AKADEMIA GÓRNICZO–HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

Praca dyplomowa

inżynierska

System do akwizycji, przetwarzania i wyświetlania stereoskopowych sekwencji wideo

Imię i nazwisko	Maciej Bielski
Kierunek studiów	Elektronika i Telekomunikacja
Opiekun pracy	dr inż. Jarosław Bułat

Kraków, rok 2014

OŚWIADCZENIE AUTORA

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

.....
(Podpis autora)

Spis treści

Wstęp

W niniejszej pracy przedstawiono system do rejestracji, przetwarzania i ostatecznie wyświetlania stereoskopowych sekwencji wideo na ekranie 3D w technologii pasywnej. Zaprezentowano system działający w czasie rzeczywistym. Obraz jest rejestrowany na bieżąco przez zestaw dwóch kamer i wyświetlany na monitorze w rozdzielczości FullHD (1920x1080). Kamery umieszczone zostały nad monitorem i rejestrują scenę przed nim tak, że cały zestaw działa na kształt lustra 3D.

W ramach projektu zaprojektowano aplikację napisaną na platformę Linux w językach C i C++ z wykorzystaniem biblioteki OpenCV [1] oraz systemowych narzędzi do przetwarzania wielowątkowego. Aplikację wyposażono w prosty interfejs użytkownika napisany z wykorzystaniem pakietu GTK [2]. Podczas pisania kodu wspomagano się przykładami zaprezentowanymi w [3].

W praktycznych zastosowaniach zagadnienie widzenia stereoskopowego można spotkać na przykład w produkcji filmów oraz gier komputerowych, które w połączeniu z odpowiednim sposobem wyświetlania oferują obraz stwarzający wrażenie trójwymiarowości. Innym zastosowaniem stereowizyjnych układów kamer jest robotyka. Dzięki analizie powierzchni odtworzonej z takiego układu kamer, roboty potrafią orientować się w otaczającej przestrzeni oraz omijać przeszkody [4,5]. Umożliwia to zmniejszenie ryzyka kolizji z przeszkodami na przykład podczas przeprowadzania rozpoznania w terenie niebezpiecznym dla człowieka.

W kolejnych rozdziałach przedstawiono poszczególne etapy projektowania i implementacji systemu. W rozdziale pierwszym omówiono od strony teoretycznej proces rejestracji sceny przez kamerę, a więc przeniesienia obserwowanej przestrzeni trójwymiarowej na płaszczyznę dwuwymiarową. Scharakteryzowano zniekształcenia wprowadzone w trakcie tego procesu. Ponadto przedstawiono zależność pomiędzy dwoma kamerami pracującymi w systemie stereoskopowym. Rozdział drugi dotyczy systemu od strony praktycznej. Opisano wykorzystany sprzęt, jego konfigurację, użyte narzędzia programistyczne oraz kolejne etapy wykonywania programu. Analizę otrzymanego rezultatu zamieszczono w rozdziale trzecim, w którym zamieszczono również opis napotkanych problemów i ograniczeń.

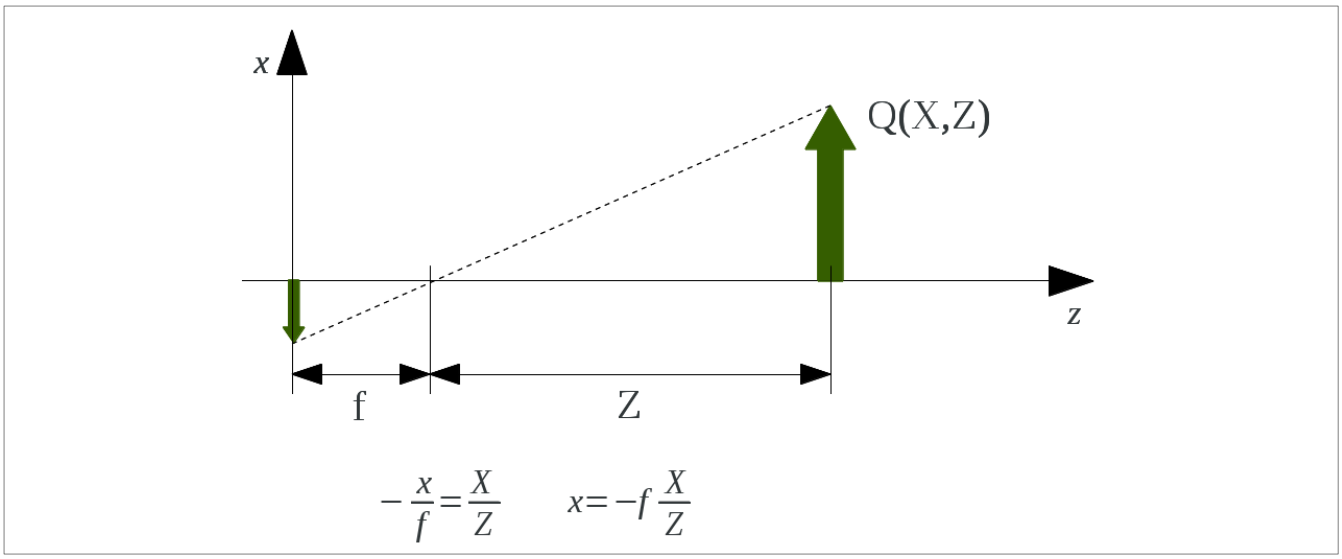
1. Obraz stereoskopowy w teorii

W rozdziale wprowadzono model kamery otworkowej, czyli zależności matematyczne łączące pozycję rzeczywistego punktu w przestrzeni z jego pozycją na otrzymanym obrazie. Następnie opisano proces rzutowania oraz wprowadzone zniekształcenia, które należy uwzględnić. Na końcu przedstawiono pojęcie kalibracji. Kalibracja pojedynczej kamery to proces, w którym otrzymuje się parametry wewnętrzne kamery oraz wektor zniekształceń.

Kalibracja dwóch kamer w układzie stereoskopowym dostarcza informacji o wzajemnym położeniu obrazów tych kamer względem siebie. W pierwszej kolejności wykonano kalibrację każdej z kamer osobno. Następnie, w oparciu o otrzymane parametry, skalibrowano układ stereoskopowy utworzony przez te kamery. Wszystkie operacje i ilustracje opisane w niniejszym rozdziale wraz z przedstawionymi wyrażeniami matematycznymi sporządzono w oparciu o [6].

1.1 Model kamery otworkowej

Model kamery otworkowej opisuje odwzorowanie na płaszczyźnie punktu w trójwymiarowej przestrzeni. Model można opisać jako sześcian, którego przednia ściana posiada umieszczony centralnie okrągły otwór, a równoległa do niej tylna ściana tworzy płaszczyznę obrazu. Prosta łącząca punkt w przestrzeni z punktem na płaszczyźnie obrazu to promień. Dowolny, widoczny z perspektywy kamery punkt emituje pewną ilość światła. Odbija część światła padającego absorbując resztę lub je generuje. Każdy punkt obrazu to miejsce przecięcia się promienia światła pochodzącego od punktu w przestrzeni z płaszczyzną obrazu. Schematyczny obraz modelu przedstawiono na rys. 1.1:



Rys. 1.1. Model kamery otworkowej

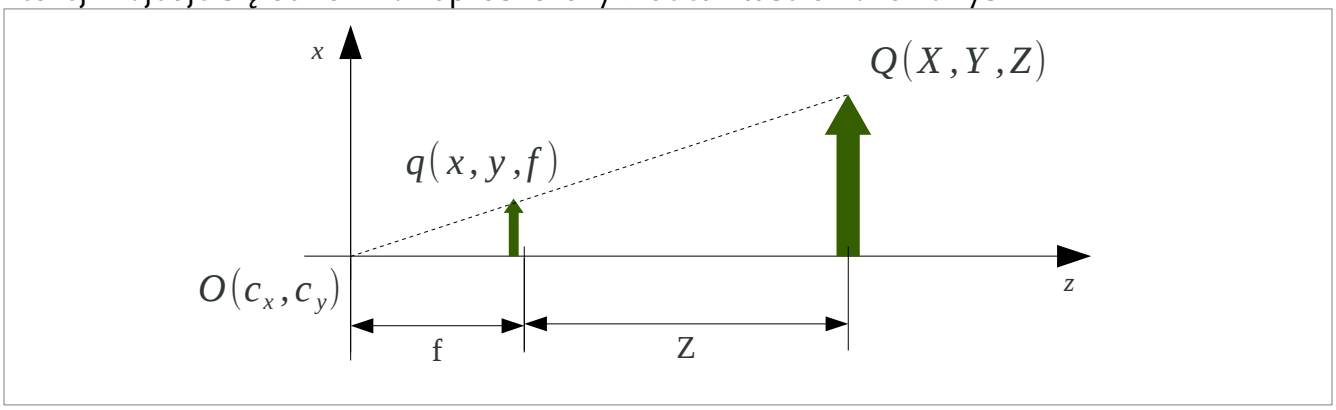
gdzie:

- Q – punkt w przestrzeni o współrzędnych rzeczywistych,
- q – punkt obrazu, którego pozycja jest określona we współrzędnych obrazu,
- f – długość ogniskowej kamery.

Od ilości światła absorbowanego przez punkt w przestrzeni zależy to w jakim kolorze i intensywności dany punkt jest postrzegany przez wzrok. Podobnie w przypadku kamery, matryca tworząca płaszczyznę obrazu jest wykonana z materiału światłoczułego co

umożliwia odwzorowanie rzeczywistych kolorów punktów obserwowanych przez kamerę.

Idąc w drugą stronę, każdy punkt na płaszczyźnie obrazu jest odwzorowaniem punktu leżącego na prostej wyznaczonej przez promień. Uwidacznia to naturalne ograniczenia wynikające z obserwacji przestrzeni za pomocą jednej kamery. Utracona zostaje informacja o jednym z wymiarów. Niezależnie od odległości punktów od kamery wszystkie są obserwowane tak, jakby leżały na płaszczyźnie. Ponadto, tylko najbliższy z wszystkich punktów leżących wzdłuż tego samego promienia jest widoczny zasłaniając pozostałe. Dla uproszczenia analizy matematycznej płaszczyznę obrazu zamienia się z płaszczyzną, na której znajduje się otwór. Tak uproszczony model zilustrowano na rys. 1.2:



Rys. 1.2. Uproszczony model kamery otworkowej

Przyjęto następujące założenia:

- wielkie litery *X*, *Y*, *Z* oznaczają pozycję punktu *Q* we współrzędnych rzeczywistych względem punktu *O*,
- małe litery *x*, *y*, *f* opisują pozycję punktu *q* we współrzędnych płaszczyzny rzutowania względem punktu *O*,
- c_x i c_y to odległości położenia środka płaszczyzny rzutowania względem osi optycznej odpowiednio wzdłuż osi *x* i *y*,
- punkt *O* jest środkiem rzutowania, miejscem skupienia wszystkich promieni tworzących obraz na płaszczyźnie rzutowania. Jest także środkiem układów współrzędnych dla współrzędnych rzeczywistych oraz współrzędnych płaszczyzny obrazu. W modelu idealnym $c_x = c_y = 0$, co oznacza, że środek matrycy pokrywa się idealnie z osią optyczną *z*,
- dla długości ogniskowej małej w porównaniu z odległością obiektu stosuje się przybliżenie $Z + f \approx Z$.

Wobec powyższych punktów pozycja punktu w rzeczywistości i jego pozycja na płaszczyźnie obrazu jest związana w następujący sposób:

$$x = f_x \frac{X}{Z} + c_x$$
$$y = f_y \frac{Y}{Z} + c_y \quad (1)$$

Długości ogniskowych wzdłuż osi *x* i *y* nie muszą być równe, oznacza to, że piksele obrazu nie są wtedy kwadratami.

1.2 Rzutowanie przestrzeni 3D na płaszczyznę 2D

Relacja pozycji punktu we współrzędnych rzeczywistych oraz jego obrazu na płaszczyźnie rzutowania da się przedstawić w następującej postaci:

$$q = M Q \quad (2)$$

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad (3)$$

Macierz *M* jest macierzą wewnętrzną kamery. Jej składowe zostaną otrzymywane w procesie kalibracji. Powyższą transformację zapisano we współrzędnych jednorodnych. Pozycję punktu na płaszczyźnie opisano za pomocą trzech współrzędnych. Umożliwia to zapis operacji rzutowania za pomocą działań na macierzach. Współrzędne jednorodne punktu na dwuwymiarowej płaszczyźnie rzutowania tworzą trójwymiarowy wektor $[x', y' w]^T$ taki że:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{x'}{w} \\ \frac{y'}{w} \end{bmatrix} \quad (4)$$

Wektory zawierające współrzędne o wartościach proporcjonalnych opisują ten sam punkt. Z (3) wynika: $w = Z$. Jest to matematyczny zapis rzutu wszystkich punktów $Q(\alpha X, \alpha Y, \alpha Z)$ leżących wzdłuż jednego promienia na ten sam punkt $q(x, y)$ na płaszczyźnie obrazu. W ten sposób informacja o odległości punktu od kamery została utracona.

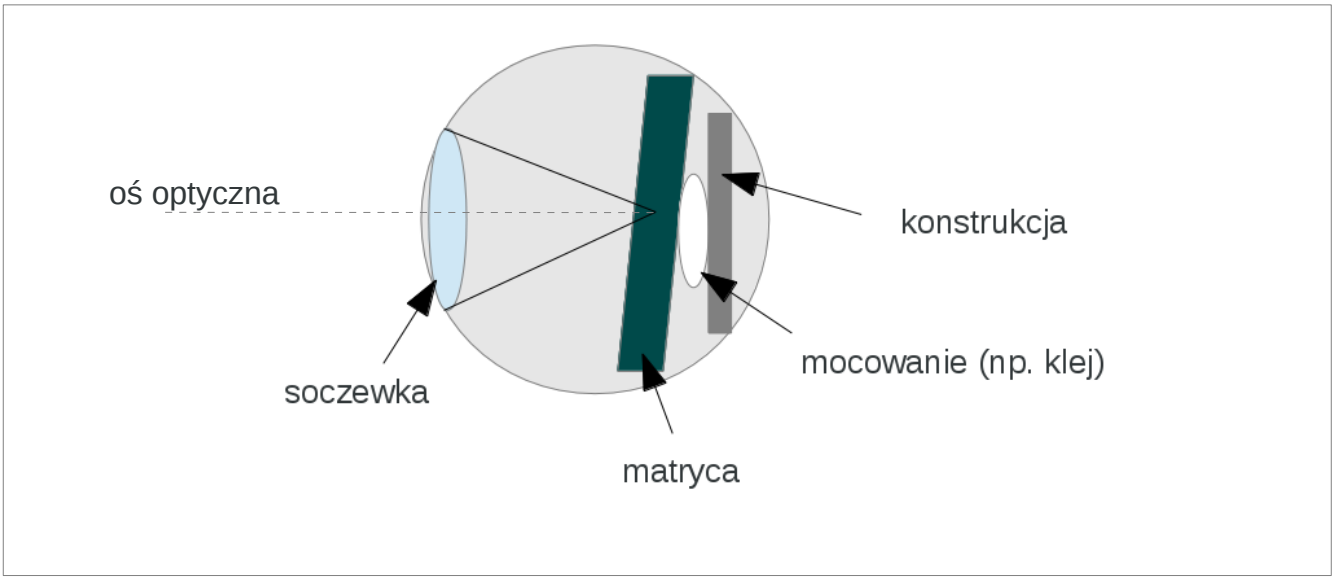
Idealny model kamery otworkowej dobrze nadaje się do analizy procesu rzutowania, natomiast w praktyce jest niewystarczający. Przez wąską szczelinę dostaje się mało światła, rejestrowanie obrazu w taki sposób trwałoby zbyt długo.

Poprawność odwzorowania punktu na płaszczyźnie obrazu jest uzależniona od

natężenia strumienia fotonów padającego na powierzchnię matrycy oraz od jej czułości [7]. Czułość matrycy jest parametrem fizycznym i w przypadku używanych kamer ma on wartość stałą. Im większe jest natężenie padającego strumienia fotonów tym krótszy jest czas konieczny do zgromadzenia w komórce matrycy odpowiedniej ilości elektronów. Odpowiedniej, to znaczy takiej, która pozwoli na prawidłowe odwzorowanie punktu na płaszczyźnie obrazu. Aby przyspieszyć proces naświetlania używa się soczewki, która ma znacznie większą aperturę i skupia większą ilość promieni na płaszczyźnie rzutowania, zwiększając intensywność padającego strumienia fotonów. W ten sposób obraz jest rejestrowany szybciej. Niestety, wraz z użyciem soczewki konieczne jest uwzględnienie wprowadzanych przez nią zniekształceń.

1.3 Zniekształcenia wprowadzone przez kamerę

Schemat konstrukcji kamer zastosowanych w projekcie przedstawiono na rys. 1.3:



Rys. 1.3. Schemat budowy prostej kamery

W modelu idealnym przyjęto brak soczewki, w praktyce jest ona używana. Przyjęto, że kształt soczewki jest idealny, co w rzeczywistości nie jest prawdą. Ponadto założono, że płaszczyzna rzutowania (powierzchnia matrycy - przetwornika CCD/CMOS) jest zamocowana idealnie prostopadle do osi optycznej i położenie to jest stałe. W rzeczywistości zamocowanie może nie być idealnie prostopadłe oraz w pewnym zakresie może się zmieniać (pod wpływem temperatury lub drgań).

Zasadniczym źródłem zniekształceń obrazu kamery jest zastosowanie soczewki lub układu soczewek [8,9]. Zniekształcenia są związane ze zmianą wartości powiększenia w zależności od odległości elementu obrazu od osi optycznej. Elementy najbliżej brzegów obrazu są najsilniej zniekształcone, mogą być słabiej powiększone (zniekształcenie

beczkowe) lub mocniej (zniekształcenie poduszkowe). Możliwe jest także występowanie obydwu efektów jednocześnie.

Istnieje więc kilka potencjalnych przyczyn zniekształceń. Istotne są zmiana długości promienia soczewki oraz brak prostopadłości powierzchni chip'u do osi optycznej.

Zniekształcenia radialne

Dokładność wykonania soczewki nie jest idealna. Jest to przyczyna zniekształceń radialnych. Biblioteka OpenCV opisuje te zniekształcenia za pomocą szeregu Taylora, którego współczynniki otrzymuje się w procesie kalibracji. Pod uwagę zostały wzięte trzy współczynniki: k_1,k_2, k_3. Poznanie ich wartości umożliwia korekcję położenia punktu obrazu zgodnie z poniższym wyrażeniem:

$$\begin{gathered} x_c = x\left(1+k_1 r^2+k_2 r^4+k_3 r^6\right) \\ y_c = y\left(1+k_1 r^2+k_2 r^4+k_3 r^6\right) \\ (x,y)\rightarrow(x_c,y_c) \end{gathered} \tag{5}$$

gdzie x i y to współrzędne przed korekcją a x_c i y_c to współrzędne po korekcji.

Zniekształcenia styczne

Przetwornik CCD/CMOS tylko w modelu idealnym jest usytuowany idealnie prostopadle i współosiowo względem osi optycznej (tak, że oś optyczna przechodzi dokładnie przez punkt środkowy płaszczyzny obrazu). Jak wspomniano wcześniej, położenie środka może się różnić od zakładanego (odległości c_x i c_y). Ponadto płaszczyzna może być także nieco odchylona od położenia idealnego. Wielkość zniekształceń stycznych również rośnie wraz ze wzrostem odległości od środka soczewki. Korekcja tych zniekształceń jest uwzględniona w bibliotece OpenCV, w tym wypadku wprowadzono dwa współczynniki p_1 i p_2. Ich wartości także otrzymuje się w procesie kalibracji kamery. Umożliwia to korekcję położenia punktu obrazu zgodnie z następującymi wyrażeniami:

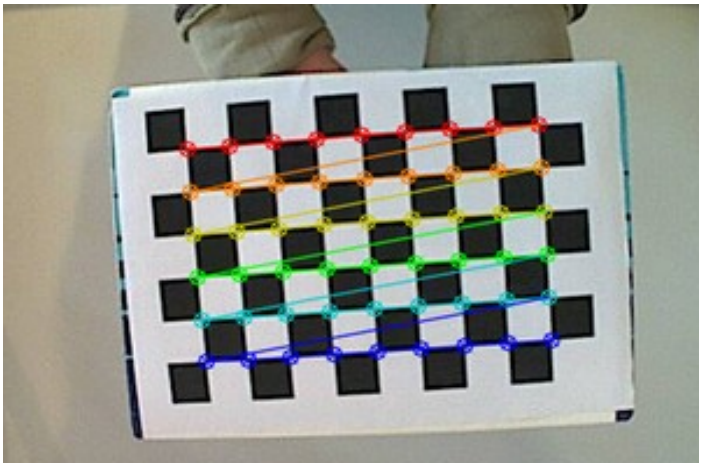
$$\begin{gathered} x_c = x+\left[2\mathrm{p}_1 y+p_2\left(r^2+2\mathrm{x}^2\right)\right] \\ y_c = y+\left[p_1\left(r^2+2\mathrm{y}^2\right)+2\mathrm{p}_2 x\right] \\ (x,y)\rightarrow(x_c,y_c) \end{gathered} \tag{6}$$

Łącznie w procesie kalibracji otrzymano pięć współczynników zaprezentowanych w (5) i (6). Biblioteka OpenCV gromadzi je razem w wektor zniekształceń opisany parametrami:

$$\left[k_1,k_2,p_1,p_2,k_3\right] \tag{7}$$

1.4 Kalibracja pojedynczej kamery

Krok pierwszy procesu kalibracji systemu to kalibracja każdej kamery z osobna. W jej rezultacie otrzymano macierz parametrów wewnętrznych kamery *M* i wektor zniekształceń (7).



Rys. 1.5 Szablon używany podczas kalibracji, kolorem zaznaczone wykryte narożniki

Kalibrację z użyciem biblioteki OpenCV przeprowadzono poprzez wykonanie serii ujęć szablonu o znanych wymiarach i zawierającego elementy łatwe do wykrycia z dużą precyzją. Wykorzystano tzw. szachownicę o wymiarach 9x6 (liczba narożników) i boku długości 25 mm. Taki szablon zaprezentowano na rys. 1.5. Kolejno, wykonane ujęcia poddano analizie polegającej na wykryciu narożników białych i czarnych pól, określeniu ich współrzędnych rzeczywistych i odniesieniu ich do położenia wyliczonego z wprowadzonych danych. Powierzchnia szachownicy jest płaska. Wykorzystano zagadnienie **odwzorowania homograficznego płaszczyzny**.

Zagadnienie dotyczy rzutowania punktów szablonu leżących w jednej płaszczyźnie na płaszczyznę obrazu.

W poniższych wyrażeniach przyjęto następujące oznaczenia:

- Q – punkt szablonu opisany w współrzędnych jednorodnych (8),
- q – punktu na płaszczyźnie obrazu we współrzędnych jednorodnych (9),
- s – współczynnik skalujący,
- Q' – punkt Q określony dla obserwowanej płaszczyzny Z=0,
- M – macierz wewnętrzna kamery (11),
- W – macierz rotacji i przesunięcia płaszczyzny szablonu względem płaszczyzny obrazu opisana wyrażeniem (10),
- H – macierz homografii (12) łącząca wyrażenia (7) i (8),
- R_{3x3} – macierz obrotu wokół trzech osi współrzędnych (r_1, r_2, r_3 to jej kolumny),

- t_{3x1} – wektor przesunięcia,

$$Q=[X\ Y\ Z\ 1]^T \tag{8}$$

$$q=[x\ y\ 1]^T \tag{9}$$

Przekształcenie można wyrazić następująco:

$$q=s\ H\ Q \tag{10}$$

gdzie:

$$W=[R_{3x3}\ \ t_{3x1}]=[r_1\ r_2\ r_3\ t_{3x1}], \tag{11}$$

$$M=\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \tag{12}$$

$$H=M\ W \tag{13}$$

Odwzorowanie (13) składa się z dwóch etapów. W pierwszym następuje obrót i przesunięcie płaszczyzny szablonu tak, by znajdowała się równolegle do płaszczyzny obrazu w punkcie $Z=0$, ustawiona centralnie. W drugim odbywa się proces rzutowania, w którym pod uwagę brane są parametry wewnętrzne kamery.

Macierz homografii rzutuje punkty płaszczyzny szablonu na płaszczyznę obrazu. Odwzorowanie homograficzne płaszczyzny przekształca czworobok w inny czworobok. Dzięki przyjęciu $Z=0$ macierz homografii ma rozmiar 3x3. Przyjęto oznaczenia:

- p_{src} – pozycja pierwotna punktu,
- p_{dst} – pozycja docelowa punktu.

Każdy punkt płaszczyzny szablonu poddano następującemu przekształceniu:

$$p_{dst}=\begin{bmatrix} x_{dst} \\ y_{dst} \\ 1 \end{bmatrix}=H\ p_{src}=H\begin{bmatrix} x_{src} \\ y_{src} \\ 1 \end{bmatrix} \tag{14}$$

a dzięki założeniu $Z=0$ wyrażenie (10) przyjmuje postać:

$$q = s\ M\ W\ Q, \tag{15}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = s\ M \begin{bmatrix} r_1\ r_2\ r_3\ t \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z=0 \\ 1 \end{bmatrix} = s\ M \begin{bmatrix} r_1\ r_2\ t \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}, \tag{16}$$

$$q = s\ M\ W\ Q' \tag{17}$$

Znalezienie parametrów wewnętrznych kamery i wektora zniekształceń jest celem procesu kalibracji.

W tym celu obliczono macierz przekształcenia homograficznego dla wielu ujęć. W pierwszej kolejności szukano składników wektora zniekształceń (7) związanych z dwuwymiarową geometrią przyjętego modelu zniekształceń. Do ich obliczenia z dużą dokładnością wystarcza jedno ujęcie szablonu. Ten etap wykonano osobno. Następnie obliczono parametry zewnętrzne: 3 kąty (obrót wokół każdej osi współrzędnych) i 3 współczynniki przesunięcia (dla każdego wektora jednostkowego) co daje 6 niewiadomych zmiennych dla każdego ujęcia. Kolejno otrzymano parametry wewnętrzne kamery tworzące macierz (12), które są stałe dla danej kalibracji. Kalibrację każdorazowo przeprowadza się dla określonej rozdzielczości akwizycji i ustawionej pozycji kamer.

Jeżeli *K* jest liczbą ujęć z różnych perspektyw, to:

- wykorzystując macierz homografii (13), dla każdego ujęcia tylko 4 punkty dostarczają użytecznej informacji (reszta punktów na tej samej płaszczyźnie nie dostarcza informacji), szablon użyty w projekcie spełnia ten warunek (54 wierzchołków do wykrycia)
- *K* ujęć dostarcza 2*4*K* równań (2- dla osi *x* i *y*),
- poszukiwano 4 parametry wewnętrzne oraz 6*K* parametrów zewnętrznych,
- minimalna liczba ujęć spełnia nierówność $2*4K \geqslant 4+6K$

Teoretycznie minimalną liczbą koniecznych ujęć jest *K*=2. Z uwagi na szumy, ograniczone pokrycie kadru szablonem i dokładność obliczeń dostarczono większej ilości ujęć (około 15). Istotne jest aby szablon był usytuowany w kadrze w różnych konfiguracjach ułożenia.

Kalibrację każdej z kamer przeprowadzono osobno korzystając z funkcji biblioteki OpenCV. Krok kolejny stanowi poznanie wzajemnego położenia kamer pracujących w układzie stereoskopowym względem siebie.

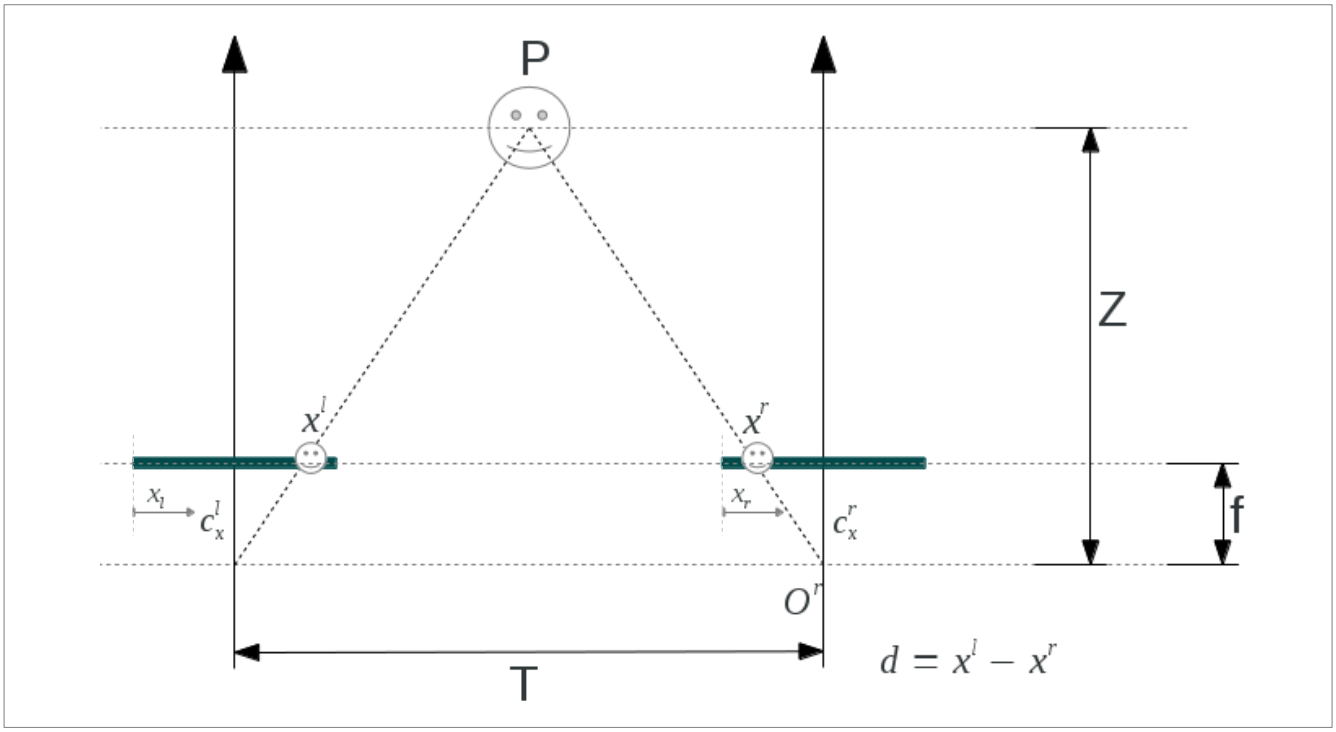
1.5 Kalibracja układu stereoskopowego

Celem kalibracji układu stereoskopowego jest znalezienie macierzy rotacji i przesunięcia

pomiędzy lewą i prawą kamerą. Aby otrzymać pożądane zależności przeanalizowano geometrię układu, którego nadrzędną cechą jest możliwość opisania obserwowanych przedmiotów w 3 wymiarach. Byłoby to niemożliwe za pomocą obrazu z jednej kamery.

Głębia sceny

W poniższym rozważaniu założono, że istnieje zestaw stereoskopowy, w którym obrazy kamer są idealnie wyrównane w poziomie (wspólna pionowa oś współrzędnych), leżą w jednej płaszczyźnie (osi optyczne idealnie równoległe), kamery mają tą samą długość ogniskowej $f^l = f^r$ i takie samo przesunięcie środka płaszczyzny obrazu względem osi optycznej $c_x^l = c_x^r$. Punkt *P* jest widoczny na obrazach obydwu kamer i posiada odpowiednio współrzędne w poziomie x^l i x^r dla lewego i prawego obrazu. Schemat tak ustawionego zestawu kamer zilustrowano na rys. 1.6.



Rys. 1.6. Schemat idealnego zestawu stereo

Różnicę położeń obrazów tego samego punktu została oznaczona jako *d*. Jest ona tym większa im mniejsza jest odległość *Z* przedmiotu od zestawu kamer. Z podobieństwa trójkątów wynikają następujące zależności:

$$\frac{T-\left(x^l-c_x^l\right)+\left(x^r-c_x^r\right)}{Z-f}=\frac{T}{Z} \tag{17}$$

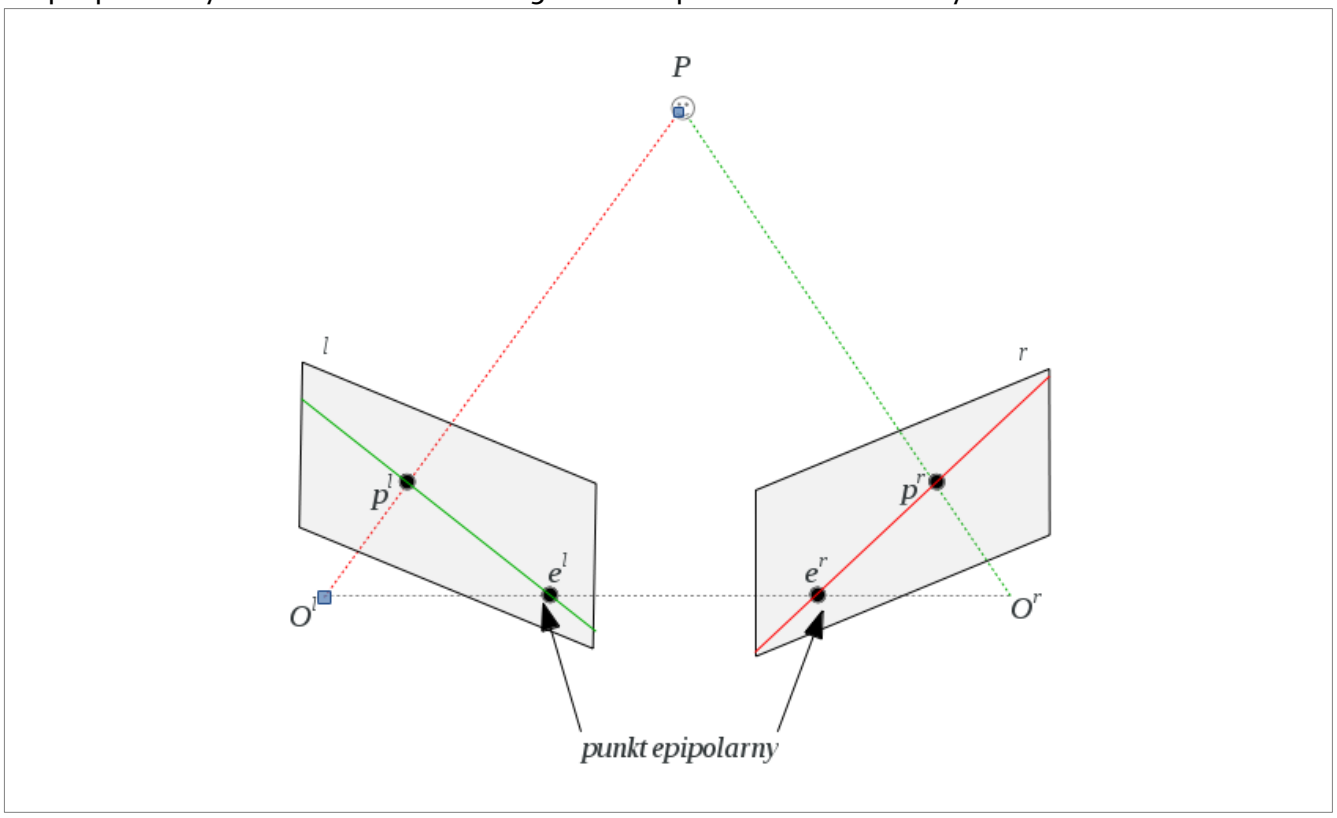
$$Z=\frac{f\ T}{x^l-x^r-\left(c_x^l-c_x^r\right)}=\frac{f\ T}{d-\left(c_x^l-c_x^r\right)}=\ \left|c_x^l=c_x^r\right|\ =\frac{f\ T}{d} \tag{18}$$

Głębia jest więc odwrotnie proporcjonalna do odległości przedmiotu dla osi optycznych idealnie równoległych. Dla przedstawionego idealnego układu ta różnica zmierza do zera dla $Z \rightarrow \infty$.

W rzeczywistości tak idealne warunki są niemożliwe do osiągnięcia. Konieczna jest więc metoda umożliwiająca przekształcenie obrazów odbieranych przez kamery do postaci, w której będą wyglądać tak, jakby płaszczyzny rzutowania kamer były współpłaszczyznowe i wyrównane w poziomie. Aby dokonać tych przekształceń posłużono się geometrią układu rzeczywistego.

Geometria epipolarna

Przeanalizowano model geometryczny układu dwóch kamer otworkowych (z zastrzeżeniem dodatkowych zniekształceń dla każdej z kamer wnoszonych przez soczewkę) w rzeczywistej konfiguracji układu stereoskopowego. Powierzchnie rzutowania nie są współpłaszczyznowe. Schemat takiego układu przedstawiono na rys 1.7:



Rys. 1.7. Geometria epipolarna układ dwóch kamer

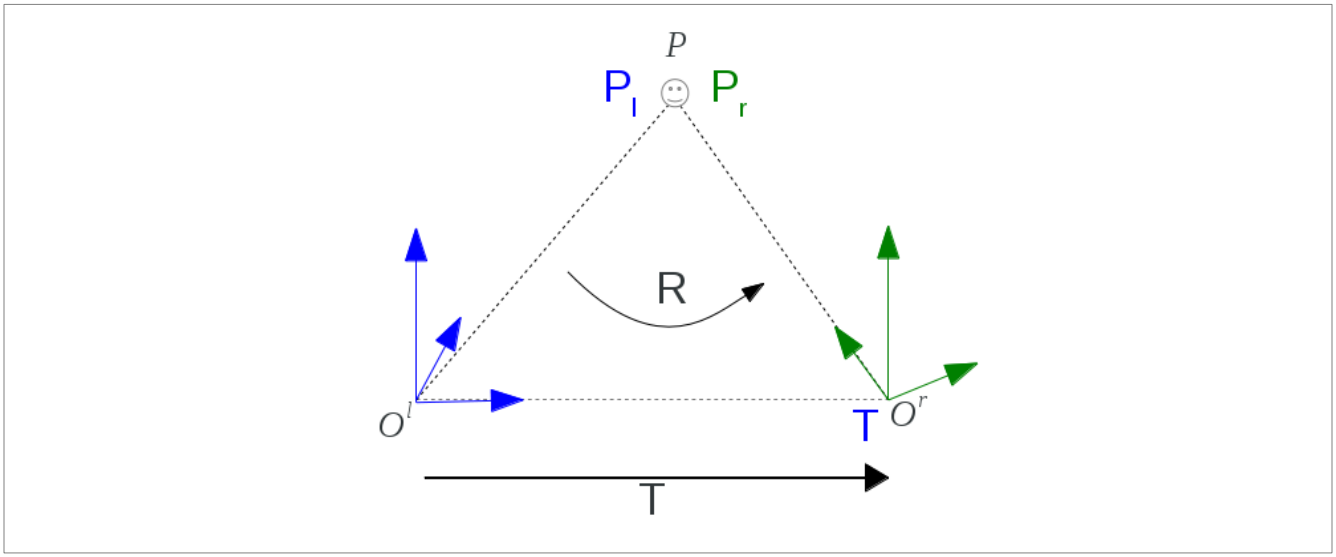
Wprowadzono następujące oznaczenia: punkty p^l i p^r są obrazami tego samego punktu *P* odpowiednio dla lewej i prawej kamery. **Punkt epipolarny** dla lewej kamery e^l to odwzorowanie środka rzutowania kamery prawej O^r na płaszczyznę rzutowania l . Innymi słowy jest to punkt, w którym na obrazie lewej kamery byłby widoczny środek rzutowania

kamery prawej (jeżeli znajdowałby się w obszarze kadru). **Linia epipolarna** dla płaszczyzny l przechodząca przez p^l i e^l jest rzutem odcinka PO^r na tą płaszczyznę. Płaszczyzna zawierająca punkty *P*, O^r , O^l jest nazywana **płaszczyzną epipolarną**.
Z perspektywy punktu O^r punkt *P* jest widoczny jako obraz p^r , nie można określić jego odległości od O^r . Wiadomo natomiast, że rzut wszystkich możliwych lokalizacji punktu *P* tworzy linię epipolarną na płaszczyźnie l . Wzdłuż tej linii należy szukać obrazu punktu *P* obserwowanego z perspektywy punktu O^l czyli punktu p^l . Analogicznie dla drugiej strony.

Oznacza to, że znając geometrię danego układu stereoskopowego możliwe jest wyznaczenie odcinka, wzdłuż którego leży punkt *P* (jak dla pojedynczej kamery) oraz odległości punktu *P* od O^r na podstawie położenia odpowiadającego mu punktu p^l wzdłuż linii epipolarnej na płaszczyźnie l i jego odległości od punktu e^l .

W celu opisania tych przekształceń wartościami liczbowymi wprowadzono pojęcia macierzy *E* (eng. *Essential matrix*) i *F* (eng. *Fundamental matrix*). Macierz *E* opisuje geometryczne przesunięcie i rotację kamer względem siebie (łącząc pozycję punktu na obrazie jednej kamery z linią na drugim obrazie), macierz *F* zawiera dodatkowo informacje o parametrach wewnętrznych obydwu kamer - opisuje te relacje w pikselach.

Niech P_l i P_r oznaczają fizyczną lokalizację punktu *P* w układach współrzędnych o środkach odpowiednio w O^l i O^r jak przedstawiono na rys. 1.8. Położenie punktu O^r we współrzędnych układu o środku O^l oznaczone będzie jako *T*, jest to zarazem wektor przesunięcia układu O^r względem O^l . Obrót oznaczony będzie macierzą *R*.



Rys. 1.8. Związek między układami współrzędnych O^l i O^r

Związek między współrzędnymi punktu *P* w obydwu układach jest następujący:

$$P_r = R(P_l - T) \tag{19}$$

Mając wektor prostopadły do płaszczyzny epipolarnej:

$$(T \times P_l) \tag{20}$$

równanie płaszczyzny zawierającej punkt P_l i przechodzącej przez punkt T można zapisać:

$$(P_l - T)^T (T \times P_l) = 0 \tag{21}$$

a także:

$$(P_l - T) = R^{-1} P_r \ , \tag{22}$$

$$R^T = R^{-1} \tag{23}$$

podsumowując, prawdziwe jest wyrażenie:

$$(R^T P_r)^T (T \times P_l) = 0 \tag{24}$$

zapisując iloczyn wektorowy jako mnożenie:

$$S = \begin{bmatrix} 0 & -T_z & T_y \\ T_y & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix} \Rightarrow (R^T P_r)^T S P_l = P_r^T R S P_l = 0 \tag{25}$$

wyrażenie opisujące macierz E ma postać:

$$E = R S \tag{26}$$

$$P_r^T E P_l = 0 \tag{27}$$

Macierz E (26) łączy fizyczne położenie tego samego punktu względem dwóch układów współrzędnych o środkach O^l i O^r . Docelowo szukane jest wyrażenie na relację punktów obserwowanych na płaszczyźnie obrazu. Zgodnie z (1) gdy $c_x = c_y = 0$:

$$\frac{p_l}{f_l} = \frac{P_l}{Z_l} \Rightarrow P_l = p_l \frac{Z_l}{f_l} \tag{28}$$

$$\frac{Z_r}{f_r} p_r^T E \frac{Z_l}{f_l} p_l = 0, \quad \frac{Z_r}{f_r} \neq 0 \wedge \frac{Z_l}{f_l} \neq 0 \tag{29}$$

$$p_r^T E p_l = 0 \tag{30}$$

W następnej kolejności należy znaleźć wyrażenie na macierz F. Macierz M wyraża relację pomiędzy punktem obrazu we współrzędnych rzeczywistych p a współrzędnymi tego punktu w pikselach:

$$q = M p \Rightarrow M^{-1} q = p \tag{31}$$

Podstawiając (31) do (30) otrzymuje się:

$$(M_r^{-1} q_r)^T E M_l^{-1} q_l = 0 \tag{32}$$

$$q_r^T \underbrace{M_r^{-T} E M_l^{-1}}_{F} q_l = 0 \Rightarrow q_r^T F q_l = 0 \tag{33}$$

Kalibracja układu stereoskopowego polega na znalezieniu relacji geometrycznej między dwoma kamerami na podstawie obserwacji tego samego punktu jednocześnie. W tym celu , jednocześnie, obydwoma kamerami wykonano ujęcie szablonu (szachownicy – patrz rys. 1.5). Wykorzystano funkcje biblioteki OpenCV, parametry wejściowe to macierze *M* i wektory zniekształceń dla lewej i prawej kamery otrzymane w procesie kalibracji pojedynczej. Parametrami wyjściowymi będą macierze *R*, *T*, *E*, *F*.

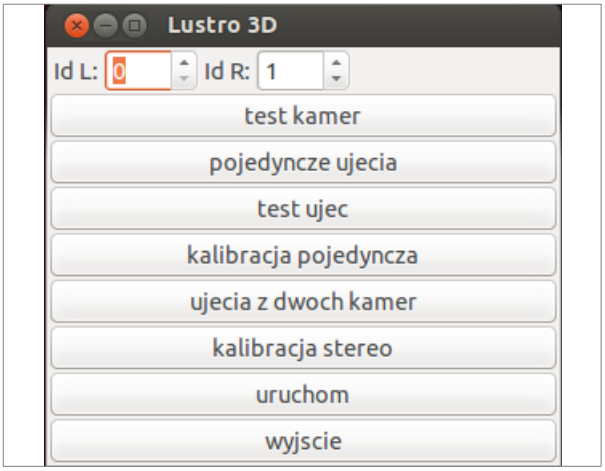
Otrzymane dane umożliwiają przekształcenie obrazów z obydwu kamer tak, by były współpłaszczyznowe a w następnej kolejności dopasowanie ich do siebie w poziomie. Dzięki temu odpowiadające sobie elementy obrazu w obydwu ramkach znalazły się w tej samej linii.

2. System do wyświetlania stereoskopowych sekwencji wideo

W poniższym rozdziale opisano stworzony system, którego zadaniem jest przejęcie ramek ze strumieni wideo, przetworzenie ich i wyświetlenie jako sekwencji ramek stereoskopowych. Na początku omówiono architekturę systemu od strony sprzętowej i programowej. Następnie przedstawiono strukturę wewnętrzną programu oraz sposób otrzymania plików wykonywalnych z plików źródłowych i użytych bibliotek zewnętrznych. Scharakteryzowano po krótce poszczególne moduły programu. W kolejnej sekcji od strony logicznej podzielono proces na kolejne etapy. Na końcu zamieszczono fragmenty kodu implementujące najważniejsze funkcje programu wraz z opisem.

2.1 Architektura systemu

Stanowisko, na którym opracowano system tworzą dwie kamery USB podłączone do komputera PC i monitor 3D wyświetlający obraz wyjściowy. Oprogramowanie napisano w języku C i C++ na platformę Linux. Wykorzystano bibliotekę OpenCV do operacji kalibracji kamer, przetwarzania i wyświetlania obrazu. W celu zrównoleglenia obliczeń, wykorzystując procesor wielordzeniowy, użyto biblioteki pthread. Dzięki temu w łatwy sposób można było zaimplementować równoległe przetwarzanie odpowiadających sobie ramek rejestrowanych przez obydwie kamery. Program wyposażono w prosty interfejs użytkownika zaprezentowany na rys. 2.1. Umożliwia on przypisane poprawnego identyfikatora odpowiednio dla lewej i prawej kamery, przeprowadzenie testu ustawienia kamer, wykonanie kalibracji i ostatecznie uruchomienie wyświetlania sekwencji stereoskopowych.

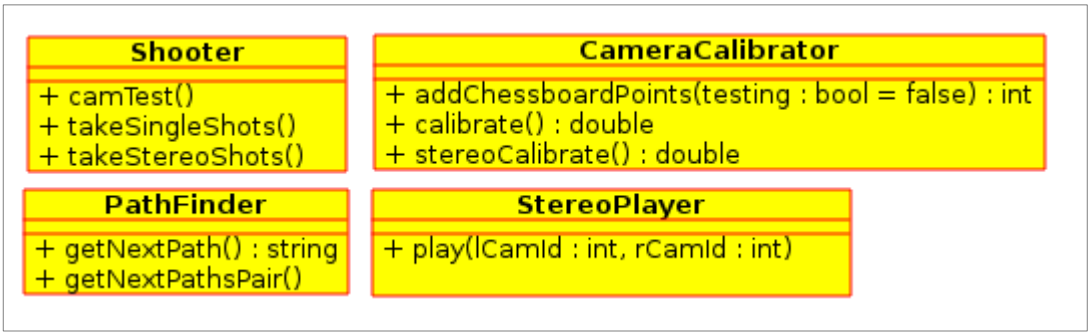


Rys. 2.1 Wygląd stworzonego interfejsu graficznego

2.2 Struktura programu

Kompilacja kodu źródłowego i proces linkowania z używanymi bibliotekami jest wykonywana przez program *cmake*. Aplikacja zawiera potrzebne skrypty konfiguracyjne. Ostatecznie, otrzymywane są dwa pliki wykonywalne *play* i *sv*. Plik *sv* jest uruchamiany przez użytkownika. Odpowiada za wyświetlenie interfejsu graficznego, uruchomienie wszystkich funkcji oraz pliku *play* w osobnym procesie. Plik *play* zajmuje się częścią zasadniczą programu jaką jest akwizycja, przetwarzanie ramek i wyświetlanie obrazu stereoskopowego.

Na rys. 2.3 przedstawiono uproszczony diagram klas wraz z funkcjami publicznymi. W oparciu o niego omówiono poszczególne funkcjonalności programu.



Rys.2.3. Uproszczony diagram klas z funkcjami publicznymi

Kod źródłowy można podzielić na kilka modułów ze względu na funkcjonalność, którą implementuje każdy z nich:

1. Test poprawnego ustawienia kamer
   Umożliwia sprawdzenie czy identyfikatory dla kamery lewej i prawej, ustawiane z poziom interfejsu graficznego, są poprawne. Odpowiada za to funkcja `camTest` w klasie `Shooter`. Przykładowo, identyfikator kamery lewej (przypisany przez system

operacyjny) ma wartość 1. Jeżeli taką sama wartość ustawiono z poziomu interfejsu graficznego to podczas testu w oknie "L" pojawi się obraz z kamery lewej. Wartość identyfikatora jest liczbą całkowitą z zakresu 0-2. Jeżeli urządzenie o danym identyfikatorze nie jest podłączone to program zostanie przerwany z błędem. Ponadto, test służy ręcznemu ustawieniu kamer. Ustawiono je tak, aby były jak najlepiej wyrównane w poziomie i obserwowały mniej więcej tą samą scenę.

2. Wykonanie zdjęć dla celów kalibracji
   Wykonanie zdjęć dla kalibracji pojedynczej odbywa się wewnątrz funkcji `takeSingleShots` w klasie `Shooter`. Obraz aktualnie obserwowany przez daną kamerę jest wyświetlany w utworzonym oknie. Po wykryciu szablonu kalibracyjnego poszczególne narożniki zostają zaznaczone kolorem jak na rys. 1.5 oraz uruchamiany jest licznik czasu. Położenie 4 skrajnych narożników zostaje zapamiętane. Po upływie 2 sekund ich współrzędne są sprawdzone ponownie i jeżeli nie zmieniły się o więcej niż 25 pikseli w pionie i poziomie to następuje zapis ramki do pamięci operacyjnej. Gdy licznik ramek w pamięci osiąga wartość 15 wtedy wszystkie są zapisane na dysku a ścieżki tych plików w pliku konfiguracyjnym `calibConf.yml`. Umożliwia to przeprowadzenie kalibracji przez jedną osobę. Procedura dla jednoczesnych ujęć z obydwu kamer jest analogiczna z tą różnicą, że przechowywanie ramek nie jest wykonywane automatycznie a za pomocą naciśnięcia klawisza. Przejmowane są tylko 3 ujęcia a usytuowanie szablonu w kadrze nie ma tak istotnej wagi dla jakości kalibracji (ważne aby w obydwu kadrach widoczny był cały szablon) dlatego wybrano prostszą metodę. Te operacje są wykonywane wewnątrz funkcji `takeStereoShots` w klasie `Shooter`.

3. Test wykonanych ujęć
   W momencie automatycznego przechwytywania ujęć zapisywane są tylko te, na których możliwe jest wykrycie wszystkich narożników. Pomimo tego, może zdarzyć się, że detekcja narożników na zapisanym ujęciu jest niemożliwa, czego doświadczono w trakcie testowania programu. Przykładowo ujęcie może zostać nieznacznie poruszone w momencie zapisu do pamięci, wskutek czego narożniki będą rozmyte co utrudnia ich detekcję z dużą precyzją. Wystąpienie tego błędu jest także możliwe podczas wykonywania ujęć, na których szablon znajduje się blisko krawędzi obrazu, na granicy poprawnego działania algorytmu wykrywającego. Wtedy nawet nieznacznie poruszenie może spowodować, że zbyt mały obszar szablonu będzie widoczny. Powoduje to błąd i przerwanie kalibracji. Dlatego przed przystąpieniem do tego procesu należy sprawdzić poprawność wszystkich przejętych ujęć dla lewej i prawej kamery. Odpowiada za to funkcja `addChessboardPoints` wywołana z argumentem typu `bool` o wartości `true`. Funkcja wyświetla po kolei ujęcia szablonu wraz z

zaznaczonymi narożnikami. Wykonanie zostanie przerwane błędem lub część narożników zostanie zaznaczona kolorem czerwonym jeżeli na którymś ujęciu niemożliwa jest detekcja wszystkich narożników. W tym wypadku wiadomo jednak dokładnie, które ujęcie jest wadliwe. Ścieżkę do niego należy wtedy ręcznie usunąć z pliku konfiguracyjnego `calibConf.yml`. Po tym należy przeprowadzić test ponownie, do momentu aż całość zostanie wykonana poprawnie.

4. Kalibracja
   Proces ten, jak wspomniano wcześniej, podzielono na dwa etapy. Najpierw przeprowadzona się kalibrację każdej z kamer indywidualnie w oparciu o wykonane wcześniej ujęcia oraz znane wymiary szablonu. Odczyt i zwrócenie ścieżek kolejnych ujęć wykonuje funkcja `getNextPath` w klasie `PathFinder`. Kalibracja pojedyncza jest wykonywana w dwóch iteracjach dla każdej z kamer. Początkowo parametry wewnętrzne kamery oraz wektor zniekształceń są obliczane z pewnym przybliżeniem a otrzymane wartości zapisane w pliku `calibInit.yml`. W drugim powtórzeniu tego samego procesu służą one jako wartości początkowe algorytmu, który oblicza je tym razem z większą precyzją. Po tym dwuetapowym procesie ostateczny rezultat jest zapisywany do pliku `calibConf.yml`. Za kalibrację pojedynczą każdej z kamer odpowiada funkcja `calibrate` z klasy `CameraCalibrator`. Krok kolejny to kalibracja układu stereoskopowego i obliczenie parametrów koniecznych do późniejszego dopasowania odbieranych ramek. Z pliku odczytywane są parametry otrzymane w efekcie kalibracji pojedynczych. Kalibrację układu stereoskopowego przeprowadza funkcja `stereoCalibrate` w klasie `CameraCalibrator`. Odczytanie ścieżek wykonanych wcześniej jednoczesnych ujęć z obydwu kamer wykonuje funkcja `getNextPathsPair` z klasy `PathFinder`. Funkcja zwraca pary ścieżek do odpowiadających sobie ujęć. Rezultatem kalibracji układu stereoskopowego są macierze *R*, *T*, *E*, *F* omówione w rozdziale 1.5. Na końcu następuje zapis tych parametrów w pliku `stereoCalib.yml`.

5. Akwizycja, przetworzenie i wyświetlenie sekwencji ujęć stereoskopowych
   Główną funkcjonalność programu stanowi wyświetlenie stereoskopowych sekwencji wideo rejestrowanych przez zestaw kamer. Wszystkie poprzednie kroki są konieczne do poprawnej realizacji tego zadania. Uruchomienie, podobnie jak wszystkich innych etapów, następuje z poziomu interfejsu graficznego. Tworzony jest proces potomny, w którym zostaje uruchomiony program `play`. Zostało to zilustrowane w tabeli 2.1. Jako argumenty przyjmuje identyfikatory lewej i prawej kamery ustawione na etapie testowania. Proces rodzica oczekuje na zakończenie procesu potomnego. Za całość wykonania odpowiada funkcja `play` w klasie `StereoPlayer`, do której przekazane są argumenty wywołania programu jak pokazano w tabeli 2.2.

```
void playStereo(int lId, int rId){
    pid_t pid;
    pid=fork();
    if(!pid){
        char left[2];
        char right[2];
        sprintf(left,"%d",lId);
        sprintf(right,"%d",rId);
        char* argv[]={"bin/play",left,right,NULL};
        execvp(argv[0],argv);
        std::cerr<<"execvp error"<<std::endl;
    }
    else{
        int exit_code;
        wait(&exit_code);
        std::cout<<" parent finish "<<std::endl;
    }
}
```

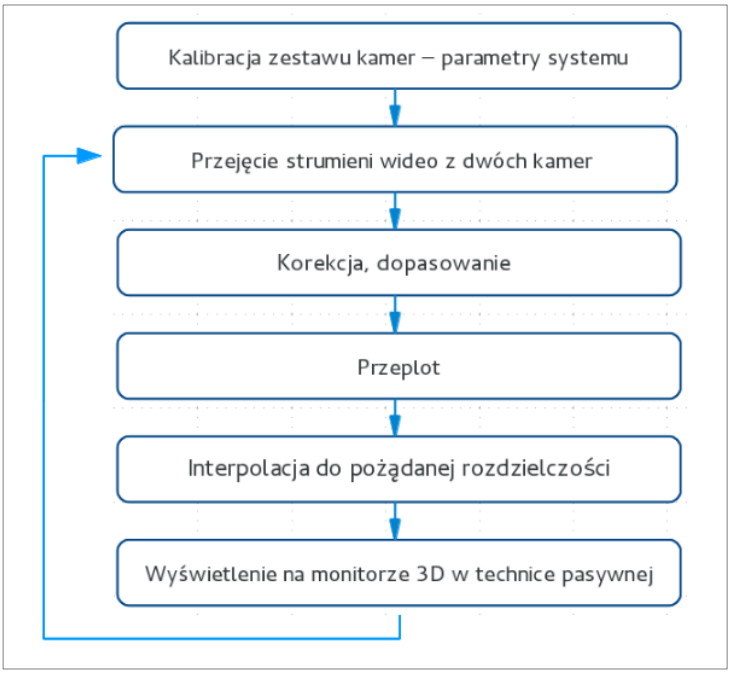
Tabela 2.1 Utworzenie procesu potomnego uruchamiającego drugi program

```
int main(int argc, char** argv){
    int l = atoi(argv[1]);  //identyfikator lewej kamery
    int r = atoi(argv[2]);  //identyfikator prawej kamery
    std::cout<<"child hello: "<<l<<" "<<r<<std::endl;
    /* uruchomienie wyświetlania */
    StereoPlayer sp;
    sp.play(l, r);
    return 0;
}
```

Tabela 2.2 Kod programu uruchamianego w nowym procesie

2.3 Akwizycja ramek, przetwarzanie i kompozycja sceny 3D

W trakcie montażu kamery ustawiono tak, by ich obrazy były jak najbardziej współpłaszczyznowe i wyrównane w poziomie na tyle na ile jest to możliwe ręcznie. W pierwszym etapie przeprowadza się kalibrację układu. Następne kroki wykonywane są w pętli. Porządek kolejnych etapów działania systemu ilustruje rys. 2.4:



Rys.2.4. Kolejne etapy wykonania programu

Najpierw odpowiadające sobie ramki są odbierane przez każdą z kamer. Zakłada się, że są przejęte w tym samym momencie choć nie ma takiej gwarancji. Kamery tej klasy nie są wyposażone w sprzętowy wyzwalacz. Jeżeli momenty przejęcia obrazu między kamerami będą się nieco różnić to może to powodować widoczne błędy wyświetlania w przypadku szybko zmieniającego się obrazu.

Kolejno, dla przejętych ujęć następuje korekcja zniekształceń w oparciu o wektor zniekształceń danej kamery. Po tym następuje etap dopasowania. Dla obydwu ramek wybierany jest fragment obrazu taki, by miały maksymalną część wspólną oraz były wyrównane w poziomie. Na tym etapie tracona jest część przejmowanego obrazu. Od ustawienia kamer i procesu kalibracji zależy jak duża część będzie użyteczna. W efekcie otrzymuje się dwa obrazy przejęte z różnych perspektyw, ale o tych samych rozmiarach i odpowiadających sobie elementach w tych samych liniach.

Operacja przeplotu, wykonywana w następnej kolejności, polega na wytworzeniu ramki wynikowej z dwóch ramek z poprzedniego etapu w ten sposób, że z jednej ramki wzięte są linie parzyste a z drugiej nieparzyste.

Powstała ramka wynikowa jest w tym momencie niemal gotowa do wyświetlenia. Ponieważ jednak rozdzielczość, w której rejestrują kamery (1280x720) jest mniejsza od rozdzielczości wyświetlania monitora (1920x1080) konieczna jest interpolacja do rozdzielczości wyjściowej. Po wykonaniu tej operacji ramka zostaje wyświetlona.

Przedstawione kroki są wykonywane dla każdej ramki. Wyświetlanie odbywa się z prędkością 30 ramek na sekundę. Jako efekt na monitorze pracującym w trybie wyświetlania 3D otrzymujemy pożądaną sekwencję ramek. Oglądając obraz w przystosowanych do tego okularach, wyposażonych w filtr polaryzacyjny, obserwujemy wrażenie trójwymiarowości.

2.4 Implementacja

W rozdziale tym omówiono implementację kluczowych elementów systemu. Składają się na nie następujące czynności: przejęcie ramek ze strumienia, przetworzenie ich i wyświetlenie stereoskopowej sekwencji ramek na ekranie.

Założono, że wyświetlanie powinno odbywać się z minimalną prędkością 25 ramek na sekundę. Oznacza to, że w ciągu każdej sekundy należy odebrać i przetworzyć 50 ramek a następnie dokonać operacji przeplotu i interpolacji aby otrzymać 25 ramek wyjściowych gotowych do wyświetlenia. Narzuca to określone wymagania względem szybkości działania programu. Należy jednak zauważyć, że część operacji jest wykonywana na niezależnych ramkach lub liniach wewnątrz ramki. Z tego powodu ich wykonanie zostało zrównoleglone wykorzystując wiele wątków procesora do wykonania tych operacji jednocześnie. Efektem jest otrzymanie ramki wyjściowej szybciej. Zastosowane rozwiązanie opiera się na wątku głównym i dwóch wątkach przetwarzających. Do utworzenia dodatkowych wątków w obrębie tego samego procesu wykorzystano systemową bibliotekę pthread.

Wątki przetwarzające są tworzone za pomocą funkcji `pthread_create` jak pokazano w tabeli 2.3.

```
    /* utworzenie wątków przetwarzających ramki */
    for(size_t i=0;i<2;i++)
        if(pthread_create(&tid[i],NULL,&StereoPlayer::threadRunner,
            static_cast<void*>(tData+i)))
        {
            std::cerr<<"thread "<<i<<" creation error"<<std::endl;
        };
```

Tabela. 2.3 Utworzenie dwóch wątków przetwarzających w obrębie tego samego procesu

Jako argumenty przekazywany jest między innymi adres funkcji `threadRunner` oraz wskaźnik do obiektu struktury `ThreadData`, którą przedstawia tabela 2.4.

```
struct ThreadData{
    int threadId;    //identyfikator wątku
    int info;        //identyfikator strumienia
    StereoPlayer *obj;    //obiekt klasy
    /* wskaźnik do funkcji składowej klasy*/
    void (StereoPlayer::*threadFun)(int tid,int maxFrames);
};
```

Tabela. 2.4 Składniki struktury ThreadData, wskaźnik do obiektu klasy StereoPlayer jest konieczny, aby móc wywołać funkcję składową tej klasy

Funkcja `threadRunner` jest wywoływana przez funkcję `pthread_create` w nowo utworzonym wątku i pełni rolę pośredniczącą. Kod tej funkcji zaprezentowano w tabeli 2.5. W oparciu o dane dostarczone przez strukturę `ThreadData` uruchamia właściwą funkcję wykonującą operację przetwarzania.

```
void* StereoPlayer::threadRunner(void *p){
    /* funkcja pośrednicząca
     * uruchamiana przez pthread_create,
     * uruchamia właściwą funkcję wykonywaną przez wątek;
     * jest to funkcja składowa klasy StereoPlayer
     * dlatego konieczny jest specjalny wskaźnik funPtr
     * oraz obiekt tej klasy, na rzecz którego można tą
     * funkcję wywołać; wszystko dostarczone
     * wewnątrz struktury ThreadData
     */
    ThreadData *td = static_cast<ThreadData*>(p); //struktura z danymi
    StereoPlayer *sm = td->obj;
    void (StereoPlayer::*funPtr)(int t,int m) = td->threadFun;
    (sm->*funPtr)(td->threadId,td->info);
    return NULL;
}
```

Tabela 2.5 Wywołanie właściwej funkcji wykonywanej przez wątki przetwarzające wewnątrz funkcji pośredniczącej threadRunner

Z tego powodu struktura `ThreadData` musi zawierać wskaźnik do aktualnego obiektu klasy `StereoPlayer`. Na rzecz tego obiektu wywołana jest funkcja składowa, na którą ustawiono wskaźnik `threadFun`. Funkcja przyjmuje dwa argumenty, które także znajdują się wewnątrz struktury `ThreadData`. Składowa `threadId` przechowuje identyfikator aktualnego wątku natomiast składowa `info` identyfikator strumienia ramek otwieranego w danym wątku (ustawiany z poziomu interfejsu).

W momencie kończenia pracy programu wątek główny czeka aż utworzone wątki się zakończą wewnątrz funkcji `pthread_join` co ilustruje tabela 2.6.

```
   /* czekaj na zakończenie wątków przetwarzających */
    for(size_t i=0;i<2;i++)
        pthread_join(tid[i],NULL);
```

Tabela 2.6 Oczekiwanie na zakończenie utworzonych wątków w wątku głównym

W wątku głównym następuje utworzenie okna wyświetlającego przetworzone ramki wyjściowe a także uruchomienie dwóch dodatkowych wątków przetwarzających. Wątki przetwarzające otwierają strumień ramek odpowiednio dla lewej i prawej kamery. Po tym następuje odebranie ramki ze strumienia, korekcja i dopasowanie, przekopiowanie odpowiednich linii do ramki wyjściowej i zasygnalizowanie wątkowi głównemu obecność nowej ramki w buforze. Sygnalizacja odbywa się poprzez mechanizm semaforów dla wątków. Kolejne ramki wyjściowe są umieszczane w buforze kołowym i z niego są następnie odczytywane przez wątek główny. Bufor jest tablicą 5 elementową zawierającą obiekty typu `cv::Mat` zainicjalizowane wartościami początkowymi. Wątki przetwarzające nadpisują elementy bufora w sposób sekwencyjny po jednym dla każdej pary odebranych ramek począwszy od elementu o indeksie 0. Po nadpisaniu ostatniego elementu następna ramka nadpisze znów element o indeksie 0.

Mechanizm semaforów dla wątków opiera się na funkcjach: `sem_init`, `sem_wait` oraz `sem_post`, które operują na wspólnej zmiennej `sem` typu `sem_t`. Funkcja `sem_init` inicjalizuje semafor wartością początkową 0, `sem_post` inkrementuje jego wartość a `sem_wait` czeka do momentu aż wartość semafora będzie większa od 0, wtedy zmniejsza ją o jeden i kończy się. Wątek główny oczekuje wewnątrz funkcji `sem_wait` aż co najmniej jedna ramka będzie gotowa do wyświetlenia (wartość zmiennej `sem` będzie większa od 0), wtedy dokonuje interpolacji ramki do większej rozdzielczości i wyświetla ją po czym powraca do `sem_wait`. Dodatkowo po wyświetleniu sprawdza, czy nie został wciśnięty klawisz oznaczający wyjście z pętli i zakończenie programu. Pobranie znaku z klawiatury wykonuje funkcja `cv::waitKey`. Jako argument przyjmuje liczbę milisekund, przez którą będzie oczekiwać na pobranie znaku. Jeżeli pobrano znak 'q' to nastąpi ustawienie flagi `quitFlag` i wyjście z pętli wyświetlania.

Flaga jest używana zarówno przez wątek główny jak i wątki przetwarzające, dlatego operacje odczytu i zmiany jej wartości są traktowane jako sekcja krytyczna. Synchronizacja wejścia i wyjścia z sekcji krytycznej w tym przypadku odbywa się przy użyciu mechanizmu blokad. Blokadą jest zmienna `fin` typu `pthread_mutex_t` i jest współdzielona przez wszystkie wątki. Na początku konieczna jest inicjalizacja blokady za pomocą funkcji `pthread_mutex_init`. W momencie wejścia do sekcji krytycznej blokada zostaje założona przez funkcję `pthread_mutex_lock`. Przed wyjściem z sekcji krytycznej następuje zdjęcie blokady przez funkcję `pthread_mutex_unlock`. Mechanizm zapewnia, że w ten sposób tylko jeden wątek, który założył blokadę z powodzeniem, znajduje się wewnątrz sekcji krytycznej. Inne wątki chcące założyć blokadę w tym samym czasie muszą czekać aż zostanie ona zdjęta.

Kod implementujący działanie wątku głównego zaprezentowano w tabeli 2.7:

```
void StereoPlayer::play(int lCamId, int rCamId){
    /* utworzenie i wypełnienie struktur z danymi dla
     * wątków przetwarzających */
    ThreadData tData[2];
    int info[]={lCamId, rCamId};
    for(int k=0; k<2; k++)
        tData[k] = (ThreadData){k+1,info[k],this,&StereoPlayer::remapThread};
    /* inicjalizacja barier, blokad i semafora */
    pthread_barrier_init(&b1,NULL,2);
    pthread_barrier_init(&b2,NULL,2);
    pthread_barrier_init(&bOpen,NULL,2);
    pthread_mutex_init(&fin,NULL);
    pthread_mutex_init(&openMtx,NULL);
    sem_init(&sem,0,0);
    /* wskazanie głównego wątka, w którym będzie
     * odbywać się wyświetlanie */
    cv::startWindowThread();
    /* utworzenie okna uruchamianego w trybie pełnego ekranu */
    cv::namedWindow("test",CV_WINDOW_NORMAL);
    cv::setWindowProperty("test", CV_WND_PROP_FULLSCREEN, CV_WINDOW_FULLSCREEN);
```

```
    cv::waitKey(500);
    /* utworzenie wątków przetwarzających ramki */
    for(size_t i=0;i<2;i++)
        if(pthread_create(&tid[i],NULL,&StereoPlayer::threadRunner,
        static_cast<void*>(tData+i))){
            std::cerr<<"thread "<<i<<" creation error"<<std::endl;
        };
    int counter = 0; //licznik wyświetlonych ramek
    double lastTime = 0.0;
    cv::Mat out(cv::Size(1920,1080), CV_8UC3,cv::Scalar(255,255,255));
    for(;!quitFlag && counter<fNum;counter++){
        /* oczekiwanie na pierwszą ramkę gotową do wyświetlenia */
        sem_wait(&sem);
        /* ustawienie wskaźnika wyświetlanej ramki na odpowiedni element bufora */
        cv::Mat *buff = &this->buffers[counter%5];
        cv::resize(*buff,out,out.size());    //interpolacja
        cv::imshow("test",out);    //wyświetlenie ramki
        char c=cv::waitKey(2);
        if(c=='q'){
            pthread_mutex_lock(&fin);
            quitFlag=true;
            pthread_mutex_unlock(&fin);
        }
    }
    cv::destroyWindow("test");    //zamknięcie okna
    /* czekaj na zakończenie wątków przetwarzających */
    for(size_t i=0;i<2;i++)
        pthread_join(tid[i],NULL);
    sem_destroy(&sem);    //zwolnienie zasobów używanych przez semafor
}
```

Tabela 2.7 Kod funkcji wykonywanej przez wątek główny

Powyżej opisano synchronizację wątku głównego z wątkami przetwarzającymi. Osobnym aspektem jest synchronizacja wątków przetwarzających między sobą. Po utworzeniu każdy wątek przetwarzający wykonuje funkcję `remapThread`, na którą ustawiony został wskaźnik `threadFun` w strukturze `ThreadData`. Argumentami są identyfikator wątku i identyfikator strumienia wideo. W pierwszej kolejności otwierany jest odpowiedni strumień za pomocą funkcji `open` wywołanej na obiekcie klasy `cv::VideoCapture`. Kolejno, funkcja `set` ustawia parametry strumienia, w tym wypadku szerokość i wysokość ramki.

Na etapie tworzenia i testowania programu okazało się, że ta operacja nie może być wykonywana w dwóch wątkach jednocześnie a więc jest sekcją krytyczną. Synchronizacja dostępu została zabezpieczona, podobnie jak wcześniej, za pomocą blokady – zmiennej `openMtx` typu `pthread_mutex_t`.

Kiedy pierwszy wątek otworzy poprawnie strumień oczekuje aż drugi zrobi to samo. Nieistotne jest, który wątek wykona tą czynność pierwszy, ważne aby przed rozpoczęciem pobierania ramek obydwa strumienie zostały poprawnie otwarte. Tym zajmuje się mechanizm barier. Bariera to zmienna `bOpen` typu `pthread_barrier_t`. Na początku bariera jest

zainicjalizowana za pomocą funkcji `pthread_barrier_init` jeszcze przed utworzeniem wątków w funkcji `play`. Podczas inicjalizacji wartość maksymalna licznika bariery ustawiana jest na 2 a wartość początkowa na 0. Każdy z wątków wykonując funkcję `pthread_barrier_wait` zwiększa wartość licznika bariery o 1. Wyjście z funkcji następuje tylko wtedy, gdy licznik osiągnie wartość maksymalną. Wobec tego, kiedy pierwszy wątek otworzy swój strumień poprawnie, wejdzie do funkcji `pthread_barrier_wait` i będzie w niej oczekiwał. W tym czasie analogiczne operacje wykona drugi wątek i także wywoła funkcję `pthread_barrier_wait`. W tym momencie licznik bariery `bOpen` osiągnie wartość maksymalną, obydwa wątki zakończą oczekiwanie i przejdą do następnej funkcji. W wypadku błędu zostanie zwrócony komunikat. Wątki są identyfikowane za pomocą wartości identyfikatora `tid` typu `int`. Bariera `bOpen` nie będzie już więcej używana więc zajmowane przez nią zasoby są zwalniane przez jeden z wątków za pomocą funkcji `pthread_barrier_destroy`. Opisany fragment kodu jest częścią funkcji `remapThread` i został przedstawiony w tabeli 2.8.

```
cv::VideoCapture cap;
int counter=0;
bool qF=false;
/* otwarcie odpowiedniego strumienia wideo
 * wejście do sekcji krytycznej */
pthread_mutex_lock(&openMtx);
switch(tid){
   case 1:
      cap.open(info);    //otwarcie strumienia
      if(!cap.isOpened())
         std::cerr<<"left stream opening error, info: "<<info<<std::endl;
      cap.set(CV_CAP_PROP_FRAME_WIDTH, 1280);    //ustawienie parametrów
      cap.set(CV_CAP_PROP_FRAME_HEIGHT, 720);
      break;
   case 2:
      cap.open(info);    //otwarcie strumienia
      sleep(2.0);
      if(!cap.isOpened())
         std::cerr<<"right stream opening error, info: "<<info<<std::endl;
      cap.set(CV_CAP_PROP_FRAME_WIDTH, 1280);    //ustawienie parametrów
      cap.set(CV_CAP_PROP_FRAME_HEIGHT, 720);
      break;
}
pthread_mutex_unlock(&openMtx);
pthread_barrier_wait(&bOpen);    //zwolnienie zasobów
```

Tabela 2.8 Otwarcie strumienia i ustawienie jego parametrów wewnątrz sekcji krytycznej

Następne etapy pracy wątków przetwarzających odbywają się w pętli, której kod został przedstawiony w tabeli 2.9. Jest to dalsza część funkcji `remapThread`. Wyjście z pętli następuje gdy w wątku głównym flaga `quitFlag` zostanie ustawiona na wartość `true`. Początkowo, w każdym z wątków ramki są pobierane ze strumienia i zapisywane do zmiennej `frame` za pomocą pary funkcji `grab` i `retrieve`. Następnie funkcja `cv::remap` dokonuje

przetworzenia ramek i wyjęcia największej części wspólnej ramki. Na końcu odpowiednie linie ramki są kopiowane, parzyste w jednym wątku i nieparzyste w drugim z uwzględnieniem przesunięcia ramek w poziomie względem siebie. Kopiowanie odbywa się przy użyciu wskaźników do ramki źródłowej (`src`) i do ramki docelowej (`dst`).

Wartość przesunięcia została ustalona na podstawie testów. Przykładowo, jeżeli przesunięcie wynosi 21 kolumn to oznacza to, że zmienna `lShift` ma wartość 11 a zmienna `rShift` 10. Wtedy ramka pochodząca z lewej kamery jest przesunięta o 11 kolumn w prawo a ramka z prawej kamery o 10 kolumn w lewo. Dzięki temu otrzymana ramka wyjściowa jest usytuowana centralnie na wyświetlanym obrazie z dokładnością do jednej linii. Każdy punkt obrazu zapisany jest w notacji BGR, składa się więc z trzech wartości składowych odpowiednio dla koloru niebieskiego, zielonego i czerownego. Dlatego aby przejść do następnej kolumny ramki należy przesunąć wskaźnik o 3 elementy.

Wskaźniki `src` i `dst` początkowo wskazują na początek danej linii odpowiednio w ramce źródłowej i docelowej i nie mogą zostać przesunięte poza obszar pamięci tej linii. Mogą więc być jedynie inkrementowane. Stąd, dla dodatniego przesunięcia, dla ramki pochodzącej z kamery lewej przesuwany jest w prawo wskaźnik `dst` a dla ramki pochodzącej z kamery prawej przesuwany jest wskaźnik `src`. Odwrotnie, dla przesunięcia ujemnego, dla ramki pochodzącej z kamery lewej przesuwany jest w prawo wskaźnik `src` a dla ramki pochodzącej z kamery prawej przesuwany jest wskaźnik `dst`. Przesunięcie jest także uwzględnione przy określaniu ilości kopiowanych danych wewnątrz funkcji `memcpy`, aby nie skopiować wartości, które nie należą już do danej linii. Obydwa wątki kopiują linie do tej samej ramki wyjściowej. Nie jest wiadome, który z nich skończy pierwszy, natomiast wymagane jest określenie momentu, w którym ramka jest gotowa do wyświetlenia. Kolejny raz jest do tego użyty mechanizm barier. W tym wypadku zmiennymi typu `pthread_barrier_t` są `b1` i `b2`. Wyjście z pierwszej bariery następuje po zakończeniu kopiowania linii przez obydwa wątki. Następnie sprawdzana jest w sekcji krytycznej flaga `quitFlag` i ustawiana lokalna flaga wyjścia z pętli `qF`. Po tym jeden z wątków zwalnia zasoby bariery `b1` za pomocą funkcji `pthread_barrier_destroy` oraz, jeżeli pętla ma być kontynuowana, inicjalizuje ją ponownie wywołując `pthread_barrier_init`. W tym czasie drugi wątek inkrementuje wartość semafora `sem`. W ten sposób przekazuje wątkowi głównemu informacje o tym, że ramka jest gotowa do wyświetlenia. Druga bariera `b2` jest zastosowana aby żaden z wątków nie przeszedł do kolejnej iteracji zanim bariera `b1` nie zostanie ustawiona ponownie. Analogicznie, po osiągnięciu bariery `b2` przez obydwa wątki ona również jest niszczona i inicjalizowana ponownie z wartościami początkowymi.

```
    for(;;counter++){
        if(qF)
            break;
        cv::Mat frame,remapped;    //ramka przejęta i przetworzona
        /* ustawienie wskaźnika na odpowiedni element bufora */
        cv::Mat *buff = &this->buffers[counter%5];
        /* pobranie ramki ze strumienia i zapis do zmiennej frame */
        cap.grab();
        cap.retrieve(frame);
        if(tid==1){
            /* przetwarzanie: korekcja i wyjęcie części wspólnej */
            cv::remap(frame,remapped,Lmap,nullMat1,CV_INTER_LINEAR);
            /* kopiowanie linii parzystych */
            for(size_t r=0; r<720;r+=2){
                uchar *dst = buff->ptr<uchar>(r);
                uchar *src = remapped.ptr<uchar>(r);
                /* uwzględnienie przesunięcia między ramkami */
                if(lShift>0)
                    dst += 3*lShift;
                else if(lShift<0)
                    src += 3*(-lShift);
                memcpy(dst,src,(1280-abs(lShift))*3);
            }
        }
        else if(tid==2){
            /* przetwarzanie: korekcja i wyjęcie części wspólnej */
            cv::remap(frame,remapped,Rmap,nullMat2,CV_INTER_LINEAR);
            /* kopiowanie linii nieparzystych */
            for(size_t r=1; r<720;r+=2){
                uchar *dst = buff->ptr<uchar>(r);
                uchar *src = remapped.ptr<uchar>(r);
                /* uwzględnienie przesunięcia między ramkami */
                if(rShift>0)
                    src += 3*rShift;
                else if(rShift<0)
                    dst += 3*(-rShift);
                memcpy(dst,src,(1280-abs(rShift))*3);
            }
        }
        /* ramka gotowa do wyświetlenia */
        pthread_barrier_wait(&b1);
        /* sprawdzenie flagi współdzielonej przez wszystkie wątki
         * wewnątrz sekcji krytycznej */
        pthread_mutex_lock(&fin);
        if(quitFlag)
            qF=true;
        pthread_mutex_unlock(&fin);
        if(tid==1){
            /* reset bariery b1 */
            pthread_barrier_destroy(&b1);
            if(!qF)
                pthread_barrier_init(&b1,NULL,2);
        }
        else if(tid==2){
```

```
            /* inkrementacja semafora */
            sem_post(&sem);
        }
        /* zabezpieczenie kolejnej iteracji */
        pthread_barrier_wait(&b2);
        if(tid==1){
            pthread_barrier_destroy(&b2);
            if(!qF)
                pthread_barrier_init(&b2,NULL,2);
        }
    }
```

Tabela 2.9 Dalsza część funkcji remapThread. Przetwarzanie kolejno przejmowanych ramek i synchronizacja między wątkami.

Zastosowane rozwiązanie zapewnia, że pobieranie ramek, przetwarzanie i kopiowanie linii jest wykonywane równocześnie w dwóch wątkach, a wątki oczekują na siebie wzajemnie po każdej iteracji.

3. Analiza rezultatu

W poniższym rozdziale dokonano analizy otrzymanego rezultatu wraz z omówieniem napotkanych trudności i ograniczeń. Na początku opisano problem braku synchronizacji między strumieniami, z których przejmowane są równolegle kolejne ramki. Następnie poruszono kwesta ograniczenia rozmiaru rejestrowanej ramki przy założonej szybkości, które wynika z możliwości zastosowanych kamer. Kolejno, omówiono problem osiągnięcia założonej wydajności systemu, zastosowane rozwiązania i trudności z nimi związane. Na końcu przedstawiono przykładowe pomiary oraz opisano sposób w jaki zostały wykonane.

3.1 Synchronizacja strumieni ramek

Wykorzystane kamery nie posiadają synchronizacji sprzętowej. Po rozpoczęciu rejestrowania przez kamerę zwracane są kolejne ramki ze strumienia. Ponadto obydwie kamery podłączone są do portów USB, które także nie są wyposażone w mechanizm synchronizacji dostępny z poziomu programu, zajmuje się tym system operacyjny. Są to fizyczne ograniczenia, których nie da się w tak zaprojektowanym systemie wyeliminować. Mogą wpływać na błędy wyświetlania widoczne tym bardziej im szybciej zmienia się obserwowana scena. Podczas wyświetlania obrazów o umiarkowanej szybkości, nieporządanych artefaktów nie zaobserwowano.

3.2 Ograniczenia rozmiaru ramki

Istnieje różnica pomiędzy rozdzielczością, w której ramki są rejestrowane a rozdzielczością wyświetlania. Maksymalny rozmiar z jakim użyte kamery mogą rejestrować

obraz to 1280x720 pikseli. Rozmiar, ramki, która ma zostać wyświetlona to 1920x1080 pikseli. Dlatego konieczna jest interpolacja. Zastosowano interpolację biliniową [10]. Zachowane zostają proporcje obrazu 16:9. W jej wyniku otrzymano pewne pogorszenie ostrości obrazu.

3.3 Szybkość wyświetlania i wielowątkowość

Założono minimalną szybkością wyświetlania 25 ramek na sekundę. Początkowo problem stanowiły czasy operacji przetwarzania ramek, dlatego zdecydowano się na wykonywanie ich w dwóch równoległych wątkach. Różnice między tymi operacjami w obydwu wątkach obejmują identyfikator strumienia, z którego pobierane są ramki, parametry wewnętrzne danej kamery, wektor zniekształceń oraz to, który wątek wypełnia linie parzyste a który nieparzyste. Mechanizm zrównoleglania operacji przy użyciu wątków nadaje się więc doskonale, ponieważ obydwa wątki wykonują niemal identyczne zadania.

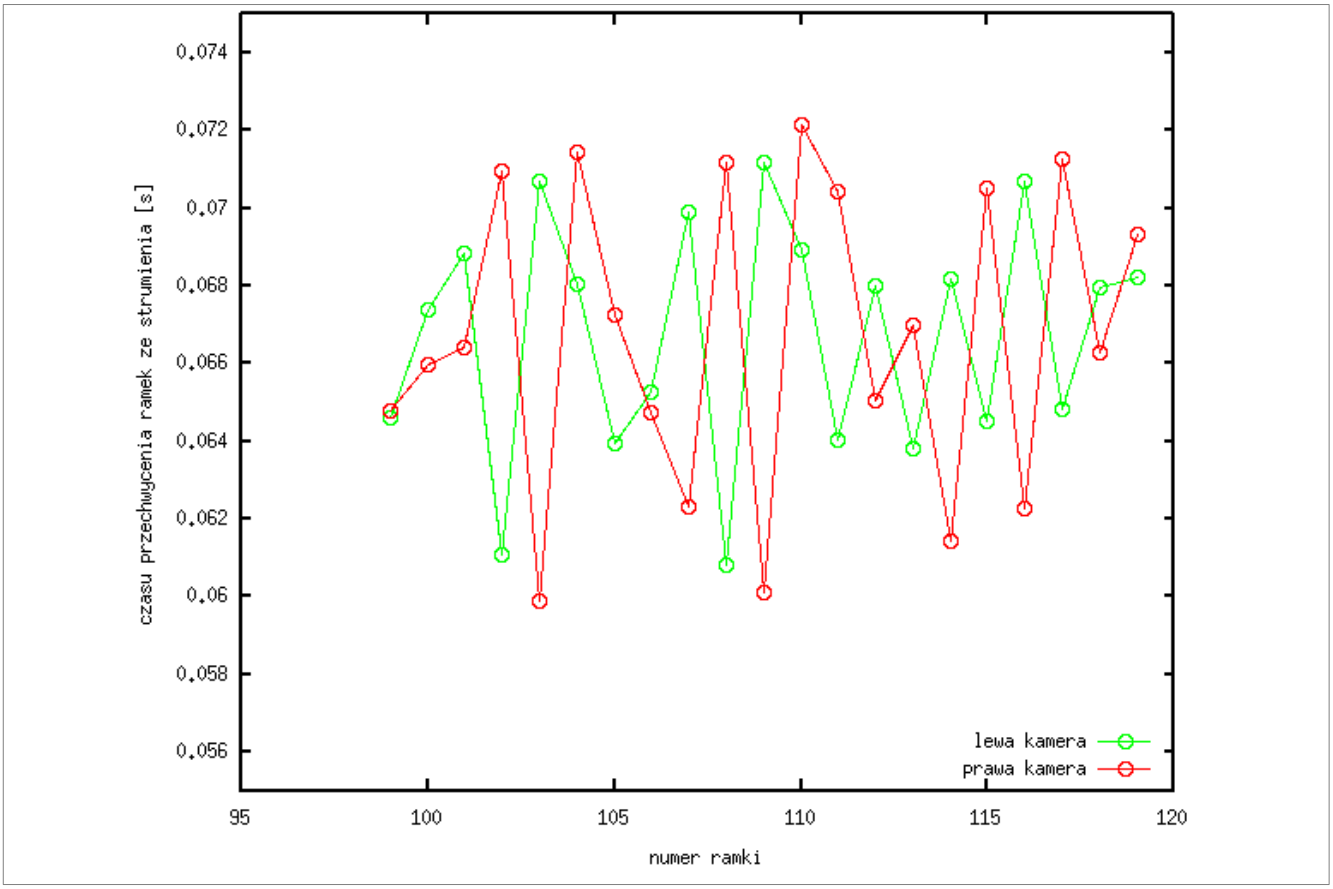
Aby wątki mogły wykonywać się z mniej więcej podobną ilością operacji do wykonania dla każdej pary ramek zdecydowano, że prócz dwóch wątków przetwarzających należy wprowadzić jeszcze wątek nazwany głównym, który jest odpowiedzialny za utworzenie i poprawne zakończenie wątków przetwarzających oraz interpolację i wyświetlanie gotowych ramek. Dzięki temu wątki przetwarzające są obarczone w przybliżeniu jednakową ilością operacji do wykonania. Problem, który się wówczas pojawił polegał na tym, że okno wyświetlające gotowe ramki nie pojawiało się w ogóle na ekranie. Prawdopodobnie przyczyna leży w wewnętrznej implementacji modułu highgui biblioteki OpenCV, który zawiera funkcje zajmujące się współpracą ze środowiskiem graficznym. Rozwiązaniem okazało się wywołanie w wątku głównym funkcji `cv::startWindowThread` przed otworzeniem jakiegokolwiek okna oraz przed uruchomieniem innych wątków w danym procesie. Wątek głowy został w ten sposób mianowany jedynym wątkiem współpracującym ze środowiskiem graficznym. Okno wyświetlające czasami nie wyświetlało się również jeżeli po jego utworzeniu nie wywołano funkcji `cv::waitKey`. Jako argument funkcja przyjmuje wartość milisekund, którą należy określić empirycznie. Znalezienie minimalnego czasu oczekiwania, dla którego okno zostaje poprawnie otwarte rozwiązało ten problem.

Wśród założeń projektu było utworzenie prostego interfejsu graficznego do komunikacji z użytkownikiem. Biblioteka OpenCV oferuje kilka prostych funkcji stworzonych do tego celu w ramach modułu highgui. Podjęto próbę ich użycia jednak po raz kolejny ich działanie nie było zgodne z oczekiwaniami. Wskutek tego, zdecydowano się na utworzenie prostego interfejsu z wykorzystaniem pakietu GTK. Biblioteka OpenCV w stosowanej konfiguracji używa dokładnie tego samego pakietu w wersji 2 w wewnętrznej implementacji funkcji modułu highgui. Z tego powodu możliwe jest użycie funkcji tego pakietu bezpośrednio ale z zastrzeżeniem do tej samej wersji. Interfejs graficzny został więc zaprojektowany w oparciu o GTK w wersji 2.

W momencie wywoływania funkcji GTK bezpośrednio oraz wywoływania ich za

pośrednictwem funkcji biblioteki OpenCV w tym samym wątku znów pojawił się problem niewidocznego okna wyświetlającego ramki. Rozwiązaniem okazało się rozdzielenie tych zadań na dwa procesy. Niewykluczone, że istnieje metoda pogodzenia tych zadań w jednym procesie. W momencie uruchomienia programu `sv` zostaje utworzony proces, który wyświetla interfejs graficzny oraz po naciśnięciu odpowiednich przycisków wykonuje wszystkie etapy procesu konfiguracji systemu. Jednak ostateczne uruchomienie systemu zostało przeniesione do osobnego procesu potomnego, który uruchamia program `play`. To rozwiązało wcześniejszy konflikt.

Po uruchomieniu działającego systemu i wykonaniu pierwszych pomiarów okazało się, że szybkość wyświetlania wynosi 15 ramek na sekundę a więc zdecydowanie poniżej oczekiwań. Pomiary dowiodły, że wąskim gardłem nie był czas przetwarzania ramek ale czas odbierania ich ze strumienia, który zilustrowano na rys. 3.1. Dla 20 kolejnych ramek o indeksach od 99 do 119 odczytano czas przejęcia z obydwu strumieni (wykonanie operacji `cap.grab()`). Następnie otrzymany zbiór 20 wartości poddano operacji różniczkowania i otrzymane wyniki przedstawiono na wykresie. Wartości są wierzchołkami krzywych i zostały zaznaczone kółkami. Krzywa czerwona przedstawia wartości dla strumienia ramek z kamery prawej a zielona z lewej. Oś pozioma wskazuje numer ramki, natomiast na osi pionowej zaznaczono czas przechwycenia ramki ze strumienia podany w milisekundach. Dla szybkości odczytu 15 ramek na sekundę średni czas odczytu wynosi w przybliżeniu 67ms. Widać, że odczytane wartości oscylują w pobliżu takiego wyniku.



Rys. 3.1 Wykres czasów przejmowania ramek ze strumienia dla wybranych kolejnych 20 ramek

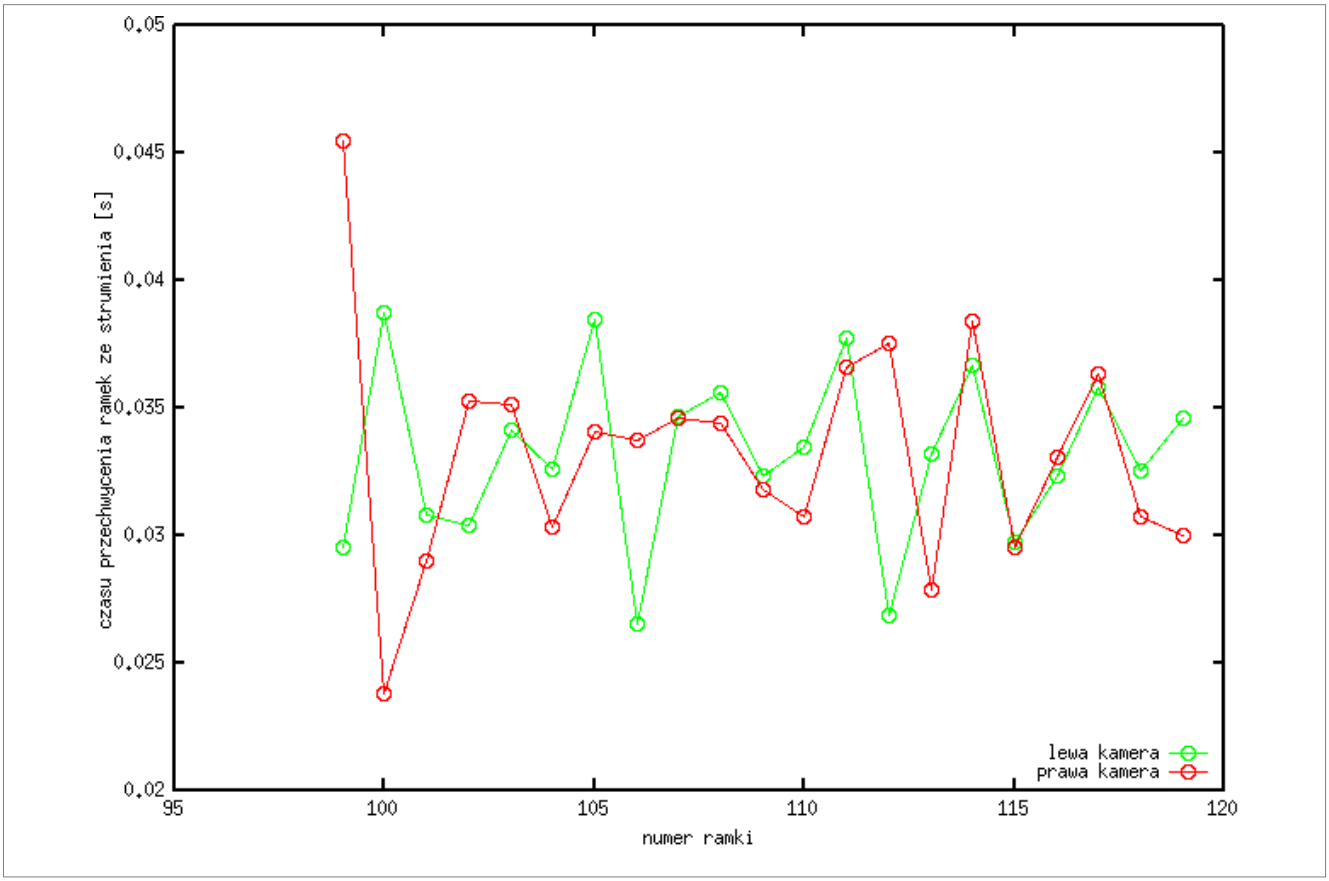
Co więcej, parametry takie jak typ kodowania i ilość pobieranych ramek na sekundę podobnie jak wymiary ramki zostały ustawione z poziomu biblioteki OpenCV za pomocą funkcji `set` z odpowiednimi argumentami.

```
cv::VideoCapture cap;
//linie pominięte
cap.set(CV_CAP_PROP_FOURCC,CV_FOURCC('M','J','P','G'));    //nie działa
cap.set(CV_CAP_PROP_FRAME_WIDTH, 1280);
cap.set(CV_CAP_PROP_FRAME_HEIGHT, 720);
cap.set(25);  //nie działa
```

Tabela 3.1 Kod ustawiający parametry strumienia

Okazało się, że niektóre parametry funkcji `set` nie działają. W tabeli 3.1 przedstawiono dwa przykłady wywołania tej funkcji, które pierwotnie planowano użyć jednak nie działały zgodnie z oczekiwaniami. Jest to uzależnione od modelu kamery i wersji sterownika **v4l** (video for linux), którego na platformie Linux używa biblioteka OpenCV. Rozwiązaniem okazało się ustawienie tych parametrów poprzez aplikację **guvcview**, która zapisała je w systemie operacyjnym. Różnica polega na tym, że w wykorzystywanej rozdzielczości, stosując kodowanie MJPG, otrzymano szybkość wyświetlania 30 ramek na sekundę a nie 25

co jest wynikiem lepszym od zakładanego minimum. Wykonano ponownie takie same pomiary, których rezultat prezentuje rys. 3.3. Przy otrzymanej szybkości średni czas odczytu ramki to nieco ponad 33 ms i w tym przypadku także odczytane wartości wahają się wokół tej wartości.



Rys. 3.3 Wykres czasów przejmowania ramek ze strumienia dla wybranych kolejnych 20 ramek, ustawiono parametry strumienia spełniające minimalne wymagania

3.4 Pomiary

Wykresy przedstawione na rys. 3.1 i rys. 3.3 sporządzono w oparciu o czasy zarejestrowane podczas działania systemu. Do tego celu wykorzystano systemową funkcję `gettimeofday`. Przykładowe użycie zaprezentowano w tabeli 3.2.

```
timeval tp;
//linie pominięte
for(;;counter++){
    //linie pominięte
    /* przykładowa funkcja, której czas wykonania chcemy zmierzyć */
    cap.grab();
    gettimeofday(&tp, NULL);
    if(counter<fNum)
        /* konwersja */
        lFramesGrab[counter]= tp.tv_sec+(tp.tv_usec/1000000.0);
    //linie pominięte
}
```

Tabela 3.2. Przykład użycia funkcji gettimefday do zarejestrowania czasu przejęcia ramki ze strumienia

Zmienna `tp` jest strukturą, która zawiera dwa pola: `tv_sec` i `tv_usec` wypełniane w momencie użycia funkcji `gettimeofday`. Pierwsze z nich zapisuje ilość sekund a drugie mikrosekund. Przy użyciu tych dwóch pól możliwa jest konwersja do typu `double`, w której otrzymuje się aktualny odczyt jako jedną wartość. Pomiary dla kolejnych iteracji są zapamiętane w tablicach o ustalonej długości. Podczas kończenia pracy programu wartości tablic są zapisywane do pliku tekstowego. Przykładowy fragment pliku zawierającego wyniki pomiarów przedstawiono w tabeli 3.3.

```
100 6.890149 0.016804 0.000851 6.910836 0.008106 0.000842 3.463995
101 6.928867 0.017796 0.001537 6.934649 0.015995 0.000822 3.495672
102 6.959667 0.020671 0.000978 6.963635 0.017696 0.000931 3.526429
103 6.990093 0.010408 0.001547 6.998903 0.014544 0.001125 3.558959
104 7.024255 0.010737 0.001605 7.034031 0.008934 0.000882 3.588068
105 7.056874 0.010142 0.001503 7.064334 0.009734 0.000785 3.619073
106 7.095360 0.011909 0.001293 7.098427 0.012405 0.000717 3.655721
107 7.121861 0.020046 0.001600 7.132145 0.009732 0.001578 3.687721
108 7.156493 0.010411 0.001471 7.166744 0.008623 0.000748 3.720331
109 7.192087 0.010548 0.001543 7.201110 0.008231 0.000744 3.754310
110 7.224406 0.009855 0.001485 7.232880 0.009241 0.000802 3.787111
111 7.257889 0.012492 0.001509 7.263604 0.011775 0.000692 3.820287
112 7.295615 0.015876 0.001349 7.300211 0.011693 0.001214 3.857343
113 7.322503 0.012110 0.001155 7.337744 0.008627 0.000766 3.891355
114 7.355705 0.010937 0.001300 7.365584 0.008600 0.000887 3.919267
115 7.392394 0.009457 0.001915 7.403967 0.008687 0.000965 3.957809
116 7.422134 0.015240 0.001208 7.433489 0.011235 0.000855 3.989783
117 7.454470 0.011522 0.001820 7.466578 0.011261 0.000939 4.023001
118 7.490270 0.012975 0.001288 7.502928 0.011328 0.000894 4.059337
119 7.522775 0.009872 0.002081 7.533632 0.013880 0.000900 4.092574
120 7.557371 0.014272 0.001520 7.563627 0.017442 0.000939 4.126220
```

Tabela 3.3 Fragment pliku z wynikami pomiarów

Znaczenie kolejnych kolumn jest następujące:

- 1 – numer ramki,
- 2,5 – czas odebrania ramki ze strumienia liczony od momentu uruchomienia programu,

- 3,6 – czas korekcji i dopasowania ramki liczony od czasu w poprzedniej kolumnie,
- 4,7 – czas operacji kopiowania linii liczony od czasu w poprzednie kolumnie,
- 8 – czas wyświetlenia ramki liczony od momentu wyświetlenia pierwszej ramki.

Kolumny opisane dwoma numerami oznaczają ten sam pomiar odpowiednio dla lewej i prawej kamery. Porównując wartości w kolumnie 2,5 i 8 widać, że ramki z obydwu kamer są odebrane ze strumienia w odstępie czasu nie większym niż 2ms. Wyświetlenie pierwszej ramki jest opóźnione względem uruchomienia programu o ponad 3 sekundy. Brak synchronizacji wyzwalania nie powoduje istotnych błędów. Nie mamy wpływu na opóźnienie momentu odebrania ramki ze strumienia względem momentu jej zarejestrowania ale obserwacja wyświetlanego obrazu pozwala stwierdzić że nie jest ono zauważalne. Opóźnienie momentu rozpoczęcia wyświetlania także nie jest problemem. Występuje ono tylko podczas uruchamiania programu `play`, później ramki są wyświetlanie na bieżąco.

Wnioski

W pracy zaprezentowano przykładowe wykonanie systemu stereoskopowego z wykorzystaniem urządzeń, które nie były optymalizowane jako zestaw do rejestracji filmów 3D. Użyte elementy oprogramowania są dostępne i bezpłatne. Aplikacja docelowo działa pod dystrybucją Ubuntu w wersji 13.10 przy wykorzystaniu biblioteki OpenCV w wersji 2.4.5.

Jako efekt końcowy projektu powstał system imitujący działanie lustra 3D. Użytkownik stając przed ekranem i zakładając okulary wyposażone w odpowiedni filtr polaryzacyjny może zaobserwować swój obraz odnosząc wrażenie jego trójwymiarowości. Obraz jest wyświetlany w sposób płynny a jego jakość jest zadowalająca. System jest obsługiwany z poziomu aplikacji wyposażonej w interfejs do komunikacji z operatorem. Od strony technicznej napisany kod wykorzystuje przetwarzanie równoległe z użyciem procesorów wielordzeniowych a proces budowania kodu do postaci plików wykonywalnych został skonfigurowany z użyciem narzędzia *cmake*.

Problemy związane z wydajnością uzasadniają potrzebę właściwego wsparcia sprzętowego dla operacji przetwarzania grafiki. Przykładem jest zastosowanie procesorów wielordzeniowych. Operacje te często polegają na wykonaniu tej samej czynności na niezależnych fragmentach obrazu, tak więc doskonale nadają się do przetwarzania wielowątkowego.

Projekt nadaje się do dalszego rozwijania. Przykładowym usprawnieniem byłoby lepsze dopasowanie do siebie równolegle odbieranych ramek pod kątem parametrów obrazu takich jak jasność, kontrast czy balans bieli. Istnienie różnic wartości tych parametrów wynika z faktu używania dwóch kamer, które nigdy nie mają identycznych parametrów. Korekcja pozwoliłaby te efekty zminimalizować. Inną możliwością byłoby zastosowanie kamer rejestrujących w większej rozdzielczości, najlepiej FullHD (1920x1080) co umożliwiłoby pominięcie etapu interpolacji. Zmniejszyłoby to złożoność operacji przetwarzania każdej pary ramek a ponadto poprawiło jasność wyświetlanego obrazu. W kwestii rozbudowania

programu o kolejną funkcjonalność warte uwagi byłoby umożliwienie rejestrowania nagrań i zapisu ich na dysk. To jednak uzależnione jest od możliwości oferowanych przez bibliotekę OpenCV i kompatybilności używanych przez nią kodeków z kamerami. W dotychczasowych wersjach biblioteki bardzo często funkcje zapisu nagrań na dysk nie działały prawidłowo z kamerami USB.

Podsumowując, rezultatem pracy jest działający prawidłowo system. Zarówno szybkość jego działania jak i jakość obrazu pozwalają uznać efekt satysfakcjonujący.

Literatura

Poniższa lista zawiera spis pozycji wykorzystanych podczas projektowania systemu, pisania kodu oraz powyższej pracy:

1. Biblioteka OpenCV (*http://opencv.org/*, z dnia 19.12.2013).
2. Dokumentacja pakietu GTK (*https://developer.gnome.org/gtk2/2.24/*).
3. *"OpenCV2 Computer Vision Application Programming Cookbook"*, Robert Laganiere, Packt Publishing 2011.
4. *"Urbie"*- robot do rozpoznania wojskowego w terenie miejskim (*http://www.jpl.nasa.gov/news/news.php?feature=485*, z dnia 18.11.2013).
5. *"Firefighting Robot"*- robot- asystent straży pożarnej do analizy i rekonstrukcji sceny 3D na miejscu interwencji; Coordinated Robotics Lab, University of California, San Diego (*http://ucsdnews.ucsd.edu/pressrelease/firefighting_robot_paints_3d_thermal_imaging_picture_for_rescuers*, z dnia 18.11.2013).
6. *"Learning OpenCV"*, Gary Bradski and Adrian Kaehler, O'Reilly 2008.
7. *"System do estymacji mapy głębokości na podstawie stereoskopowych sekwencji wideo dla platformy Android"* (strona 21-22), praca magisterska, Konrad Kurzawski, AGH Kraków, 2013.
8. Strona www: *http://www.digitalcameraworld.com/2012/10/12/lens-distortion-everything-every-photographer-must-know/2/*
9. Zniekształcenia geometryczne obrazu rejestrowanego przez obiektyw, strona www: *http://en.wikipedia.org/wiki/Distortion_%28optics%29*
10. Interpolacja biliniowa, strona www: (*https://en.wikipedia.org/wiki/Bilinear_interpolation)*.