

Poznań, 12.06.2019

Filip Wicha

Robert Zaranek

Marcin Wiktorowski

Fruit Viking

dokumentacja techniczna

Spis treści

Spis treści	1
1. Motywacja	3
2. Podział prac	3
3. Oferowane funkcjonalności	4
4. Wybrane technologie	5
5. Architektura rozwiązania	5
5.1 Moduł śledzenia	6
5.2 Moduł gry	17
5.2.1 Klasa Game	17
5.2.2 Klasy State	18
5.2.3 Plik resources	20
5.2.4 Plik util	21
5.2.5 Klasy Controller	21
5.2.6 Klasy Target	21
5.2.7 Klasa Spawner	23
5.2.8 Plik state_types	24
6. Interesujące problemy	25
7. Instrukcja użytkowania aplikacji	27

1. Motywacja

Główną motywacją do napisania niniejszej aplikacji było przetestowanie mechanizmu umożliwiającego kontrolowanie aplikacji z wykorzystaniem źródła światła. Zaobserwowano, że z wykorzystaniem maski można odseparować z obrazu wybrany kolor. Następnie dzięki algorytmom wykrywającym kontury w obrazie, można znaleźć pozycję największego z nich i traktować znalezioną pozycję jako pozycja kontrolera w aplikacji.

W związku z tym, na potrzeby przetestowania wyżej wymienionego mechanizmu zdecydowano się na desktopową implementację gry Fruit Ninja znanej z środowiska iOS oraz Android. Gra Fruit Ninja pojawiła się na iOS oraz Androidzie w 2010 roku i był jedną z pierwszych gier na smartfony, która osiągnęła taki sukces. W ciągu pierwszego roku sprzedano ją w 20 milionach egzemplarzy. Z uwagi na zamiłowanie do toporów i nienawiść do owoców, aplikację nazwano Fruit Viking.

Symulacja działania opisanego mechanizmu wymaga użycia narzędzi pozwalających na jednoczesne nagrywanie obrazu i wyświetlanie obrazu gry, dlatego z uwagi na popularność biblioteki OpenCV zdecydowano, by aplikacja działała na Desktopie w środowisku Python.

2. Podział prac

1. Filip Wicha:

- moduł wykrywania koloru;
- moduł śledzenia obiektu;
- moduł kamery;
- strona wizualna;
- muzyka;
- efekty dźwiękowe;
- moduł przecinania owoców;
- dokumentacja.

2. Robert Zarnek:

- przygotowanie logiki gry;
- menu główne;

- moduł owoców (pozytywne cele);
- moduł jedzenia typu fast food (negatywne cele);
- moduł poruszania się ruchem parabolicznym;
- moduł przecinania owoców;
- poziomy gry;
- przygotowanie gry do dodania muzyki oraz grafiki;
- efekty rozcinania owoców;
- dokumentacja.

3. Marcin Wiktorowski:

- moduł śledzenia obiektu;
- moduł kamery;
- moduł predykcji ruchu (filtr Kalmana);
- przekazywanie danych do gry;
- optymalizacja przetwarzania obrazu;
- dokumentacja;

3. Oferowane funkcjonalności

Gra Fruit Viking oferuje niewielki wachlarz funkcjonalności, ale za to jest zorientowana na to, aby dawać jak najwięcej radości z rozgrywki oraz umożliwiać innowacyjną rozgrywkę z wykorzystaniem innego źródła interakcji niż klawiatura i mysz. Do głównych możliwości należą:

- możliwość sterowania przy użyciu miecza świetlnego lub mocnego źródła światła z wykorzystaniem modułu śledzenia. Polega to na tym, że obraz z kamery jest przetwarzany w takim stopniu, aby na wyjściu moduł podawał współrzędne największej białej plamy, tj.: promienia światła skierowanego w oko kamery.
- wykorzystanie wygodnego, czytelnego oraz -co najważniejsze- prostego w użyciu interfejsu menu. Dzięki dodanym dźwiękom gra zyskała na atrakcyjności oraz daje użytkownikowi wyraźny komunikat o zmianach zachodzących na ekranie.
- wygładzenie ruchu filtrem Kalmana, które niweluje tzw. skakanie wskaźnika, spowodowane niedokładnymi współrzędnymi podanymi przez algorytm, poprzez predykcję następnego położenia.

- pobieranie koloru wskaźnika przez kamerę, aby zmienić element, który będzie wskaźnikiem.
- gra w jednym z wielu poziomów, o różnym stopniu trudności. Na następnych poziomach pojawiają się “złe” fast foodowe posiłki, które utrudniają dalszą rozgrywkę.

4. Wybrane technologie

Jako użyty język programowania wybrany został Python w wersji 3.7, ze względu na dostęp do biblioteki OpenCV poprzez sprawdzone, kompletne wiązania jak i bezproblemowe uruchamianie skryptów na każdym systemie operacyjnym wykorzystywanym podczas rozwoju aplikacji (Windows 10, MacOS, Ubuntu 18.04).

Wykorzystane biblioteki języka:

Biblioteka OpenCV została wybrana, ponieważ zawiera bardzo bogaty zestaw funkcji wykorzystywanych podczas obróbki obrazu, wiązania języka Python kryją pod sobą wydajne funkcje języka C++ oraz skupia się ona na przetwarzaniu obrazu w czasie rzeczywistym, co jest jednym z głównych wymagań naszej aplikacji.

Biblioteka numpy została użyta ze względu na bycie standardem jeśli chodzi o przetwarzanie macierzy, świetnie współpracuje z danymi uzyskanymi dzięki bibliotece OpenCV.

Biblioteka Pygame została wybrana, ponieważ była reklamowana jako nakładka na bibliotekę SLD (ang. *Simple DirectMedia Layer*) napisaną w języku C, międzyplatformowy zbiór modułów służący do pisania gier wideo, zawierającą takie funkcjonalności jak wyświetlanie grafiki, odtwarzanie dźwięków czy renderowanie czcionek. Niestety, co okazało się później biblioteka korzysta ze starszej wersji SDL i prace nad zaktualizowaną wersją Pygame 2 nadal nie zostały ukończone.

5. Architektura rozwiązania

Opis architektury zostanie podzielony na dwa moduły współpracujące ze sobą, jednak będące funkcjonalnie niezależne. Informacja odnośnie integracji znajduje się w obydwu modułach, z punktu widzenia danego modułu.

5.1 Moduł śledzenia

1. Klasa Tracker - śledzenie obiektów

Wykorzystuje ona bibliotekę OpenCV do znajdowania na obrazie największego konturu o zadanym kolorze. Do swojego działania potrzebuje instancji klasy Camera, która służy do pobierania aktualnego obrazu z kamery, oraz z uwagi na wysoką ilość błędnych pomiarów instancję klasy Prediction, która z wykorzystaniem poprzednich pomiarów przewiduje aktualną pozycję kontrolera oraz wygładza tor ruchu.

```
def loop(self):
    while self.running:
        start = time.time()

        # Heavy load

        img = self.camera.image()
        position = self.get_position(img)

        timestamp = time.time() * 1000

        predicted = self.prediction.process(position, 50, timestamp)

        self.position = predicted

        # End of heavy load

        sleep_time = 1. / self.frequency - (time.time() - start)

        if sleep_time > 0:
            time.sleep(sleep_time)
```

Z uwagi na to, że pobieranie i przetwarzanie obrazu z kamery jest zadaniem czasochłonnym, obliczenia te zostały przeniesione do osobnej pętli, która wykonywana jest w wątku pobocznym. Aktualna pozycja przekazywana jest przez pole position.

Pojedyncza iteracja polega na pobraniu obrazu z kamery, określenie pozycji kontrolera na podstawie szczytowego punktu największego konturu o zadanym kolorze, następnie przetworzenie tej pozycji przez algorytm predykcyjny, a na koniec przypisanie obliczonej pozycji do pola position.

```
def get_position(self, img):
    if self.is_color_set():
        img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

        mask = cv2.inRange(img_hsv, self.lower_color, self.upper_color)

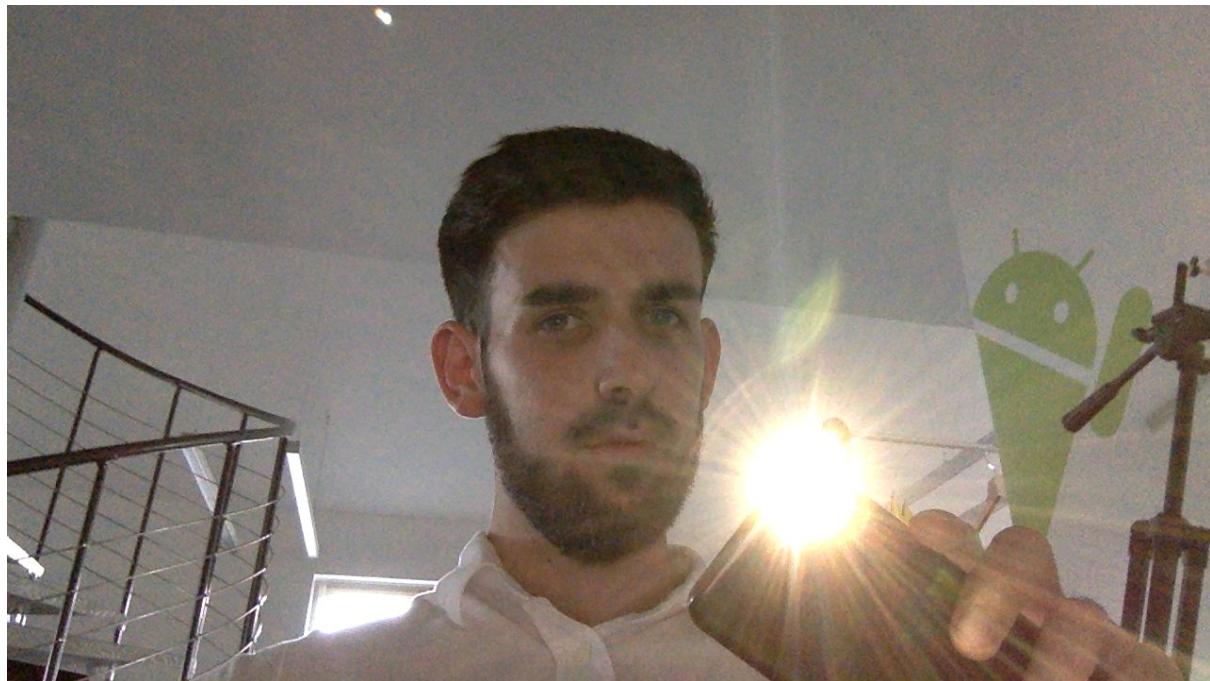
        mask_open = cv2.morphologyEx(mask, cv2.MORPH_OPEN, self.kernel_open)
        mask_close = cv2.morphologyEx(mask_open, cv2.MORPH_CLOSE, self.kernel_close)

        contours, h = cv2.findContours(mask_close, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

        if len(contours) > 0:
            largest_contour = max(contours, key=cv2.contourArea)

            return self.highpoint(largest_contour)
        else:
            return 0, 0
    else:
        return 0, 0
```

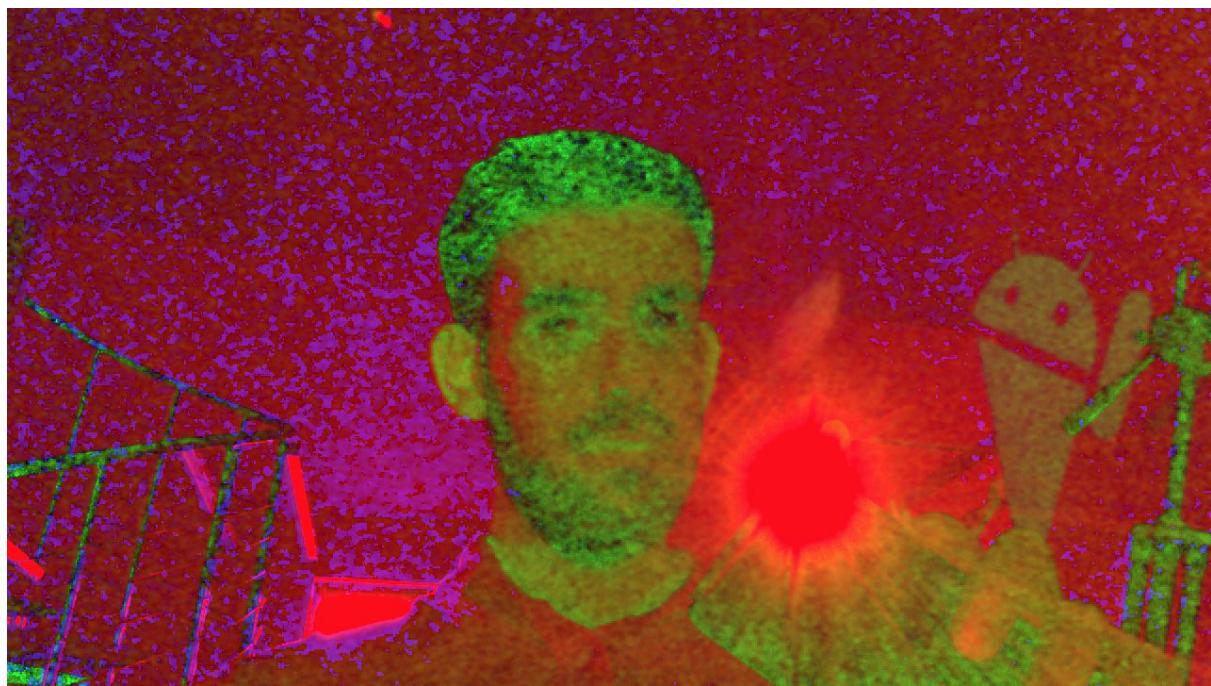
Metoda `get_position` służy do obliczenia pozycji kontrolera na podstawie obrazu.



Obraz pobrany z kamery

Kolejne kroki algorytmu przedstawiają się następująco:

1. Metoda `cvtColor` zamienia obraz na przestrzeń barw HSV, która pozwala na łatwiejsze i dokładniejsze znalezienie zakresu kolorów.



Obraz po zmianie przestrzeni barw na HSV

2. Metoda `inRange` nakłada na zdjęciu maskę z zakresu kolorów od `lower_color` do `upper_color`. Nałożenie maski pozwala na wyekstrahowanie ze zdjęcia tylko poszukiwanego koloru.



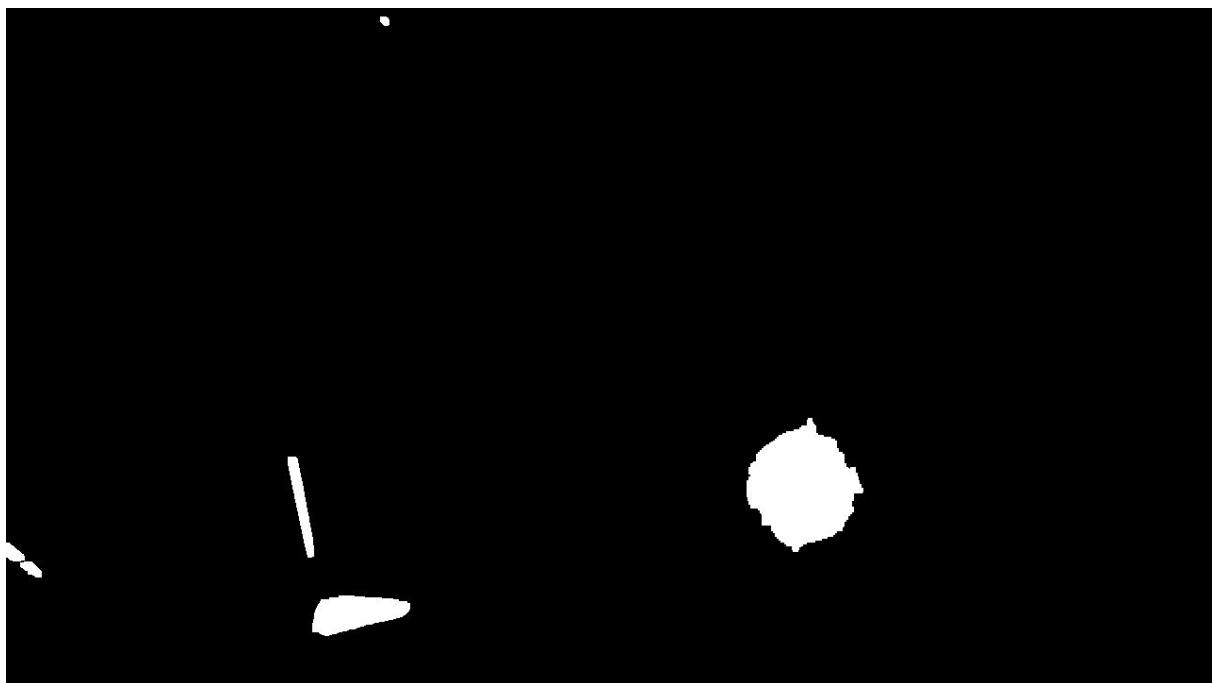
Obraz po nałożeniu maski

3. Metoda `morphologyEx(Opening)` - wykonuje na obrazie erozję, a następnie dylatację.

Zjawisko erozji polega na tym, że przesuwamy nad obrazem macierz zwaną kernel (tak jak w konwolucji), a następnie w przypadku gdy wszystkie piksele pod kernelem mają wartość 1, ustawiana jest wartość pikselu jako 1, w przeciwnym przypadku jest wartość 0 (piksel jest erodowany).

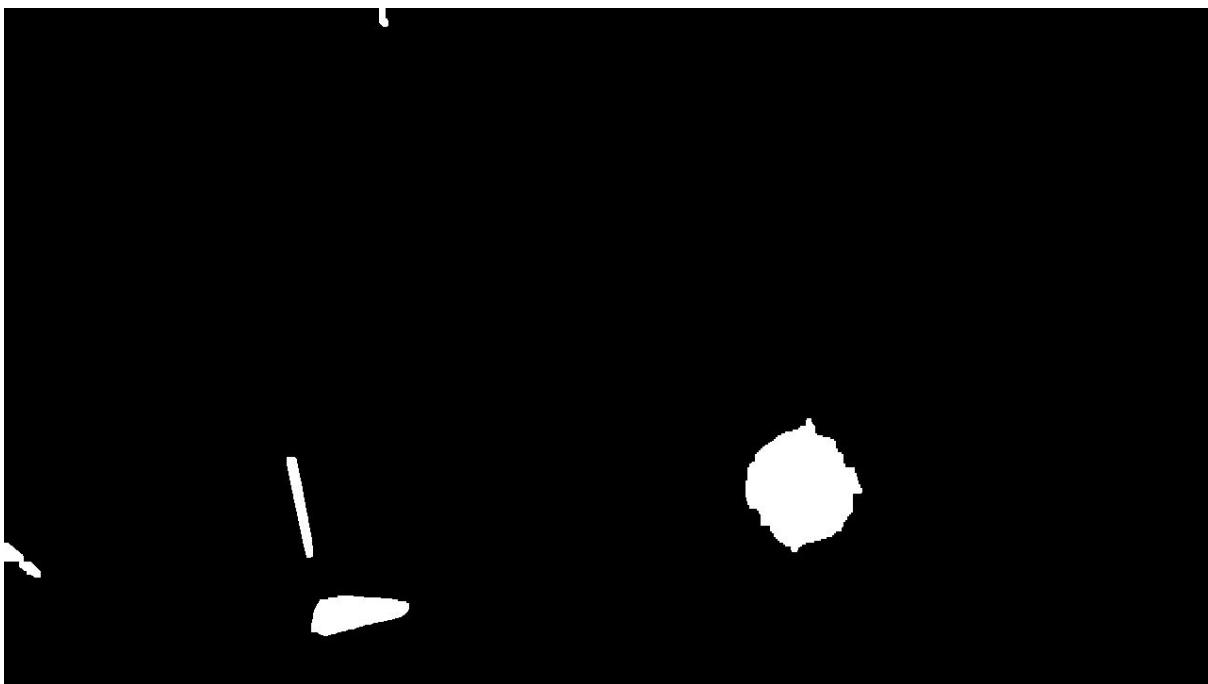
Pozwala to na odrzucenie pikseli na granicy obiektu, co prowadzi do zmniejszenia szumów na obrazie lub rozłączenia dwóch połączonych obiektów.

Zjawisko dylatacji jest przeciwieństwem erozji. Piksel ustawiony będzie jako 1 jeśli wszystkie piksele pod kernelem będą 1. Prowadzi to do zwiększenia powierzchni białych obszarów na obrazie. Zazwyczaj dylatację stosuje się po erozji, z uwagi na to, że pomimo, że erozja usuwa szum to także zmniejsza obiekty. Dylataция pozwala na powrót do oryginalnych rozmiarów obiektów z pominięciem szumów, a także na połączenie rozłączonych obiektów.



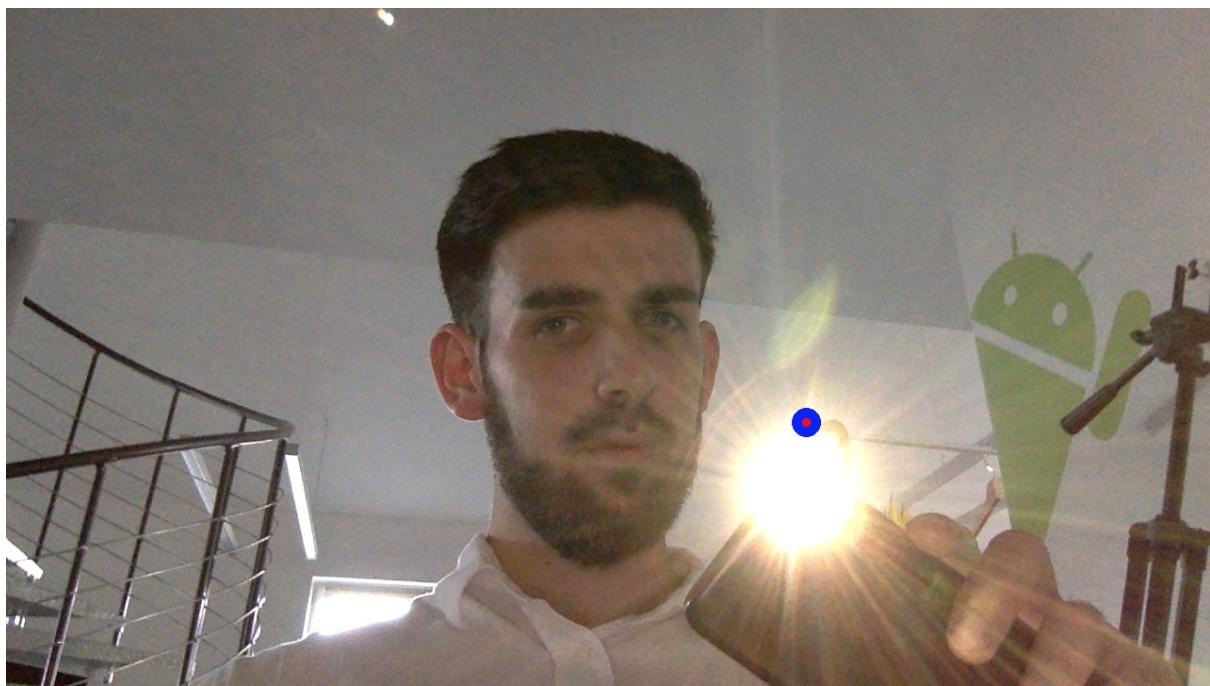
Obraz po odfiltrowaniu szumów

4. Metoda morphologyEx(Closing) - jest odwrotnością morphology(Opening). Metoda wykonuje na obrazie dylatację, a następnie erozję, co pozwala na usunięcie małych czarnych obszarów na obrazie.



Obraz po usunięciu małych czarnych obszarów

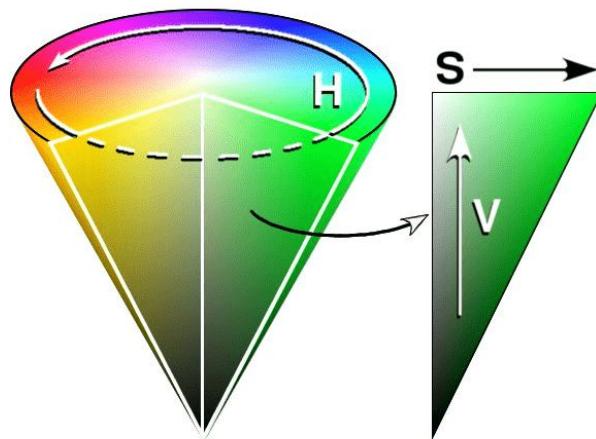
5. Metoda findContours - służy do znajdowania na obrazie konturów. Kontur to krzywa łącząca sąsiadujące punkty o tym samym kolorze (lub intensywności). Dla lepszej dokładności należy wykorzystywać przetworzone zdjęcia w postaci binarnej. W OpenCV, wykrywanie konturów polega na znalezieniu białych obiektów pośród czarnego tła, dlatego obraz wejściowy musi być czarno-biały.
6. Następnym etapem algorytmu jest sprawdzenie czy ilość znalezionych konturów jest dodatnia, po czym wśród znalezionych konturów szukany jest największy z nich.
7. Po znalezieniu największego konturu wyznaczany jest jego najwyższy punkt, który zwracany jest jako pozycja kontrolera.



Finalny obraz

2. Przestrzeń barw.

W przeciwieństwie do najczęściej wybieranej w informatyce przestrzeni RGB, do wykrywania obiektów w OpenCV najczęściej stosuje się przestrzeń HSV, której składowymi są (hue – barwa, saturation – nasycenie, value – wartość). Reprezentację geometryczną tego modelu stanowi stożek lub cylinder. Choć na pierwszy rzut oka wydaje się to dziwne, jest jednak jak najbardziej naturalne, gdyż model ten nawiązuje do sposobu, w jakim widzi ludzki narząd wzroku. Zgodnie z tym modelem wszystkie barwy wywodzą się ze światła białego (środek stożka).



Jak widzimy na powyższym rysunku składowa H – czyli barwa (hue) jest określana jako kąt od 0 do 360 stopni i określa barwę jaką ma dany kolor. Składowa

S – nasycenie (saturation), czyli odległość od środka na promieniu podstawy określa nam nasycenie koloru, a więc im mniejsza wartość tym bliższy białemu) nasz kolor. Natomiast składowa V – wartość (value) oznacza wysokość na stożku, a właściwie możemy traktować tą wartość jako jasność naszego koloru, im mniejsza jej wartość tym kolor ciemniejszy.

Jeśli popatrzymy chwilę na sposób w jakim grupowane są poszczególne barwy w model RGB i HSV szybko znajdziemy wyjaśnienie.



Otoż chodzi o to, że dla modelu HSV, aby określić dany kolor wystarczy jedna składowa H. Na przykład, kolor żółty na stożku wyżej jest zgrupowany w jednym miejscu, wystarczy wskazać zakres kątów odpowiadających żółtemu, przykładowo od 15 do 30 stopni, aby wyselekcjonować obiekty o danym kolorze z naszego obrazu.

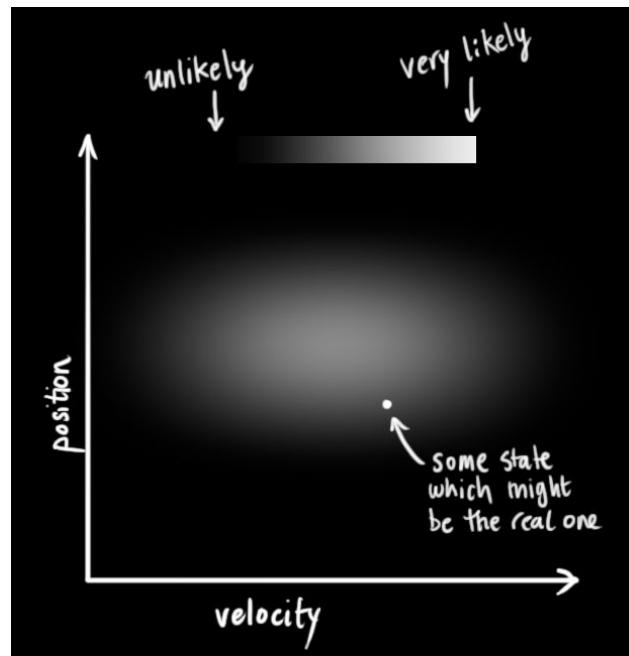
3. Wygładzanie toru ruchu, usuwanie szumów

Do wygładzania toru ruchu i odrzucania błędnych pomiarów wykorzystano rozszerzony filtr Kalmana. Rozszerzony filtr Kalmana wyznacza optymalne oszacowanie stanu dla układów nieliniowych na podstawie pomiarów zakłóconych białym szumem. Otrzymana w ten sposób estymata stanu minimalizuje błąd średniookwadratowy, czyli jest optymalna statystycznie.

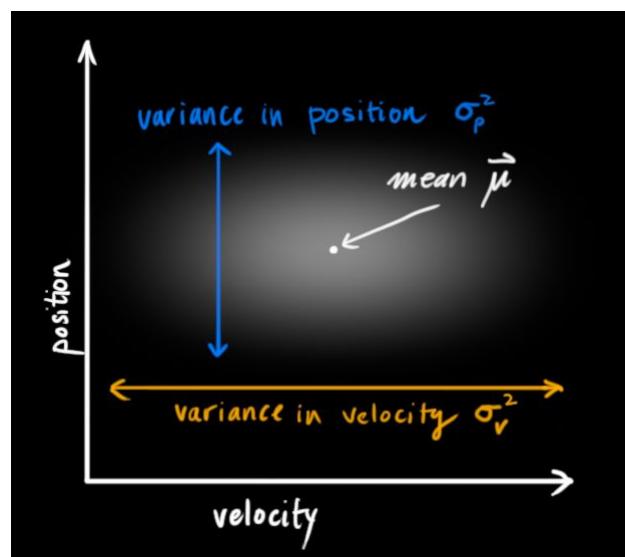
Rozszerzony filtr Kalmana jest algorytmem typu rekursywnego to znaczy nie przechowuje on wszystkich danych z przeszłości i nie dokonuje przeliczania w każdym kroku. Informacje są przetwarzane sukcesywnie, bazując na wartościach obliczonych w poprzednim kroku. Kolejną cechą, która świadczy o optymalności filtra to fakt, że korzysta on z wszystkich dostępnych pomiarów bez względu na to, z jaką dokładnością i precyzją zostały one wykonane. Ostatecznie na ich podstawie dokonuje najlepszej estymacji stanu.

Metodę nazywamy filtrem, gdyż jest on optymalnym estymatorem stanu to znaczy, że uzyskamy możliwie optymalna wartość, na podstawie wielu pomiarów pochodzących z zaszumionego środowiska.

Rozważmy przykład zastosowania filtru Kalmana dla pomiaru pozycji i prędkości.



Z uwagi na to, że pomiar obarczony jest błędem, nie wiemy, jaka jest rzeczywista pozycja i prędkość. Istnieje szereg możliwych kombinacji pozycji i prędkości, które mogą być prawdziwe, ale niektóre z nich są bardziej prawdopodobne niż inne.



Rozszerzony filtr Kalmana zakłada, że obydwie wartości błędu pomiarowego (w podanym przykładzie pozycji i prędkości) są losowe i rozproszone są według rozkładu normalnego. Obydwie mają swoją wartość średnią μ , która jest punktem centralnym rozkładu normalnego, oraz wariancję σ^2 , która odpowiada wielkości niepewności pomiarowej.

4. Wybieranie koloru.

Aby istniała możliwość wykorzystania do sterowania różnych obiektów zaimplementowaliśmy możliwość ustawienia koloru obiektu.

```
class ColorPicker:
    def from_area(self, img, size):
        center = self.center_point(img)

        hsv_roi = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
        crop_img = hsv_roi[center[1] - 1:center[1] + 1, center[0] - 1:center[0] + 1]
        lower = np.array([crop_img[:, :, 0].min(), crop_img[:, :, 1].min(), crop_img[:, :, 2].min()])
        lower_threshold = np.array([lower[0] - size, lower[1] - size, lower[2] - size])
        upper = np.array([crop_img[:, :, 0].max(), crop_img[:, :, 1].max(), crop_img[:, :, 2].max()])
        upper_threshold = np.array([upper[0] + size, upper[1] + size, upper[2] + size])
        return lower_threshold, upper_threshold
```

W tej opcji na ekranie wyświetla się obraz z kamery ze znacznikiem na środku. Gdy w ognisku znacznika znajdzie się obiekt, którego kolor chcemy ustawić jak ten do śledzenia, musimy nacisnąć przycisk “q”. Wtedy wyzwala się metoda z klasy ColorPicker, która pobiera aktualną klatkę obrazu, a następnie wybiera 4 piksele znajdujące się w środku punktu. Z pikseli wybiera się najmniejsze oraz największe wartości Hue, Value oraz Saturation. Algorytm wówczas tworzy 2 listy, w których skład wchodzą właśnie minimalne (dla dolnego przedziału koloru) oraz maksymalne (dla górnego przedziału) wartości Hue, Value oraz Saturation z pobranych pikseli. Następnie te listy przekazywane są do modułu wykrywania koloru. Pozwalają one na wykrywanie obiektów, których powierzchnia niejednolicie odbija światło, co pozwala zapobiec braku wykrycia obiektu ustawionego pod innym kątem. Ze względu na to, że wykrywane obiekty mogą odbijać światło mocniej, niż się spodziewaliśmy, zdecydowaliśmy się na możliwość zwiększenia zakresu o wartość podaną w parametrze “size”.

W naszym wypadku podajemy jej wartość 10.

Przykład działania:

- w ognisku znacznika dajemy obiekt, naciskamy przycisk “q”, który inicjuje pobranie 4 pikseli z wartości HSV:

1 piksel:

H	S	V
20	12	56

2 piksel:

H	S	V
24	13	57

3 piksel:

H	S	V
19	14	59

4 piksel:

H	S	V
28	16	54

- z 4 pikseli wybierane są odpowiednio największe i najmniejsze wartości do ostatecznych list kolorów do zakresu (zielony - najmniejsze, czerwony największe):

tabela z najmniejszymi wartościami:

H	S	V
19	12	54

tabela z największymi wartościami:

H	S	V
28	16	59

- dodawanie parametru “size” (dla tabeli z najmniejszymi wartościami odejmujemy, a dla tabeli z największymi wartościami dodajemy parametr):

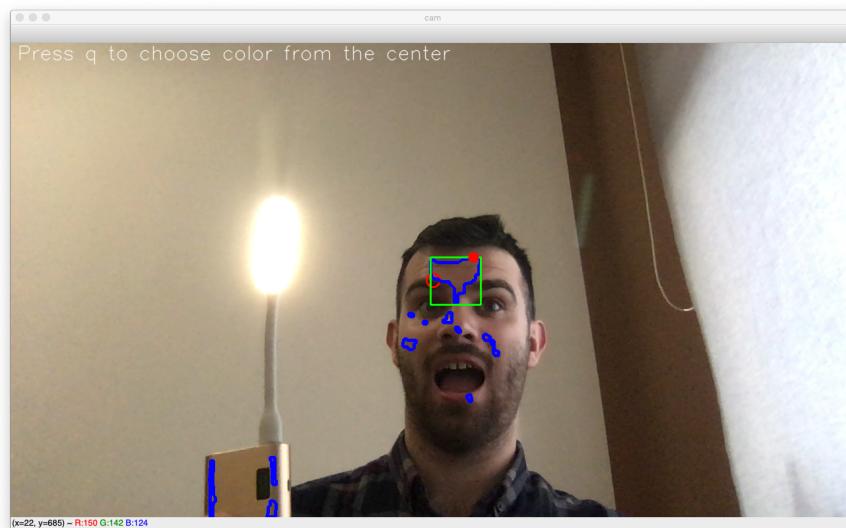
ostateczna tabela z najmniejszymi wartościami:

H	S	V
9	2	44

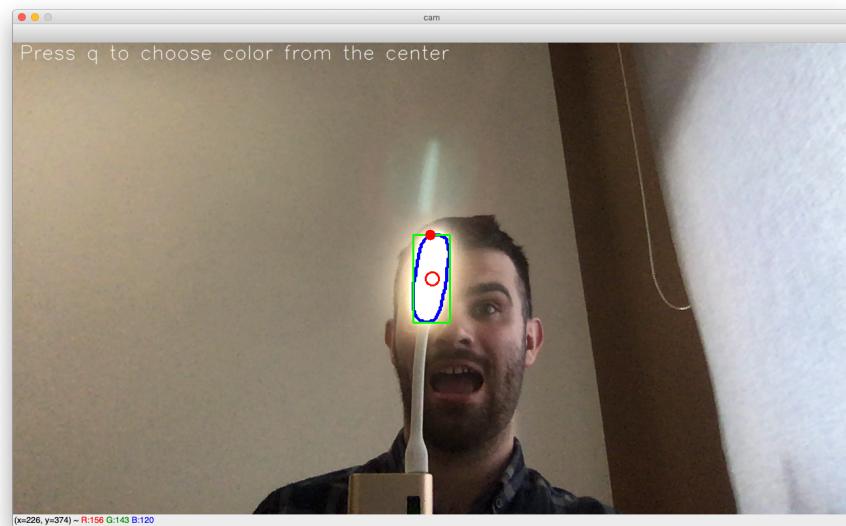
ostateczna tabela z największymi wartościami:

H	S	V
38	26	69

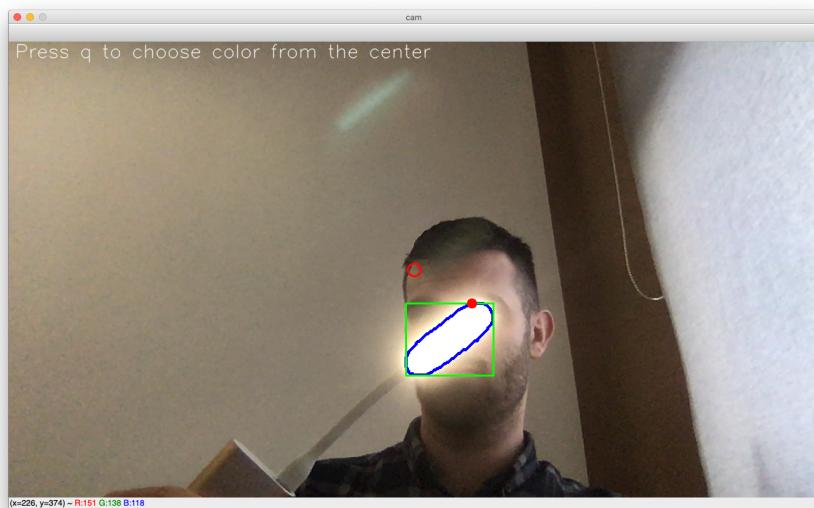
5. Widok z ekranu gry.



Jak widać na powyższym obrazku aktualny kolor jest zbliżony do koloru twarzy (niebieskie obwody wskazują wykrywane obiekty, zielony kwadrat największy element znaleziony na obrazie z kamery, czerwona pusta kropka to znacznik używany do wykrycia koloru, a czerwona kropka to najwyższy punkt w wykrytym największym obiekcie).



Po umieszczeniu obiektu (w tym wypadku latarka) w ognisku znacznika i przyciśnięciu "q" pobierane są wartości kolorów, następnie tworzone minimalne i maksymalne zakresy kolorów. Te przekazywane są do modułu śledzenia.



Aplikacja ustawia kolor do śledzenia ze stworzonych zakresów.

5.2 Moduł gry

5.2.1 Klasa Game

Klasa zawiera w sobie podstawowe parametry gry takie jak wymiary okna, wykorzystywany kontroler, ograniczenie ilości klatek na sekundę oraz flagi okna biblioteki Pygame - `pygame.DOUBLEBUF` i `pygame.HWSURFACE` mające na celu wykorzystanie karty graficznej kiedy tylko jest to możliwe. Automat ze stosem jest również częścią funkcjonalności tej klasy, nie jest on osobną klasą ze względu na prostotę wykorzystanej implementacji i dużej integracji z pętlą gry.

Obiekt tej klasy hermetyzuje wszelkie ogólne parametry gry, jego metody definiują kolejność operacji w pojedynczym wywołaniu pętli gry.

W konstruktorze klasy inicjalizowana jest biblioteka pygame, zdefiniowane wcześniej parametry gry, inicjalizowany jest zegar, ustawiany jest początkowy stan automatu - `MenuState` (menu gry) oraz ustawiana jest flaga `running` sterująca zakończeniem pętli gry.

Główna logika gry w pętli podzielona jest na 3 odrębne funkcje wywoływanie kolejno podczas każdej jej iteracji w następującej kolejności - tick, render i events.

Sama pętla w celu ręcznego testowania wydajności wyświetla w konsoli co 2 sekundy liczbę klatek na sekundę. Po wyłączeniu flagi *running* następuje zakończenie pętli, wyłączenie modułu kamery jeśli jest uruchomiony i wyłączenie całej aplikacji.

Metoda *tick* wywołuje metodę o tej samej nazwie z czubku stosu, tylko dla aktywnych stanów gry. Przekazywanym parametrem jest *dt* (*ang. delta time*) będący czasem który upłynął od ostatniej iteracji pętli, dzięki temu bez względu na częstotliwość wyświetlania klatek logika przetwarzana jest bardzo podobnie, przykładowo obiekty rozgrywki przemieszczają się z tą samą prędkością.

Metoda *render* zajmuje się rysowaniem aktualnego stanu aplikacji, zgodnie z zaleceniami biblioteki Pygame przekazywana jest do niej lista prostokątów będących miejscami na ekranie wymagającymi ponownego narysowania. Rysowany jest tylko stan gry znajdujący się na czubku stosu, jednak z wykorzystaniem atrybutu *propagate_render* istnieje możliwość narysowania większej ilości stanów w pojedynczej iteracji. Właściwość ta nie została jednak wykorzystana w finalnym produkcie. Rysowane są tylko stany aktywne. W przypadku zmiany stanu jednorazowo zostanie narysowany całe okno aplikacji.

Metoda *events* ma za zadanie obsłużyć wydarzenia które zaszły w obecnej iteracji pętli. W klasie *Game* obsługiwane jest zdarzenie związane z wyłączeniem okna aplikacji jak i uruchomieniem trybu pełnoekranowego lub wyłączenie go (kolejno klawisze F12 i F10). Następnie wydarzenia zostaną przekazane do obsłużenia przez stan gry znajdujący się na samej górze stosu.

Automat ze stosem zaimplementowany został z wykorzystaniem kolekcji Lista języka Python, posiada on 3 metody - *push_state* przyjmujący instancję klasy *States* (opisana w 5.2.8 Plik *state_types*) oraz argumenty słowa kluczowego przekazane do tworzonego obiektu klasy *State* mapowanego z enumeracją *States*, *remove_top_state* usuwająca ze stosu ostatnio dodany stan gry oraz *swap_state* która wywołuje kolejno metody *remove_top_state* oraz *push_state*.

5.2.2 Klasy State

Klasa *State* ma za zadanie być klasą bazową po której dziedziczą inne stany gry, definiuje interfejs który powinien implementować każdy stan - metody *render*, *tick* oraz *event*. Dodatkowo przechowuje ona referencję do obiektu klasy *Game* aby

każdy ze stanów miał dostęp do metod automatu stanowego ze stosem i do okna aplikacji, aby mógł w nim rysować.

Wszelkie klasy dziedziczące z klasy bazowej *State* nazywane dalej będą stanami gry.

Stan gry *MenuState* hermetyzuje menu aplikacji, posiada 2 opcje Start oraz Quit. Przy wybraniu opcji Quit pętla gry otrzymuje sygnał o zakończeniu aplikacji, a po wybraniu opcji Start zostaje odłożony na stos stan *LevelSelectionState*. W związku z posiadanym tylko dwóch opcji górna opcja zostanie ustawiona przy wciśnięciu strzałki w góre, a dolna przy wciśnięciu strzałki w dół.

Stan gry *LevelSelectionState* hermetyzuje menu wyboru poziomu, wykorzystuje on pomocniczą klasę *MenuOption*, każda pozycja w menu jest obiektem tej klasy, przechowywane są one na liście i w metodzie *tick* obecnego stanu gry znajduje się logika odpowiadająca za zmianę aktywnej pozycji menu. Każdy obiekt *MenuOption* wskazuje na stan gry do jakiego następuje przejściu po wciśnięciu przycisku Enter kiedy jest on aktywny.

Stan gry *ControllerTestState* jest prostym stanem mającym na celu dać użytkownikowi możliwość sprawdzenia czy kontroler działa odpowiednio, w tym celu umieszczone zostały 3 statyczne cele w kształcie kół ze zdefiniowaną kolizją przy wejściu na ich powierzchnię. Kolizja zachodzi kiedy odległość pomiędzy promieniem statycznego koła, a kołem oznaczającym kursor jest mniejsza niż suma ich promieni.

Stan gry *ResultScreenState* wyświetla wyniki przekazane w formie słownika języka Python ze stanu odpowiedzialnego za rozgrywkę, liczony jest % trafionych celów w stosunku do całkowitej ich ilości, jednak cele oznaczone jako niekorzystne (np. frytki) nie zostaną wzięte pod uwagę w tych obliczeniach. Alternatywą byłoby liczyć je jako ujemne punkty, jednak nadanie im negatywnych efektów podczas samej rozgrywki okazało się ciekawszym podejściem, a karanie gracza podwójnie za pojedynczy błąd nie miało dobrego uzasadnienia.

Stan gry *GameLevelState* posiada możliwość dostosowania do różnych poziomów rozgrywki za pomocą przekazywanych parametrów. Podczas inicjalizacji przekazywane są następujące parametry:

- spawners - domyślnie pusta lista, jest to lista obiektów klasy Spawner definiujących wzorzec pojawiania się celów w grze,

- targets - domyślnie pusta lista, jest to lista celów w grze które mają pojawić się oprócz tych stworzonych przez spawners, wykorzystywane głównie podczas testów,
- debug - zmienna boolowska domyślnie ustawiona na wartość fałsz, sygnalizuje czy poziom powinien zostać uruchomiony w trybie odpluskowania, pokazuje to dodatkowe informacje,
- start_timer - domyślnie sekunda, czas od rozpoczęcia rozgrywki przez jaki nie pojawiają się żadne cele,
- finish_timer - domyślnie 3 sekundy, czas po zakończeniu rozgrywki który poziom odczeka przed przejściem do stanu gry ResultScreenState, po spełnieniu wszystkich innych warunków zakończenia poziomu,
- music - utwór grany podczas trwania poziomu,
- background - tło widoczne podczas trwania poziomu.

Ten stan gry zawiera w sobie mechanizm sterujący całą logiką rozgrywki, w rezultacie zgrywa ze sobą sposób działania listy obiektów klasy *Spawner*, tworzonych przez nie obiektów klasy *Target* oraz obiektów klasy *Remains* które pojawiają się po zniszczeniu celu.

Kolejnym zadaniem stanu jest przygotowanie i przekazanie słownika rejestrującego czy cele zostały trafione, czy też nie do stanu kolejnego przy zakończeniu rozgrywki.

Po utworzeniu przez spawner wszystkich jego obiektów zostaje on usunięty z listy aktywnych spawnerów. W przypadku celów zostają one usunięte z listy aktywnych celów gdy ich współrzędne znajdują się poza oknem aplikacji, lub gdy zostaną zniszczone przez gracza.

Warunkiem zakończenia rozgrywki jest to, aby listy zawierające cele jak i spawnery były puste.

5.2.3 Plik resources

Plik ten zawiera zbiór metod wykorzystywanych do wczytywania zasobów. Mają one odgórnie zdefiniowaną ścieżkę do folderu w którym te zasoby powinny się znajdować. W skład zasobów wchodzą: grafika, czcionki, dźwięk i muzyka.

Zasób wczytany z dysku raz zapisywany jest w pamięci operacyjnej (RAM) i przy ponownym odniesieniu się do niego nie jest on wczytywany z dysku ponownie.

5.2.4 Plik util

Plik ten zawiera pomocnicze funkcje, które nie przynależały do żadnej konkretnej części aplikacji. Dzięki tej architekturze w miarę wzrostu liczby funkcji istnieje możliwość wprowadzenia bardziej rozbudowanej struktury folderów.

W pliku tym znajduje się funkcja *blit_rotate* odpowiedzialna za rotację banana z menu głównego, *text_format* odpowiedzialna za wczytanie czcionki i zastosowaniu jej do tekstu oraz *circle_collision* implementująca kolizję między dwoma kołami.

5.2.5 Klasy Controller

Klasa *Controller* ma za zadanie być klasą bazową po której dziedziczą inne kontrolery, reprezentuje kursor którym gracz steruje na ekranie w czasie rozgrywki, w metodzie render hermetyzuje całą logikę rysowania kurSORA, klasy potomne mają za zadanie jedynie podanie jego lokacji. Do aktualizacji pozycji wykorzystywana jest funkcja *update_position* którą implementują klasy potomne. Klasy potomne będą dalej nazywane kontrolerami.

Kontroler *MouseController* jest osłoną na pozycję myszy, pobiera ją podczas każdego wywołania metody *update_position*, oraz przy inicjalizacji sprawia, że kursor myszy staje się niewidoczny z wykorzystaniem funkcji biblioteki *Pygame*. Wykorzystywany jest tylko w ramach testów z powodu o wiele większej swobody manewrowania nim i braku wymogu używania kamery.

Kontroler *CameraController* wykorzystuje interfejs klasy *Tracker*, zaimplementowanej w ramach modułu przetwarzającego dane z kamery. Logika obiektu klasy *Tracker* funkcjonuje asynchronicznie, a jedyna komunikacją między wątkami jest pobieranie ostatniej pozycji kurSORA, w celu symulowania działania myszy. Przy zakończeniu aplikacji wątek zostaje przyłączony do głównego z wykorzystaniem metody *clean_up* która zatrzymuje instancję *Tracker'a*.

5.2.6 Klasy Target

Klasa *Target* ma za zadanie być klasą bazową po której dziedziczą inne cele, celami nazywane są klasy dziedziczące po tej klasie i implementujące jej interfejs. Klasy dziedziczące dzielone są na pozytywne i negatywne, zależnie od wartości

boolowskiej jaką zwraca statyczna metoda `is_fruit`. W przypadku prawdy są to obiekty pozytywne, w innym negatywne.

Bazowa klasa implementuje większość logiki celów, warunki ich rysowania, renderowania, kiedy wychodzą poza ekran i jaka jest ich aktualna pozycja. Cele mogą zdefiniować swoją logikę przy zniszczeniu przeciążając metodę `on_defeat` która pozwala wpływać na inne cele znajdujące się na ekranie.

Występują następujące cele pozytywne:

Apple:



Melon:



Avocado:



Peach:



Banana:



Pear:



Cherry:



Pineapple:



Eggplant:



Strawberry:



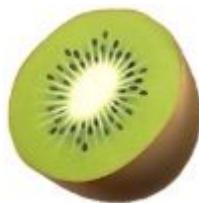
Grapes:



Tangerine:



Kiwi:



Watermelon:



Lemon:



Cele te różnią się jedynie grafiką i miejscami w których następuje kolizja z kursorem, jednak w początkowych założeniach projektowych planowane były osobne efekty dla każdego z nich, a więc dziedziczenie wydawało się najbardziej logicznym podejściem. Pomysł ten został zaniechany z powodów czasowych przy implementacji, jak i poprzez trudność zaprezentowania wszystkich efektów w łatwy do zapamiętania sposób. Natomiast został on wprowadzony w następujących celach negatywnych:

Fries:



Steak:



- Fries - co drugi cel zwiększa swoją prędkość w dół o stałą wartość,
- Steak - co trzeci cel przyspiesza dwukrotnie.

5.2.7 Klasa Spawner

Klasa Spawner ma za zadanie zdefiniować część wzorca pojawiania się celów w czasie rozgrywki. Posiada metody `spawn`, `finished`, `update` oraz `get_spawn_name`.

Przy tworzeniu obiektu należy podać następujące parametry:

- `spawn_type` - klasę dziedziczącą po type Target,
- `munition` - ilość celów tej klasy które mają się pojawić,
- `initial_delay` - początkowe opóźnienie w pojawieniu się celów,

- cooldown - okres czasu między pojawieniem się kolejnych celów,
- min_velocity - krotkę zawierającą zakres prędkości minimalnych celu,
- max_velocity - krotkę zawierającą zakres prędkości maksymalnych celu,
- screen - okno aplikacji, przekazywane przez *GameLevelState* automatycznie,
- strategy_right - flaga boolowska, jeśli jest włączona to obiekty będą lecieć od prawej strony, w innym wypadku od lewej,
- debug - flaga boolowska, czy cele mają pojawiać się w trybie odpluskowania z dodatkowymi informacjami, domyślnie wyłączona.

Wszystkie te parametry pozwalają na bardzo zróżnicowane sposoby pojawiania się celów na ekranie.

Metoda *spawn* odpowiedzialna jest za pojawianie się celu na ekranie, korzysta z klasy przekazanej w parametrze *spawn_type* aby stworzyć jegoinstancję, prędkość jest wartością losową z zakresu pomiędzy minimalną i maksymalną prędkością podaną winicjalizacji spawnera.

Metoda *finished* zwraca wartość prawda jeżeli spawner nie ma amunicji oraz fałsz jeżeli jest nadal aktywny.

Metoda *update* przyjmuje jako parametr czas od poprzedniej aktualizacji (ang. *delta time*) i jej zadaniem jest wywołanie metody *spawn* w odpowiednim momencie.

5.2.8 Plik state_types

Plik ten zawiera klasę States która dziedziczy po dostarczonej przez bibliotekę języka Python klasie Enum naśladującej typ enumeracyjny. Każda wartość przypisana jest konkretnemu stanowi gry, z uwzględnieniem faktu iż stan z innymi parametrami traktowany jest jako osobny stan gry. Zostały utworzone następujące stany:

- MENU,
- CONTROLLER_TEST,
- LEVEL_SELECTION,
- RESULT_SCREEN,
- LEVEL1,
- LEVEL2,
- LEVEL3,

- LEVEL_TEST.

Wartość wyliczeniowa przypisywana jest automatycznie poprzez funkcję auto() stanowiącą część modułu enum języka Python.

6. Interesujące problemy

1. Problem wykrywania miecza.

W grze jako wskaźnika (śledzonego elementu) chcieliśmy użyć toporu. Niestety podczas początkowych testów okazało się, że zwykły topór odbija zbyt dużo światła, przez co tworzą się artefakty, a sam kolor zmienia się podczas poruszania toporu. Rozwiązaniem tego problemu musiałaby być jednolita matowa powierzchnia toporu, która pochłania całe światło, co uniemożliwiałoby zmianę postrzegania koloru przez naszą kamerę. Niestety taki materiał nie istnieje. Następnym krokiem było stworzenie toporu z świecącą lampką na końcu, podobne do PlayStation Move.



Takie urządzenie generuje niezbyt mocne, kolorowe światło, które pod każdym kątem ma tą samą barwę. Myśląc, że problem jest rozwiązany, na starszych laptopach okazało się, że taka kolorowa żarówka nie zdaje egzaminu, gdyż kamera komputera sama dostosowuje poziom ekspozycji do otoczenia, co powoduje, że nie da się jej ustawić na stałe. Oznacza to, że w słabym świetle obraz z kamery ma wywindowaną opcję ekspozycji, co powoduje to, że na ekranie komputera żarówka mimo mocnej barwy jest i tak biała. Uniemożliwia to śledzenie jej po barwie.

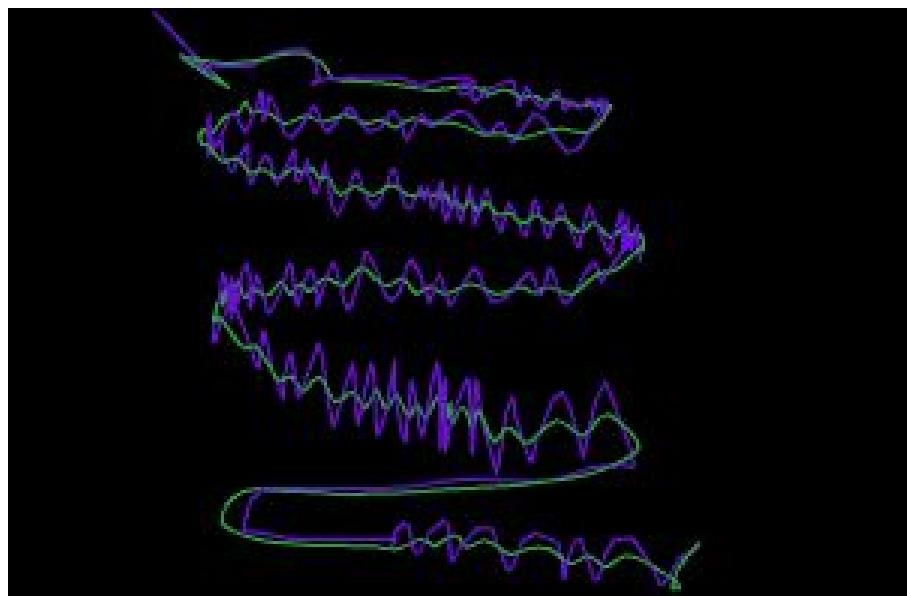


W ostateczności zdecydowaliśmy się na białe światło, które w każdych warunkach dalej pozostaje białe. Jedynym wymaganiem jest to, aby było ono wystarczająco mocne. Ognisko światła powinno wydzielać około 200 kandeli na niewielkiej powierzchni, np. 2 cm x 5 cm. W naszym wypadku egzamin najlepiej zaliczyła lampka używana do czytania podłączana do portu USB firmy Xiaomi (dostęp na dzień 11.06.2019). Podłączona do mobilnego źródła zasilania (np. bank energii 2000 mAh) oraz ustawiona na maksymalną jasność pozwala na godziny zabawy z naszą grą.



2. Problem wygładzania ruchu.

Z uwagi na to, że śledzonym obiektem jest źródło światła napotkano na problemy związane z flarami emitowanymi przez to źródło światła, a także wykrywanie innych źródeł światła. Ruch kontrolera w aplikacji nie był płynny i stosunkowo często diametralnie zmieniał swoją pozycję. Było to spowodowane tym, że zmieniające swój kąt padania źródło światła momentalnie stawało się mniejsze, niż inne wykryte obszary białego światła, dlatego algorytm błędnie akceptował inne obszary jako pozycję kontrolera. Dodatkowo, z uwagi na flary występujące wokół źródła światła, pomimo poprawnie wykrytego obiektu, trajektoria kontrolera nie była płynna.



Rozwiązaniem było zastosowanie rozszerzonego filtru Kalmana, który wyznacza optymalne oszacowanie stanu dla układów nieliniowych na podstawie pomiarów zakłóconych białym szumem. Otrzymana w ten sposób estymata stanu minimalizuje błąd średniokwadratowy, dzięki czemu przebieg danych wyjściowych jest wygładzony w stosunku do przebiegu danych wejściowych. Na powyższym rysunku fioletowym kolorem oznaczono przebieg wejściowy, a zielonym kolorem oznaczono przebieg wyjściowy po zastosowaniu rozszerzonego filtru Kalmana.

7. Instrukcja użytkowania aplikacji

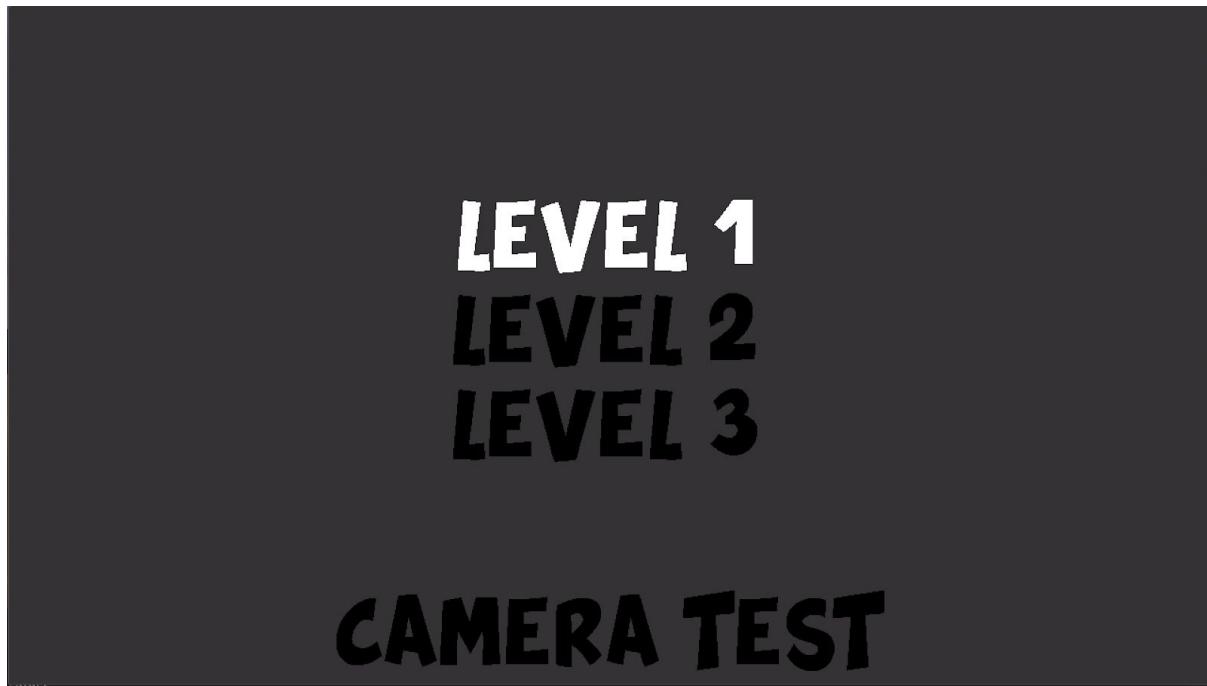
Gra polega na przecięciu jak największej liczby owoców w jednym z wielu poziomów. Owoce pojawiają się na ekranie i poruszają się ruchem parabolicznym lub prostoliniowym przez określony czas po ekranie. Użytkownik do sterowania w menu używa przycisków klawiatury, a do ścinania owoców aplikacja wykorzystuje wykryty z użyciem kamery miecz z umieszczoną na końcu latarką. Aby przeciąć element na ekranie, wskaźnik sterowany mieczem musi znaleźć się w polu, a następnie z niego wyjść. Jest to symulacja przecięcia.

1. Menu główne.



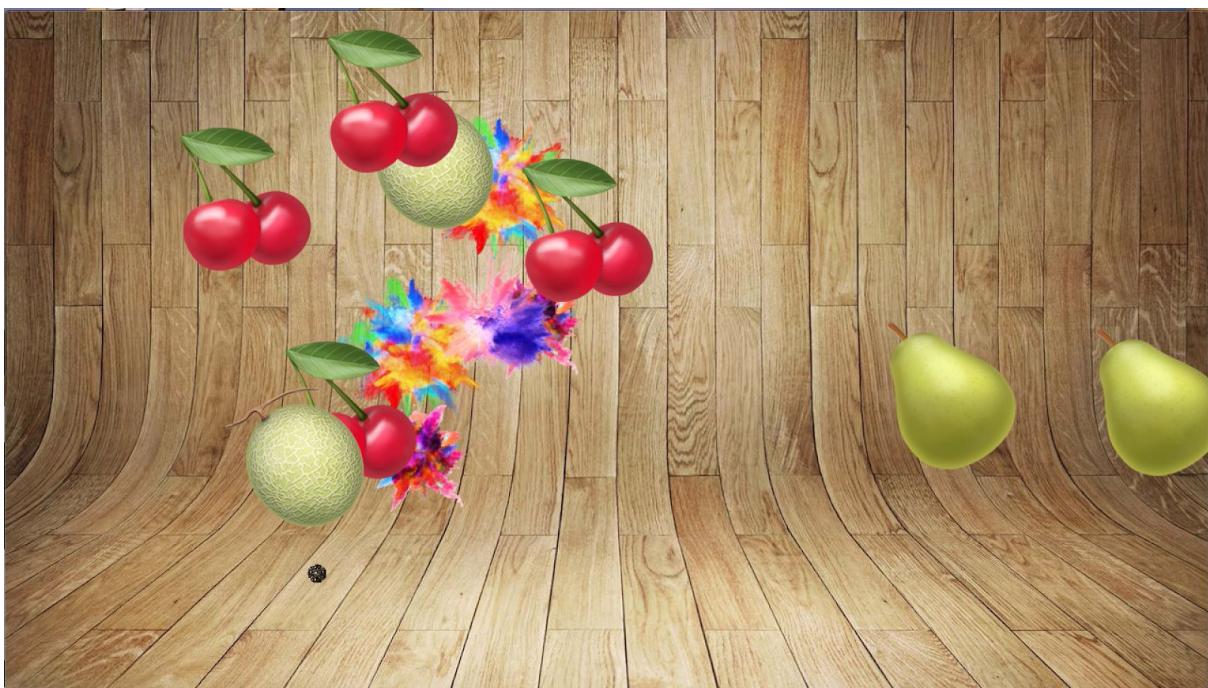
W menu głównym znajdują się 2 opcje wyboru, takie jak "Start", który pozwala przejść do menu wyboru poziomów, a także "Quit", którego wybranie jest jednoznaczne z zakończeniem rozgrywki i wyłączeniem gry.

2. Menu z wyborem poziomów.



W menu wyboru poziomów mamy dostępne 3 poziomy z różnymi trudnościami rozgrywki. Na samym dole mamy opcjonalny test kamery, który pozwala zweryfikować poprawność oraz dokładność odczytu pozycji miecza przez kamerę.

3. Rozgrywka



Rozgrywka zaczyna się 3 sekundy po wybraniu poziomu. Jest to czas, w którym powinniśmy oddalić się od oka kamery na wystarczającą odległość (1-1,5m). Rozpoczęcie rozgrywki jest jednoznaczne z pojawiением się pierwszego owocu lub jedzenia na ekranie.



Po tym czasie na ekranie pojawiają się owoce. Naszym zadaniem teraz jest poruszanie mieczem w taki sposób, aby czarny wskaźnik będący tarczą wikinga w małej wersji przeciął jak najwięcej owoców. Najlepszą taktyką są szybkie ruchy wzdłużne, które pozwalają przeciąć parę owoców na raz. Musimy mieć na uwadze, że w trudniejszych poziomach pojawiają się mniej zdrowe formy jedzenia takie jak

frytki i steki, których przecięcie powoduje przyśpieszenie oraz zmiana ich kierunku ruchu owoców oraz utrudnia ich ścięcie. Powinniśmy unikać przecinania tych posiłków, aby zwiększyć swoje szanse.

4. Podsumowanie rozgrywki.



Ekran podsumowujący rozgrywkę pojawia się po zakończeniu poziomu. Pokazuje on statystyki w formacie stosunku przeciętych owoców do wszystkich, jakie pojawiły się na ekranie oraz zabawną dwuznaczna notkę, która dodaje nam motywacji oraz informuje jak daleko nam brakuje do perfekcyjnego wyniku.

5. Przydatne informacje:

- nawigacja po menu jest realizowana przez przyciski strzałek w dół i w górę;
- przycisk "Enter" pozwala wybrać zaznaczoną opcję;
- w każdym momencie możemy zakończyć rozgrywkę przyciskając przycisk "Esc".