ELEKTRONIKA I TELEKOMUNIKACJA
WYDZIAŁ ELEKTRONIKI I TELEKOMUNIKACJI
POLITECHNIKA POZNAŃSKA

INŻYNIERSKA PRACA DYPLOMOWA

SZYBKA ESTYMACJA MAP GŁĘBI NA PROCESORACH GRAFICZNYCH

PAWEŁ MANIA

Promotor:
dr inż. Tomasz Grajek

Poznań. 2017

Spis treści

Streszczenie

Praca dotyczy estymacji map głębi stereoskopowej z wykorzystaniem procesorów graficznych. W pracy wybrano dwa reprezentatywne algorytmy umożliwiające wyznaczanie głębi: metoda krzyżowa oparta na dynamicznie konstruowanym obszarze wokół każdego punktu oraz metoda iteracyjna oparta na masce z odpowiednio przypisanymi wagami charakteryzującymi podobieństwo obrazu. Wybrane algorytmy zaimplementowano w języku OpenCL. Za pomocą obrazów testowych z renomowanej bazy Middlebury, przetestowano przygotowaną implementacje. Otrzymane rezultaty pokazały, iż możliwa jest estymacja wysokiej jakości map głębi w czasie rzeczywistym. Dodatkowo w pracy przeanalizowano przygotowaną implementacje i wskazano najbardziej czasochłonne etapy.

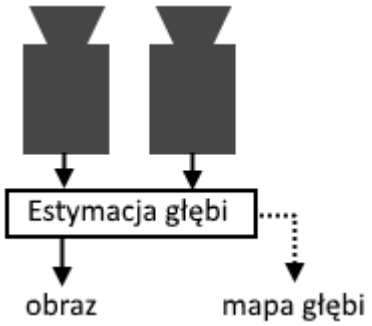
Summary

This work deals with stereoscopic depth map estimation on graphics processing unit (GPU). Two representative algorithms have been selected: Cross based local stereo-matching and stereo-matching using iterative refinement method. Both methods have been implemented using OpenCL language and thoroughly tested with help of widely recognised Middlebury datasets. Obtained results have shown that real time estimation of high quality depth maps is obtainable. Additionally, developed implementations have been examined in order to point out most time consuming steps of algorithms.

1. Wprowadzenie

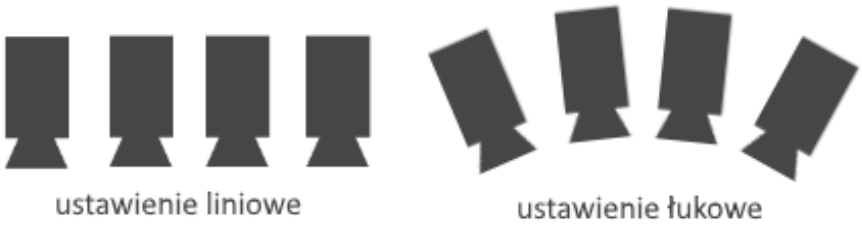
1.1. Systemy wielowidokowe i telewizja swobodnego punktu widzenia

Systemy wielowidokowe [1] składają się z wielu dokładnie ze sobą zsynchronizowanych kamer i pozwalają na zmiany punktu obserwacji sceny oraz dają wrażenie jej trójwymiarowości. Jakość sceny utworzonej przy pomocy wielu kamer w dużej mierze zależy od liczby dostarczonych widoków. Ze względu na wysoką cenę urządzeń istotna jest redukcja liczby kamer, a zarazem by umożliwić generowanie widoków wirtualnych z miejsc, w których nie można ustawić kamery korzysta się z danych o głębi obiektów reprezentowanych przez mapy głębi. Synteza widoków umożliwia przesyłanie jedynie kilku obrazów wraz z odpowiadającymi im mapami głębi co znacznie zmniejsza wymaganą prędkość bitową w porównaniu z przesyłaniem pojedynczych widoków z wielu kamer (rys.1.1.)[2].



Rys.1.1. Redukcja wymaganej liczby widoków poprzez generacje map głębi.

Mapa głębi zawiera informacje o odległości obiektów w scenie od kamery, a wykorzystanie procesu syntezy widoków pozwala na znaczną redukcję widoków, które należy przesłać. Oczywiście do przeprowadzenia syntezy widoków niezbędna jest także informacja o parametrach systemu kamerowego, a także rektyfikacja i odpowiednia korekcja otrzymanych widoków. W tego typu systemach kamery mogą być rozstawione w różny sposób, gdzie największą możliwość wyboru dowolnego punktu obserwacji dają kamery rozłożone na powierzchni półsfery co oczywiście sprawia problemy ze względu na wymagane rozmiary takiego systemu. Najłatwiejsze w realizacji są ustawienia liniowe i łukowe przedstawione na rysunku (rys.1.2.).

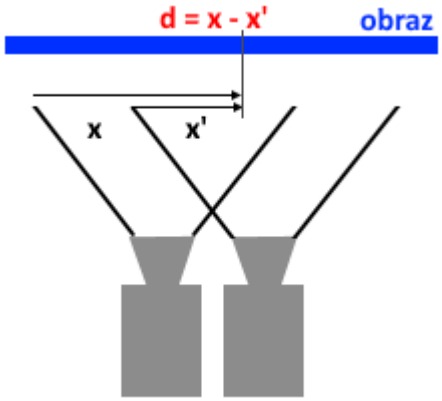


Rys.1.2. Przykładowe rozstawienia kamer - liniowe i łukowe.

1.2. Estymacja głębi

Mapy głębi odwzorowują odległość obiektów sceny od kamery, a ich wyznaczenie w czasie rzeczywistym ma duże znaczenie dla silnie rozwijających się dziedzin nauki i wielu zagadnień takich jak pomiary, robotyka, modelowanie sceny czy synteza widoków. Mapę głębi [3] można uzyskać za pomocą specjalistycznego sprzętu takiego jak kamery *ToF (Time of Flight)* wyznaczające głębie na podstawie czasu powrotu impulsu świetlnego. Głębia uzyskana tego typu kamerą charakteryzuję się dużą szybkością, lecz jest podatna na odbicia fali co może wprowadzić zakłócenia. Popularne są kamery działające na podstawie światła strukturalnego którego rzutowany wzór jest deformowany w zależności od naświetlanej powierzchni i jej odległości od kamery. Jedną z najbardziej znanych tego typu kamerą jest *kinect*, kamery tego typu potrafią uzyskać dokładność na poziomie jednego milimetra. Niestety przy użyciu światła strukturalnego obraz musi zostać odpowiednio przetworzony co przekłada się na obniżenie prędkości działania, dodatkowo scena na którą rzutowana jest siatka musi być odpowiednio oświetlona. Obie wymienione kamery wykorzystują niewidzialne dla ludzkiego oka światło podczerwone. Innym i tańszym rozwiązaniem jest estymacja rozbieżności na podstawie analizy dwóch zrektyfikowanych obrazów bez potrzeby korzystania z dodatkowych urządzeń pomiarowych, możliwe jest to jednak za cenę dużej złożoności obliczeniowej. Najczęściej zakłada się, iż obrazy na podstawie których wyznaczana jest mapa głębi są już wstępnie zrektyfikowane. Rektyfikacja jest operacją która pozwala na przetransformowanie pary obrazów do postaci obrazów zarejestrowanych przez idealną parę równolegle ustawionych kamer. Pozwala to na uproszczenie procesu estymacji głębi, gdyż obrazy każdego punktu znajdują się w tej samej linii w obrazach z obu kamer. A więc w celu odnalezienia obrazu punktu w sąsiedniej kamerze wystarczy wyszukać go w drugim obrazie w tej samej linii. Ze

względu na parametry kamer nie poszukuje się identycznych punktów a najbardziej podobne. W celu zwiększenia pewności dopasowania punktów obrazu, porównuje się ze sobą nie pojedyncze punkty, a większe fragmenty np. bloki, mówi się wtedy o agregacji kosztu. Rozbieżnością nazywamy względne przesunięcie punktu w obrazach, im wartość rozbieżności jest większa, tym bliżej kamery znajduje się dany obiekt (rys.1.3.). Reprezentacja głębi za pomocą rozbieżności powinna być skuteczna dla obiektów położonych blisko kamery co jest istotne dla telewizji trójwymiarowej



Rys.1.3. Obliczanie rozbieżności d, pomiędzy obrazami z sąsiednich kamer dla konkretnego punktu rejestrowanego obrazu.

2. Cele i założenia pracy

Celem pracy jest:

- Implementacja metody wyznaczania głębi stereoskopowej umożliwiająca działanie w czasie rzeczywistym przy użyciu mocy obliczeniowych procesorów graficznych.
- Analiza wydajności zaimplementowanych metod oraz ocena jakości wyznaczonych za ich pomocą map głębi.
- Wskazanie najbardziej czasochłonnych etapów, celem wskazania kierunków przyszłej optymalizacji.

W pracy przyjęto następujące założenia:

- Na podstawie rankingu M*iddlebury* [4] zostaną wybrane dwie metody, które będą miały największy potencjał do wydajnych obliczeń na procesorach graficznych.
- Wykorzystane zostaną obrazy z bazy danych *Middlebury* [5], dla których znane są prawdziwe, dokładne mapy głębi, pozwalające na weryfikację dokładności wyznaczania map głębi.
- Algorytmy zaimplementowane zostaną w języku *OpenCL*.

3. Opis wybranych metod

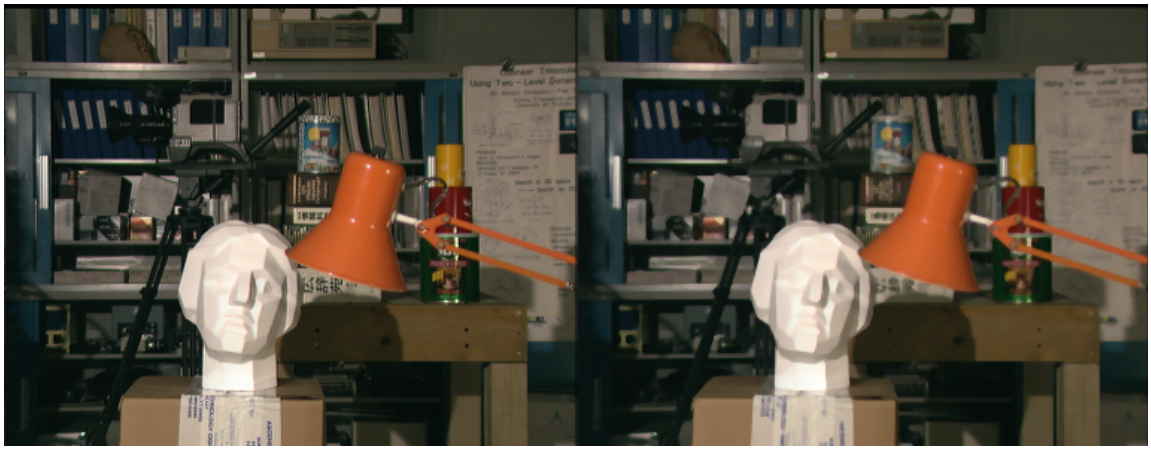
Na podstawie rankingu *Middlebury*, zostały wybrane dwie metody: metoda krzyżowa [6] i metoda iteracyjna [7]. Wybrane metody charakteryzują się wysoką wydajnością i posiadają duży potencjał umożliwiający przyśpieszenie obliczeń. Metody wyróżniają się dobrą jakością generowanych map głębi - liczba punktów z błędnie wyznaczoną wartością głębi nie przekracza *7%*.

3.1. Metoda krzyżowa

Uproszczony schemat blokowy metody został przedstawiony na rysunku (rys.3.1.). Najistotniejszym elementem metody krzyżowej są wyznaczane dla każdego punktu jednowymiarowe bloki danych, które wartościami muszą odwzorować określone podobieństwo z punktem środkowym. Obliczane bloki krzyżują się w miejscu wyznaczanego punktu obrazu i posłużą do konstrukcji dwuwymiarowego obszaru obejmującego podobne punkty obrazu przy zachowaniu liniowej złożoności obliczeń.

Rys.3.1. Schemat blokowy metody krzyżowej, gdzie linią przerywaną oznaczono argumenty wejściowe i jej końcowy rezultat, bloki zapisane linią ciągłą oznaczają konkretne etapy obliczeniowe algorytmu .

Wstępnie w celu redukcji szumów i zakłóceń wykonano filtrację medianową przedstawioną na rysunku (rys.3.2.). Filtracji medianowej dokonuje się na obrazie lewym i prawym, gdzie jako obraz lewy rozumiemy obraz przeznaczony dla lewego oka (zarejestrowany przez kamerę znajdującą się po lewej stronie systemu wielowidokowego), jako obraz prawy rozumiemy obraz przeznaczony dla prawego oka (zarejestrowany przez kamerę znajdującą się po prawej stronie systemu wielowidokowego).



Rys.3.2. Efekt filtracji medianowej (po prawej) na obrazie Tsukuba (po lewej).

Po przeprowadzeniu filtracji, w celu wykrycia podobieństw pomiędzy konkretnymi fragmentami dwóch obrazów, należy w każdym z obrazów pogrupować podobne do siebie punkty. Zamiast tworzyć dwuwymiarową maskę dla każdego punktu obrazu, wyznaczono liniowe bloki danych. W późniejszym etapie bloki posłużyły do dynamicznej konstrukcji całego obszaru zawierającego punkty o określonym podobieństwie. Takie podejście pozwala zmniejszyć redundancję obliczeń, a także zmniejszenie całkowitego czasu wymaganego na ponowny odczyt i zapis danych. Algorytm wyznaczył dla każdego punktu obszar obejmujący w linii ciągłej punkty będące podobnymi do punktu dla którego jest wyznaczany bieżący obszar. Obszary wyznaczono dla obrazu lewego i prawego. (rys.3.3.).

Rys.3.3. Krzyże przedstawiono dla obrazu tsukuba [6]. Przykład przedstawia bloki obejmujące punkty podobne do punktu leżącego w ich przecięciu.

Zasięg ramion krzyża reprezentują punkty obrazu znajdujące się w odległości *r** od punktu środkowego, wartość *r** obliczono według wzoru (1), gdzie podobieństwo punktu obrazu zostało określone poprzez wartości *1* i *0*, dzięki czemu kolejne punkty należące do konkretnego bloku zachowują ciągłość w posiadaniu właściwości bycia wystarczająco podobnym do punktu odniesienia.

$$r^* = \max_{r \in [1, L]} \left(r \prod_{i \in [1,r]} \delta(p, p_i) \right) \qquad (1)$$

gdzie:

*r** - zasięg obliczonego ramienia,

L - maksymalna przyjęta długość,

$\delta(p, p_i)$ - funkcja określająca podobieństwo punktu obrazu p i p_i (2).

$$\delta(p_1, p_2) = \begin{cases} 1, & \max\limits_{c \in \{R, G, B\}} \left(\left| I_c(p_1) - I_c(p_2) \right| \right) \le \tau \\ 0, & \text{dla wszystkich innych} \end{cases} \qquad (2)$$

gdzie:

p_1, p_2 - punkty obrazu,

I_c - składowa koloru punktu p_x,

τ - określa maksymalną dopuszczalną różnice pomiędzy składowymi koloru.

Skrzyżowane w ten sposób bloki obejmują punkty które są podobne do siebie pod względem koloru, są one niezbędne by potencjalnie małym kosztem obliczeń

wyznaczyć nieregularne maski dla każdego punktu obrazu. Nieregularny kształt zapewnia większą szanse na poprawne wyznaczenie rozbieżności, czyli odległości w jakiej znajduje się punkt obrazu prawego, względem tego samego punktu w obrazie lewym. Istotnym etapem jest agregacja kosztów, czyli obliczenie podobieństwa pomiędzy punktem obrazu lewego i prawego pod względem różnic koloru. Podobieństwo pomiędzy punktami sprawdza się dla *n* przyjętych rozbieżności. By obliczyć koszt (3) wyliczona została suma wartości bezwzględnych różnicy składowych koloru punktu obrazu *p* i *p'* zwana dalej jako pomiar *SAD (Sum of Absolute Differences)*. Koszta przechowywane są w trójwymiarowej strukturze danych, gdzie dwa wymiary odpowiadają powierzchni obrazu, a trzeci należy do rozbieżności.

$$cost(x,\ y,\ d) = \sum_{c \in R,G,B} \left| (\ p_c(x,\ y) - p_c{}'(x - d,\ y)) \right| \tag{3}$$

gdzie:

x, y - numer wiersza i kolumny obrazu,

d - wartość rozbieżności.

Posiadając obliczone koszta i bloki odzwierciedlające podobieństwo względem konkretnych punktów obrazu, można wyznaczyć całkowity koszt dla punktu uwzględniając jego otoczenie. W celu znacznej redukcji obliczeń unikamy sumowania kosztów każdego okna osobno, każdy wyznaczony koszt zostaje zastąpiony sumą kosztów które zostały wyznaczone dla wszystkich punktów obrazu poprzedzających obecny punkt w jego wierszu lub kolumnie (akumulacja). Akumulacja została wykonana na bloku danych wyznaczonych wzorem (3) przeprowadzając dla każdej rozbieżności najpierw akumulacje wierszy(4), a następnie wykorzystując jej wyniki przeprowadzono akumulacje kolumn (5) (rys.3.4.).

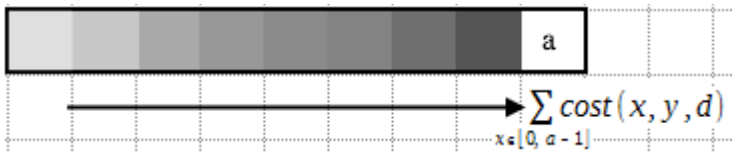
$$S^H(x, y, d) = \sum_{0 \le i \le x} cost(i,\ y,\ d) = S^H(x-1,\ y,\ d) + cost(x,\ y,\ d) \tag{4}$$

$$S^V(x, y, d) = \sum_{0 \le j \le y} cost^H(x,\ j,\ d) = S^V(x,\ y-1,\ d) + cost^H(x,\ y,\ d) \tag{5}$$

gdzie:

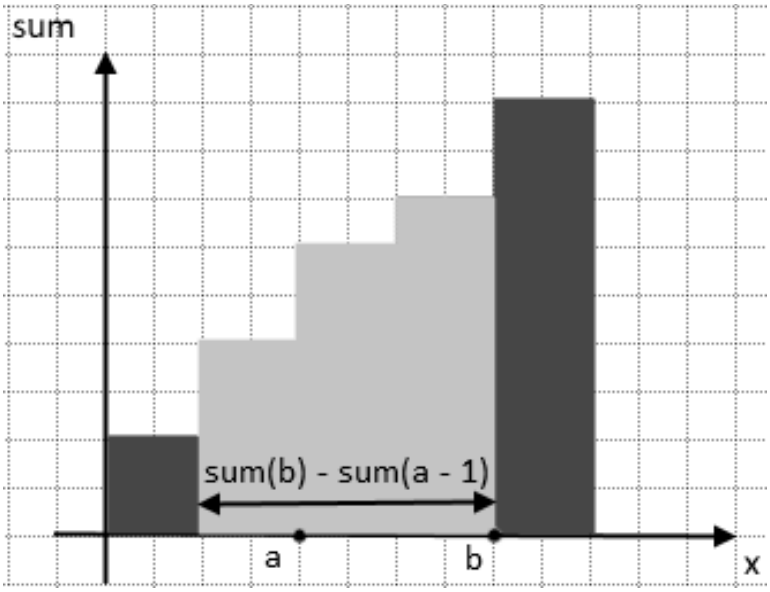
S^H, S^V - kolejno akumulacja kosztów wierszy i kolumn,

$cost^H$ - koszt wartości wierszy wyznaczony wzorem (6).



Rys.3.4. Akumulacja kosztów dla wiersza o długości a -1, przedstawia funkcje sumującą koszta kolejno wszystkich napotkanych punktów. Po kolejno dodawanych kosztach, przypisano im nowy koszt w postaci obecnego stanu sumy.

Po przeprowadzeniu każdej z akumulacji korzystając z właściwości całki oznaczonej odejmując dalszą granice bloku od bliższej (rys.3.5.) (względem początku układu współrzędnych obrazu) wyznaczone zostały koszta bloków poziomych (6), a następnie bloków pionowych składając w ten sposób koszt całkowitego okna (7) , tak jak to przedstawiono na rysunku (rys.3.6.).



Rys.3.5. Obliczenie kosztu dla pojedynczego elementu krzyża korzystając z akumulacji. Na osi X znajdują się kolejne punkty obrazu tej samej linii. Na osi Y znajdują się wyniki uzyskane podczas akumulacji kosztów. Przedstawiony przykład pokazuje sposób obliczenia kosztu pojedynczego bloku o zasięgu a i b wykorzystując akumulacje.

$$cost^{H}(x,\ y,\ d) = S^{H}(x + r^{H+},\ y,\ d) - S^{H}(x - r^{H-} - 1,\ y,\ d) \tag{6}$$

gdzie:

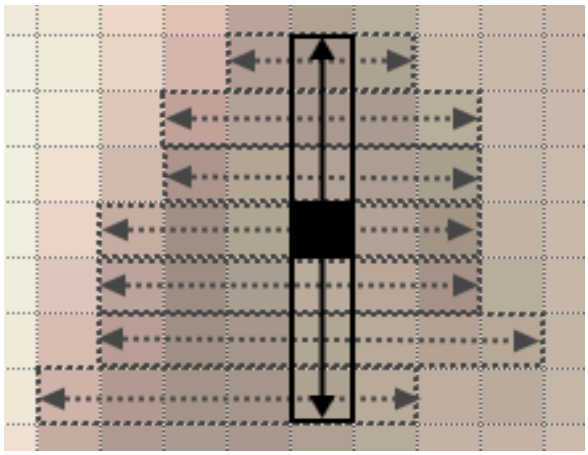
r^{H-}, r^{H+} - są kolejno lewym i prawym poziomym zasięgiem krzyża.

$$cost^{V}(x,\ y,\ d) = S^{V}(x,\ y + r^{V+},\ d) - S^{V}(x,\ y - r^{V-} - 1,\ d) \tag{7}$$

gdzie:

r^{V-}, r^{V+} - są kolejno dolnym i górnym pionowym zasięgiem krzyża.

Obliczony w ten sposób koszt wprowadza znaczną redukcje obliczeń, ponieważ bazuje na dwóch jednowymiarowych blokach i na ich podstawie generuje dwuwymiarową maskę. W przypadku, gdy podczas wyznaczania kosztu, określone bloki podobnych punktów nie pokrywały się wzajemnie, brana pod uwagę była tylko część wspólna każdego z nich.



Rys.3.6.Wyznaczenie całkowitego kosztu adaptacyjnie określonych obszarów. Po agregacji poziomej, każdy punkt trójwymiarowej struktury przechowującej dane zawiera w sobie koszt dla całego poziomego bloku (oznaczone na szaro). Dokonując agregacji pionowej, każdy punkt zawiera w sobie koszt całego otaczającego go obszaru skonstruowanego na podstawie jednowymiarowych bloków (oznaczony na czarno).

Wstępną mape głębi uzyskano korzystając z strategii *WTA (Winner Takes All - Zwycięzca bierze wszystko).* Korzystając z obliczonej wcześniej trójwymiarowej struktury danych zawierającej koszta zlokalizowano w osi rozbieżności koszt o najmniejszej wartości i przypisano wartość rozbieżności obecnemu punktowi obrazu (8).

$$d_p(x,y) = arg\ min_d(cost^V(x,y,d)) \tag{8}$$

gdzie:

$d_p(x,y)$ - wstępna głębia dla współrzędnych x, y,

$arg\ min_d()$ - zbiór *argumentów d* funkcji dla jakich osiąga ona minimum.

Najniższy możliwy koszt dla rozbieżności oznacza, że różnica pomiędzy kolorami punktów w obrazie lewym i prawym jest tam najmniejsza. W ten sposób otrzymano wstępną mapę głębi. Otrzymane rozbieżności wymagały ulepszeń ze względu na małe niedokładności i zostały poddane dodatkowemu przetwarzaniu wykorzystując algorytmy przetwarzania wtórnego - filtracji medianowwej oraz operacji na histogramie (9) [8] - gdzie na podstawie wcześniej wyznaczonych obszarów przeprowadzono operacje histogramu wykorzystując wstępną mapę głębi. Dla każdego punktu obrazu wybrano z otaczającego go obszaru najczęstszej występujące rozbieżności, w rezultacie otrzymano końcową mapę głębi.

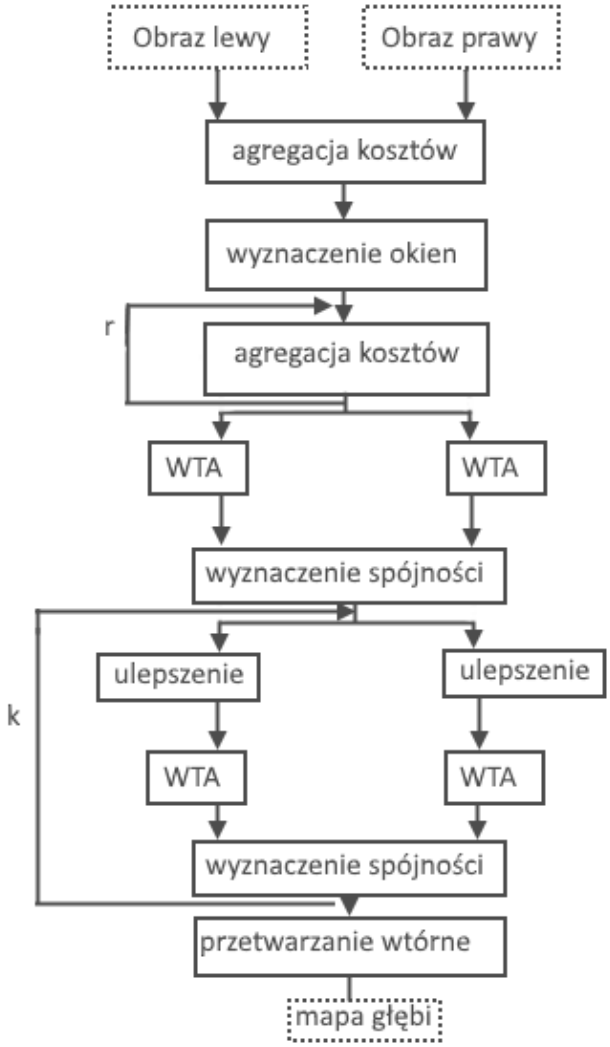
$$d'_p = arg\ max\ \varphi_p(d) \tag{9}$$

gdzie:

φ_p - jest utworzonym histogramem rozbieżności dla okna punktu *p.*

3.2. Metoda iteracyjna

Uproszczony schemat blokowy metody został przedstawiony na rysunku (rys.3.7.). Metoda iteracyjna wyróżnia się możliwością wielokrotnej agregacji kosztów, umożliwiając estymacje znacznie dokładniejszych map głębi. Charakterystyczny sposób wiązania w grupy podobnych punktów w obrazie polega na stworzeniu dla każdego punktu maski, której współczynniki odzwierciedlają prawdopodobieństwo przynależności punktu do analizowanego fragmentu obrazu.



Rys.3.7. Uproszczony schemat blokowy metody iteracyjnej, gdzie linią przerywaną oznaczono argumenty wejściowe i jej końcowy rezultat, bloki zapisane linią ciągłą oznaczają konkretne etapy obliczeniowe algorytmu. Zmienne k i r oznaczają liczbę ponownych iteracji kolejno ulepszeń i agregacji kosztów.

Algorytm rozpoczyna działanie od wstępnej agregacji kosztów obliczanej miarą *SAD.* Dla wszystkich punktów obrazu i ich możliwych rozbieżności obliczony został koszt, tworząc trójwymiarowy blok danych odzwierciedlający dopasowanie wszystkich możliwych rozbieżności obrazów. Algorytm wyznacza dla każdego punktu obrazu obszar (maskę) w którym każdy punkt ma określoną wagę której wartość silnie zależy od jego położenia i różnicy koloru względem punktu środkowego. Wyznaczone wagi mają na celu określenie jak bardzo dany punkt obrazu może należeć do tego samego obiektu co punkt znajdujący się w środku okna, tak określone wagi są zgodne z regułami teorii percepcji *gestalt* [9] określającej w jaki sposób ludzie potrafią łączyć w grupy (całość) konkretne elementy obrazu. Wagi (10) obliczone zostały osobno dla bloków w pionie i poziomie (idea krzyży) dla obrazu lewego i prawego.

$$w(p,\ q) = e^{-\frac{\Delta_c(p,\ q)}{\gamma_c} - \frac{\Delta_g(p,\ q)}{\gamma_g}} \tag{10}$$

gdzie:

p - rozpatrywany punkt obrazu

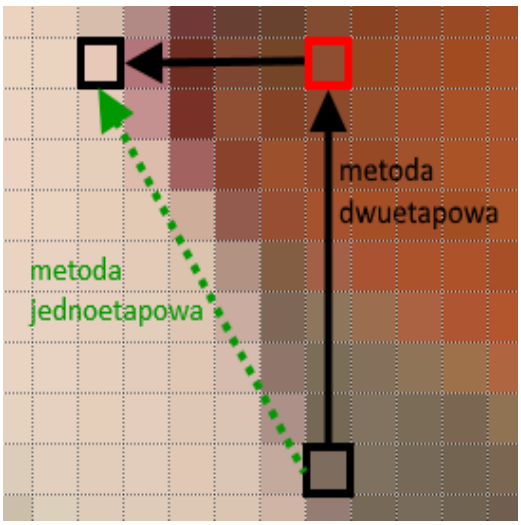
q - bieżący rozpatrywany punkt okna

$\Delta_c(p,q)$ - różnica kolorów punktu p i q wyznaczona metodą *SAD*

$\Delta_g(p,q)$ - odległość euklidesowa pomiędzy punktami p i q

γ_c, γ_g - stałe wyznaczone empirycznie

Po obliczeniu wstępnej agregacji kosztów dla każdego punktu obrazu i możliwych rozbieżności, metoda iteracyjna [7] w dużym stopniu opiera się na obliczeniu dla każdej maski średniej ważonej z kosztów uzyskanych dla wszystkich punktów obrazu i ich rozbieżności. Wagi są wartościami obszaru wyznaczonymi na podstawie różnicy koloru i odległości geometrycznej względem analizowanego punktu w środku maski, więc w dużym stopniu eliminują niedokładności powstałe podczas wstępnej agregacji. Obliczanie poszczególnych kosztów korzystając z średniej ważonej odbywało się dwuetapowo w celu redukcji złożoności obliczeń sprowadzając obliczenie dwuwymiarowego okna do wyznaczenia dla każdego punktu jednowymiarowych bloków, co wiąże się w niektórych przypadkach z niewielką niedokładnością (rys.3.8.), która nie wpływa znacząco na jakość wyników. Agregacje wykonuje się najpierw dla pionowych linii obrazu (kolumn) (11), a następnie dla otrzymanego kosztu przeprowadza się ponownie agregacje poziomych linii obrazu (wierszy).



Rys.3.8. Niedokładność dopasowania wagi przypisanej danemu punktowi obrazu, która powstała podczas korzystania z bloku punktu pośredniego (oznaczony na czerwono). Kolorem zielonym oznaczono przykład dopasowania wag indywidualnie dla każdego punktu w obszarze zainteresowania, co dla trójwymiarowego bloku danych oznacza znaczny wzrost ilości obliczeń wykonywanych przez algorytm iteracyjnej agregacji.

$$cost^{V}(p,p')=\frac{\sum_{q\in v,\,q'\in v'} w(p,q)\,w(p',q')\,\delta(q,q')}{\sum_{q\in v,\,q'\in v'} w(p,q)\,w(p',q')} \tag{11}$$

gdzie:

p, q - kolejno punkt obrazu lewego oraz punkt znajdujący się w jego oknie,

p′, q′ - kolejno punkt obrazu prawego oraz punkt znajdujący się w jego oknie,

v, v′ - pojedynczy pionowy blok okna obrazu lewego i prawego,

δ(q, q′) - wstępny koszt pomiędzy punktem w oknie obrazu lewego i prawego.

Po wyznaczeniu pionowej agregacji kosztów przeprowadzana zostaję pozioma agregacja (12) z wykorzystaniem wag przypisanym odpowiednim punktom w ich maskach oraz wcześniej wyznaczone koszty, dzięki czemu otrzymujemy pełny dwuwymiarowy obszar danych, w przypadku ponownych iteracji wyznaczony koszt zastępuje wstępną agregacje.

$$\delta(p,p') = \frac{\sum_{q \in h, q' \in h'} w(p,q) w(p',q') cost^{V}(q,q')}{\sum_{q \in h, q' \in h'} w(p,q) w(p',q')} \tag{12}$$

gdzie:

p, q - kolejno punkt obrazu lewego oraz punkt znajdujący się w jego oknie,

p′, q′ - kolejno punkt obrazu prawego oraz punkt znajdujący się w jego oknie,

h, h′ - pojedynczy poziomy blok okna obrazu lewego i prawego.

Metoda nazywana jest iteracyjną, ponieważ agregacja kosztów może być powtarzana wielokrotnie wykorzystując ponownie wagi wyznaczone dla poszczególnych obszarów obrazu. Końcowe iteracyjne ulepszenie oparte zostały na unikalności ostatnio wybranego kosztu co skutkowało dodaniem kary do nowo wybranych kosztów. Wyznaczenie wstępnej mapy głębi korzystając z strategii WTA przebiega analogicznie jak podczas metody krzyżowej, co dla obecnej sytuacji opisano wzorem (13). Podczas estymacji wstępnych map głębi obliczony został również współczynnik ufności (14).

$$d_p(x,y) = arg\,min_d(\delta(x,y,d)) \tag{13}$$

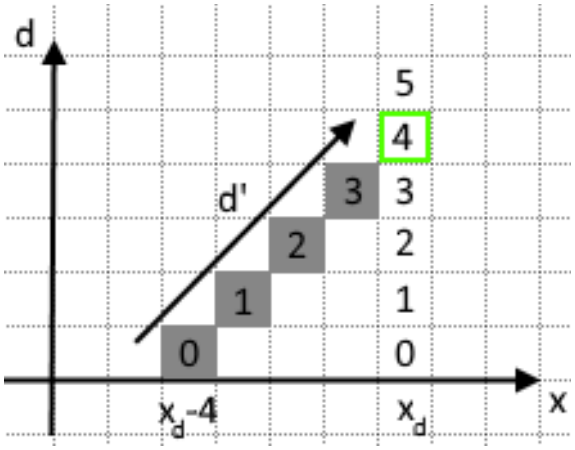
$$F_p^0 = \frac{min_2 - min_1}{min_2} \tag{14}$$

gdzie:

min_1 - najmniejszy otrzymany koszt

min_2 - drugi najmniejszy otrzymany koszt

W celu eliminacji niezgodności wynikających z przesłaniania się obiektów, głębię wyznaczono również dla obrazu prawego, by to zrobić najpierw należało znaleźć dla danego punktu najmniejszy koszt przeszukując wszystkie możliwe rozbieżności, a następnie sięgając do punktu wskazanego przez znalezioną rozbieżność sprawdzono wszystkie punkty obrazu znajdujące się pomiędzy nimi sprawdzając koszta wyznaczone dla kolejnych rozbieżności i wybierając najmniejszy z nich [rys.3.9.].



Rys.3.9. Wartości d oznaczają kolejne rozbieżności odszukane dla obrazu lewego. Oś X oznacza kolejne punkty linii obrazu. Rozbieżność oznaczona na zielono posiada najmniejszy koszt z całej kolumny x_d, odnosząc się odpowiednio do wskazywanego przez rozbieżność punktu obrazu x_d-4, można odnaleźć kolejne koszta dla rozbieżności obrazu prawego. Oznaczone w układzie punkty wzdłuż d' oznaczają koszta dla kolejnych rozbieżności obrazu prawego.

W kolejnym etapie wykorzystano iteracyjne ulepszenie, gdzie w końcowym etapie algorytmu punktom obrazu które zostały oznaczone jako niezgodne, czyli takie których punkty się nie pokrywają przypisano wartość przesłoniętej głębi (15).

$$d_p(x,y) = min\left(d_p^L(x,y),\ d_p^R(x,y) \right) \tag{15}$$

gdzie:

d_p^L, d_p^R - estymacje głębi obrazu lewego i prawego

Podczas obliczania obu rozbieżności w przypadku ich niezgodności zerowany był współczynnik ufności który wpływa znacząco na przebieg etapu iteracyjnego ulepszania map głębi. W celu eliminacji błędów spójności oraz poprawy jakości estymacji przeliczono uzyskane rozbieżności dla obu obrazów korzystając z wag określonych dla wyznaczonych obszarów, oraz poziomu ufności. W celu redukcji ilości obliczeń, podzielono je na kolumny (16) i wiersze (17).

$$\varepsilon_p{}' = \frac{\sum_{q \in v} w(p,q) F_q^{i-1} D_q^{i-1}}{\sum_{q \in v} w(p,q) F_q^{i-1}} \tag{16}$$

gdzie:

D_q^{i-1} - rozbieżność która została otrzymana w poprzedniej iteracji

F_q^{i-1} - poziom ufności wyznaczony w poprzedniej iteracji

i - numer iteracji

$$\varepsilon_p = \frac{\sum_{q \in h} w(p,q) F_q^{i-1} \varepsilon'^{d}_{q} \varepsilon'_q}{\sum_{q \in h} w(p,q) F_q^{i-1} \varepsilon'^{d}_{p}} \tag{17}$$

gdzie:

ε'^{d}_{q} - mianownik działania agregacji kolumn (16)

Estymacja rozbieżności została przeprowadzona dla obu obrazów, dodatkowo do każdego kosztu dodając kare (18) składającą się z agregacji otrzymanej w wcześniejszym etapie algorytmu. Na końcu estymacji wyznaczono współczynnik ufności dla obecnej iteracji, po czym całe iteracyjne ulepszenie jest powtarzane dla zaaktualizowanych map głębi.

$$\Lambda = \alpha\ \varepsilon_p^d \left| \varepsilon_p - d \right| \tag{18}$$

gdzie:

α - stała wartość, przyjęta jako *0,085*

ε_p^d - mianownik otrzymanej agregacji dla punktu p

ε_p - otrzymana agregacja dla punktu p

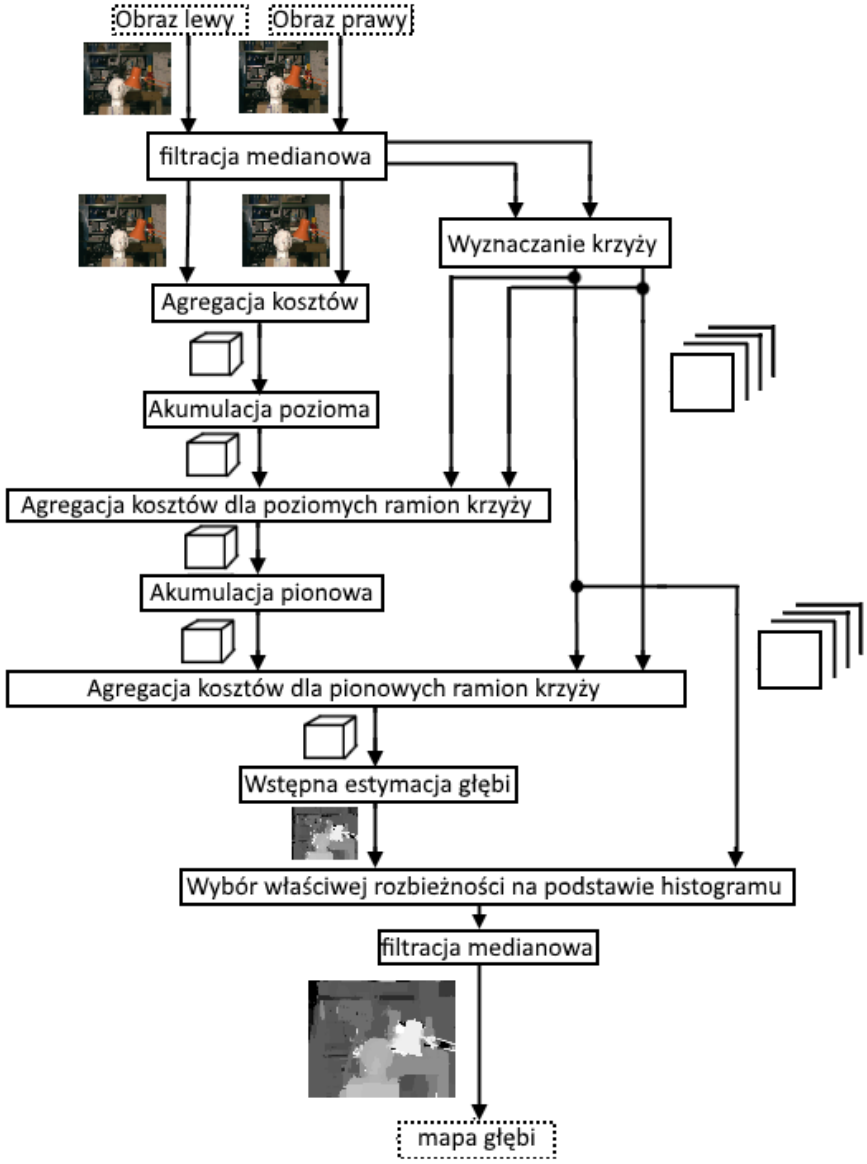
d - aktualnie rozpatrywana rozbieżność

4. Implementacja

Implementacji dokonano korzystając z języka *OpenCL* który umożliwił masowe zrównoleglenie obliczeń na procesorach graficznych. *OpenCL* umożliwił pomiar czasu konkretnych etapów algorytmu, etapy te zostały zaimplementowane jako osobne funkcje wywoływane na procesorze masowo równoległym.

4.1. Szybka estymacja głębi metodą krzyżową

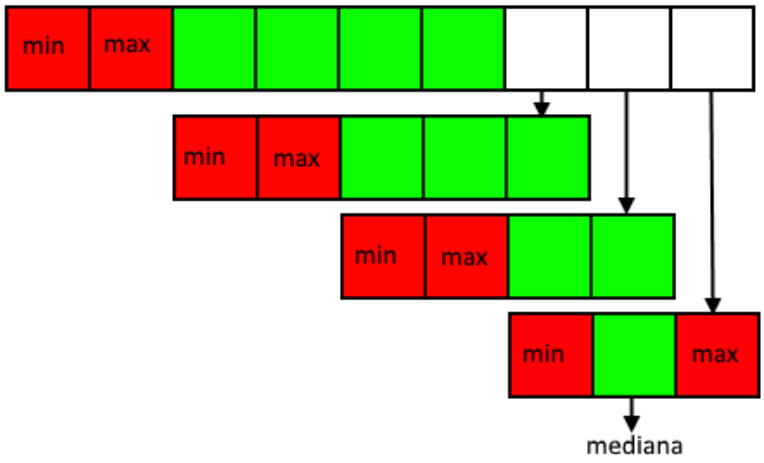
Schemat blokowy zaimplementowanej metody przedstawiono na rysunku (rys.4.1.).



Rys.4.1. Schemat blokowy zaimplementowanej metody. Przy istotnych etapach algorytmu umieszczono typy danych jakie są przesyłane pomiędzy funkcjami wyróżniając obrazy i trójwymiarowe bloki

4.1.1. Filtr medianowy

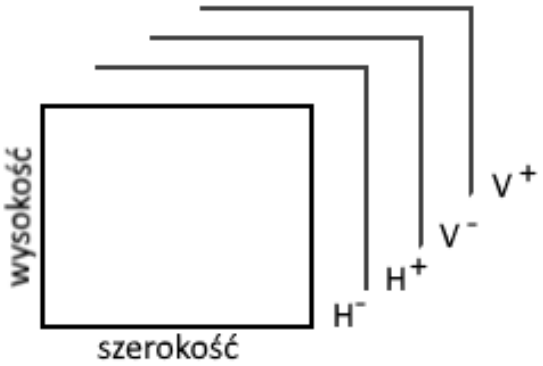
Zaimplementowany filtr został opracowany na podstawie algorytmu [10] który traktuje okno jako jednowymiarowy wektor i poczynając od pierwszych sześciu wartości, znajdował najmniejszy i największy element wektora i je odrzucał, powtarzał proces dla pięciu pozostałych elementów i kontynuował do momentu otrzymania wartości środkowej (rys.4.2.). Wybrany algorytm sprawdza się doskonale podczas masowego zrównoleglania obliczeń, ponieważ wymaga jedynie dwudziestu prostych dla procesora graficznego operacji. Wybierając wartość środkową z sąsiednich punktów wyeliminowano pojedyncze wartości będące zakłóceniami wpływającymi na jakość końcowych wyników. Każdy z obrazów był filtrowany w osobnej funkcji, przeprowadzone obliczenia zostały wykonane równolegle dla każdego z punktów obrazu.



Rys.4.2. Filtr medianowy - wyznaczanie mediany dla okna 3x3. Tak zaprojektowany filtr wymaga w kolejnych etapach tylko 7, 6, 4 i 3 operacje. Wykorzystując jedynie wbudowane funkcje min(a, b), max(a, b) oraz operacje podstawienia zmiennej pod wartość tymczasową, by nie utracić ważnych danych.

4.1.2. Wyznaczenie zasięgu krzyży

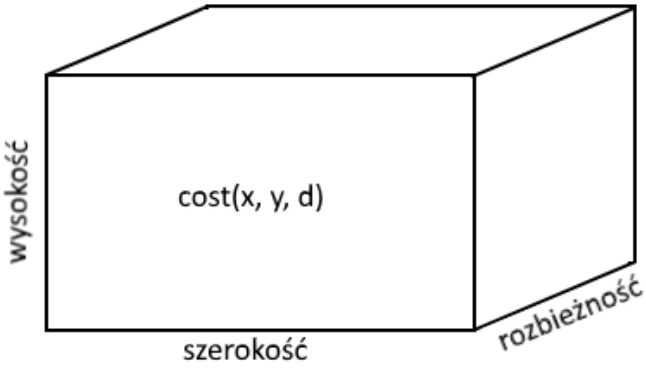
Wyniki dla każdego punktu zostały zapisane w pamięci w postaci bufora, który na każdy punkt obrazu posiada informacje o czterech wyznaczonych zasięgach skrzyżowanych bloków (rys.4.3.). Określone w ten sposób wartości są niezbędne by w późniejszych krokach poprawnie zbudować adaptacyjne okno pozwalające na dokładniejsze wyznaczenie rozbieżności pomiędzy obrazami. Krzyże zostały wyznaczone dla parametrów $L = 25$ i $\tau = 0.1$. Zaimplementowana funkcja wywoływana była dwukrotnie dla obrazu lewego i prawego. Obliczenia wszystkich czterech wartości wykonały się równolegle dla każdego punktu obrazu. Wyznaczenie zasięgu krzyży wymagało odczytu danych dla każdego z sprawdzonych punktów, co oznacza że dla każdego z punktów dokonano około *100* odczytów wartości obrazu przechowywanego w pamięci.



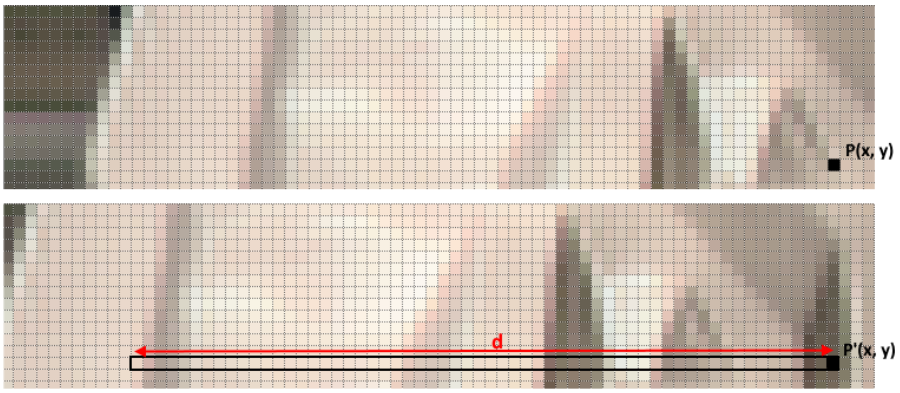
Rys.4.3. Wyznaczone krzyże zapisane w buforze wyjściowym. Gdzie kolejno dla każdego punktu obrazu można rozróżnić lewą (H^-) i prawą (H^+) oraz dolną (V^-) i górną (V^+) granicę bloku.

4.1.3. Agregacja kosztów

Koszta obliczone zostały dla $n = 60$ przyjętych rozbieżności punktu obrazu lewego $p = (x, y)$ i punktu obrazu prawego $p' = (x - d, y)$, gdzie d jest rozbieżnością. Rezultatem przedstawionej agregacji kosztów jest trójwymiarowy blok danych (rys.4.4.) przechowujący koszt dla każdego punktu obrazu $p(x, y)$ przy uwzględnieniu każdej z możliwych rozbieżności. Obliczanie kosztów zostało zrównoleglone dla każdego punktu obrazu. Dla każdego wykonania funkcji przeliczano koszt dla przyjętej liczby rozbieżności (rys.4.5.) i wynik w postaci pomiaru *SAD* zapisywano dla pozycji punktu obrazu i aktualnie liczonej rozbieżności. Realizacja agregacji kosztów wymagała $n+1$ odczytów danych dla każdego z analizowanych punktów obrazu.



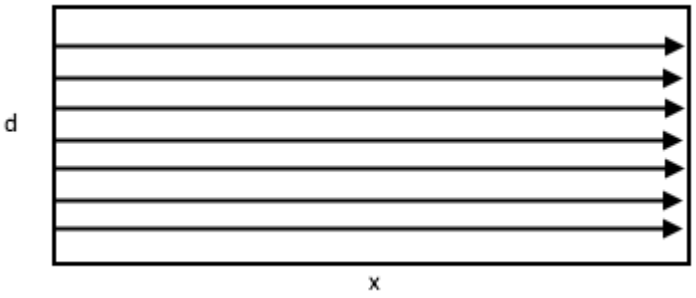
Rys.4.4. Otrzymany trójwymiarowy blok agregacji kosztów(cost). Blok zawiera pomiary SAD pomiędzy punktami obrazu lewego i prawego. Pomiary przeprowadzono dla n możliwych rozbieżności.



Rys.4.5. Przykładowy obszar dla którego wyznaczone zostały koszty dla punktu x i y.

4.1.4. Akumulacja kosztów

Akumulacja kosztów polega na zastąpieniu każdego kosztu sumą wszystkich poprzedzających go kosztów w danej linii. Akumulacje wykonano w dwóch osobnych funkcjach, osobno dla kolumn i wierszy obrazu. Równolegle przeprowadzono obliczenia dla całych konkretnych linii obrazu i ich rozbieżności, odpowiednio wierszy i kolumn. Każda z wykonanych funkcji musiała wykonać obliczenia dla całej szerokości (rys.4.6.) lub wysokości obrazu w zależności od tego czy wykonywana jest obecnie akumulacja kolejno pozioma lub pionowa.



Rys.4.6. Rysunek przedstawia akumulacje wierszy, gdzie dla każdej wybranej rozbieżności (d) sumowane są kolejne wartości całego wiersza (x).

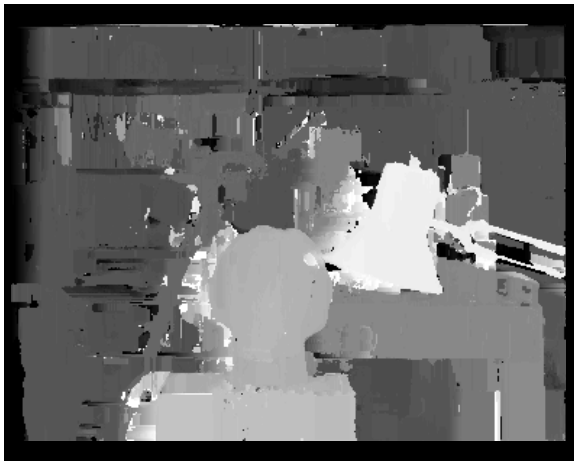
Wyniki akumulacji przepisywane były na bieżąco do nowego bloku o takiej samej pojemność jak trójwymiarowy blok kosztów. Każda z konkretnej pozycji która została dodana podczas akumulacji została przypisana nowemu blokowi kosztów w dokładnie do tych samych współrzędnych. Obie akumulacje, pionowa i pozioma odbywały się analogicznie, mimo to wymagały osobnych implementacji by nie obciążać całego procesu dodatkowym odczytem danych.

4.1.5. Wyznaczenie kosztów krzyży

Wyznaczenie kosztów krzyży polega na określeniu całkowitego kosztu punktu uwzględniając także podobne punkty leżące w jego obszarze. Wyznaczenie całkowitego kosztu punktu rozpoczyna wyznaczenie kosztu poziomej części krzyża, korzystając z obliczonego zasięgu ramion oraz akumulowanych kosztów wierszy S^H. Wykorzystano właściwość całki oznaczonej i obliczono różnice pomiędzy akumulacją kosztu znajdującego się w punkcie prawego ramienia krzyża oraz akumulacją kosztu w punkcie lewego ramienia krzyża. Po obliczeniu kosztu poziomego zasięgu krzyży obliczona została akumulacja kolumn. Całkowity koszt krzyży został obliczony korzystając z pionowego zasięgu ramion oraz posiadanej akumulacji kolumn. Obliczony został koszt bazujący na adaptacyjnym oknie zbudowanym z krzyża punktu rozpatrywanego oraz krzyży punktów znajdujących na jego powierzchni. Zaimplementowany algorytm w przypadku wyznaczania kosztów krzyży wykonywał się równolegle dla każdej wartości trójwymiarowego bloku kosztów. Jest to równoznaczne z jednoczesnym obliczaniem kosztu wszystkich punktów obrazu i ich rozbieżności. Do wyznaczenia kosztów użyto dwóch osobnych funkcji ze względu na rozbicie operacji na pionową i poziomą.

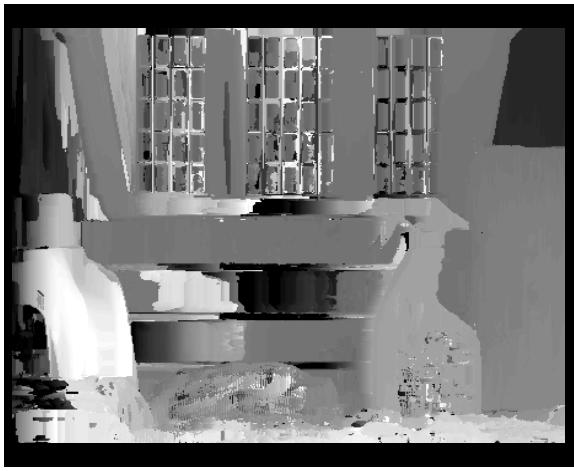
4.1.6. Wyznaczenie wstępnej mapy głębi

Mapa głębi (rys.4.7. - 4.10.) została wyznaczona za pomocą strategii WTA, polegającej na przeszukaniu wszystkich możliwych rozbieżności i wybraniu tej z najmniejszym kosztem. Realizacja algorytmu została wykonana określając jednocześnie najlepsze rozbieżności dla każdego z punktów obrazu.



Rys.4.7. - Tsukuba

Rys.4.8. - Teddy



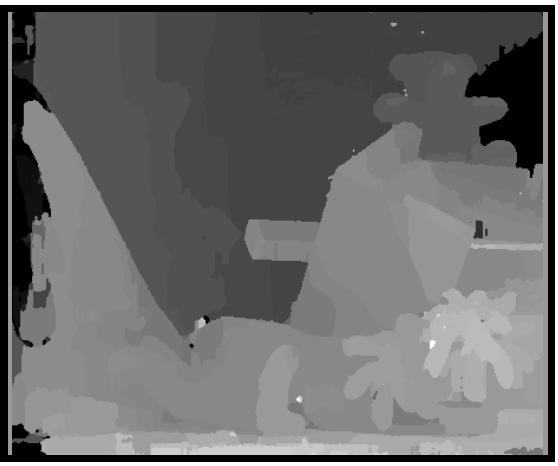
Rys.4.9. - Laundry

Rys.4.10. - Cones

4.1.7. Udoskonalenie wyników

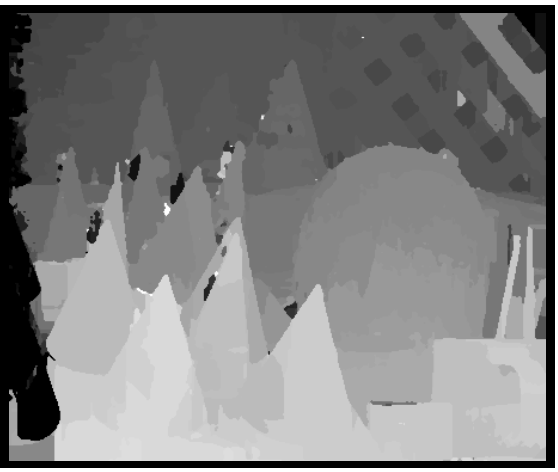
By polepszyć dokładność mapy głębi zaimplementowano algorytm wyznaczający histogram dla każdego punktu obrazu i jego okna. Histogram przedstawiał liczebność wystąpień konkretnych rozbieżności w danym oknie. Rozbieżność charakteryzującą się największą częstością wystąpień uznano za właściwą i przypisano punktowi znajdującemu się w środku okna. By usunąć drobne zakłócenia ponownie zastosowany został filtr medianowy (rys.4.11. - rys.4.14.). Budowa histogramu została wykonana jednocześnie dla wszystkich punktów wstępnej mapy głębi i wymagała wykonania licznych pętli niezbędnych do stworzenia histogramu.

Rys.4.11. - Tsukuba



Rys.4.12. - Teddy

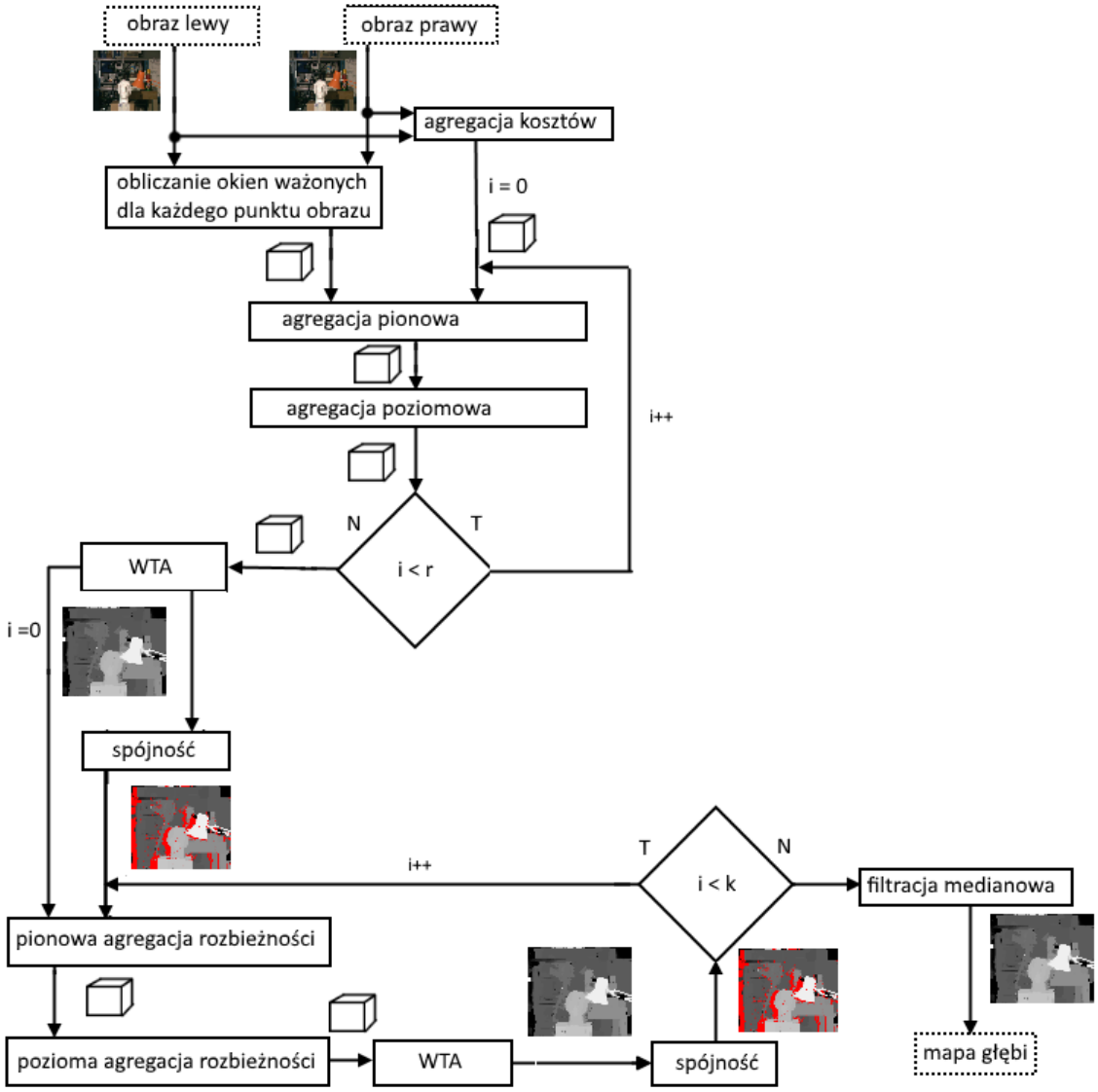
Rys.4.13. - Laundry



Rys.4.14. - Cones

4.2. Szybka estymacja głębi metodą iteracyjną

Schemat blokowy zaimplementowanej metody przedstawiono na rysunku (rys.4.15.).



Rys.4.15. Schemat blokowy zaimplementowanej metody. Przy istotnych etapach algorytmu umieszczono typy danych jakie są przesyłane pomiędzy funkcjami wyróżniając obrazy i trójwymiarowe bloki

4.2.1. Obliczanie wag dla poszczególnych okien

Wagi odzwierciedlają podobieństwo koloru i prawdopodobieństwo, że punkt o konkretnej pozycji może należeć do tego samego obiektu. Funkcja określająca wagi została wykonana czterokrotnie. Wymagała obliczeń dla dwóch obrazów i dla każdego obliczono wagi w pionie i poziomie. Wyznaczona maska posiadała po *33* wartości w pojedynczym bloku. Taka implementacja pozwoliła jednocześnie obliczyć pojedyncza wagę dla każdego punktu w oknie unikając użycia jakiejkolwiek pętli co jest istotne podczas zrównoleglania obliczeń na procesorach graficznych. Rezultatem był bufor pamięci który dla każdego punktu obrazu przechowywał wszystkie wartości maski. Kolejno maski pionowe i poziome dla obrazu lewego i prawego zostały przesłane do funkcji odpowiedzialnej za iteracyjną agregacje kosztów.

4.2.2. Iteracyjna agregacja kosztów

Po wyznaczeniu wstępnej agregacji kosztów analogicznie jak w metodzie krzyżowej, przeprowadzana została pionowa agregacja z wykorzystaniem wag przypisanym odpowiednim punktom w oknie *1x33*. Podczas realizacji algorytmu licznik i mianownik został umieszczony w osobnym buforze w celu przyśpieszenia obliczeń agregacji poziomej. Algorytm był wykonywany dla wszystkich możliwych punktów w obrazie i ich rozbieżności. W przypadku ponownych iteracji wstępny uzyskany koszt został zastąpiony przez końcowy rezultat kosztu agregacji. Implementacja funkcji zawierała pętle obliczającą średnią ważoną kolejnych linii obrazów. Biorąc pod uwagę, że pętla zostanie wykonana dla każdego punktu w obrazie i wszystkim odpowiadającym im rozbieżnością istotne było maksymalne zoptymalizowanie tej funkcji ze względu na możliwy długi czas obliczeń.

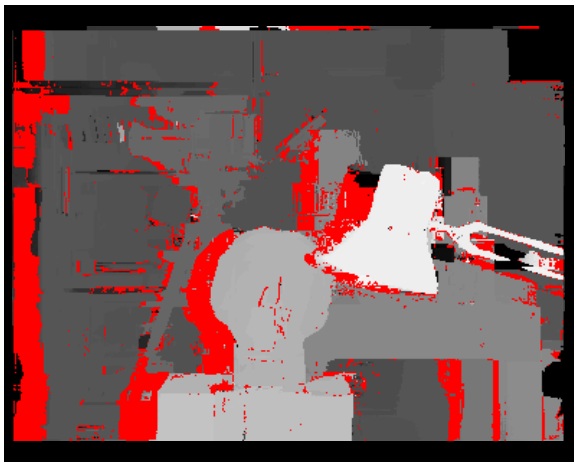
4.2.3. Estymacja wstępnych map głębi

Wyznaczenie wstępnej mapy głębi korzystając z strategii WTA przebiega analogicznie jak podczas metody krzyżowej, z otrzymanych kosztów na podstawie określonej rozbieżności obrazu lewego obliczono także mapę głębi dla obrazu prawego, gdzie w celu wyznaczenia rozbieżności skorzystano z funkcji liniowej. Obliczenie mapy głębi dla obrazu lewego i prawego wykonano przy użyciu tej samej funkcji. Udało się uniknąć niepotrzebnej redundancji obliczeń, która powstała by w momencie

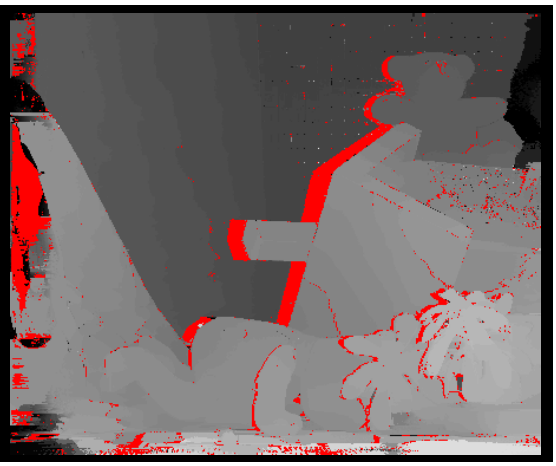
wyznaczania mapy głębi dla obrazu prawego, która wymaga w pierwszej kolejności wyznaczenia najmniejszego kosztu dla obrazu lewego.

4.2.4. Spójność map głębi

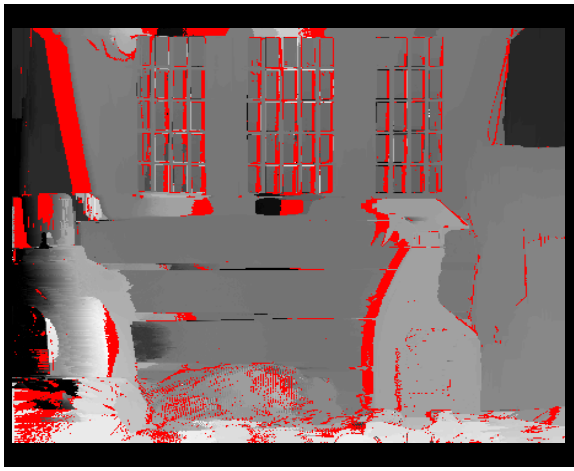
Obliczenie mapy głębi dla obrazu lewego i prawego posłużyło do określenia ich spójności. Ze względu na efekt przesłaniania utracono część informacji co spowodowało błędy (rys.4.16.-4.19.) podczas obliczania rozbieżności pomiędzy obrazami. Obliczenie spójności zostało wykonane równolegle dla każdego punktu obrazu, gdzie przy każdym wywołaniu funkcji wymagany był dwukrotny odczyt informacji z obrazu.



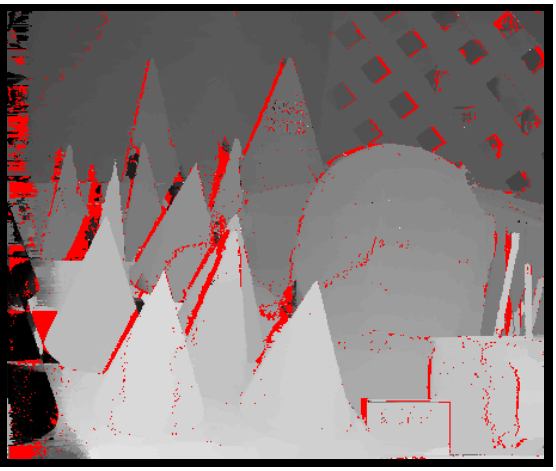
Rys.4.16. - Tsukuba



Rys.4.17. - Teddy



Rys.4.18. - Laundry



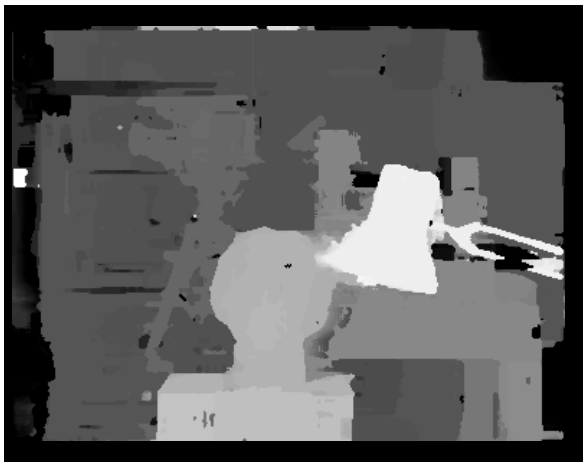
Rys.4.19. - Cones

4.2.5. Iteracyjne ulepszanie map głębi - Agregacja rozbieżności

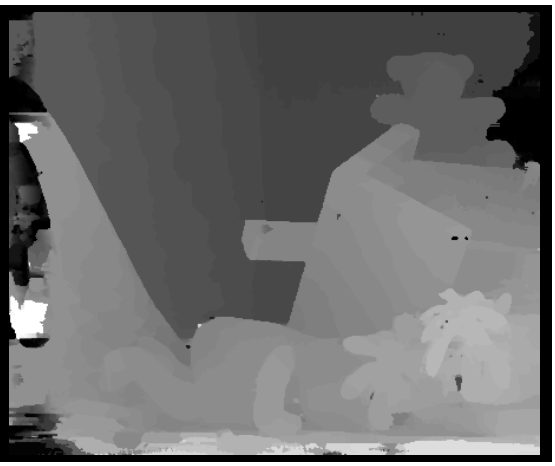
W celu eliminacji błędów spójności oraz poprawy jakości estymacji przeliczono uzyskane rozbieżności dla obu obrazów korzystając z wyznaczonych okien wagowych. Po przeliczeniu rozbieżności otrzymano kare która została dodana do wcześniej wyznaczonych kosztów, dzięki czemu możliwe było uzyskanie delikatnej poprawy otrzymanej mapy głębi. Rezultat zależy od różnicy pomiędzy rozbieżnościami w wyznaczanym oknie obrazu, co zostanie użyte w celu wyznaczenia kolejnych.

4.2.6. Iteracyjne ulepszanie map głębi - Estymacja rozbieżności

Po przeprowadzeniu estymacji rozbieżności sprawdzona została spójność map głębi i wyzerowano współczynnik ufności dla punktów obarczonych błędem. Podczas ostatniej iteracji błędy zostają wyeliminowane poprzez wybór przesłoniętych punktów. W celu polepszenia mapy głębi została przeprowadzona filtracja medianowa, wyniki przedstawiono na rysunkach poniżej (rys.4.20.-4.23.).

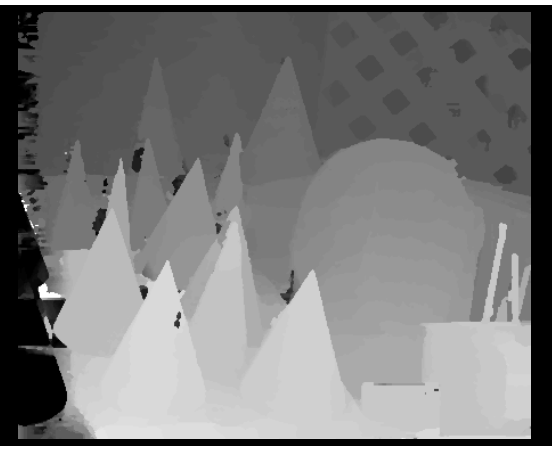


Rys.4.20. - Tsukuba



Rys.4.21. - Teddy

Rys.4.22. - Laundry



Rys.4.23. - Cones

5. Wyniki

5.1. Metodologia

Wykorzystywany do implementacji język *OpenCL,* posiada wbudowany system pomiarowy potrafiący rejestrować dokładny czas rozpoczęcia i zakończenia wykonywanych obliczeń dla konkretnego zdarzenia. Osobne zdarzenia zostały przypisane do każdego z omówionych etapów algorytmu, dzięki czemu uzyskano dokładny czas trwania pojedynczych jąder obliczeniowych, czyli funkcji wywoływanych na procesorze masowo równoległym. Prędkość wykonania całych zaimplementowanych metod została zmierzona od początku pierwszego zdarzenia do końca ostatniego zdarzenia to znaczy, że całkowity pomiar zawiera również pośrednie czasy kolejkowania kolejno wykonywanych zdarzeń. Każdy z podanych pomiarów został wykonany 10 razy dla każdego z obrazów by zmniejszyć wpływ czynników zewnętrznych do których należą usługi działające w tle systemu operacyjnego. Przez *ilość rozbieżności na sekundę* oznaczam wszystkie istniejące punkty pojedynczego obrazu wraz z ich rozbieżnościami podzielone przez całkowity czas trwania algorytmu. Metoda iteracyjna została wykonana dla $r = 7$ iteracji agregacji oraz $k = 6$ iteracji ulepszeń. Procent błędnych punktów został obliczony na podstawie punktów odbiegających od posiadanej idealnej mapy głębi.

5.2. Szybka estymacja głębi metodą krzyżową

Całkowity czas estymacji mapy głębi dla wybranych obrazów przedstawiono w tabeli (tab.5.1.) dla najkorzystniejszych wyników CPU, zintegrowanych GPU i dedykowanych GPU. Dodatkowa analiza wyników została przedstawiona w tabelach (tab.5.2. - 5.3.).

tab.5.1. Najszybsze wyniki estymacji map głębi

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i7-6700K @4.00GHz	274,23	389,94	388,53	394,39
Intel HD Graphics 530	145,08	201,97	212,99	203,62
GeForce GTX 970	28,97	42,78	45,21	44,30

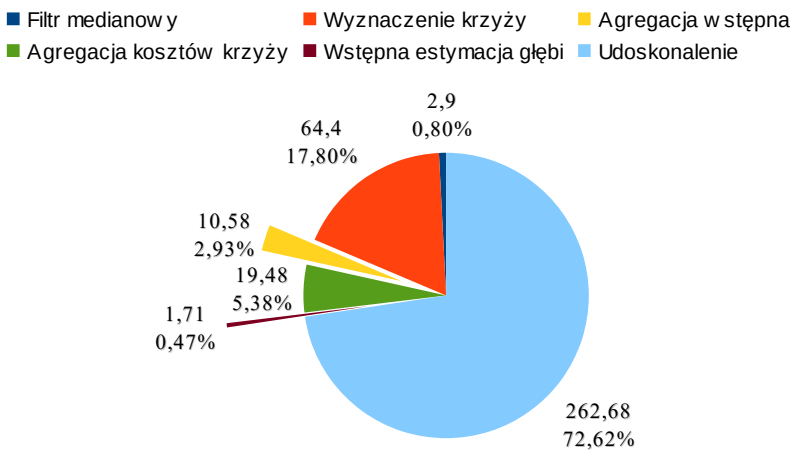
tab.5.2. Procent błędnych punktów

tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
13,90%	5,04%	22,05%	10,08%

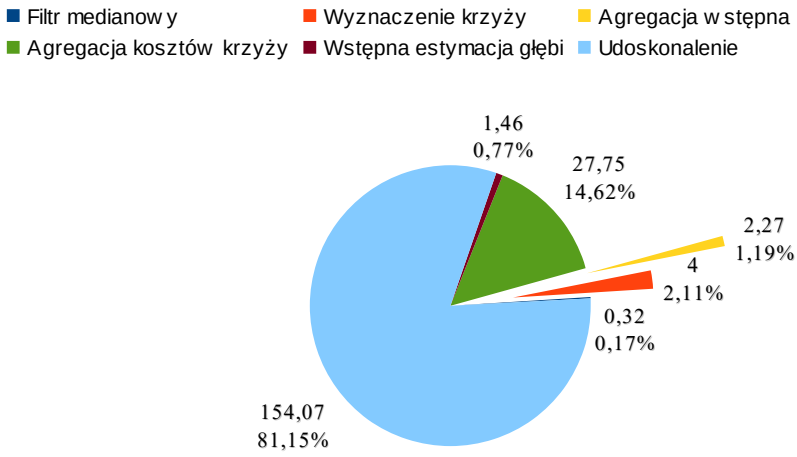
tab.5.3. Analiza wyników wszystkich obrazów

Nazwa jednostki obliczeniowej	średni czas [ms]	rozbieżność na sekundę [10^6]
Intel Core i7-6700K @4.00GHz	361,77	25,50
Intel HD Graphics 530	190,92	48,30
GeForce GTX 970	40,32	230,80

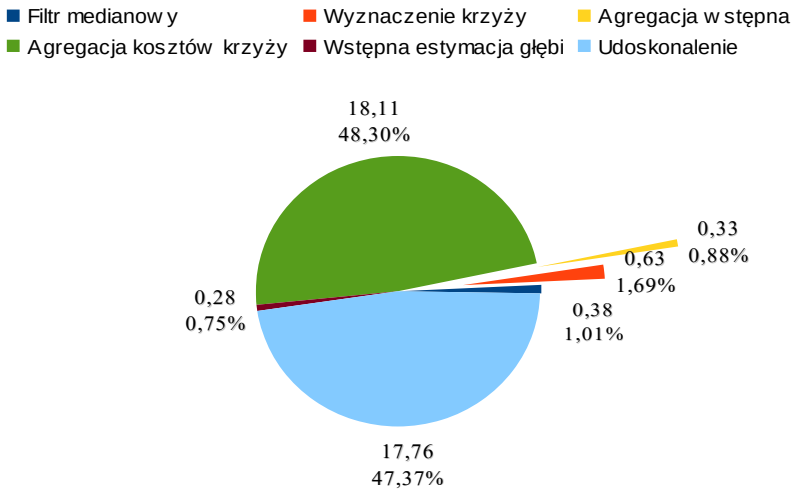
Wykresy kołowe (rys.5.1. - 5.3.) przedstawiają uśredniony procentowy rozkład czasu metody iteracyjnej na różne czynniki, natomiast uzyskane mapy głębi wraz z wzorcami przedstawiono na rysunkach (rys.5.4. - 5.7.).



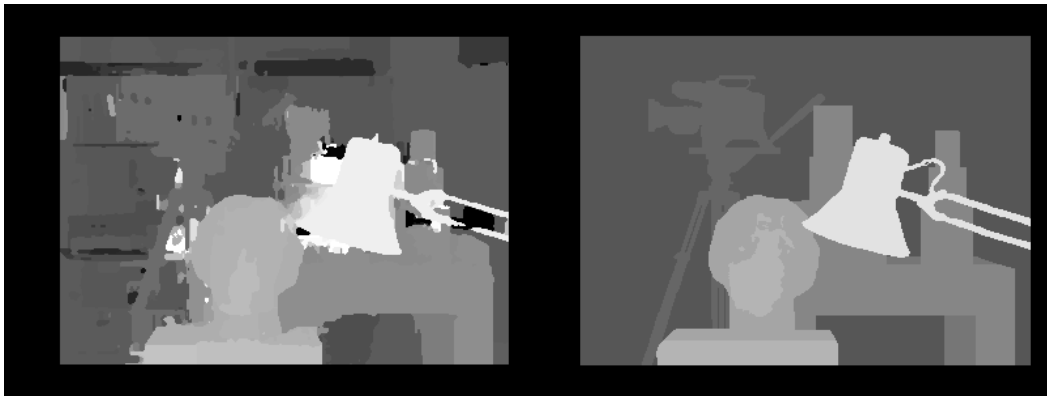
Rys.5.1. Intel Core i7-6700K @4.00GHz



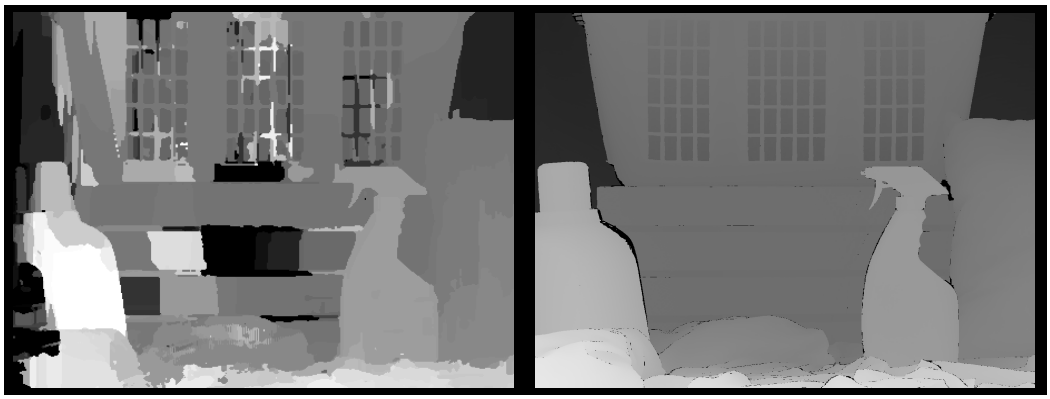
Rys.5.2. Intel HD Graphics 530



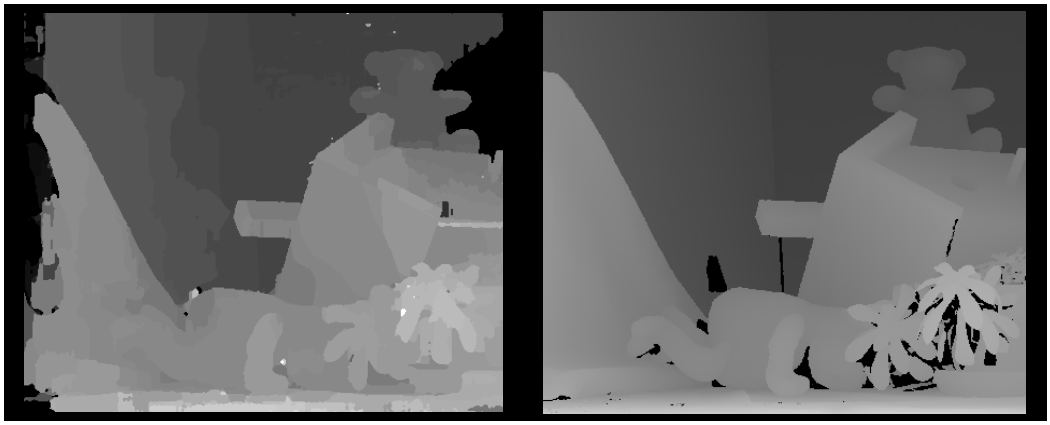
Rys.5.3. GeForce GTX 970



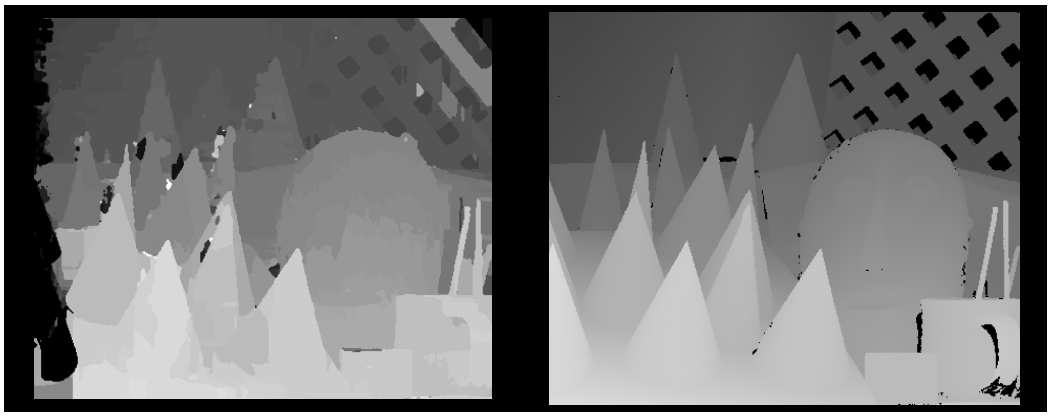
*Rys.*5.4. - *Tsukuba (wzorzec po prawej)*



*Rys.*5.5. - *Laundry*



Rys. 5.6. - *Teddy*



Rys. 5.7. - *Cones*

Dla procesorów graficznych znacząco wyróżnił się czas *Agregacji kosztów krzyży* co jest spowodowane dużą ilością iteracji w pętlach co nie jest dla GPU najwydajniejszą implementacją, natomiast CPU miał większy problem z *Wyznaczeniem krzyży* z powodu dużej ilości zapisu i odczytu danych. Dla obu jednostek obliczeniowych najdłuższa okazuje się implementacja *udoskonaleń,* stworzenie histogramu dla każdego okna jest procesem bardzo złożonym obliczeniowo, ale także niezbędnym do otrzymania zadowalających rezultatów.

5.2.1. Filtr medianowy

Uzyskany czas realizacji na wybranych obrazach przedstawiono osobno dla CPU (tab.5.4.), zintegrowanych GPU (tab.5.5.) i dedykowanych GPU (tab.5.6.).

tab.5.4. CPU

Nazwa jednostki obliczeniowej	tsukuba[ms]	teddy[ms]	laundry[ms]	cones[ms]
Intel Core i3-4170 3.70GHz [11]	8,20	10,81	9,97	10,46
Intel Core i7-4710HQ2.50GHz [12]	1,87	2,86	2,81	2,84
Intel Core i7-6700K 4.00GHz [13]	1,17	1,72	1,70	1,73

tab.5.5. zint. GPU

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	0,10	0,16	0,15	0,16
Intel HD Graphics 4400	0,35	0,56	0,53	0,57
Intel HD Graphics 4600	0,41	0,60	0,55	0,61

tab.5.6. GPU

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870 [14]	0,02	0,03	0,03	0,03
GeForce GTX 760 [15]	0,04	0,06	0,06	0,06
GeForce GTX 970 [16]	0,03	0,04	0,04	0,04

Filtr medianowy został zastosowany na obrazie lewym i prawym by zniwelować drobne rozbieżności wstępnych obliczeń które wpływają znacząco na wyniki kolejnych etapów algorytmu. Realizując metodę krzyżową filtr został użyty jednocześnie na obrazie lewym i prawym. Dedykowane procesory graficzne są jednoznacznie szybsze od wybranych jednostek obliczeniowych, kolejno o rząd i o dwa rzędy wielkości szybsze od zintegrowanych kart graficznych i centralnych jednostek obliczeniowych.

5.2.2. Wyznaczenie zasięgu krzyży

W tabelach Przedstawiono wyniki osobno dla CPU (tab.5.7.), zintegrowanych GPU (tab.5.8.) i dedykowanych GPU (tab.5.9.). Parametry zostały odpowiednio dopasowane biorąc pod uwagę czas realizacji jak i dokładność w odwzorowaniu krzyży i płaszczyzn przedstawianych przez obraz.

tab.5.7. CPU - Wyznaczanie krzyży

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	209,06	312,94	279,77	292,82
Intel Core i7-4710HQ@2.50GHz	96,79	154,69	153,18	155,09
Intel Core i7-6700K @4.00GHz	46,30	70,48	69,91	70,91

tab.5.8. zint. GPU- Wyznaczanie krzyży

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	2,85	4,42	4,31	4,44
Intel HD Graphics 4400	7,96	12,90	12,33	12,97
Intel HD Graphics 4600	9,66	13,04	12,59	13,47

tab.5.9. GPU- Wyznaczanie krzyży

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	1,08	2,88	2,81	2,30
GeForce GTX 760	0,68	1,16	1,13	1,13
GeForce GTX 970	0,47	0,70	0,68	0,68

Podczas wyznaczania krzyży czas obliczeń jest silnie zależny od wybranego obrazu. Najszybciej obliczanym obrazem był *tsukuba*, ponieważ posiada on najmniejsze okna obrazu w których punkty są do siebie bardzo podobne.

5.2.3. Agregacja kosztów

Czas obliczenia kosztów na wybranych obrazach przedstawiono w tabelach osobno dla CPU (tab.5.10.), zintegrowanych GPU (tab.5.11.) i dedykowanych GPU (tab.5.12.). Poprawne wyznaczenie kosztów jest najważniejszym elementem algorytmu, ponieważ determinuje on jak bardzo konkretna rozbieżność może być tą właściwą. Znając więcej informacji o obrazie można przyśpieszyć liczenie kosztów ograniczając zakres badanych rozbieżności wartościami d_{min} *i* d_{max}.

tab.5.10. CPU - Agregacja kosztów

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	41,84	60,63	55,25	57,12
Intel Core i7-4710HQ@2.50GHz	13,72	21,21	21,23	21,17
Intel Core i7-6700K @4.00GHz	11,68	11,35	11,65	7,65

tab.5.11. zint. GPU - Agregacja kosztów

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	0,80	2,74	2,77	2,75
Intel HD Graphics 4400	5,84	9,33	8,95	8,92
Intel HD Graphics 4600	3,93	6,35	6,41	6,41

tab.5.12. GPU - Agregacja kosztów

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	0,46	1,13	1,17	1,12
GeForce GTX 760	0,24	0,44	0,47	0,44
GeForce GTX 970	0,24	0,36	0,36	0,36

5.2.4. Akumulacja kosztów

Czas poziomej akumulacji kosztów na wybranych obrazach przedstawiono w tabelach osobno dla CPU (tab.5.13.), zintegrowanych GPU (tab.5.14.) i dedykowanych GPU (tab.5.15.).

tab.5.13. CPU - Akumulacja wierszy

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	5,28	7,82	7,67	7,64
Intel Core i7-4710HQ@2.50GHz	3,37	5,23	5,17	5,19
Intel Core i7-6700K @4.00GHz	1,68	2,59	2,54	2,58

tab.5.14. zint. GPU - Akumulacja wierszy

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	17,69	7,32	7,20	7,32
Intel HD Graphics 4400	25,85	35,13	36,32	35,10
Intel HD Graphics 4600	72,64	126,55	123,76	124,99

tab.5.15. GPU - Akumulacja wierszy

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	18,01	19,37	18,93	19,32
GeForce GTX 760	21,19	23,14	23,53	23,16
GeForce GTX 970	9,67	15,79	15,38	15,84

Czas pionowej akumulacji kosztów na wybranych obrazach przedstawiono w tabelach osobno dla CPU (tab.5.16.), zintegrowanych GPU (tab.5.17.) i dedykowanych GPU (tab.5.18.).

tab.5.16. CPU - Akumulacja kolumn

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	8,88	19,44	17,74	18,76
Intel Core i7-4710HQ@2.50GHz	3,49	6,23	6,16	6,51
Intel Core i7-6700K @4.00GHz	1,68	2,78	2,74	2,77

tab.5.17. zint. GPU - Akumulacja kolumn

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	1,67	4,28	4,191	4,25
Intel HD Graphics 4400	5,16	12,65	11,52	12,54
Intel HD Graphics 4600	3,61	20,83	20,19	20,61

tab.5.18. GPU - Akumulacja kolumn

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	0,67	1,11	1,06	1,07
GeForce GTX 760	0,70	1,22	1,21	1,22
GeForce GTX 970	0,48	1,16	1,15	1,16

Mimo, że użyto tych samych obrazów w akumulacji poziomej i pionowej czasy znacząco się różnią, wynika to z sposobu zrównoleglania obliczeń i w przypadku poziomej akumulacji procesory GPU nie okazały się wystarczająco wydajne ze względu na dużą liczbę iteracji na każdym rdzeniu z czym procesor CPU nie ma najmniejszych problemów, dlatego też znacząco wyprzedziły dedykowane karty graficzne które nadrobiły swoje swoje braki podczas agregacji pionowej ponownie osiągając czasy obliczeń poniżej jednej milisekundy.

5.2.5. Wyznaczanie kosztów krzyży

Czas obliczeń kosztu poziomego zasięgu krzyży na wybranych obrazach przedstawiono w tabelach osobno dla CPU (tab.5.19.), zintegrowanych GPU (tab.5.20.) i dedykowanych GPU (tab.5.21.).

tab.5.19. CPU - koszty wierszy

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	5,28	7,82	7,67	7,64
Intel Core i7-4710HQ@2.50GHz	8,38	13,16	13,10	13,18
Intel Core i7-6700K @4.00GHz	5,43	8,34	8,27	8,34

tab.5.20. zint. GPU - koszty wierszy

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	3,31	6,60	11,79	6,67
Intel HD Graphics 4400	6,29	22,10	23,53	20,42
Intel HD Graphics 4600	8,63	17,25	18,68	17,11

tab.5.21. GPU - koszty wierszy

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	1,54	2,37	2,32	2,22
GeForce GTX 760	1,26	2,31	2,35	2,33
GeForce GTX 970	1,05	1,61	1,60	1,64

Czas obliczeń kosztu pionowego zasięgu krzyży na wybranych obrazach przedstawiono w tabelach osobno dla CPU (tab.5.22.*)*, zintegrowanych GPU (tab.5.23.*)* i dedykowanych GPU (tab.5.24.*)* .

tab.5.22. CPU - koszty kolumn

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	56,23	80,66	71,49	79,96
Intel Core i7-4710HQ@2.50GHz	9,08	15,56	15,47	15,80
Intel Core i7-6700K @4.00GHz	4,77	7,77	7,83	7,81

tab.5.23. zint. GPU - koszty kolumn

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	3,38	7,14	11,05	7,15
Intel HD Graphics 4400	7,40	23,47	23,13	22,66
Intel HD Graphics 4600	9,94	20,10	22,11	20,09

tab.5.24. GPU - koszty kolumn

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	1,40	2,11	2,11	2,16
GeForce GTX 760	1,64	3,27	3,25	3,35
GeForce GTX 970	1,05	1,62	1,61	1,63

Obliczanie kosztów kolumn i wierszy zostało zrealizowane niemalże identycznie pod względem obliczeniowym, procesory zintegrowane nie mają w tym przypadku znacznych odchyleń od centralnych jednostek obliczeniowych, podczas obliczania kosztów dokonuje się wiele zapisu i odczytu danych, a to daje przewagę obliczeniową dedykowanym procesorom graficznym.

5.2.6. Wyznaczenie wstępnej mapy głębi

Czas obliczeń wstępnej mapy głębi przedstawiono w tabelach osobno dla CPU (tab.5.*25.)*, zintegrowanych GPU (tab.5.26*.)* i dedykowanych GPU (tab.5.27.).

tab.5.25. CPU - Wstępna mapa głębi

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	8,57	18,39	10,29	18,30
Intel Core i7-4710HQ@2.50GHz	2,39	4,53	4,46	4,54
Intel Core i7-6700K @4.00GHz	1,07	1,92	1,91	1,92

tab.5.26. zint. GPU- Wstępna mapa głębi

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	1,04	1,61	1,59	1,60
Intel HD Graphics 4400	2,58	4,07	4,01	3,95
Intel HD Graphics 4600	2,74	4,14	3,87	4,16

tab.5.27. GPU- Wstępna mapa głębi

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	0,65	0,54	0,54	0,54
GeForce GTX 760	0,25	0,46	0,47	0,46
GeForce GTX 970	0,20	0,31	0,31	0,31

Czasy otrzymane przy użyciu CPU oraz zintegrowanych GPU są do siebie zbliżone natomiast dedykowane procesory graficzne były w stanie policzyć tę samą mapę głębi o rząd wielkości krótszym czasie.

5.2.7. Udoskonalenie wyników

Czas obliczeń końcowych udoskonaleń wyników przedstawiono w tabelach osobno dla CPU (tab.5.28.*)*, zintegrowanych GPU (tab.5.29.*)* i dedykowanych GPU (tab.5.*30.*).

tab.5.28. CPU - Udoskonalenie wyników

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	1030,07	1319,15	1284,39	1297,44
Intel Core i7-4710HQ@2.50GHz	381,68	545,04	544,07	553,33
Intel Core i7-6700K @4.00GHz	203,43	281,24	280,83	285,20

tab.5.29. zint. GPU- Udoskonalenie wyników

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	112,94	166,57	168,66	168,11
Intel HD Graphics 4400	166,51	245,40	251,14	245,48
Intel HD Graphics 4600	384,20	553,67	559,66	548,66

tab.5.30. GPU- Udoskonalenie wyników

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	19,75	29,20	30,99	29,08
GeForce GTX 760	52,02	72,52	89,11	80,10
GeForce GTX 970	13,02	17,79	20,83	19,40

Udoskonalenie wyników czyli operacja na histogramie jest najbardziej spowalniającą częścią metody krzyżowej której optymalizacja dałaby olbrzymie korzyści.

5.3. Szybka estymacja głębi metodą iteracyjną

Całkowity czas estymacji głębi metodą iteracyjną dla siedmiokrotnej agregacji oraz sześciokrotnego ulepszenia został przedstawiony w tabeli (tab.5.31.). Dodatkowa analiza wyników została przedstawiona w tabelach (tab.5.32. - tab.5.33.).

tab.5.31. Najszybsze wyniki estymacji map głębi

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i7-6700K @4.00GHz	906,10	1416,71	1426,09	1420,98
Intel HD Graphics 530	894,76	1521,65	1502,98	1521,56
GeForce GTX 970	180,17	291,76	288,83	293,66

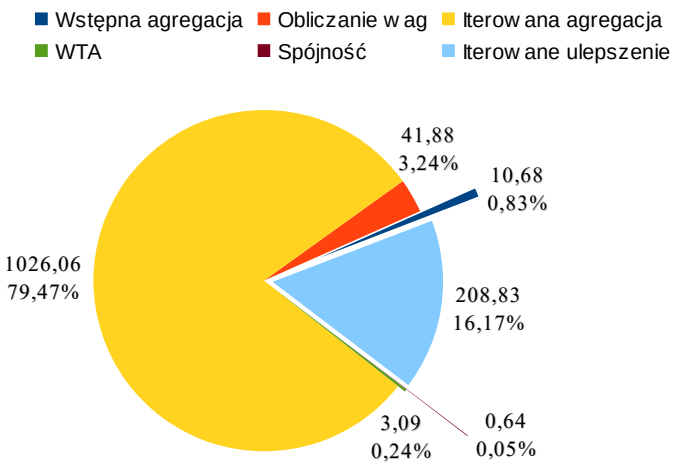
tab.5.32. Procent błędnych punktów

tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
14,04%	6,92%	13,02%	9,04%

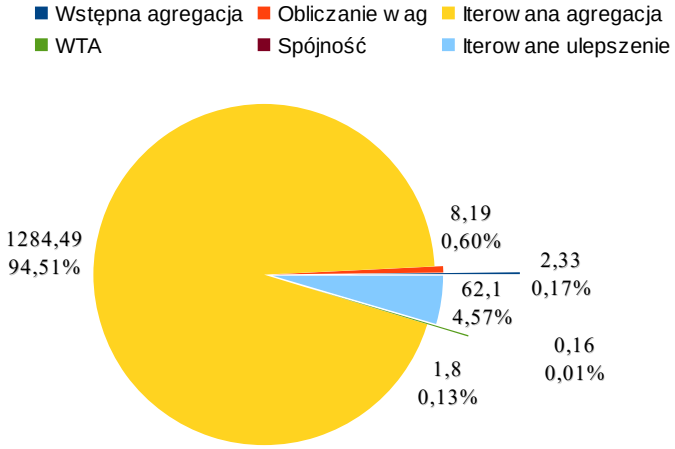
tab.5.33. Analiza wyników wszystkich obrazów

Nazwa jednostki obliczeniowej	średni czas [ms]	rozbieżność na sekunde [10^6]
Intel Core i7-6700K @4.00GHz	1292,47	7,14
Intel HD Graphics 530	1360,24	6,79
GeForce GTX 970	263,61	35,02

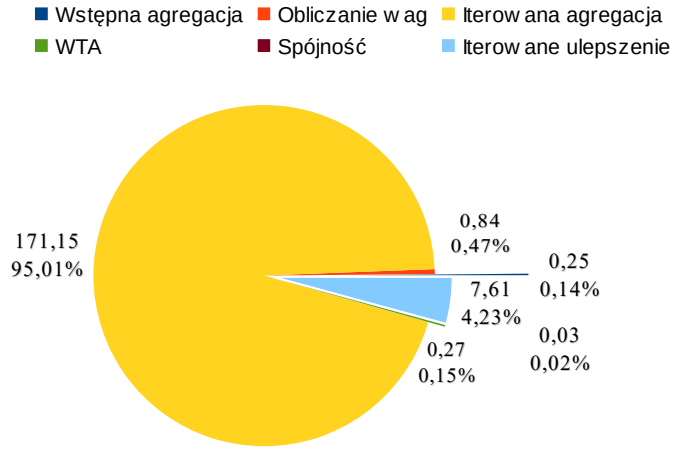
Metoda iteracyjna cechuje się wysoką dokładnością, lecz czas obliczeń jest wydłużony przez kilkukrotne powtarzanie agregacji kosztów oraz ulepszenie rozbieżności. Wykresy kołowe (rys.5.8. - 5.10.) przedstawiają uśredniony procentowy rozkład czasu metody iteracyjnej na różne czynniki, natomiast uzyskane mapy głębi wraz z wzorcami przedstawiono na rysunkach (rys.5.11. - 5.14.).



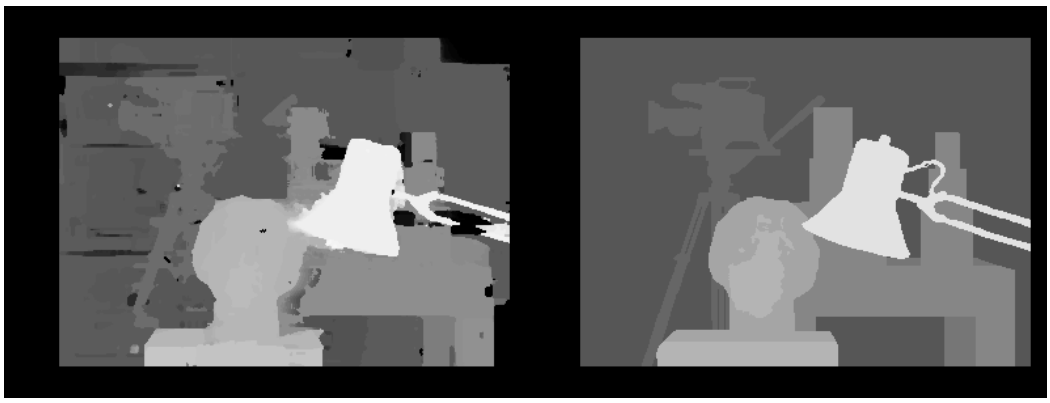
Rys.5.8. Intel Core i7-6700K @4.00GHz



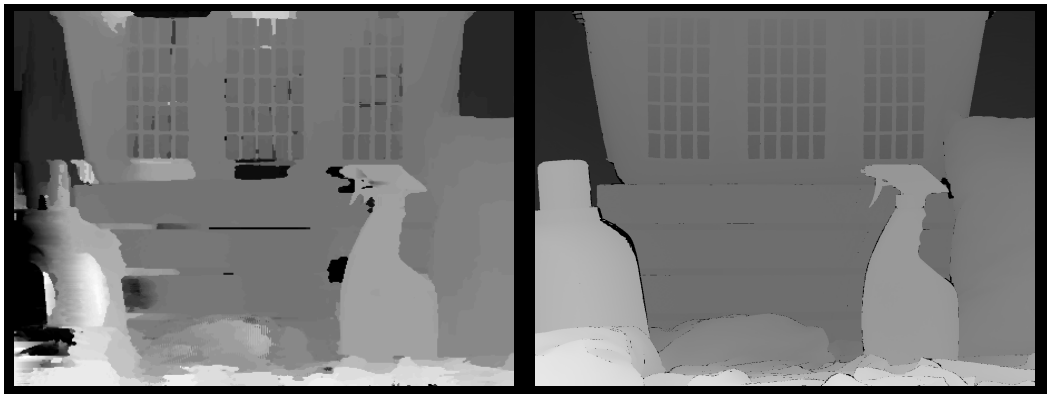
Rys.5.9. Intel HD Graphics 530



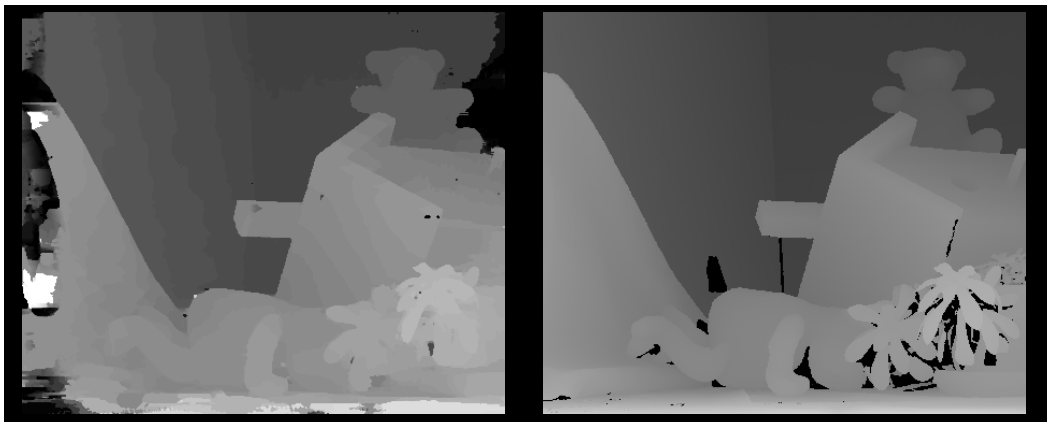
Rys.5.10. GeForce GTX 970



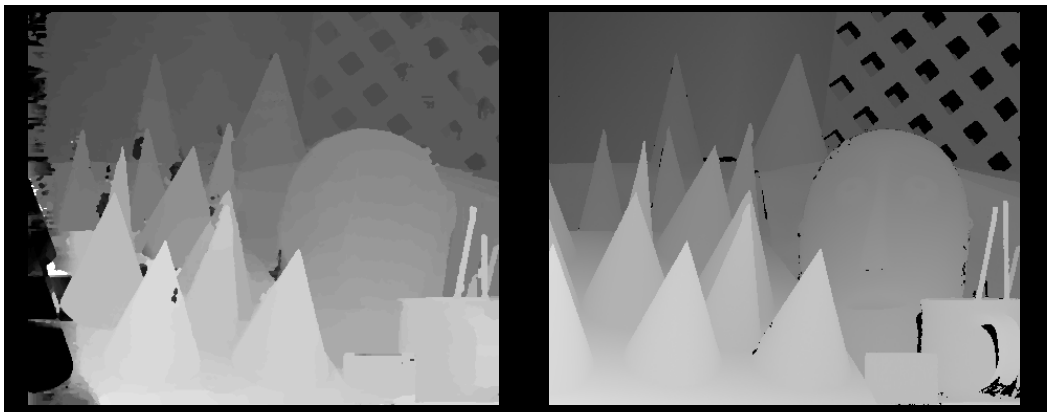
Rys.5.11. - Tsukuba



Rys.5.12. - Laundry



Rys.5.13. - Teddy



Rys.*5.14.* - Cones

Procesory graficzne wykazały się znacznie dłuższym czasem *iterowanej agregacji* niż jakikolwiek inny czynnik składający się na użytą metodę, procesory CPU natomiast, miały dodatkowy problem podczas obliczania wag dla okien obrazu co jest spowodowane dużą ilością odczytu danych. Metoda krzyżowa ze względu sposób tworzenia nieregularnych okien działa mniej efektywnie na obrazach z dużą ilością podobnych do siebie elementów takich jak *tsukuba*, ale znacznie lepiej radzi sobie z obrazami o bardzo jednolitych pod względem koloru elementach takich jak *laundry*. Metoda iteracyjna wykorzystuje całe okno jako maskę z odpowiednio wyliczonymi wagami co pozwala na lepszą identyfikacje nawet mniejszych i mniej różniących się od tła obszarów obrazu.

5.3.1. Obliczanie wag dla poszczególnych okien

Czas wyznaczenia wag na wybranych obrazach przedstawiono w tabelach osobno dla CPU (tab.5.34.), zintegrowanych GPU (tab.5.35.) i dedykowanych GPU(tab.5.36.).

tab.5.34. CPU - Obliczanie wag

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	316,69	472,71	430,97	476,42
Intel Core i7-4710HQ@2.50GHz	60,66	94,74	93,66	94,66
Intel Core i7-6700K @4.00GHz	30,06	45,89	45,58	45,97

tab.5.35. zint. GPU - Obliczanie wag

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	5,62	9,16	8,80	9,19
Intel HD Graphics 4400	14,82	22,96	22,02	22,66
Intel HD Graphics 4600	18,53	29,81	26,91	29,06

tab.5.36. GPU - Obliczanie wag

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	1,55	1,88	2,12	1,50
GeForce GTX 760	1,87	3,14	3,11	3,14
GeForce GTX 970	0,84	1,34	1,33	1,35

Wyznaczone w ten sposób wagi określają jak duża jest szansa na to że punkt w obrębie okna należy do tego samego obiektu w rzeczywistości, co składa się zarówno na różnorodność koloru jak i odległość (im dalej od punktu bazowego tym mniejsza szansa na przynależność do tej samej powierzchni). Przeprowadzenie obliczeń wymagało znacznej ilości odczytu danych co sprawia, że dedykowane karty graficzne znacząco prowadzą pod względem szybkości.

5.3.2. Iterowana agregacja kosztów - kolumny

Czas obliczeń przedstawiono w tabelach osobno dla CPU (tab.5.37.), zintegrowanych GPU (tab.5.38.) i dedykowanych GPU (tab.5.39.).

tab.5.37. CPU - Agregacja kosztów kolumn

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	617,67	1003,71	745,20	1075,80
Intel Core i7-4710HQ@2.50GHz	104,47	196,57	210,69	193,93
Intel Core i7-6700K @4.00GHz	47,42	78,39	79,05	78,83

tab.5.38. zint. GPU- Agregacja kosztów kolumn

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	59,75	102,30	101,27	102,29
Intel HD Graphics 4400	170,08	273,02	266,24	266,89
Intel HD Graphics 4600	231,54	373,19	360,31	372,44

tab.5.39. GPU- Agregacja kosztów kolumn

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	34,23	56,22	55,99	56,12
GeForce GTX 760	16,39	33,14	32,99	33,10
GeForce GTX 970	12,37	19,57	19,36	19,82

5.3.3. Iterowana agregacja kosztów - wiersze

Czas obliczeń przedstawiono w tabelach osobno dla CPU (tab.5.40.), zintegrowanych GPU (tab.5.41.) i dedykowanych GPU (tab.5.42.).

tab.5.40. CPU - Agregacja kosztów wierszy

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	568,17	947,33	698,15	1012,53
Intel Core i7-4710HQ@2.50GHz	107,27	179,73	187,68	181,48
Intel Core i7-6700K @4.00GHz	55,60	82,28	82,38	82,32

tab.5.41. zint. GPU - Agregacja kosztów wierszy

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	59,15	101,34	100,21	101,36
Intel HD Graphics 4400	165,62	265,75	258,32	260,19
Intel HD Graphics 4600	216,83	330,44	316,92	329,19

tab.5.42. GPU - Agregacja kosztów wierszy

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	33,53	53,94	53,99	53,87
GeForce GTX 760	16,98	32,30	32,14	32,44
GeForce GTX 970	12,08	20,14	19,93	20,15

Czas realizacji jest wysoki, ponieważ powyższy algorytm musi zostać wykonany dla wszystkich możliwych punktów w obrazie i ich rozbieżności. Przeprowadzenie dużej liczby równoległych skomplikowanych obliczeń znacznie przerosło możliwości zintegrowanych GPU które czasowo przegrały z CPU, natomiast dedykowane GPU nadal znacząco przewyższają możliwości obliczeniowe CPU.

5.3.4. Estymacja wstępnych map głębi

Czas realizacji wstępnych map głębi przedstawiony został w tabelach osobno dla CPU (tab.5.43.), zintegrowanych GPU (tab.5.44.) i dedykowanych GPU (tab.5.45.).

tab. 5.43. *CPU - Wstępne mapy głębi*

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	13,59	22,34	15,70	22,84
Intel Core i7-4710HQ@2.50GHz	4,27	7,33	7,92	7,23
Intel Core i7-6700K @4.00GHz	1,81	3,46	3,66	3,43

tab. 5.44. *zint. GPU - Wstępne mapy głębi*

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	1,18	2,02	2,02	1,99
Intel HD Graphics 4400	2,97	5,10	4,87	4,85
Intel HD Graphics 4600	3,50	6,07	5,96	6,08

tab. 5.45. *GPU - Wstępne mapy głębi*

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	1,11	1,96	1,97	2,01
GeForce GTX 760	0,50	0,84	0,85	0,85
GeForce GTX 970	0,27	0,40	0,41	0,41

5.3.5. Spójność map głębi

Czas obliczenia spójności został przedstawiony w tabelach osobno dla CPU (tab.5.46.), zintegrowanych GPU (tab.5.47.) i dedykowanych GPU (tab.5.48.).

tab.5.46. CPU - Spójność map głębi

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	2,38	3,07	3,07	3,20
Intel Core i7-4710HQ@2.50GHz	0,90	1,42	1,43	1,41
Intel Core i7-6700K @4.00GHz	0,46	0,69	0,71	0,69

tab.5.47. zint. GPU - Spójność map głębi

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	0,11	0,17	0,18	0,17
Intel HD Graphics 4400	0,27	0,48	0,44	0,46
Intel HD Graphics 4600	0,31	0,46	0,49	0,46

tab.5.48. GPU - Spójność map głębi

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	0,04	0,06	0,06	0,06
GeForce GTX 760	0,03	0,05	0,05	0,05
GeForce GTX 970	0,03	0,04	0,04	0,04

5.3.6. Iteracyjne ulepszanie map głębi - Agregacja rozbieżności

Uśrednione czasy obliczeń dla pojedynczych obrazów zostały przedstawione w tabelach osobno dla CPU (tab.5.49.), zintegrowanych GPU (tab.5.50.) i dedykowanych GPU (tab.5.51.).

tab.5.49. CPU - Agregacja rozbieżności

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	117,87	160,60	155,26	160,11
Intel Core i7-4710HQ@2.50GHz	20,29	31,99	31,64	32,47
Intel Core i7-6700K @4.00GHz	10,01	15,33	15,39	15,40

tab.5.50. zint. GPU - Agregacja rozbieżności

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	2,42	3,88	3,70	3,90
Intel HD Graphics 4400	4,24	6,89	6,63	6,87
Intel HD Graphics 4600	6,46	10,38	9,89	10,28

tab.5.51 GPU - Agregacja rozbieżności

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	0,45	0,94	0,94	0,92
GeForce GTX 760	0,66	1,14	1,13	1,15
GeForce GTX 970	0,39	0,60	0,59	0,60

5.3.7. Iteracyjne ulepszanie map głębi - Estymacja rozbieżności

Uśredniony czas estymacji głębi został przedstawiony w tabeli osobno dla CPU (tab.5.52.), zintegrowanych GPU (tab.5.53) i dedykowanych GPU (tab.5.54.).

tab.5.52. CPU - Estymacja rozbieżności

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel Core i3-4170@3.70GHz	14,25	21,80	15,94	22,52
Intel Core i7-4710HQ@2.50GHz	4,30	6,66	8,26	6,64
Intel Core i7-6700K @4.00GHz	1,68	3,20	3,54	3,20

tab.5.53. zint. GPU - Estymacja rozbieżności

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Intel HD Graphics 530	1,07	1,76	1,74	1,76
Intel HD Graphics 4400	3,01	4,68	4,46	4,50
Intel HD Graphics 4600	3,48	6,69	6,46	6,60

tab.5.54. GPU - Estymacja rozbieżności

Nazwa jednostki obliczeniowej	tsukuba [ms]	teddy [ms]	laundry [ms]	cones [ms]
Radeon HD 7870	0,77	1,01	0,83	0,98
GeForce GTX 760	0,29	0,53	0,53	0,53
GeForce GTX 970	0,21	0,32	0,32	0,32

5.4. Porównanie metod

Metoda krzyżowa wyróżnia się o rząd szybszą estymacją, kosztem jakości otrzymanych map głębi. Zintegrowane procesory graficzne podczas estymacji głębi metodą krzyżową były szybsze od CPU. Obliczając głębię metodą iteracyjną okazało się, że procesory CPU były szybsze niż zintegrowane GPU. Znaczne różnice prędkości były spowodowane między innymi złożonym obliczeniowo algorytmem agregacji oraz dużą ilością zapisu i odczytu danych, podobne wyniki osiągnięto podczas iteracyjnego ulepszania głębi. Jako końcowe wyniki czasu estymacji wybrano najlepsze z posiadanych procesorów, gdzie wszystkie trzy urządzenia należały do tej samej platformy. W przypadku ogólniejszego porównania biorąc pod uwagę dwu rdzeniowy procesor *Intel Core i3-4170* przewaga szybkości GPU nad CPU wzrosłaby o kolejny rząd wielkości. GPU charakteryzował się dużym spadkiem wydajności podczas wykonywania słabo zrównoleglonych obliczeń które inicjowały cykliczne wykonywanie instrukcji w postaci pętli co miało miejsce podczas obliczania *akumulacji kosztów*. Liczne i złożone instrukcje warunkowe również spowalniają prace układów GPU i gdzie było to możliwe zostały zastąpione osobnymi instrukcjami zwiększając dzięki temu szybkość obliczeń. W obu przygotowanych metodach najszybsze były procesory graficzne wykonujące się o rząd wielkości szybciej niż centralne jednostki obliczeniowe i pozwalające zarazem osiągnąć prędkość estymacji na poziomie czasu rzeczywistego. Warto zaznaczyć, że sprawdzone układy graficzne nie zostały stworzone z myślą o masowym zrównoleglaniu obliczeń, a istnieją obecnie układy specjalnie do tego zaprojektowane, które posiadają kilkakrotnie więcej rdzeni niż te przedstawione w pracy. Szybkość metod w obu przypadkach zależała od konkretnego etapu algorytmu co przy użyciu metody iteracyjnej oznacza powtarzającą się agregacje kosztów. W dokonanych obliczeniach przyjęto siedem iteracji których liczbę można dostosować do potrzeb, zmniejszanie liczby iteracji wiąże się z spadkiem jakości otrzymywanych wyników. Metoda krzyżowa wykazała się długim czasem wykonania ulepszeń i operacji histogramu, mimo to jej wyniki są o rząd wielkości szybsze niż metody iteracyjnej ze względu na jej kosztowną agregacje w celu otrzymania satysfakcjonujących map głębi.

6. Podsumowanie

Celem pracy była implementacja metod estymacji mapy głębi zrównoleglając obliczania na wybranych procesorach graficznych oraz porównanie i analiza ich czasu wykonania z centralnymi jednostkami obliczeniowymi. Dodatkowo należało wskazać najbardziej czasochłonne etapy wybranych algorytmów. Zaimplementowano dwa wybrane algorytmy: metoda krzyżowa i metoda iteracyjna. Wyniki eksperymentów pokazują, że zaimplementowane techniki pozwalają na estymację mapy głębi w czasie rzeczywistym. Dodatkowo zaimplementowane algorytmy cechowały się wysoką jakością wyznaczonych map głębi. Zaimplementowane algorytmy przebadano pod kątem złożoności obliczeniowej poszczególnych etapów i wskazano najbardziej czasochłonne z nich. Podsumowując, wszystkie cele założone w pracy zostały osiągnięte.

Bibliografia

1: Krzysztof Klimaszewski, Krzysztof Wegner,
Wpływ kompresji obrazów i map głębi na syntezę widoków w systemie wielowidokowym
KKRRiT 2009, 18.06.2009, Warszawa

2: Krzysztof Klimaszewski,
Algorytmy kompresji sekwencji wielowidokowych,
Rozprawa doktorska, 2012, Poznań.

3: Marek Domański,
Materiały pomocnicze do wykładu,
http://multimedia.edu.pl/?page=teaching_vs

4: Daniel Scharstein, Richard Szeliski, Heiko Hirschmüller,
Middlebury Stereo Evaluation,
http://vision.middlebury.edu/stereo/eval/

5: Daniel Scharstein, Richard Szeliski, Chris Pal,
Middlebury Stereo Datasets,
IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2003), volume 1, pages 195-202, Madison, WI, June 2003.

6: Ke Zhang, Jiangbo Lu, Gauthier Lafruit,
Cross-Based Local Stereo Matching Using Orthogonal Integral Images,
IEEE TCSVT 2009.

7: Jedrzej Kowalczuk, Eric T. Psota, Lance C. Pérez,
Stereo matching using an iterative refinement method for ASW correspondences,
IEEE TCSVT 23(1):94-104, 2013.

8: Marek Domański,
Obraz cyfrowy,
Wydawnictwo Komunikacji i Łączności, Warszawa

9: Doug Crabtree, June Roys, Greg Stiles,
The Gestalt Principles,
http://graphicdesign.spokanefalls.edu/tutorials/process/gestaltprinciples/gestaltprinc.htm

10: Morgan McGuire, Williams College,
A Fast, Small-Radius GPU Median Filter,
ShaderX6, February 2008.

11: **Intel® Core™ i3-4170 Processor**
http://ark.intel.com/pl/products/77490/Intel-Core-i3-4170-Processor-3M-Cache-3_70-GHz

12: **Intel® Core™ i7-4710HQ Processor**
http://ark.intel.com/products/78930/Intel-Core-i7-4710HQ-Processor-6M-Cache-up-to-3_50-GHz

13: **Intel® Core™ i7-6700K Processor**
http://ark.intel.com/pl/products/88195/Intel-Core-i7-6700K-Processor-8M-Cache-up-to-4_20-GHz

14: **AMD Radeon™ HD 7870 GHZ Edition GPU**
http://www.amd.com/en-us/products/graphics/desktop/7000/7800

15: **GeForce GTX 760**
http://www.nvidia.pl/object/geforce-gtx-760-pl.html#pdpContent=2

16: **GeForce GTX 970**
http://www.nvidia.pl/object/geforce-gtx-970-pl.html#pdpContent=2