicjalizacji i wstępnego przetworzenia klatki. Po wstępny przetworzeniu umieszcza ona produkty tego przetwarzania klatki w zmiennych współdzielonych pomiędzy innymi klasami - procesorami.
2. **BallProcessor**. Klasa odpowiedzialna za detekcję i klasyfikację bil. W celu klasyfikacji wykrytych bil klasa ta przekazuje sterowanie do klasy `Classification`.
3. **CueProcessor**. Klasa odpowiedzialna za detekcję kija.

Procesy **BallProcessor** i **CueProcessor** przed rozpoczęciem przetwarzania oczekują na klatkę, jeśli nie jest ona dostępna. Dzięki funkcjonowaniu zmiennych współdzielonych, w których przekazywane są klatki przez **VideoProcessor**, oba procesory w momencie rozpoczęcia przetwarzania pobierają zawsze aktualne dane. Pozwala to na przetwarzanie niezależnie od wydajności algorytmów w poszczególnych klasach.

Po zakończeniu przetwarzania każdej klatki **BallProcessor** i **CueProcessor** wysyłają wynik swoich działań w postaci obiektów `Ball` lub `Cue` do kolejek modułu `OutputModule`.

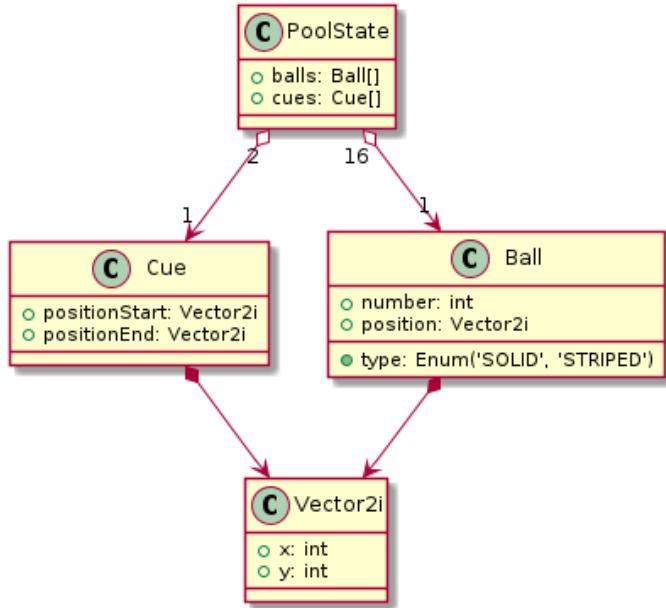


Rysunek 2: Diagram klas komponentu VideoProcessor.

3.2.1 OutputModule

OutputModule jest osobnym procesem, który obsługuje trzy procedury:

1. Serwer komunikacji zewnętrznej - **WebSocketServer**.
2. Obserwator kolejki wykrytych bili.
3. Obserwator kolejki wykrytych kijów.



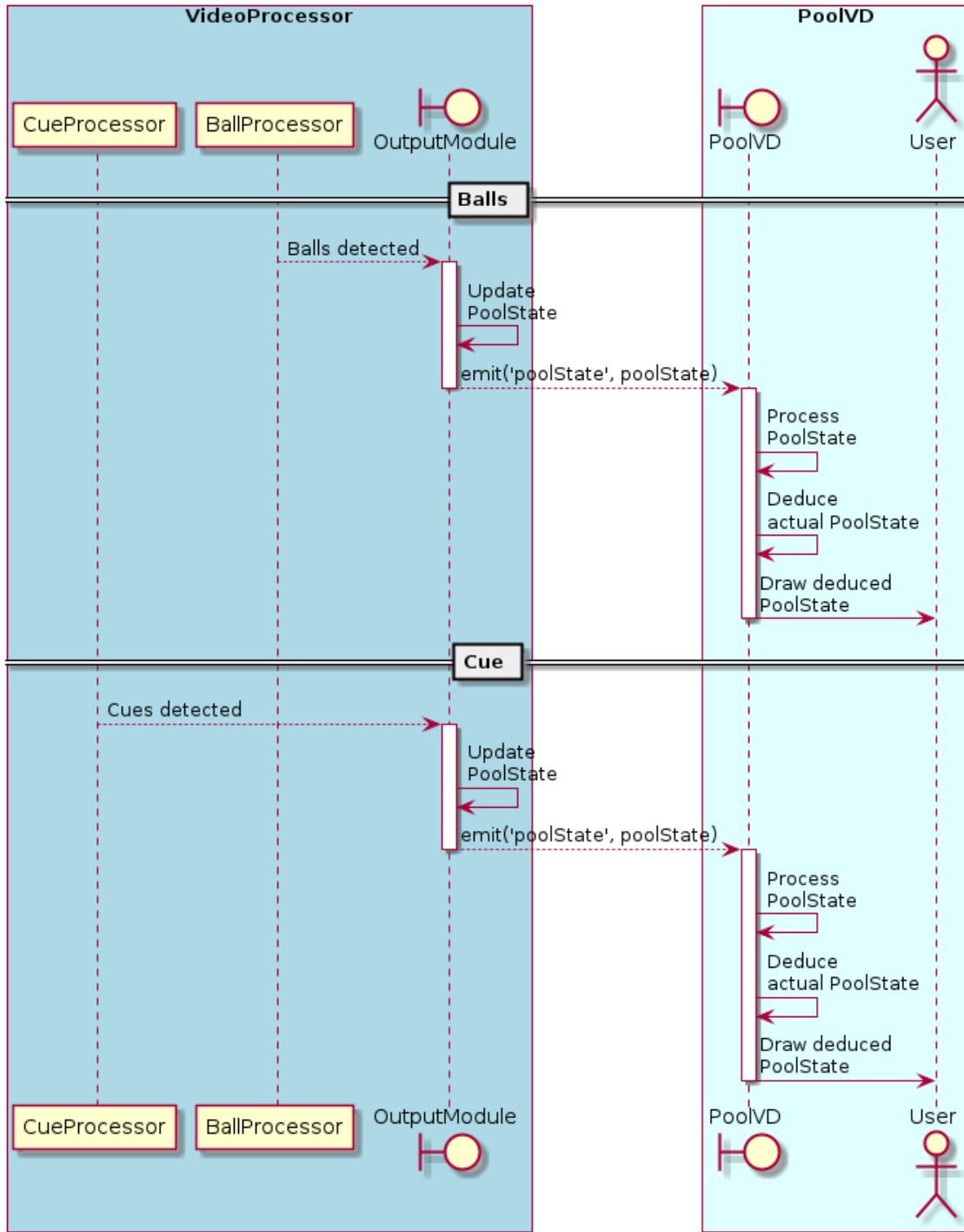
Rysunek 3: Struktura PoolState.

Moduł zawiera również ostatni aktualny wykryty stan stołu. Strukturę te prezentuje diagram na rysunku 3. Obserwatory obserwujące kolejkę, po umieszczeniu w nich danych przez moduły **BallProcessor** i **CueProcessor** aktualizują obiekt **PoolState** i wymuszają wysłanie stanu stołu do klientów przez serwer.

Protokół WebSocket umożliwia komunikację full-duplex w dowolny sposób. W celu ustanowienia komunikacji pomiędzy klientami, a serwerem zdecydowano się wybrać zorientowaną zdarzeniowo bibliotekę Socket.IO [14].

Proces przesyłania stanu stołu do komponentu **PoolVD** obrazuje diagram na rysunku 4. Obsługę zdarzenia zmiany opcji czasu trwania okresu inicjalizacji z poziomu interfejsu użytkownika prezentuje diagram na rysunku 5.

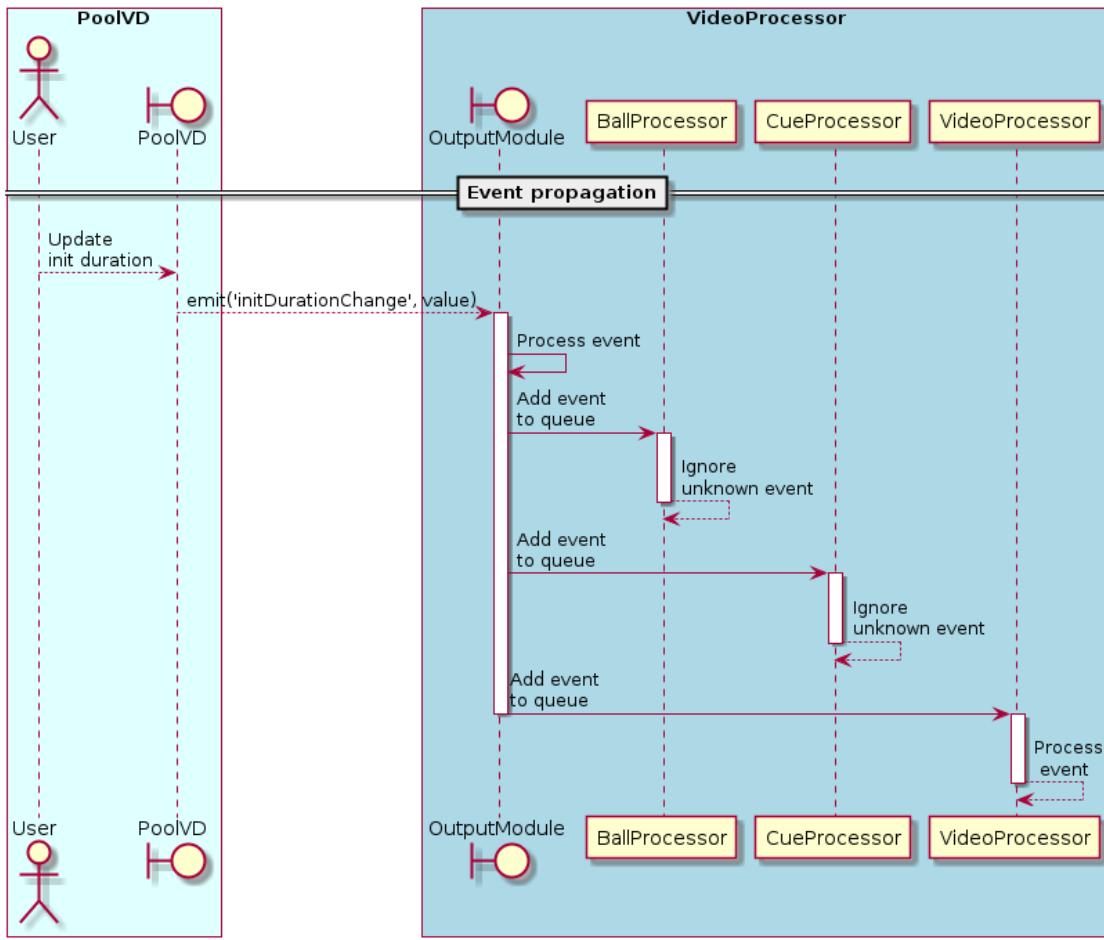
Każda z klas obsługujących zdarzenia ma dostęp do swojej kolejki, w której **OutputModule** umieszcza przychodzące obiekty zdarzeń. Klasa obsługuje tylko takie zdarzenia, których obsługa została przewidziana i ignoruje nieznane. Rozwiążanie to pozwala na łatwe rozszerzenie konfiguracji i zachowania każdej z klas bez żadnego wpływu na inne klasy - wystarczy tylko zaimplementować odbiór i obsługę nowego zdarzenia.



Rysunek 4: Proces przesyłania stanu stołu do komponentu **PoolVD**.

3.3 Komponent PoolVD

Komponent **PoolVD** odpowiada za wyświetlanie otrzymanego stanu stołu, odrzucanie stanów nieprawdopodobnych i śledzenie wpadnięć bili do luz. Wysyła on również zdarzenia do komponentu **VideoProcessor**, które aktualizują parametry detekcji i wywołują zdefiniowane akcje.



Rysunek 5: Obsługa zdarzenia wywołanego przez PoolVD.

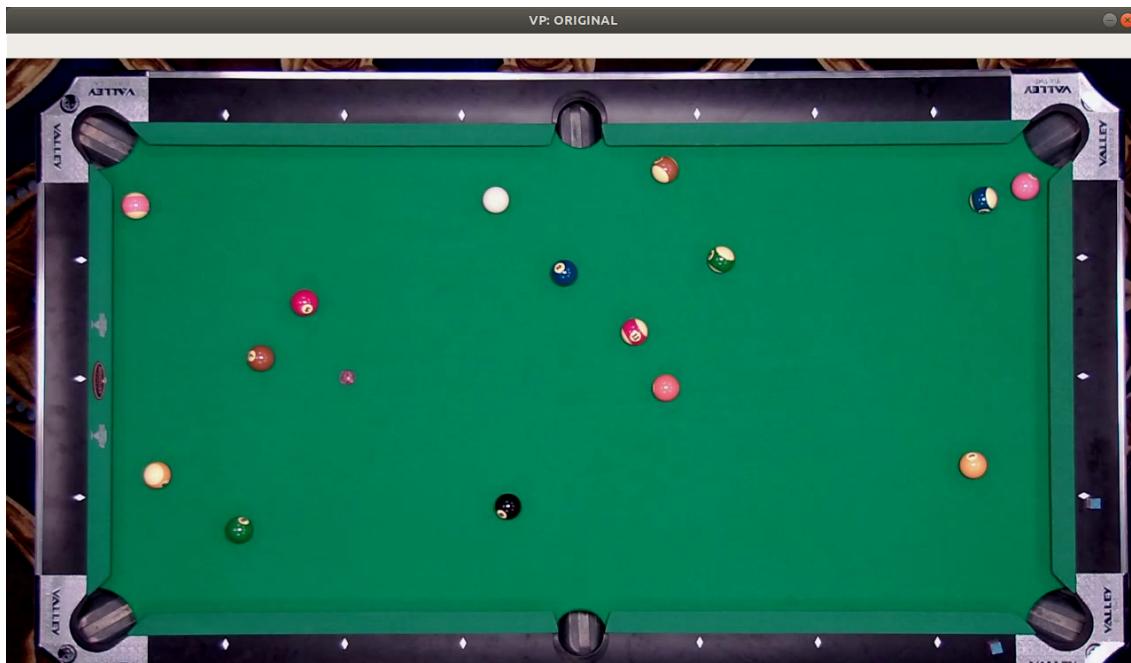
Komponent tworzy historię stanów stołu. Na tej podstawie wykonuje następujące procedury dla każdej bili:

1. Wyświetlanie ostatniego otrzymanego stanu. Eliminuje to tymczasowe zniknięcie bili spowodowane błędem detekcji, klasyfikacji, albo zasłonięciem bili przez inny obiekt.
2. Odrzucenie stanów nieprawdopodobnych. Na podstawie stanów poprzednich, przychodzący stan bili może być odrzucony, jeśli bila przemieściła się nienaturalnie szybko w inną część stołu. Odrzuca to pojedyncze błędy klasyfikacji. Heurystyką używaną w tym wnioskowaniu jest prędkość średnia bili uzyskana na podstawie poprzednich stanów.
3. Wykrycie wpadnięcia bili do luzy. Bila jest uznana za wpadniętą do luzy jeśli przez konfigurowalną liczbę stanów była niewykryta, a jej ostatnia znana pozycja była pozycją blisko danej luzy.

Każda z procedur ma konfigurowalną precyzję podaną jako liczbę stanów.

4 Detekcja i klasyfikacja

Cały proces detekcji i klasyfikacji rozpoczyna się w **VideoProcessor** gdzie pozyskiwany jest obraz (rysunek 6) i wykonywany jest wstępny proces jego przetwarzania. Przetwarzanie obrazu odbywa się z wykorzystaniem biblioteki OpenCV [6].



Rysunek 6: Oryginalna klatka pozyskiwana w **VideoProcessor**.

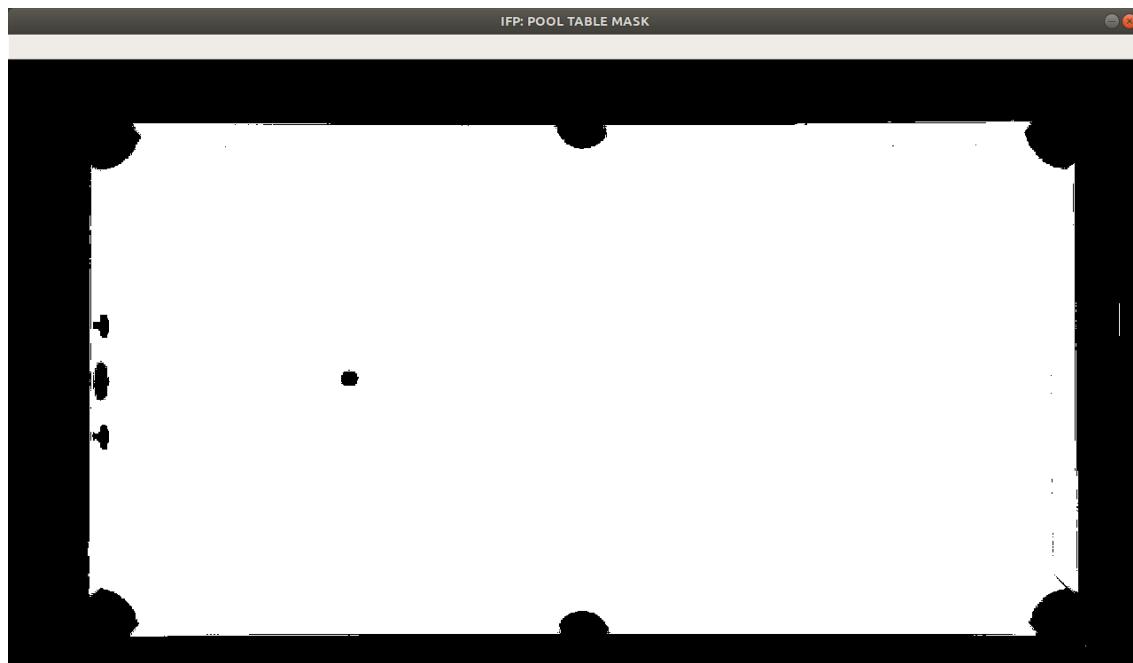
4.1 Proces inicjalizacji

Proces inicjalizacji powinien odbywać się na pustym stole, przed rozpoczęciem gry. Jego głównym celem jest uzyskanie uśrednionej klatki obrazu zawierającej pusty stół, który w kolejnych etapach odejmowany jest od aktualnej klatki obrazu w celu wyizolowania istotnych z punktu widzenia detekcji elementów – bil i kijów. Proces inicjalizacji zwraca obraz tła – uśredniony pusty stół. Ustala również sposób wycinania i obracania kolejnych klatek. Dane potrzebne do dalszego przetwarzania dostępne są już po przeanalizowaniu jednej klatki, jednak dłuższy czas jego trwania pozwala na uzyskanie lepiej uśrednionej klatki i lepszej macierzy transformacji.

Etapy przetwarzania podczas procesu inicjalizacji to kolejno:

1. Uśrednianie klatki
2. Detekcja stołu (na podstawie uśrednionej klatki).
3. Polepszanie macierzy transformacji (na podstawie uśrednionej klatki).

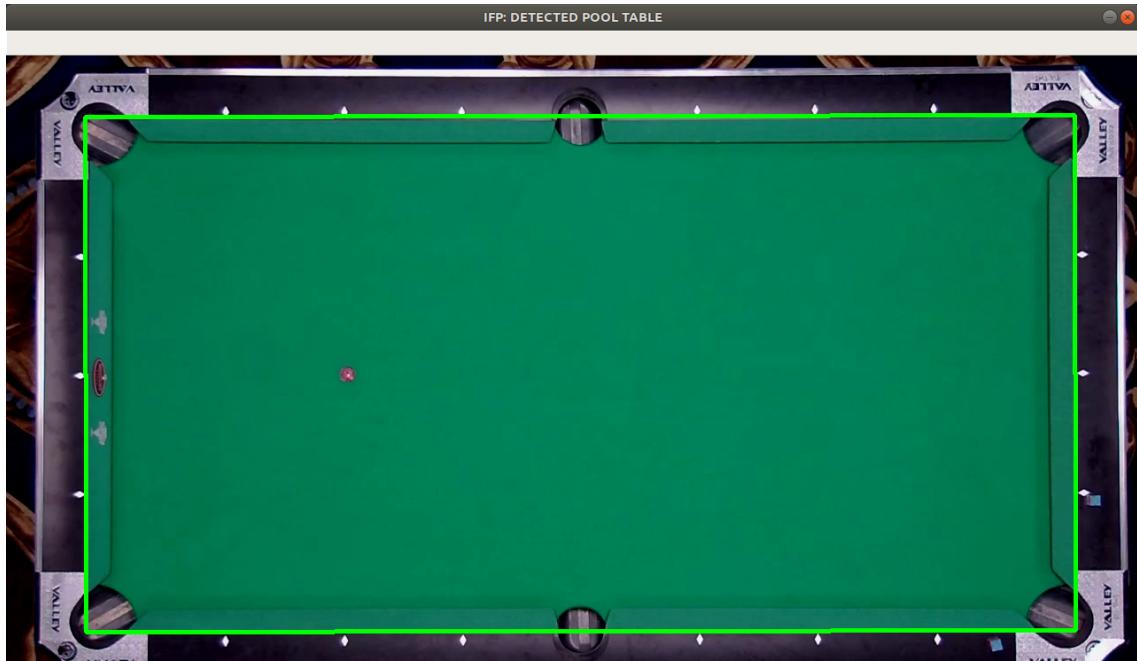
Uśredniona klatka konwertowana jest na przestrzeń kolorów HSV (hue, saturation, value), gdzie Hue reprezentuje odcień, Saturation reprezentuje intensywność, a Value reprezentuje jasność koloru. Ponieważ odcień koloru modelowany jest tylko przez kanał Hue, segmentacja części obrazu na podstawie koloru łatwiejsza jest w przestrzeni HSV niż w przestrzeni RGB, gdzie kolory kodowane są przy użyciu kanałów właściwych dla składowej czerwonej, zielonej i niebieskiej. Po konwersji uśrednionej klatki na przestrzeń barw HSV, tworzona jest maska, na podstawie dolnej i górnej granicy zakresu wartości pikseli w przestrzeni kolorów HSV. Granice te konfigurowalne są z poziomu interfejsu użytkownika. Uzyskana maska przedstawiona jest na rysunku 7.



Rysunek 7: Uzyskiwana w procesie inicjalizacji maska.

Przy użyciu funkcji **findContours**[7] w otrzymanej masce znajdywany jest największy kontur, do którego zostaje dopasowany prostokąt, co przedstawione zostało na rysunku 8.

Granice znalezionej prostokąta, a dokładniej 4 pary współrzędnych jego rogów, wykorzystywane są do utworzenia macierzy transformacji przy użyciu metody **getPerspectiveTransform**[8]. Macierz ta polepszana jest z czasem trwania procesu inicjalizacji ze względu na uśrednianie klatki,

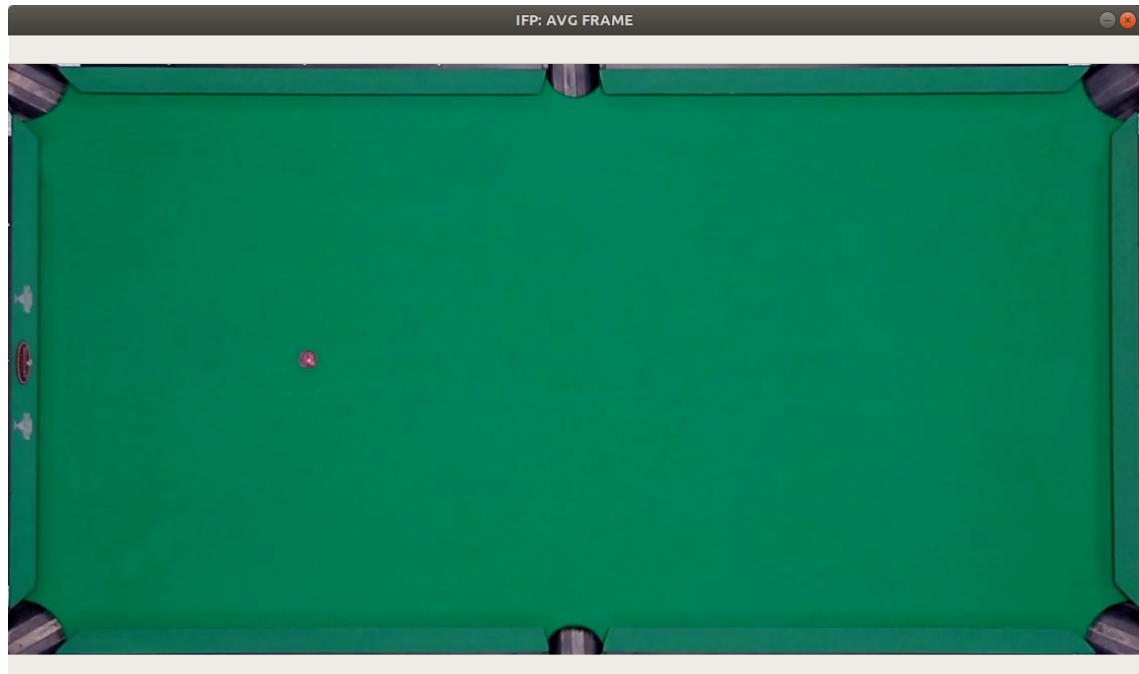


Rysunek 8: Wykryty stół.

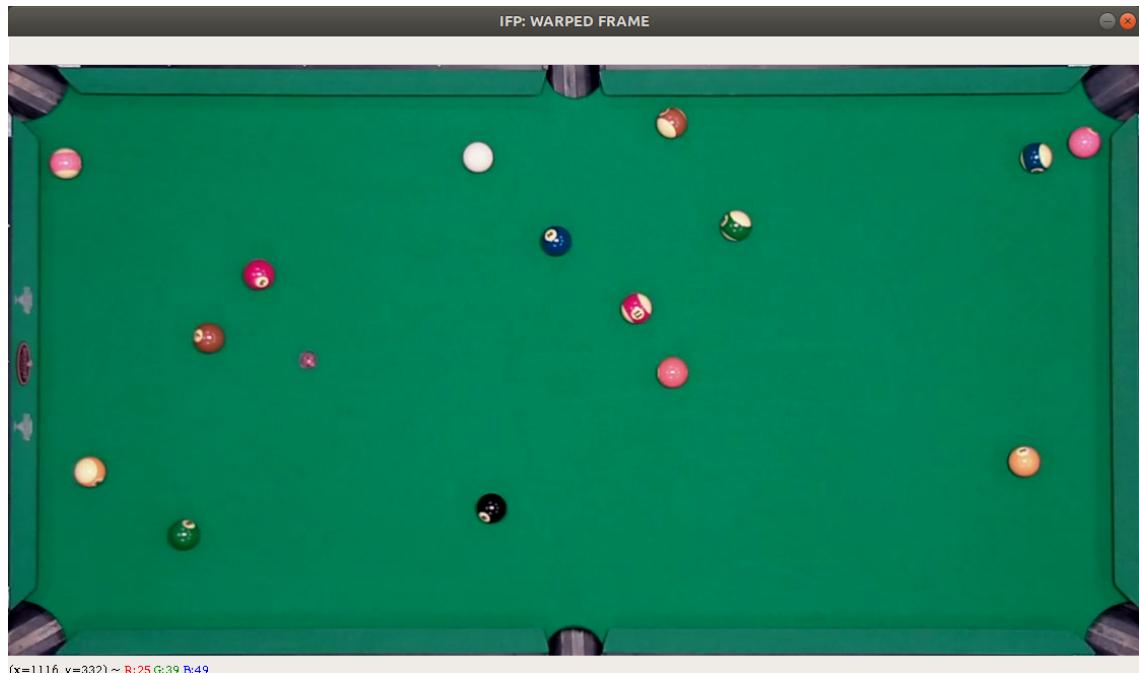
które powoduje uzyskiwanie lepszych wartości współrzędnych rogów stołu. Utworzona macierz transformacji pozwala na wykonanie transformacji perspektywicznej w celu uzyskania odpowiednio wyciętego i obróconego stołu wypełniającego całą ramkę. W tym celu wykorzystana została metoda **warpPerspective**[12].

4.2 Produkty przetwarzania wstępnego

Produktami przetwarzania wstępnego są klatka uśredniona przedstawiona na rysunku 9 oraz odpowiednio wycięta aktualna klatka obrazu przedstawiona na rysunku 10. Obie te klatki przekazywane są dalej do kolejnych procesorów – **BallProcessor** i **CueProcessor**, a **VideoProcessor** zajmuje się dalej, nie czekając, przetwarzaniem kolejnej klatki obrazu.



Rysunek 9: Uśredniona i wycięta klatka.

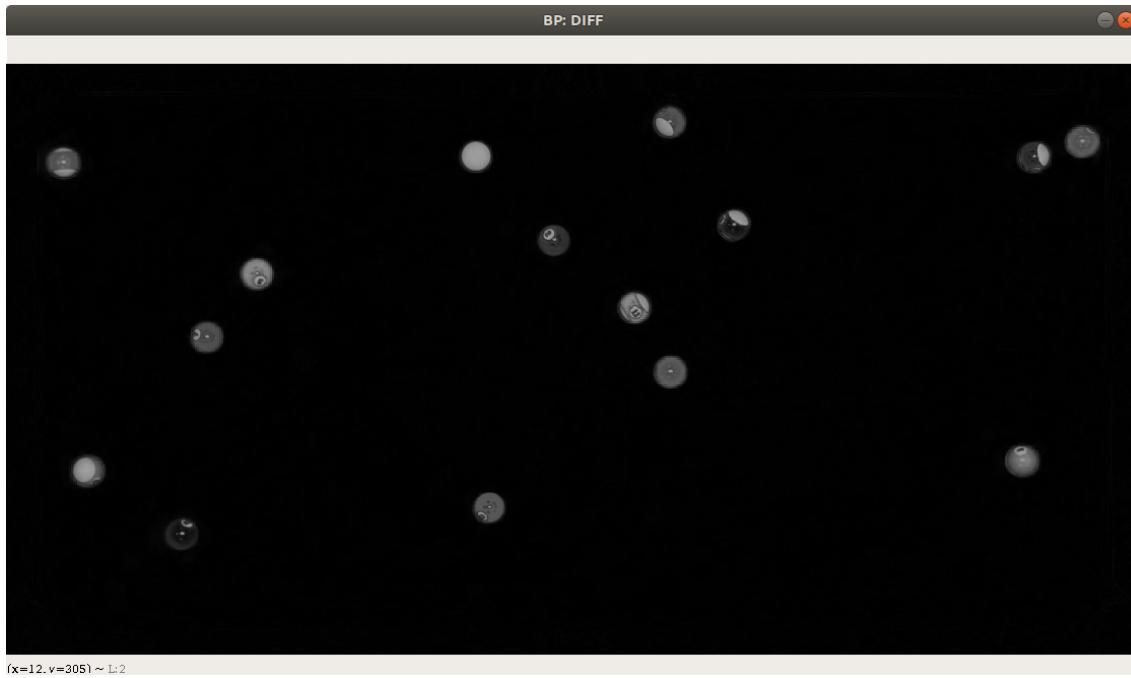


Rysunek 10: Odpowiednio wycięta aktualna klatka obrazu.

4.3 Bile

4.3.1 Detekcja

Od odpowiednio wyciętej aktualnej klatki obrazu odejmowane jest tło oraz wykonywana jest konwersja do skali szarości. Uzyskiwany w ten sposób efekt przedstawiony został na rysunku 11.



Rysunek 11: Różnica aktualnej klatki z klatką uśrednioną w skali szarsości.

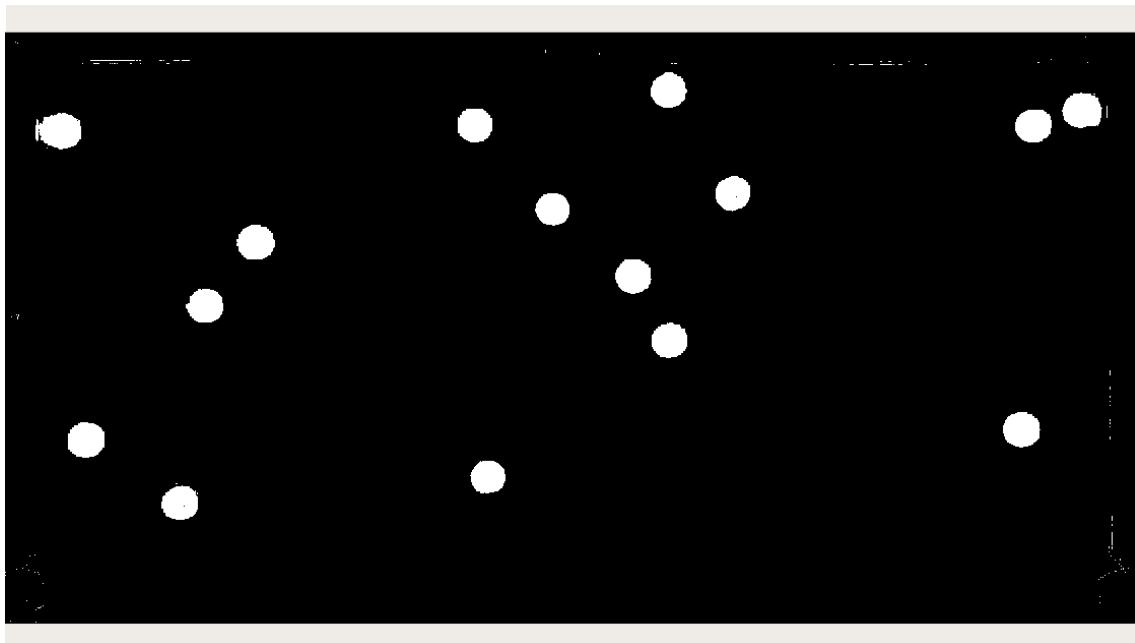
W celu uzyskania maski wykonywane jest progowanie (wartość progowa konfigurowalna jest dynamicznie z poziomu interfejsu użytkownika). W celu uwydawnienia istotnych elementów – w tym przypadku bil, oraz usunięcia szumów, wykonywane są również operacje morfologiczne.

Przetestowane zostały różne kombinacje i częstotliwości operacji otwarcia, domknięcia oraz samych erozji, które powodują usunięcie tych punktów obrazu binarnego, których sąsiedztwo opisane przez podany kernel nie jest w każdym przypadku równe 1, i dylatacji, działającej odwrotnie. Wartość piksela zmieniana jest na 1 gdy przynajmniej 1 piksel w sąsiedztwie jest równy 1. Ostatecznie operacjami, które pozwalają uzyskać najlepszy w tym przypadku efekt przy utrzymaniu odpowiedniej wydajności są operacje:

1. otwarcia – erozji, po której następuje dylatacja, która dobrze sprawdza się do usuwania szumów z obrazu.

2. domknięcia – dylatacji, po której następuje erozja. Przydatna do zapełniania/domykania konturów.

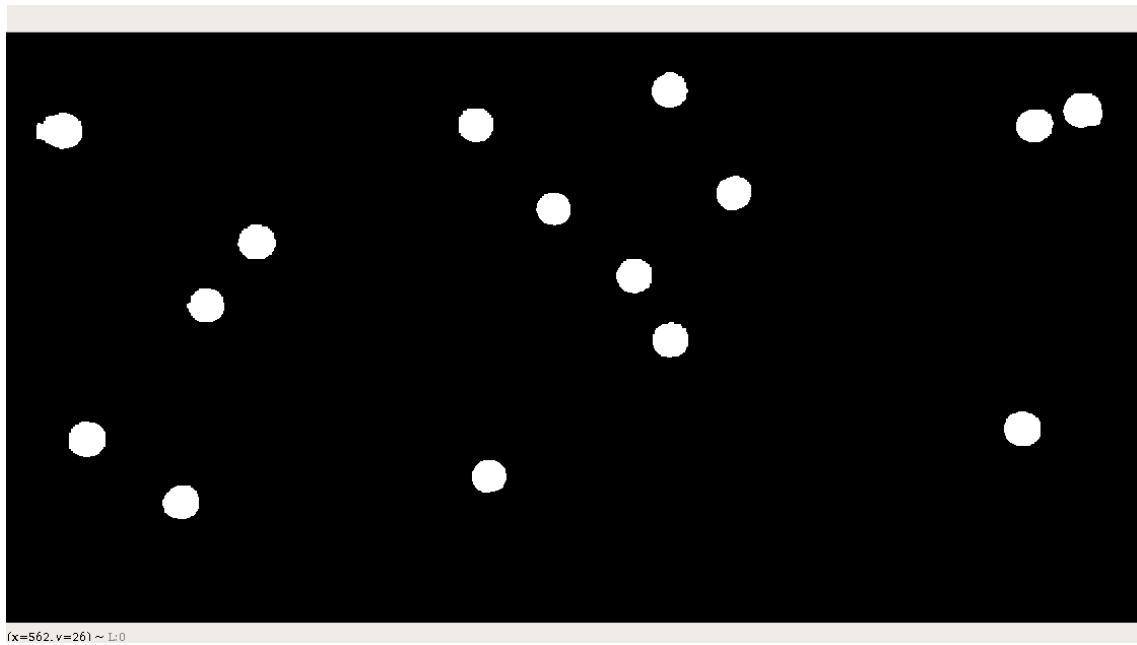
Różnice między sprogowanym obrazem przed i po operacjach morfologicznych przedstawione zostały na rysunkach 12 i 13. W celu zastosowania opisanych operacji morfologicznych wykorzystana została metoda **morphologyEx**[11].



Rysunek 12: Sprogowany obraz przed przeprowadzeniem operacji morfologicznych.

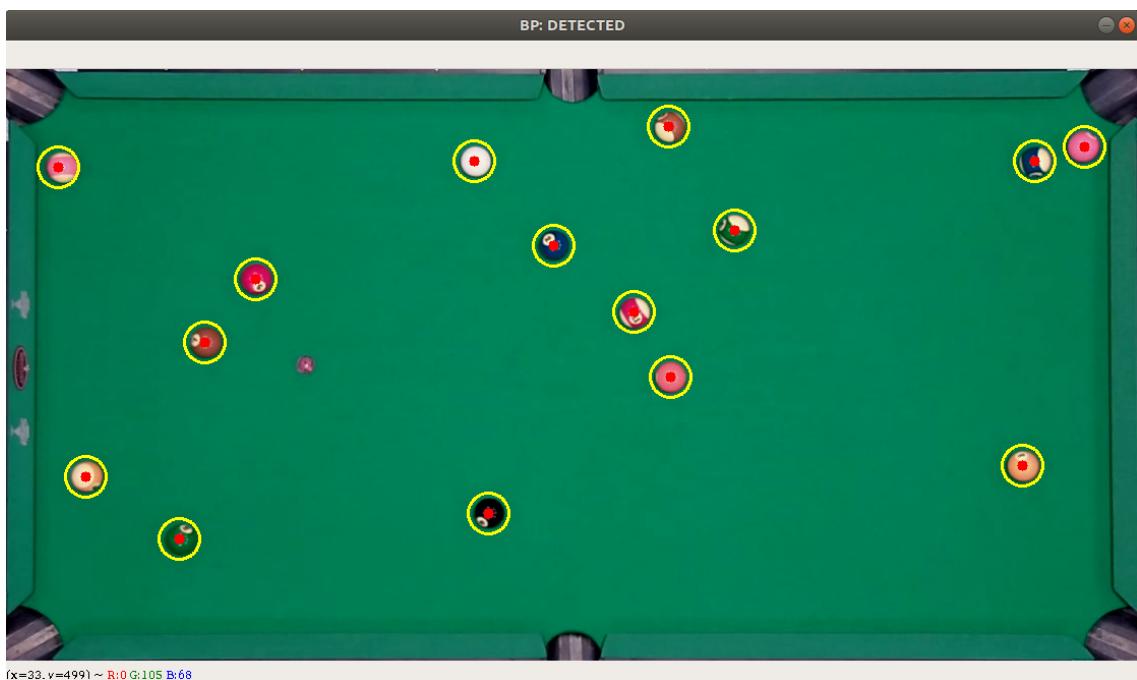
Detekcja bil odbywa się dwuetapowo wykorzystując połączenie dwóch metod – **FindContours** [7] oraz **HoughCircles** [9].

Pierwszym etapem detekcji jest metoda **FindContours**, która po odpowiednim dostrojeniu pozwala na stosunkowo szybkie wykrycie konturów bil. Dostrojenie tej metody w przypadku wykrywania bil, polega na sprawdzaniu, czy znalezione kontury spełniają warunki kolistego kształtu oraz czy długość ich promienia znajduje się w odpowiednim przedziale. Metoda **FindContours** okazała się być bardziej wydajna i stabilna jeśli chodzi o wykrywane współrzędne od metody **HoughCircles**. Doskonale radzi sobie ona z wykrywaniem bil, które znajdują się daleko od siebie, co przedstawiono zostało na rysunku 14. Nie generuje też ona fałszywych detekcji, co ma miejsce w przypadku metody **HoughCircles**. Nie radzi ona sobie jednak w przypadku bil znajdujących się blisko siebie lub blisko gracza – co powoduje sklejenie ich konturów, co widać na rysunku 15 i tym samym uniemożliwia ich wykrycie. Dla takich przypadków powstał drugi etap detekcji, czyli

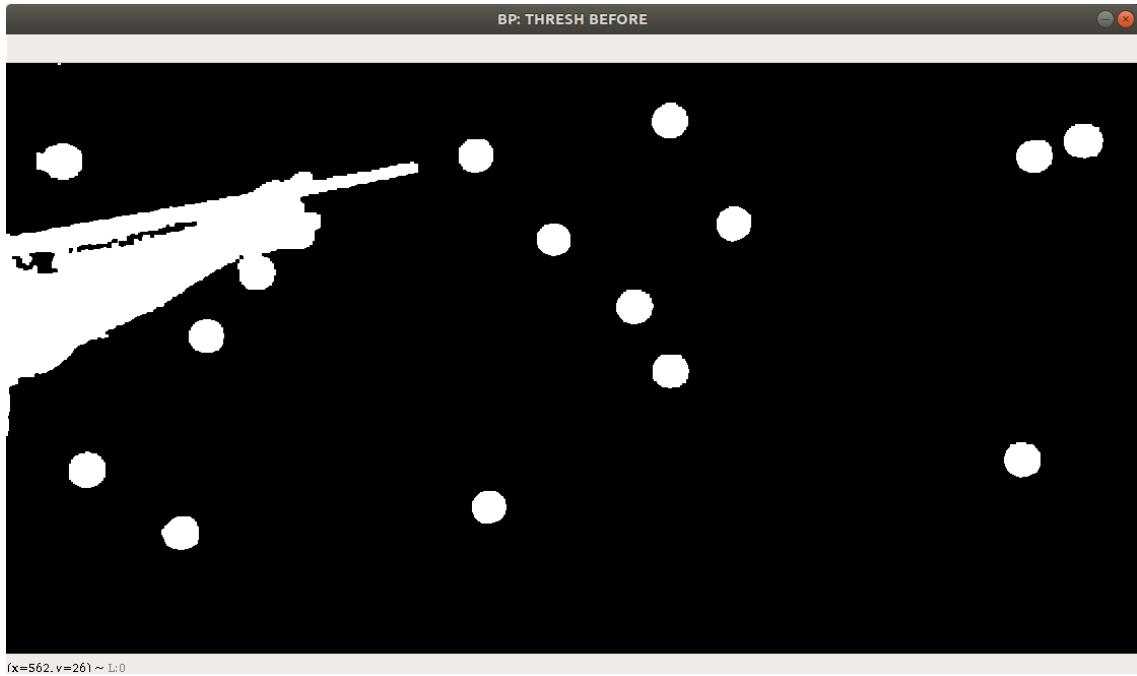


Rysunek 13: Sprogowany obraz po przeprowadzeniu operacji morfologicznych.

wykorzystanie metody **HoughCircles**. Znalezione w pierwszym etapie bile zostają zamalowane, a maska, która może zawierać niewykryte bile, przechodzi dalej do drugiego etapu. W tym etapie użycie metody **HoughCircles** pozwoli wykryć te bile, których nie udało się wykryć w pierwszym etapie.



Rysunek 14: Bile znajdujące się daleko od siebie - znalezione przy użyciu **FindContours**.



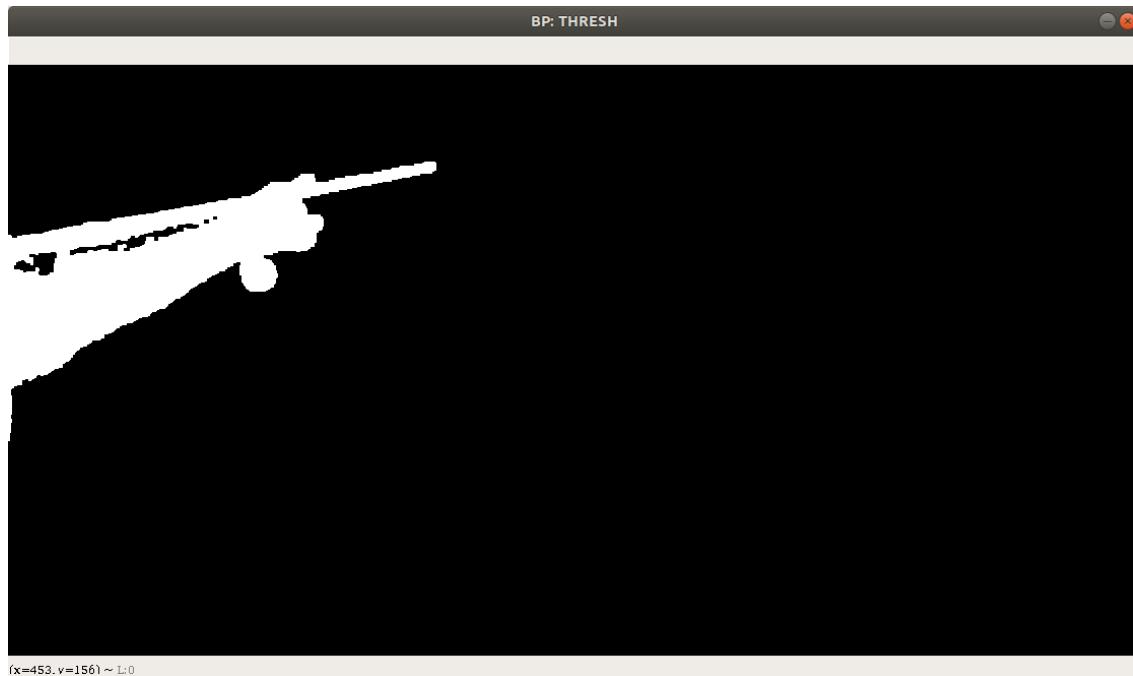
Rysunek 15: Sklejenie konturu gracza i bili znajdującej się obok niego.

Drugim etapem detekcji jest metoda **HoughCircles**[9], która wykorzystuje modyfikację oryginalnej transformacji Hougha[2] (metody wykrywania prostych) pozwalającą na wykrywanie kolistycznych kształtów. Jest ona mniej wydajna od **FindContours**, powoduje więcej fałszywych wykryć, a przede wszystkim jest mniej stabilna jeśli chodzi o wykryte współrzędne środków bil, co wyklucza ją jako podstawową i jedyną metodę detekcji. Pozwala ona jednak na detekcję nawet bardzo trudnych do wykrycia bil i bardzo dobrze spełnia się w roli detekcji pomocniczej - tam gdzie **FindContours** jest niewystarczająca. Zmodyfikowana w poprzednim etapie maska poprzez zamalowanie już wykrytych bil przedstawiona na rysunku 16 zostaje nałożona na różnicę klatek. Otrzymany w ten sposób obraz przedstawiony na rysunku 17 przekazywany jest do metody **HoughCircles**.

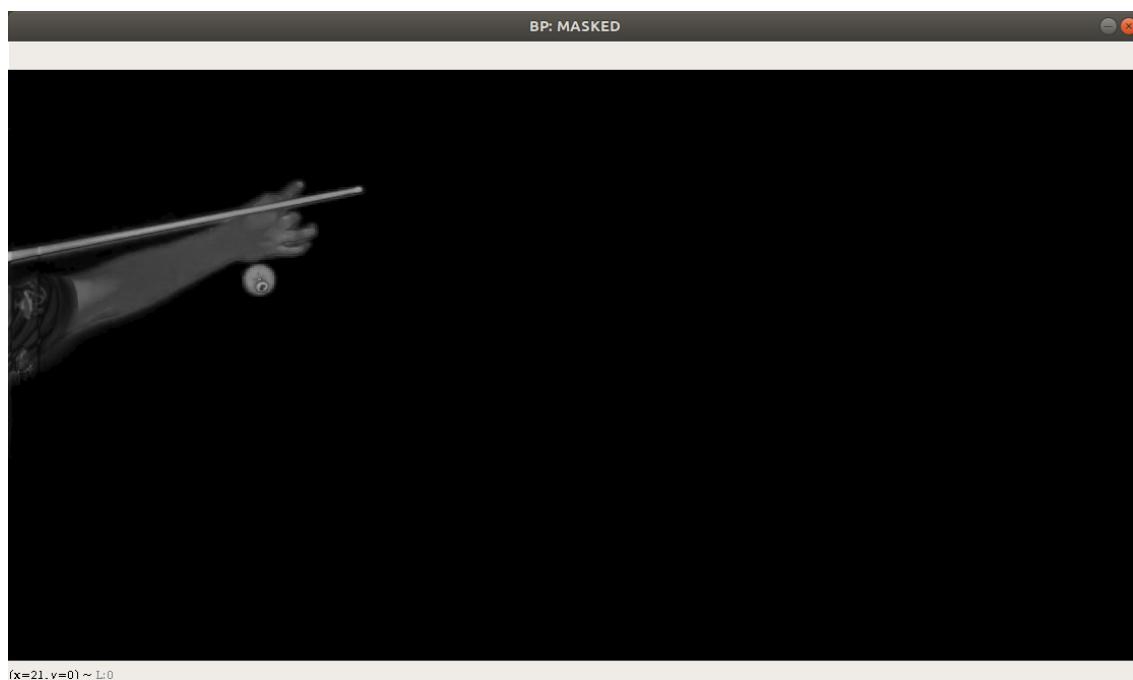
Ostatecznie otrzymywana jest lista współrzędnych środków wykrytych bil (rysunek 18). Dla każdego z nich wycinany jest obraz bili w rozmiarze 50x50, który klasyfikowany jest przy użyciu konwolucyjnej sieci neuronowej.

4.3.2 Klasyfikacja

Do utworzenia sieci neuronowej wykorzystany został framework **Tensorflow** [15] i **Keras** [4]. Głównym wyzwaniem było uzyskanie odpowiedniej jakości klasyfikacji przy jednoczesnym utrzymaniu odpowiedniej wydajności. Ostateczny model sieci składa się z powtózonej sekwencji dwóch



Rysunek 16: Uzyskana po pierwszym etapie maska z niewykrytą jedną bilą.



Rysunek 17: Nałożenie maski na różnice tła i klatki aktualnej.

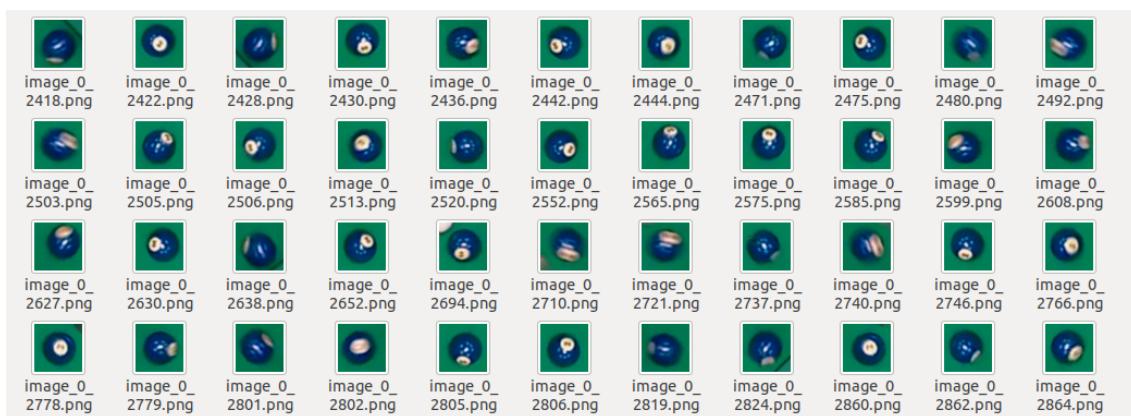
warstw konwolucyjnych, po których następują warstwa łącząca oraz warstwa dropout. Na końcu do wsparcia ostatecznej klasyfikacji wykorzystywane są dwie klasyczne warstwy gęsto rozłożonych neuronów.

Możliwość uruchomienia aplikacji w trybie, w którym wykrywane bile zapisywane są do pliku



Rysunek 18: Wykryte bile po obu etapach detekcji.

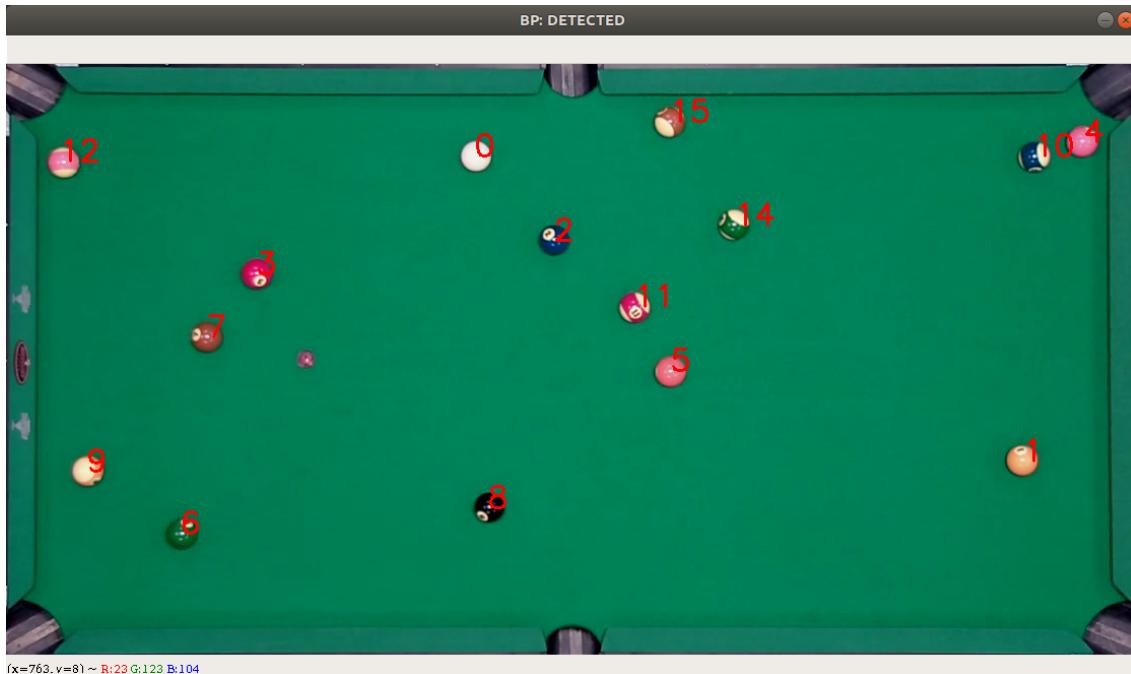
pozwoliła na zebranie własnego datasetu, który następnie został ręcznie poetykietowany. Utworzona została również metoda rozszerzania i zapewniania unikalności datasetu - każdy element datasetu mnożony jest 10 krotnie oraz poddawany jest losowym transformacjom – losowe odbicia w pionie i poziomie, przesunięcia, oraz obroty o losowy kąt z ustalonego przedziału. Pozwoliło to na utworzenie datasetu złożonego z **16.000 unikalnych obrazów bil** oraz **1.000 obrazów fałszywych – błędów detekcji** (rysunek 19).



Rysunek 19: Fragment utworzonego datasetu.

Wykryte i sklasyfikowane bile, co przedstawia rysunek 20, wysyłane są do końcowego procesu – **OutputModule**. Wartości wszystkich punktów normalizowane są do przestrzeni, gdzie prawy

dolny punkt stołu ma współrzędne (2, 1). Pozwala to uniezależnić wyświetlanie danych od rozmiaru klatki wejściowej. Parametry progowania, minimalny oraz maksymalny promień wykrywania bil, oraz wszystkie parametry metody **HoughCircles** konfigurowalne są z poziomu interfejsu aplikacji, co pozwala na dynamiczne ich dostosowywanie.

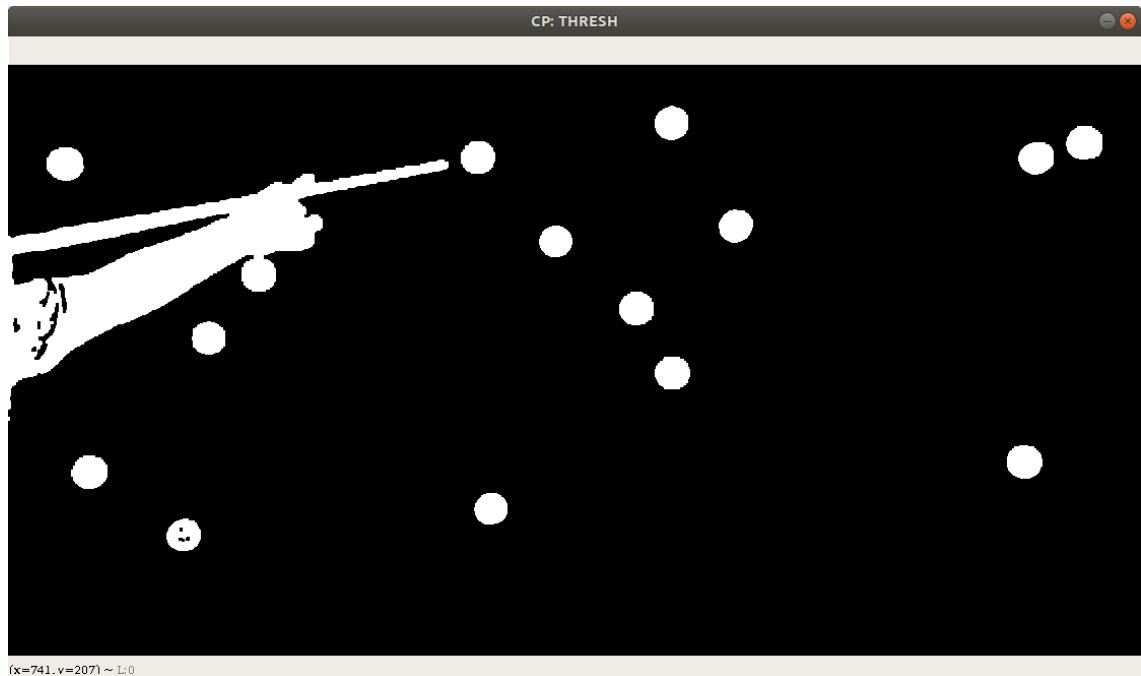


Rysunek 20: Ostateczny produkt uzyskiwany w **BallProcesor** - wykryte i sklasyfikowane bile.

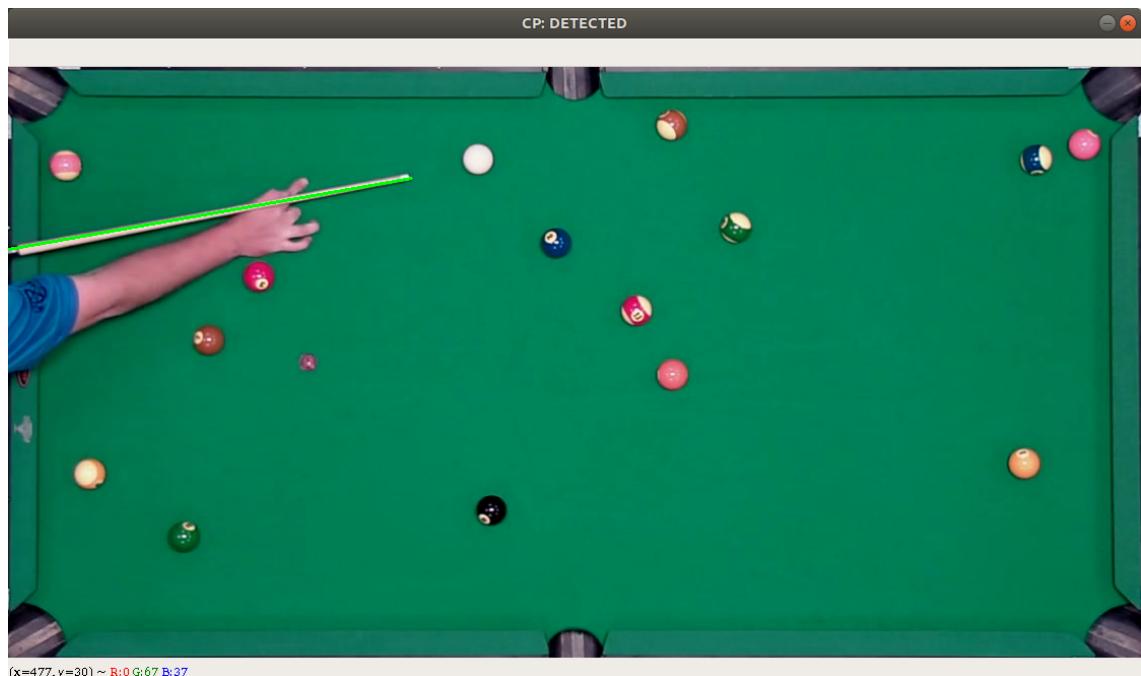
4.4 Kij

Uzyskiwanie sprogowanego obrazu odbywa się podobnie jak w przypadku detekcji bil, ale przy innych, bardziej odpowiednich dla tego problemu, parametrów, które tak jak w przypadku detekcji bil konfigurowalne są z poziomu interfejsu użytkownika. Uzyskany obraz sprogowany przedstawiony został na rysunku 21.

Do detekcji kijów wykorzystano metodę **HoughLinesP**[10], która wykorzystuje probabilistyczny algorytm transformacji Hougha[13]. Pozwala ona na uzyskanie współrzędnych początków i końców kijów, które wysyłane są do **OutputModule**. Podobnie jak w przypadku **BallProcessor**, przed wysłaniem, współrzędne początków i końców kijów normalizowane są do przestrzeni, gdzie prawy dolny punkt stołu ma współrzędne (2, 1).



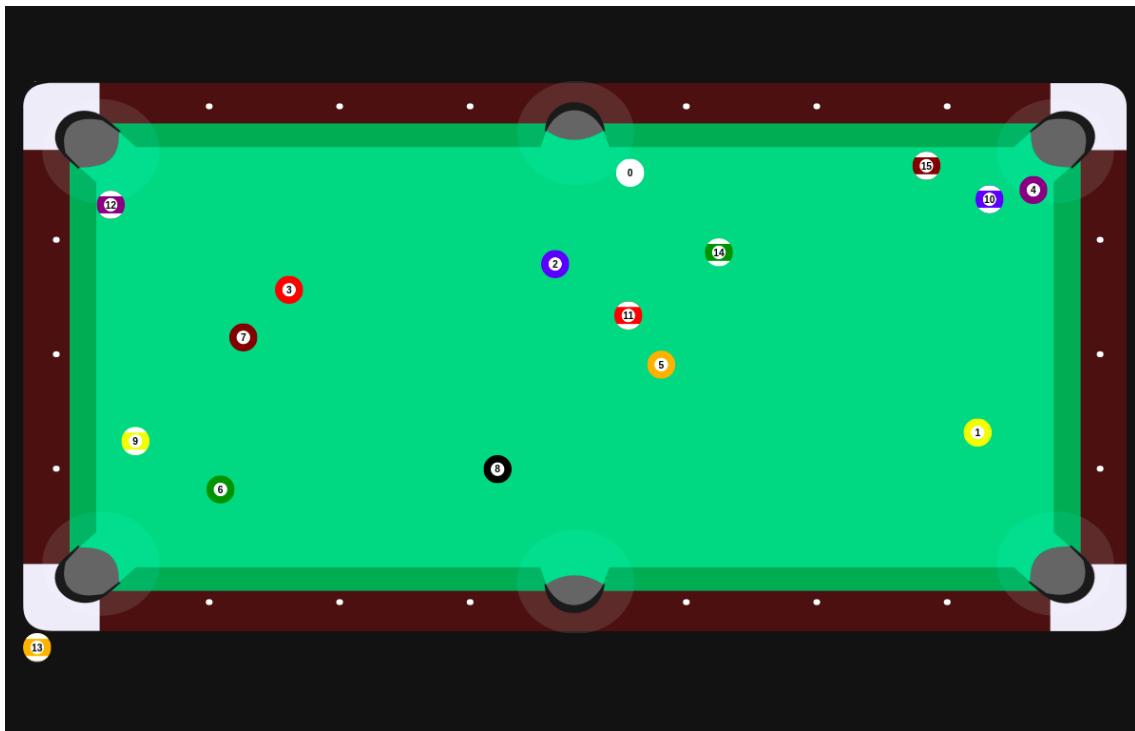
Rysunek 21: Sprogowany obraz po przeprowadzeniu operacji morfologicznych dla CueProcessor.



Rysunek 22: Wykryty kij.

5 Interfejs użytkownika

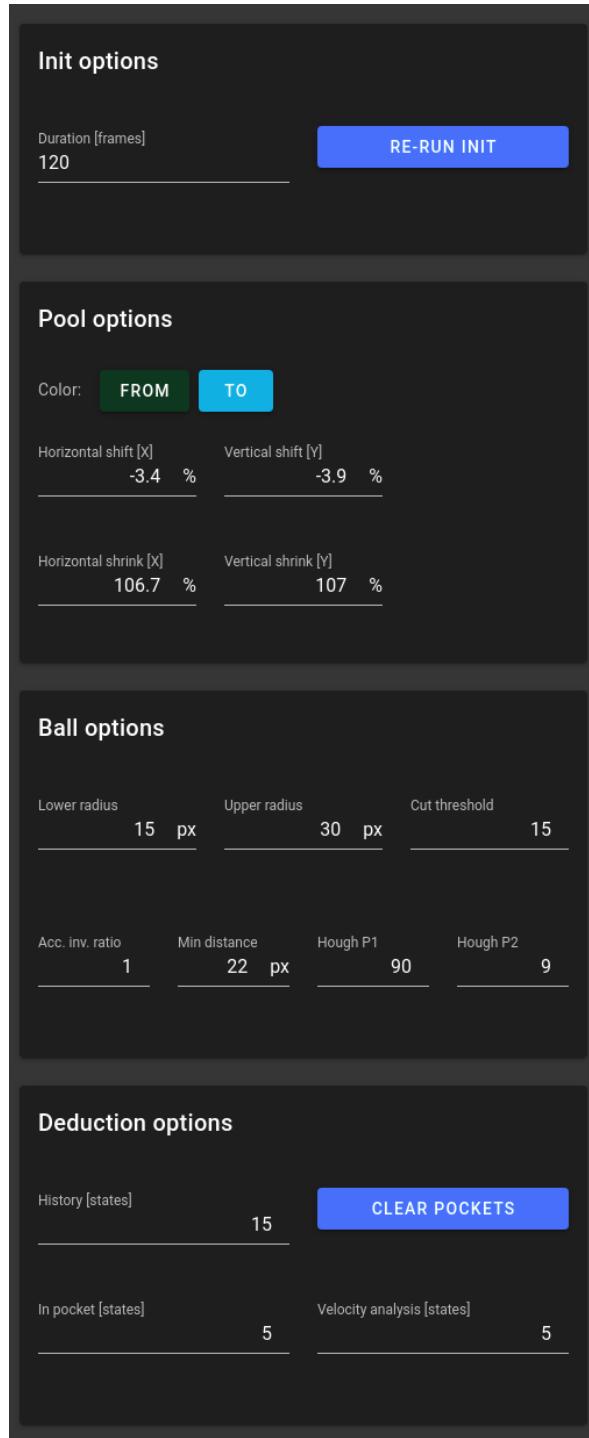
Wyświetlanie stołu odbywa się w komponentie **PoolVD**. Aplikacja webowa stworzona z wykorzystaniem frameworka Vue i Vuetify, na komponentie Canvas rysuje stół wraz z aktualną, wydedukowaną zawartością łuza (rysunek 23). Rysowanie elementów odbywa się ze wsparciem frameworka KonvaJS[5]. Białe półprzezroczyste elipsy wokół łuza obrazują obszar działania dedukcji wpadnięcia bili do łuzy. Pełny widok interfejsu graficznego przedstawia rysunek 25.



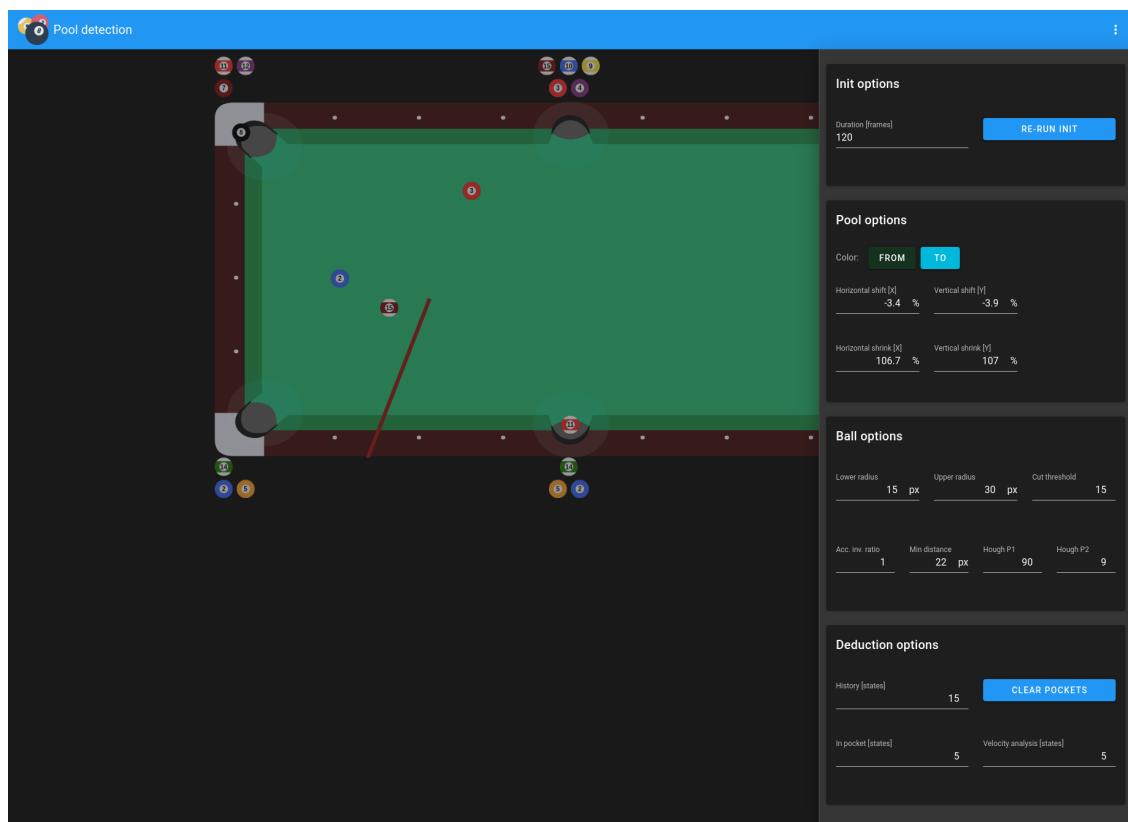
Rysunek 23: Widok stołu w komponencie **PoolVD**.

Aplikacja posiada wysuwalny panel (rysunek 24), z którego poziomu można zmieniać parametry detekcji i dedukcji. Zmiany ustawień detekcji generują opisane wcześniej zdarzenia, które odbierane są przez komponent **VideoProcessor**. Zmiany ustawień dedukcji mają efekt tylko na poziomie komponentu **PoolVD**.

Z racji na możliwość powstania przesunięcia i zaburzeń proporcji stołu w kamerze ustalenia pozwalają na zmianę przesunięcia i skali wyświetlanych elementów stołu. Jest to końcowym etapem bezpośrednio poprzedzającym wyświetlanie elementów stołu, więc dedukcja i analiza pozycji odbywa się na danych otrzymanych od komponentu **VideoProcessor**.



Rysunek 24: Panel opcji w komponentie **PoolVD**.



Rysunek 25: Interfejs komponentu **PoolVD** z testowymi danymi.

6 Uruchomienia systemu

W celu uruchomienia systemu należy udostępnić źródło danych i uruchomić dwa główne komponenty - **VideoProcessor** i **PoolVD**. Wszystkie polecenia muszą być wykonane z katalogu głównego projektu.

6.0.1 Źródło danych

Źródłem danych może być dowolny strumień wideo przesłany z użyciem protokołu UDP. Podczas pracy nad projektem źródłem danych był plik MP4 streamowany z użyciem skryptu za pomocą komendy:

```
1 $ ./video_processor/test/stream.sh ./video_processor/test/videos/test3.mp4  
127.0.0.1:8444
```

Powyższa komenda wysyła dane na port 8444 hosta lokalnego wykorzystując narzędzie ffmpeg.

6.0.2 VideoProcessor

Komponent **VideoProcessor** powinien być uruchomiony za pomocą następującej komendy:

```
1 $ pipenv install  
2 $ pipenv shell  
3 $ python3 -m video_processor.src.main -p 8444 -c 8888 -w 1280 -ht 720 -f 30
```

Argumentami komendy są kolejno port źródła danych, port nasłuchu klientów **PoolVD**, wysokość klatki, szerokość klatki i liczba klatek na sekundę materiału źródłowego. Uruchomienie modułu wymaga wcześniejszego zainstalowania zależności zdefiniowanych w pliku Pipfile narzędziem Pipenv.

6.0.3 PoolVD

Uruchomienie komponentu **PoolVD** odbywa się w oparciu o środowisko NodeJS i narzędzia Vu- eCLI4 [16] z wykorzystaniem wersji deweloperskiej serwera. Port połączenia z komponentem **VideoProcessor** (8888) w obecnej wersji jest na stałe wpisany w kod. Aplikacja dostępna jest pod adresem `localhost:8080`.

```
1 $ ( cd pool_vd && npm install )  
2 $ ( cd pool_vd && npm run serve )
```

7 Podsumowanie

7.1 Zrealizowane założenia

System w obecnej formie realizuje odbieranie materiału źródłowego, jego przetwarzanie i wysyłanie danych do użytkownika. Pozwala na detekcję i klasyfikację bili oraz na detekcję kijów. Aplikacja użytkownika pozwala na modyfikację parametrów klasyfikacji i detekcji oraz odpowiada za dedukcję stanu gry - odrzucanie błędów klasyfikacji i detekcji oraz za wykrywanie wpadnięć bili do luzy. Parametry dedukcji są również konfigurowalne.

7.2 Możliwości rozwoju

Język Python, w którym napisany został komponent **VideoProcessor** doskonale sprawdza się w prototypowaniu złożonych rozwiązań dotyczących przetwarzania obrazu ze względu na dostępność szerokiego wachlarza bibliotek. Sprawdza się nawet w przypadku systemów działających na wielu rdzeniach procesora, mimo ograniczeń jakie niesie ze sobą GIL[3]. Niestety skonstruowane przez nas rozwiązanie przesyłania stanów stołu wiązało się z użyciem korutyn, których wykonywanie w połączeniu z wieloprocesorowością i z wykorzystaniem obu kolejek bili i kilów, prowadziło do nieprzewidywalnych zachowań i braku stabilności przesyłania stanu stołu. Dlatego też nie udało się zaimplementować efektywnego przesyłania pozycji kijów. Rozwijając projekt można przebudować moduł **OutputModule** tak, aby sprawnie obsługiwał protokół WebSocket, który jest podstawa do wydajnego przesyłania danych do aplikacji webowej komponentu **PoolVD**.

Rozwój projektu może wiązać się oczywiście też z poprawianiem struktury sieci neuronowej oraz zebranych lepszych danych treningowych obejmujących różne stoły i warianty kolorów bili. Wydajność detekcji i klasyfikacji bili ma bardzo duże znaczenie w procesie dedukcji stanu stołu.

Bibliografia

- [1] *American Poolplayers Association.* URL: <https://www.youtube.com/user/apaleagues/videos>.
- [2] *Circles Hough Transform.* URL: https://en.wikipedia.org/wiki/Circle_Hough_Transform.
- [3] *Glogal Interpreter Lock.* URL: https://en.wikipedia.org/wiki/Global_interpreter_lock.
- [4] *Keras.* URL: <https://keras.io/>.
- [5] *KonvaJS.* URL: <https://konvajs.org/>.
- [6] *OpenCV.* URL: <https://opencv.org/>.
- [7] *OpenCV - FindContours.* URL: https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html#ga17ed9f5d79ae97bd4c7cf18403e1689a.
- [8] *OpenCV - getPerspectiveTransform.* URL: https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html#ga15302cbff82bdcd70158a58b73d981.
- [9] *OpenCV - HoughCircles.* URL: https://docs.opencv.org/master/dd/d1a/group__imgproc__feature.html#ga47849c3be0d0406ad3ca45db65a25d2d.
- [10] *OpenCV - HoughLinesP.* URL: https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html#ga8618180a5948286384e3b7ca02f6feeb.
- [11] *OpenCV - morphologyEx.* URL: https://docs.opencv.org/trunk/d4/d86/group__imgproc__filter.html#ga67493776e3ad1a3df63883829375201f.
- [12] *OpenCV - warpPerspective.* URL: https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html#gaf73673a7e8e18ec6963e3774e6a94b87.
- [13] *Probabilistic Hough Transform.* URL: <https://jayrambhia.com/blog/probabilistic-hough-transform>.
- [14] *Socket.IO.* URL: <https://socket.io/>.
- [15] *TensorFlow.* URL: <https://www.tensorflow.org/>.
- [16] *Vue CLI.* URL: <https://cli.vuejs.org/>.