

**AGH**

---

---

AKADEMIA GÓRNICZO-HUTNICZA W KRAKOWIE  
Wydział Fizyki i Informatyki Stosowanej

## Praca magisterska

**Paweł Jurgielewicz**

kierunek studiów: fizyka techniczna

**Akcelerator realistycznej grafiki trójwymiarowej  
w języku Vivado HLS dla układu FPGA**

Opiekun: dr inż. Daniel Lewandowski

Kraków, lipiec 2018

## Oświadczenie

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

.....  
(czytelny podpis)

**Tematyka pracy magisterskiej i praktyki dyplomowej Pawła Jurgielewicza,  
studenta V roku studiów kierunku fizyka techniczna**

**Temat pracy magisterskiej: Akcelerator realistycznej grafiki trójwymiarowej w języku Vivado HLS dla układu FPGA**

Opiekun pracy: dr inż. Daniel Lewandowski

Recenzenci pracy: dr hab. inż. Bartosz Mindur

Miejsce praktyki dyplomowej: Korporacyjne Centrum Badawcze ABB, Kraków

**Program pracy magisterskiej i praktyki dyplomowej**

1. Omówienie realizacji pracy magisterskiej z opiekunem.
2. Praktyka dyplomowa:
  - zebranie i opracowanie literatury dotyczącej tematu pracy,
  - przygotowanie wstępnej działającej wersji oprogramowania do śledzenia promieni,
  - ewaluacja przydatności stworzonego systemu dla zastosowań w czasie rzeczywistym,
  - sporządzenie sprawozdania z praktyki.
3. Dalsze usprawnianie akceleratora stworzonego w ramach praktyki dyplomowej.
4. Zmiana sposobu rozwiązania postawionego zadania.
5. Rozbudowa funkcjonalna, optymalizacje algorytmów oraz sprawdzanie poprawności używanych obrazów.
6. Zebranie i opracowanie wyników obliczeń.
7. Zatwierdzenie rezultatów przez opiekuna oraz implementacja projektu w układzie FPGA.
8. Opracowanie redakcyjne pracy.

Termin oddania w dziekanacie: 9 lipca 2018

.....  
(podpis kierownika katedry)

.....  
(podpis opiekuna)



Recenzja opiekuna



Recenzja recenzenta





Niniejsza praca została zrealizowana dzięki wsparciu technologicznemu  
Korporacyjnego Centrum Badawczego ABB w Krakowie



*Kiedy rozpoczynałem pracę nad projektem opisanym na kartach tejże pracy, nie zdawałem sobie sprawy z rangi wyzwania, przed jakim zostałem postawiony. Kiedy zaś zaczął kończyć się czas na jego rozbudowę i ulepszanie, zacząłem czuć niedosyt związanego z tym, iż pewnych funkcjonalności nie uda mi się już zaimplementować. Na początku tej drogi, gdy nic się nie udawało potrzebowałem wsparcia i pocieszenia, zaś na jej końcu pozytywnego hamulca, dzięki któremu projekt udało się utrzymać w ryzach czasowych. Właśnie dlatego moje najszczersze podziękowania należą się mojej żonie Beacie - tylko tyle i aż tyle.*

*Projekt ten nigdy by też nie powstał, gdyby nie bezgraniczna wiara Daniela Lewandowskiego w to, iż podołam postawionemu zadaniu, pomimo że przez bardzo długi okres rezultaty nie napawały optymizmem, a moje zrozumienie wielu technicznych aspektów było dalekie od oczekiwanych. Dzięki Daniel za poświęcony czas!*



---

# Spis treści

---

<b>Spis treści</b>	<b>13</b>
<b>Spis rysunków</b>	<b>15</b>
<b>Spis tabel</b>	<b>19</b>
<b>Wstęp</b>	<b>21</b>
<b>1 Śledzenie promieni</b>	<b>23</b>
1.1 Współczesne techniki generowania trójwymiarowej grafiki komputerowej . . . . .	23
1.1.1 Rasteryzacja . . . . .	23
1.1.2 Śledzenie promieni - zagadnienie widoczności . . . . .	27
1.2 Fizyczne podstawy generowania realistycznych obrazów . . . . .	29
1.2.1 Dystrybucja energii na granicy dwóch ośrodków - równania Fresnela .	37
1.2.2 Podstawowe zagadnienia radiometrii . . . . .	44
1.2.3 Modelowanie materiałów . . . . .	47
Podsumowanie . . . . .	55
<b>2 Układy programowalne</b>	<b>57</b>
2.1 Konfiguracja i języki opisu sprzętu . . . . .	61
2.2 Nowoczesne podejście do tworzenia funkcjonalnych układów elektronicznych	63
2.2.1 Synteza wysokiego poziomu . . . . .	64
2.3 Xilinx i Vivado HLS . . . . .	65
2.3.1 Vivado HLS w opracowaniach . . . . .	65
2.3.2 Podstawowe informacje związane z Vivado HLS . . . . .	66
2.3.3 Wykorzystanie dyrektyw optymalizacyjnych . . . . .	69
2.3.4 Integracja stworzonego modułu IP w układzie FPGA . . . . .	78
Podsumowanie . . . . .	83
<b>3 Akcelerator śledzenia promieni z użyciem układu FPGA</b>	<b>85</b>
3.1 Założenia . . . . .	85
3.2 Budowa modułu śledzenia promieni . . . . .	88
3.2.1 Układ procesorowy . . . . .	91
3.2.2 Wyspecjalizowany akcelerator o ustalonej funkcjonalności . . . . .	100

3.3 Implementacja modułu ViRAY w układzie FPGA . . . . .	137
3.4 Oprogramowanie mikrokontrolera . . . . .	141
Podsumowanie . . . . .	144
<b>Podsumowanie</b>	<b>147</b>
<b>Bibliografia</b>	<b>151</b>

---

# Spis rysunków

---

1.1	Stożek widoczności z pozycji obserwatora . . . . .	25
1.2	Trójkąt po rasteryzacji . . . . .	25
1.3	Triangulacja obiektów na przykładzie sfery . . . . .	26
1.4	Wsteczna rekonstrukcja propagacji fotonów za pomocą śledzenia promieni . .	28
1.5	Konstrukcja promieni . . . . .	33
1.6	Konstrukcja fali odbitej i załamanej na bazie zasady Huygenса . . . . .	34
1.7	Współczynnik załamania w funkcji częstotliwości . . . . .	36
1.8	Ciągłość pola elektrycznego na granicy ośrodków . . . . .	38
1.9	Konfiguracja pola elektrycznego $\vec{E}$ i magnetycznego $\vec{B}$ w przypadku polaryzacji poprzecznej na granicy dwóch ośrodków . . . . .	39
1.10	Odbicie i załamanie strumienia padającego promieniowania . . . . .	40
1.11	Reflektancja i transmitancja dla przypadku obu typów polaryzacji w funkcji kąta padania . . . . .	41
1.12	Reflektancja dla dobrego przewodnika dla obu typów polaryzacji w funkcji kąta padania . . . . .	43
1.13	Współczynniki $\eta$ oraz $k$ dla złota dla optycznych długości fal . . . . .	43
1.14	Zależność pola i kształtu oświetlanej powierzchni od kąta padania . . . . .	45
1.15	Funkcja dwukierunkowego rozkładu rozproszenia BSDF . . . . .	46
1.16	Rodzaje odbicia od powierzchni . . . . .	47
1.17	Przykład rozproszonego odbicia kierunkowego . . . . .	49
1.18	Topologia rzeczywistej powierzchni . . . . .	50
1.19	Przybliżenie powierzchni w modelu mikrościanek dla niskiej szorstkości . . .	51
1.20	Przybliżenie powierzchni w modelu mikrościanek dla wysokiej szorstkości .	51
1.21	Modelowanie odbić w modelu Phonga . . . . .	52
1.22	Zależność funkcji rozbłysku od parametru skupienia $e$ . . . . .	53
1.23	Modelowanie odbić w modelu Blinna-Phonga . . . . .	53
1.24	Geometria odbicia w modelu Torrance'a-Sparrowa . . . . .	54
2.1	Schemat ideowy budowy układu FPGA . . . . .	58
2.2	Budowa podstawowego bloku logicznego w układzie FPGA . . . . .	59
2.3	Rozwój układów FPGA od momentu ich konstrukcji do dziś . . . . .	60
2.4	Przykład optymalizacji wykonania iloczynu skalarnego za pomocą HLS . . .	64
2.5	Przykład działania dyrektywy PIPELINE . . . . .	73

2.6 Sposoby podziału tablicy na mniejsze elementy . . . . .	74
2.7 Schemat działania DATAFLOW . . . . .	76
2.8 Schemat blokowy prostego przykładowego systemu przetwarzania danych . . . . .	80
2.9 Czas konfiguracji i okres wstrzymania . . . . .	82
 3.1 Płytką ewaluacyjną VC707 . . . . .	86
3.2 Rozmieszczenie bitów danych opisujących podinstrukcję w stworzonym procesorze . . . . .	93
3.3 Potok przetwarzania w stworzonym procesorze . . . . .	96
3.4 Mapa głębokości wykonana na podstawie obliczonych przez procesor danych	99
3.5 Profil odległości sfery od obserwatora przechodzący przez punkt najbliższy obserwatorowi . . . . .	99
3.6 Triada zależności między sprzecznymi ze sobą charakterystykami akceleratora	101
3.7 Ogólna zasada działania modułu ViRAY . . . . .	102
3.8 Przykład zapisu danych w tablicach wejściowych . . . . .	110
3.9 Kamera w module ViRAY . . . . .	111
3.10 Optymalizacja czasu wykonania głównej pętli algorytmu . . . . .	114
3.11 Wykorzystanie CORE_BIAS w teście przecięcia . . . . .	115
3.12 Punktowe źródło światła w module ViRAY . . . . .	120
3.13 Oświetlenie płaskiej powierzchni przez punktowe źródło światła . . . . .	121
3.14 Oświetlenie powierzchni odbijającej światło całkowicie w sposób rozproszony zgodnie z modelem Orena-Nayara . . . . .	122
3.15 Rola parametru rozblysku $e$ w modelu Blinna-Phonga . . . . .	123
3.16 Powierzchnie odbijające w sposób lustrzany . . . . .	123
3.17 Powierzchnie odbijające zgodnie ze wzorami Fresnela . . . . .	124
3.18 Powierzchnie odbijające w modelu Torrance'a-Sparrowa . . . . .	124
3.19 Transformacja pozycji piksela do współrzędnych $u, v$ . . . . .	125
3.20 Efekt nałożenia tekstuury na różne typy obiektów w funkcji rodzaju zastosowanego mapowania . . . . .	126
3.21 Błąd bezwzględny oraz względny generowany przez przybliżoną implementację funkcji <code>atan2()</code> . . . . .	128
3.22 Błąd bezwzględny oraz względny generowany przez przybliżoną implementację funkcji <code>acos()</code> . . . . .	129
3.23 Translacja oraz skalowanie tekstuury . . . . .	130
3.24 Algorytm filtracji biliniowej tekstuury . . . . .	131
3.25 Upakowanie danych koloru RGB w 32-bitowej wartości w buforze ramki . . . . .	132
3.26 Porównanie obrazów wygenerowanych przez wcześniejszą wersję modułu ViRAY za pomocą liczb o różnej precyzji . . . . .	134
3.27 Wpływ konfiguracji na poziom komplikacji generowanego obrazu . . . . .	136
3.28 Uproszczony schemat blokowy pełnego systemu przetwarzania danych wykorzystujący stworzony moduł do śledzenia promieni ViRAY . . . . .	139
3.29 Typy tekstuur mogące zostać wygenerowane przez sterownik . . . . .	143

3.30 Wygenerowane w czasie rzeczywistym klatki animacji otrzymane dzięki wykorzystaniu techniki śledzenia promieni akcelerowanej przez stworzony z użyciem Vivado HLS moduł ViRAY . . . . .	144
---	-----



---

## Spis tabel

---

3.1	Porównanie podstawowych parametrów opisujących wykorzystane układy FPGA	86
3.2	Zestawienie typów danych wraz z estymacją wykorzystania poszczególnych zasobów oraz czasu wykonania dla podstawowych operacji arytmetycznych . . . . .	90
3.3	Architektura instrukcji stworzonego układu procesora . . . . .	94
3.4	Wykorzystanie zasobów sprzętowych przez procesor instrukcji . . . . .	97
3.5	Opis najważniejszych parametrów sterujących zachowaniem modułu VI <sup>R</sup> AY	104
3.6	Wykorzystanie zasobów sprzętowych układu KCU116 przez instancje funkcji sprawdzające przecięcie promienia z geometrią . . . . .	117
3.7	Wykorzystanie zasobów sprzętowych układu KCU116 przez poszczególne implementacje funkcji cykometrycznych . . . . .	127
3.8	Wykorzystanie zasobów układu KCU116 oraz szybkość finalnej wersji modułu VI <sup>R</sup> AY . . . . .	133
3.9	Wykorzystanie zasobów sprzętowych układu KCU116 przez zaprojektowany system przetwarzania danych . . . . .	141



---

# Wstęp

---

Celem najnowszych technologii jest przede wszystkim zwiększenie jakości życia, wygody oraz produktywności korzystających z nich ludzi. Gdy określony krąg specjalistów znających się na danym zagadnieniu stara się stworzyć z użyciem własnych umiejętności rozwiązanie przystępne dla szerszego grona odbiorców mamy do czynienia z postępuem.

Kiedy powstawały pierwsze próby stworzenia możliwie zrozumiałego opisu zachowania układów elektronicznych za pomocą języków opisu sprzętu, nikt nie był zainteresowany tym by upodabniać je do rozpowszechnionych sekwencyjnych języków programowania - liczyło się przede wszystkim wierne odwzorowanie sposobu, w jaki dane transferowane są pomiędzy różnymi elementami elektronicznymi. Opanowanie tych języków wiąże się z zupełnie innym postrzeganiem przetwarzania informacji.

Obecnie okazuje się za sprawą syntezy wysokiego poziomu, takiej jak Vivado HLS, że do konfiguracji układu elektronicznego można wykorzystać wysokopoziomowe języki programowania takie jak C/C++. Dzięki temu programiści mogą wykorzystywać możliwości sprzętowej akceleracji problemów bez konieczności posiadania specjalistycznej wiedzy z zakresu języków opisu sprzętu.

Niniejsza praca koncentruje się właśnie na sprawdzeniu użyteczności narzędzia jakim jest Vivado HLS z użyciem układów FPGA firmy Xilinx na przykładzie generowania realistycznych obrazów metodą śledzenia promieni. Technika ta jest kosztowna obliczeniowo za pomocą tradycyjnych technik opartych o wykonywanie algorytmu przez procesor komputera, jednak w przeciwieństwie do rastryzacji daje realistyczne obrazy w sposób bezpośredni, tylko na podstawie analizy rozchodzenia się światła w przestrzeni.

Niniejsza praca została podzielona na 3 rozdziały:

1. Opisuje, na czym polega technika śledzenia promieni, jakie podstawowe prawa i koncepcje fizyczne są przez nią wykorzystywane. Przedstawione tu zostały również pewne modele służące opisaniu zachowania się różnych typów powierzchni w momencie interakcji ze światłem.
2. Pokazuje, dlaczego układy FPGA są ciekawym narzędziem służącym do akceleracji przetwarzania danych. Opisane zostały cechy Vivado HLS odróżniające go od zwykłych języków sekwencyjnych (w tym dyrektywy optymalizacyjne) oraz fundamentalne informacje dotyczące implementacji stworzonego przy pomocy Vivado HLS modułu w strukturze układu FPGA.
3. Rozdział ten podaje, w jaki sposób próbowało stworzyć moduł dokonujący akceleracji

śledzenia promieni przy pomocy Vivado HLS. Pokazuje on wady i zalety tego narzędzia w różnych sytuacjach, a także opisuje możliwości finalnego modułu, który jest implementowalny w strukturze układu FPGA.

# Rozdział 1

---

## Śledzenie promieni

---

### 1.1 Współczesne techniki generowania trójwymiarowej grafiki komputerowej

Dostęp do informacji (również wizualnej) jest motorem napędzającym rozwój współczesnego świata. Dokładność rozumiana jako wierność oddania zachowania się obiektu po wyprodukowaniu, ma znaczenie tak w przypadku projektu kryształowego wazonu jak i budynku filharmonii. Projektant dysponujący wizualizacją jest w stanie ocenić czy jest usatysfakcjonowany rezultatami swojej pracy czy wymagane są zmiany. Do wykonania takiej wizualizacji może posłużyć się jedną z dwóch ugruntowanych na przestrzeni ostatnich dziesięcioleci technik: rasteryzacją bądź też śledzeniem promieni (ang. *ray tracing*). Ta pierwsza choć szybka pozwala uzyskiwać przybliżone rezultaty o skończonej dokładności (stąd również jej popularność w branży elektronicznej rozrywki, gdzie dla komfortowej rozgrywki wymaga się generowania obrazu w umownym tempie minimum 30 Hz). Druga zaś w prosty sposób umożliwia generowanie fizycznie poprawnych obrazów, jednak za cenę znacznego wydłużenia czasu obliczeń.

W następnych dwóch podrozdziałach zostaną przedstawione podstawy, które stoją za działaniem obu wyżej wspomnianych technik.

#### 1.1.1 Rasteryzacja

Proces rasteryzacji w ogólnym rozumieniu polega na transformacji grafiki wektorowej, czyli takiej, która opisywana jest poprzez położenia, kierunki i odległości, do płaskiego obrazu złożonego z określonej ilości pikseli w pionie i poziomie wypełniających przestrzeń między tymi położeniami (wierzchołkami). W przypadku grafiki trójwymiarowej i posługiwania się biblioteką OpenGL [1] czy wchodzącej w skład DirectX biblioteki Direct3D, wejście całego potoku przetwarzania geometrii stanowi zbiór obiektów (przeważnie trójkątów), z których każdy zbudowany jest z wierzchołków. Każdy wierzchołek  $\vec{v}_i$  posiada zaś szereg atrybutów takich jak:

- macierz przekształceń 4x4 we współrzędnych obserwatora (kamery)  $\mathbf{MV}_i$  zawierającą informację o jego położeniu, skali, obrocie oraz ewentualnych deformacjach (np. po-

chyleniu),

- współrzędne tekstury,
- normalna - o ile w przypadku pojedynczego punktu podawanie normalnej nie ma sensu, o tyle trzeba pamiętać, iż punkty te definiują odpowiednie płaszczyzny obiektów; przy takiej interpretacji można mówić o normalnej do płaszczyzny w danym jej wierzchołku, co wykorzystywane jest chociażby w procesie wizualnego bądź też rzeczywistego (wykorzystując teselacje<sup>i</sup>) wygładzania obiektów,
- parametry dotyczące charakterystyki materiału, z którego zbudowany jest opisywany obiekt<sup>ii</sup>,
- i innych<sup>iii</sup>.

Atrybuty te są następnie wykorzystywane podczas kolejnych etapów tworzenia obrazu.

Ważnym globalnym parametrem jest również *macierz projekcji*  $\mathbf{P}$ , która decyduje, jaka część świata zostanie odwzorowana na finalnym obrazie oraz w jaki sposób obiekty zostaną przez nią zdeformowane. Dokonuje ona transformacji tak, aby przekształcone za jej pomocą wierzchołki znajdujące się wewnątrz ściętego stożka widzenia<sup>iv</sup> miały współrzędne  $[x, y] \in [-1, 1]^2$ . Postać tej macierzy zależy od [2]:

- odległości od obserwatora bliskiej  $n$  oraz dalekiej  $f$  płaszczyzny odcięcia,
- kąta rozwarcia stożka widoczności  $\alpha$

i wygląda następująco:

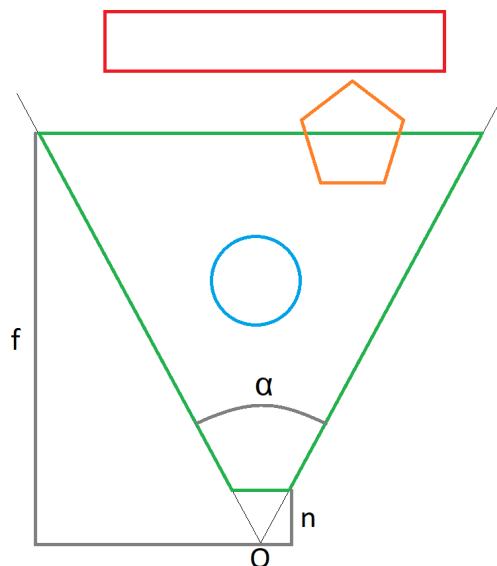
$$\mathbf{P} = \begin{bmatrix} S & 0 & 0 & 0 \\ 0 & S & 0 & 0 \\ 0 & 0 & -\frac{f}{f-n} & -1 \\ 0 & 0 & -\frac{fn}{f-n} & 0 \end{bmatrix}, \quad \text{gdzie } S = \frac{1}{\tan \frac{\alpha}{2}}. \quad (1.1.1)$$

<sup>i</sup>Technika dzielenia siatki trójkątów obiektu na mniejsze, dzięki czemu obiekt wydaje się gładzszy i dokładniejszy niż ten, który pierwotnie został przekazany do renderowania.

<sup>ii</sup>Materiał opisuje sposób zachowania powierzchni w odpowiedzi na interakcję ze światłem. Temat ten zostanie rozwinięty w podrozdziale dotyczącym radiometrii.

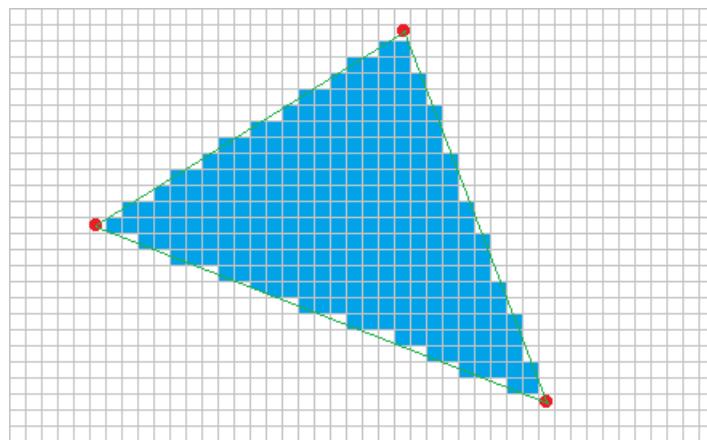
<sup>iii</sup>W dobie programowalnych potoków przetwarzania programiści oraz artyści ograniczeni są właściwie tylko przez własną wyobraźnię. Dla przykładu animacja pomiędzy dwoma położeniami punktów będzie wymagać nie tylko aktualnej macierzy przekształceń, ale również drugiej odpowiadającej następnej klatce kluczowej animacji macierzy wraz z parametrem mówiącym o położeniu pomiędzy tymi dwoma stanami.

<sup>iv</sup>Rozpatrywany przypadek dotyczy rzutowania perspektywicznego. Nic jednak nie stoi na przeszkodzie by zdefiniować macierz  $\mathbf{P}$ , która dokonywać będzie inne przekształcenie np. izometryczne.



**Rys. 1.1:** Rzut z góry na stożek widoczności z pozycji obserwatora  $O$ . Tylko obiekty we wnętrzu stożka opisanego parametrami  $n$ ,  $f$ , oraz  $\alpha$  zostaną narysowane całkowicie, te na brzegach będą wygenerowane częściowo

Wierzchołki przekształcone zgodnie z  $\vec{v}'_i = \mathbf{P} \cdot \mathbf{M}\mathbf{V}_i \cdot \vec{v}_i$  i składające się na jeden prymityw geometryczny są na kolejnych etapach łączone ze sobą tworząc powierzchnię zajmującą określone pozycje pikseli na obrazie.



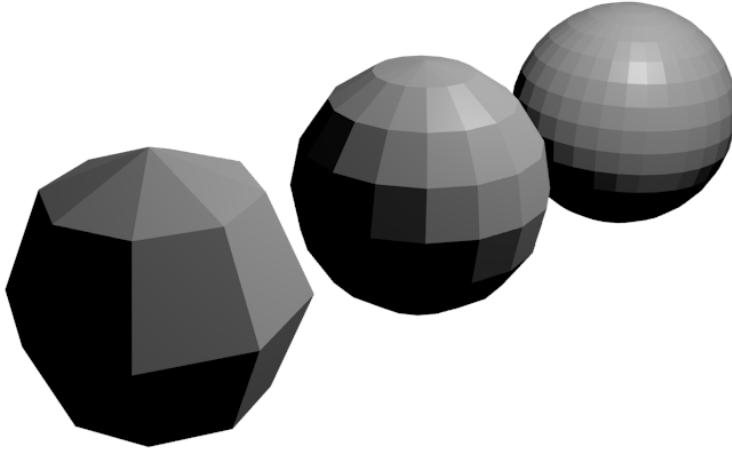
**Rys. 1.2:** Trójkąt po rasteryzacji. Siatka kwadratów odpowiada niepodzielnej siatce pikseli obrazu. Wierzchołki zostały oznaczone **czerwonymi** punktami i połączone ze sobą. Z uwagi na fakt, iż piksele są elementami dyskretnymi, odpowiednie algorytmy muszą zdecydować czy dana płaszczyzna w odpowiednio dużym stopniu zajmuje miejsce odpowiadające danemu pikselowi, w efekcie powstają postrzępione granice obiektów. Dla pikseli znajdujących się we wnętrzu (**niebieskie**) należy dokonać interpolacji atrybutów

Jako, że dokładne wartości atrybutów znane są tylko i wyłącznie dla wierzchołków to wewnątrz powierzchni atrybuty te muszą być interpolowane by móc na kolejnych etapach nadać pikselom pożądany kolor zgodny z przyjętym oświetleniem, materiałem i tekturem. Tak obliczone piksele, o ile znajdują się dalej w przestrzeni obserwatora, mogą zostać jeszcze nadpisane przez kolejne powierzchnie. Wykorzystywany jest do tego *bufor głębokości* (ang. *depth buffer*)

*buffer, z-buffer*), którego rozmiar w pikselach odpowiada generowanemu obrazowi, zaś same piksele przechowują informację o aktualnie najbliższym położeniu. Jeśli aktualnie przetwarzany obiekt (płaszczyzna) dla danego piksela znajduje się bliżej niż wszystko, co do tej pory zostało wyrenderowane, wartość bufora głębokości zostaje nadpisana w tym punkcie przez nową wartość głębi, a piksel obrazu zostanie obliczony na nowo - tym razem biorąc pod uwagę własności tego nowego obiektu. W przeciwnym razie, gdy piksel odpowiadający nowemu obiekowi jest dalej w przestrzeni widoku zostaje on odrzucony. Dzięki czemu nie są przeprowadzane dalsze obliczenia dla elementów, których obserwator i tak nie jest w stanie dostrzec, oszczędzając przy tym czas procesora.

Powyższy opis procesu rasteryzacji, mimo iż znacznie uproszczony w stosunku do wykorzystywanych obecnie potoków przetwarzania [3], zawiera sedno całej operacji i pozwala wyróżnić kilka jej najważniejszych cech.

1. Jest to proces, który idealnie nadaje się do zrównoleglenia obliczeń, dzięki temu, że każda płaszczyzna jest rysowana w sposób niezależny (synchronizacja danych jest wymagana jednak na etapie dostępu do bufora głębokości). Ma to swoje odzwierciedlenie w fakcie, że producenci procesorów graficznych od dawna produkują urządzenia zawierające setki, a nawet tysiące równoległych jednostek przetwarzających [4][5].
2. Z uwagi na fakt, iż określenie pozycji w przestrzeni widoku wymaga znajomości pierwotnego położenia wierzchołka  $\vec{v}_i$  to sam obiekt, którego część ten wierzchołek reprezentuje, musi zostać przybliżony przez sieć wierzchołków. Wynika z tego, iż chcąc uzyskać lepsze odwzorowanie krzywizny powierzchni należy dysponować gęstszą siatką punktów.



**Rys. 1.3:** Triangulacja obiektów na przykładzie sfery - im więcej podziałów powierzchni tym gładszym obiektem, jednak czas renderowania dłuższy

3. Procesor przetwarzający aktualną powierzchnię nie wie nic o pozostałych obiektach w scenie. W szczególności nie wie nic o tym czy dana powierzchnia na drodze do źródła światła jest przesłaniana przez inne płaszczyzny czy też nie. W związku z tym cienie generowane są kilkutapowo. Najprostszą metodą jest wygenerowanie tzw. *mapy cieni* (ang. *shadow map*) z pozycji źródła światła. Jest to tekstura, która określa punkty, do których dociera oświetlenie. Wartości jej pikseli odpowiadają natomiast odległości

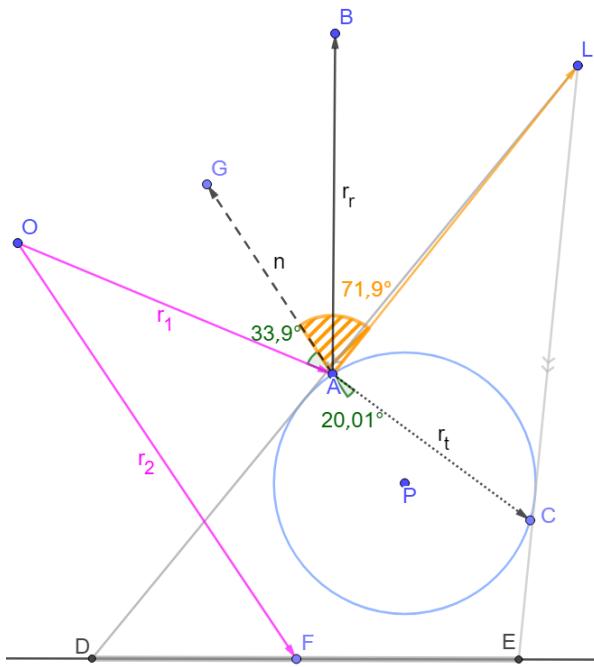
najbliższych źródłu światła obiektów. W kolejnej fazie scena jest już renderowana normalnie z pozycji obserwatora i do określenia miejsc ocienionych używa się informacji z tekstuury wygenerowanej wcześniej. Nie jest to jednak proces idealny. Z uwagi na fakt, że mapa cieni jest tekstyurą jej rozdzielcość w znaczący sposób wpływa na jakość obrazu - im większa tym granice między cieniem a powierzchnią oświetloną są mniej postrzępine. Dodatkowo stosuje się filtrację mapy cieni dla uzyskania gładzych przejść [6]. Podobne trudności sprawia również uzyskanie poprawnych (tzn. zgodnych z perspektywą i zasadami fizyki) odbić np. na powierzchniach lustrzanych czy wodzie. W prostych przypadkach, takich jak płaskie lustro, można sobie z tym poradzić poprzez narysowanie na płaszczyźnie lustrzanego odbicia sceny względem tej płaszczyzny. Jednak w większości wypadków powierzchnie odbijające posiadają skomplikowane kształty, przez co autorzy skłaniają się ku wykorzystaniu technik przybliżonych bądź też bazujących na śledzeniu promieni [7].

W związku z przytoczonymi powyżej faktami można stwierdzić, iż rasteryzacja jest stosunkowo szybkim procesem (o ile obliczenia są prowadzone równolegle) opartym o informacje lokalne (dla danej powierzchni określonej przez zbiór jej wierzchołków), która musi posiadać się różnymi (nieraz bardzo wyrafinowanymi) technikami przybliżonymi w celu stworzenia iluzji realizmu świata przedstawionego.

### 1.1.2 Śledzenie promieni - zagadnienie widoczności

Generowanie grafiki metodą śledzenia promieni jest oparte na fizycznych podstawach, które mówią o tym, w jaki sposób światło (fotony) propaguje się w przestrzeni. Cały proces polega na rekonstrukcji ścieżki, po której poruszają się fotony zanim trafią do obserwatora tworząc obraz (rysunek 1.4), a pierwsze wzmiotki o jego wykorzystaniu sięgają roku 1968 [8].

Z pozycji obserwatora  $\vec{O}$  w kierunku obserwacji  $\vec{d}_i$  ( $|\vec{d}_i| = 1$ ) wysyłane są promienie  $\vec{r}_i = \vec{O} + t\vec{d}_i$ , których celem jest znalezienie najbliższych obserwatorowi przeszkód (obiektów) na drodze propagacji - są to tak zwane *promienie pierwotne* (ang. *primary rays*). Aby znaleźć taki obiekt należy znać postać równania  $f(x, y, z) = 0$  dla każdego obiektu znajdującego się w świecie i umieć je rozwiązać dla tego promienia. Rozwiążaniem takiego równania jest najmniejsza nieujemna wartość  $t$  mówiąca o tym, jak daleko od obserwatora w zadanym kierunku  $\vec{d}_i$  znajduje się obiekt (ujemne wartości  $t$  odpowiadają sytuacji, gdy obiekt znajduje się za obserwatorem). Jeśli aktualna wartość  $t$  jest mniejsza niż znane do tej pory  $t_{min}$  następuje aktualizacja odległości  $t_{min} = t$  oraz wielkości  $p$  wskazującej na napotkany obiekt. W najprostszym przypadku (tzn. bez wykorzystania struktur akcelerujących typu siatki i drzewa [9]) znalezienie najbliższego obiektu  $p$  dla promienia  $\vec{r}_i$  wymaga sprawdzenia wszystkich obiektów w scenie.



**Rys. 1.4:** Wsteczna rekonstrukcja propagacji fotonów za pomocą śledzenia promieni (rzut 2D z boku). Z pozycji obserwatora  $\vec{O}$  wysyłane są promienie sprawdzające ścieżkę, po której poruszały się fotony zanim trafiły do obserwatora. Promień  $\vec{r}_1$  w punkcie  $A$  przecina sferę o środku  $P$ , następnie zostaje odbity  $\vec{r}_r$  względem normalnej  $\vec{n}$  w punkcie  $A$  oraz załamany  $\vec{r}_t$  (promień wtórny). Relacja pomiędzy kątami  $\angle OAG$  i  $\angle LAB$  wpływa na kolor obiektu w danym punkcie zgodnie z przyjętym modelem materiału oraz parametrami opisującymi źródło światła  $L$ . W przypadku promienia  $\vec{r}_2$  przecinającego płaszczyznę przechodzącą przez punkty  $D$  oraz  $E$  w punkcie  $F$ , promień cienia  $\vec{FL}$  napotyka na swojej drodze przeszkodę w postaci sfery, w skutek czego punkt ten będzie znajdował się w cieniu (tak jak wszystkie punkty położone na odcinku  $|DE|$ )

Znalezienie punktu  $\vec{X}$ , w którym nastąpiło przecięcie promienia  $\vec{r}_i$  z najbliższym mu obiektem pozwala na obliczenie m. in. normalnej  $\vec{n}$  w danym punkcie. Jest to kluczowy parametr, wykorzystywany w obliczeniu tak koloru wynikającego z położenia względem źródła światła jak i do stworzenia promieni, których początek  $\vec{O}'$  znajduje się w  $\vec{X}$  zaś kierunek odpowiada promieniu (tzw. *promieniu wtórne* ang. *secondary rays*):

- odbitemu,
- załamanemu,
- tzw. *promieniu cienia* (ang. *shadow ray*) sprawdzającemu czy na drodze między badanym punktem  $\vec{X}$  a źródłem światła nie znajduje się inny obiekt go przysłaniający (rzucający cień na  $\vec{X}$ ).

Dla tych promieni również sprawdzane są testy przecięcia ze wszystkimi obiektami, a efekty tych obliczeń wpływają na wynikowy kolor piksela, który powiązany jest z konkretnym promieniem pierwotnym  $\vec{r}_i$ . W szczególności promienie wtórne mogą rozdzielać się na kolejne wtórne promienie (mówiąc się o tzw. *głębokości śledzenia promieni*, która określa, ile generacji promieni wtórnego jest rozpatrywanych w procesie renderowania obrazu). Im większa głębokość śledzenia promieni tym większy stopień realizmu jak również złożoność obliczeń.

Przedstawiona powyżej technika choć bardzo prosta w swoich założeniach pozwala generować złożone wizualizacje o wysokim poziomie realizmu. Co bardzo istotne w kontekście porównania do rasteryzacji, generowanie cieni czy odbić nie wymaga specjalnego traktowania i tworzenia dodatkowych algorytmów - wynika to wprost z samej idei śledzenia promieni. Wymagana jest jedynie znajomość położenia obserwatora  $\vec{O}$  oraz listy obiektów, których przecięcie z promieniem jesteśmy w stanie obliczyć w sposób analityczny. Wynika stąd również fakt, że nie jest wymagane by obiekty sceny były opisywane poprzez listę ich wierzchołków. Dla przykładu sferę można wyrenderować jako zbiór odpowiedniej ilości trójkątów jak i rozwiązyując analityczne równanie  $(\vec{r}_i - \vec{o})^2 = R^2$ , gdzie  $\vec{o}$  i  $R$  to odpowiednio położenie środka oraz promień sfery. W pierwszym przypadku algorytm musi sprawdzić przecięcia ze wszystkimi trójkątami, podczas gdy w drugim wystarczy tylko rozwiązać równanie kwadratowe, które ponadto poda niemal dokładne rozwiązanie<sup>v</sup>.

W porównaniu do rasteryzacji, śledzenie promieni charakteryzuje się też innym pochodzeniem równoległości obliczeń. W tym przypadku żaden promień (pierwotny, wtórne, cieni), który uczestniczy w tworzeniu koloru danego piksela nie wpływa na kolor innych pikseli. Stąd idealny system śledzący promienie byłby w stanie przetwarzać wszystkie piksele jednocześnie.

Niestety największą wadą, a przez to czynnikiem niepozwalającym na adaptację śledzenia promieni w znacznej części przypadków jest złożoność obliczeniowa. Każdy stworzony promień w scenie musi na podstawie dostarczonej listy obiektów sam określić, który obiekt (jeśli którykolwiek) znajduje się najbliżej na jego drodze. Jest to zupełne przeciwnieństwo sytuacji, która ma miejsce podczas rasteryzacji, gdzie pozycja na mapie pikseli każdego obiektu geometrycznego z listy jest określana tylko raz dla danego obrazu. Stąd koniecznym jest stosowanie struktur akcelerujących [10][11], pozwalających na szybkie stwierdzenie czy i jaki podzbiór wszystkich obiektów znajduje się w danym kierunku propagacji promienia  $\vec{r}_i$ .

Śledzenie promieni jest zatem techniką pozwalającą w sposób oparty na podstawowych zasadach fizyki tworzyć fotorealistyczne obrazy bez konieczności stosowania wyrafinowanych sztuczek będących jedynie przybliżeniem rzeczywistości (a koniecznych podczas rasteryzacji). Niestety z przyczyn wyjaśnionych wcześniej nie jest ona stosowana tam, gdzie wymagane jest tworzenie płynnych animacji w czasie rzeczywistym. Celem tej pracy jest natomiast próba zaadresowania tego problemu oraz stworzenie prototypowego systemu pozwalającego na wykonywanie takich symulacji.

## 1.2 Fizyczne podstawy generowania realistycznych obrazów

Aby zrozumieć sens fizyczny algorytmów używanych podczas śledzenia promieni warto na początku przypomnieć prawa, jakie żądzą zachowaniem się fal elektromagnetycznych, i które stanowią bazę do wyprowadzenia wielu użytkowych zależności. Te zostały zebrane w XIX wieku przez Jamesa Clerka Maxwella i udowodniły ponad wszelką wątpliwość, iż swia-

<sup>v</sup>Należy mieć na uwadze, iż komputerowe obliczenia zmienoprzecinkowe charakteryzują się skończoną dokładnością.

tło (rozumiane jako pełne spektrum - nie tylko zakres widzialny) jest falą wynikającą ze współzależności pomiędzy polem elektrycznym  $\vec{E}$  i magnetycznym  $\vec{B}$ .

- Prawo indukcji Faraday'a - zmienny w czasie strumień pola magnetycznego indukuje pole elektryczne związane z tym strumieniem:

$$\oint_C \vec{E} \cdot d\vec{l} = - \iint_A \frac{\partial \vec{B}}{\partial t} \cdot d\vec{S}. \quad (1.2.1)$$

W szczególności, gdy zamknięty obwód przewodzący zostanie umieszczony w zmiennym strumieniu magnetycznym na jego końcach pojawi się siła elektromotoryczna (SEM).

- Prawo Gaussa dla elektryczności - strumień pola elektrycznego przechodzącego przez dowolną powierzchnię jest wprost proporcjonalny do całkowitego ładunku zawartego w objętości ograniczonej tą powierzchnią:

$$\iint_A \vec{E} \cdot d\vec{S} = \frac{1}{\epsilon} \iiint_V \rho dV, \quad (1.2.2)$$

gdzie  $\epsilon$  jest bezwzględną przenikalnością elektryczną ośrodka, a  $\rho$  gęstością ładunku.

- Uogólnione prawo Ampere'a - płynący prąd powoduje wytworzenie związanego z nim pola magnetycznego:

$$\oint_C \vec{B} \cdot d\vec{l} = \mu \iint_A \left( \vec{J} + \epsilon \frac{\partial \vec{E}}{\partial t} \right) \cdot d\vec{S}, \quad (1.2.3)$$

gdzie  $\vec{J}$  i  $\mu$  są odpowiednio gęstością prądu oraz bezwzględną przenikalnością magnetyczną.

- Prawo Gaussa dla magnetyzmu - nie istnieją monopole magnetyczne tzn. wkład do strumienia linii pola przechodzącego przez dowolną powierzchnię jest taki sam dla biegunów N i S tyle, że z przeciwnym znakiem:

$$\iint_A \vec{B} \cdot d\vec{S} = 0. \quad (1.2.4)$$

Powyższe wzory można również przedstawić w innej postaci, która jak się okaże będzie bardziej użyteczna dla dalszych rozważań. Równania Maxwella daje się zapisać jako zestaw równań różniczkowych korzystając z praw Stokesa (1.2.5) oraz Gaussa-Ostrogradskiego (1.2.6)<sup>vi</sup>:

$$\oint_C \vec{F} \cdot d\vec{l} = \iint_A \nabla \times \vec{F} \cdot d\vec{S}, \quad (1.2.5)$$

$$\iint_A \vec{F} \cdot d\vec{S} = \iiint_V \nabla \cdot \vec{F} dV. \quad (1.2.6)$$

Znalezienie podobieństw pomiędzy powyższymi a odpowiednimi równaniami (1.2.1)-(1.2.4) i stwierdzenie, iż aby obie strony tych równań były sobie zawsze równe, również funkcje podcałkowe muszą być sobie równe prowadzi do poniższych wzorów:

<sup>vi</sup>Odpowiednie założenia dotyczące funkcji wektorowej  $\vec{F}$  oraz obszarów całkowania i ich brzegów uznaje się za spełnione.

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t}, \quad (1.2.7)$$

$$\nabla \cdot \vec{E} = \frac{\rho}{\epsilon}, \quad (1.2.8)$$

$$\nabla \times \vec{B} = \mu \left( \vec{J} + \epsilon \frac{\partial \vec{E}}{\partial t} \right), \quad (1.2.9)$$

$$\nabla \cdot \vec{B} = 0. \quad (1.2.10)$$

W najprostszym przypadku tj. wolnej przestrzeni równania te ulegają uproszczeniu do postaci ( $\mu_0$  i  $\epsilon_0$  są odpowiednio przenikalnością magnetyczną i elektryczną próżni):

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t}, \quad (1.2.11)$$

$$\nabla \cdot \vec{E} = 0, \quad (1.2.12)$$

$$\nabla \times \vec{B} = \mu_0 \epsilon_0 \frac{\partial \vec{E}}{\partial t}, \quad (1.2.13)$$

$$\nabla \cdot \vec{B} = 0. \quad (1.2.14)$$

Rozwiązań można poszukiwać stosując lewostronne operator rotacji ( $\nabla \times$ ) po obu stronach równości dla (1.2.11) oraz (1.2.13) z wykorzystaniem poniższej tożsamości:

$$\nabla \times \nabla \times = \nabla(\nabla \cdot) - \nabla^2 = \nabla(\nabla \cdot) - \Delta, \quad (1.2.15)$$

$$\begin{aligned} \nabla \times (\nabla \times \vec{E}) &= \nabla \times \left( -\frac{\partial \vec{B}}{\partial t} \right), \\ \nabla(\nabla \cdot \vec{E}) - \Delta \vec{E} &= -\frac{\partial}{\partial t} (\nabla \times \vec{B}), \\ \Delta \vec{E} &= \mu_0 \epsilon_0 \frac{\partial^2 \vec{E}}{\partial t^2}. \end{aligned} \quad (1.2.16)$$

Postępując analogicznie:

$$\Delta \vec{B} = \mu_0 \epsilon_0 \frac{\partial^2 \vec{B}}{\partial t^2}. \quad (1.2.17)$$

W wyniku otrzymano równania na zachowanie się pól  $\vec{E}$  i  $\vec{B}$  w postaci równania falowego d'Alemberta:

$$\Delta \vec{F} = \frac{1}{v} \frac{\partial^2 \vec{F}}{\partial t^2}, \quad \vec{F} = \vec{F}(\vec{r}, t), \quad (1.2.18)$$

gdzie stała proporcjonalności:

$$v = c = \frac{1}{\sqrt{\mu_0 \epsilon_0}} \quad (1.2.19)$$

jest szybkością, z jaką porusza się fala ( $c$  - prędkość światła w próżni).

Na podstawie równań (1.2.11)-(1.2.14) można wykazać, że  $\vec{E}$  oraz  $\vec{B}$  są do siebie wzajemnie prostopadłe, a iloczyn wektorowy  $\vec{E} \times \vec{B} = \vec{s}$  wskazuje kierunek propagacji fali - fala elektromagnetyczna propagująca się w wolnej przestrzeni jest falą poprzeczną.

Zakładając, że płaska harmoniczna fala elektromagnetyczna o częstotliwości  $\omega$  porusza się w taki sposób, iż jej propagacja zachodzi w kierunku dodatnim osi  $x$  laboratoryjnego układu współrzędnych, a pole elektryczne w tym układzie posiada tylko składową  $y$ , rozwiązanie równania (1.2.16) prowadzi do:

$$E_y(x, t) = E_{0y} \cos \left[ \omega \left( t - \frac{x}{c} \right) + \phi \right]. \quad (1.2.20)$$

Szukając zależności dla pola magnetycznego  $\vec{B}$  tej fali można posłużyć się (1.2.11):

$$\frac{\partial E_y}{\partial x} = -\frac{\partial B_z}{\partial t} \rightarrow B_z = - \int \frac{\partial E_y}{\partial x} dt, \quad (1.2.21)$$

$$B_z(x, t) = \frac{1}{c} E_{0y} \cos \left[ \omega \left( t - \frac{x}{c} \right) + \phi \right] = \frac{1}{c} E_y(x, t). \quad (1.2.22)$$

W efekcie uzyskane wyrażenie pokazuje, że we wszystkich punktach przestrzeni pola elektryczne  $\vec{E}$  oraz magnetyczne  $\vec{B}$  posiadają identyczną zależność czasową - są zatem w fazie. Co więcej można pokazać, że następująca funkcja wektorowa:

$$\vec{S} = \frac{1}{\mu_0} \vec{E} \times \vec{B} = c^2 \epsilon_0 \vec{E} \times \vec{B}, \quad (1.2.23)$$

zwana wektorem Poyntinga pokazuje kierunek propagacji fali (a więc i kierunek przepływu energii fali elektromagnetycznej) zaś jego wartość  $|\vec{S}|$  opisuje chwilową powierzchniową gęstość mocy promieniowania. W wielu praktycznych zastosowaniach stosuje się uśrednioną w czasie wartość tej wielkości, która nazywana jest *oświetleniem* (ang. *irradiance*):

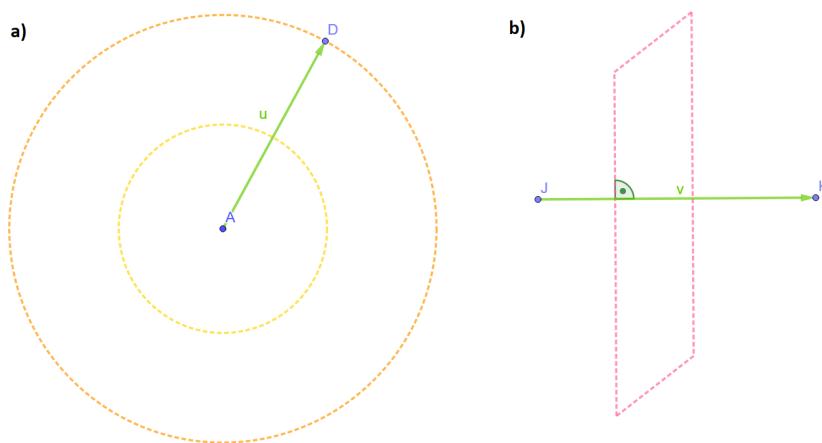
$$I \equiv \langle \vec{S} \rangle_T = c^2 \epsilon_0 \langle \vec{E} \times \vec{B} \rangle_T = \frac{c \epsilon_0}{2} E_0^2. \quad (1.2.24)$$

Czynnik  $\frac{1}{2}$  wynika z faktu uśredniania w czasie funkcji typu  $\cos^2 \alpha t$ <sup>vii</sup>.

Wektor Poyntinga pozwala również na zdefiniowanie niezwykle użytecznej konstrukcji z punktu widzenia techniki śledzenia promieni, a wywodzącej się jeszcze z czasów starożytnych - promień. Promień jest konstrukcją matematyczną (w ogólnym przypadku krzywą), którego kierunek odpowiada kierunkowi przepływu energii elektromagnetycznej. W przypadku ośrodków izotropowych (bez wyróżnionych kierunków) promień jest więc półprostą prostopadłą do frontu falowego.

---

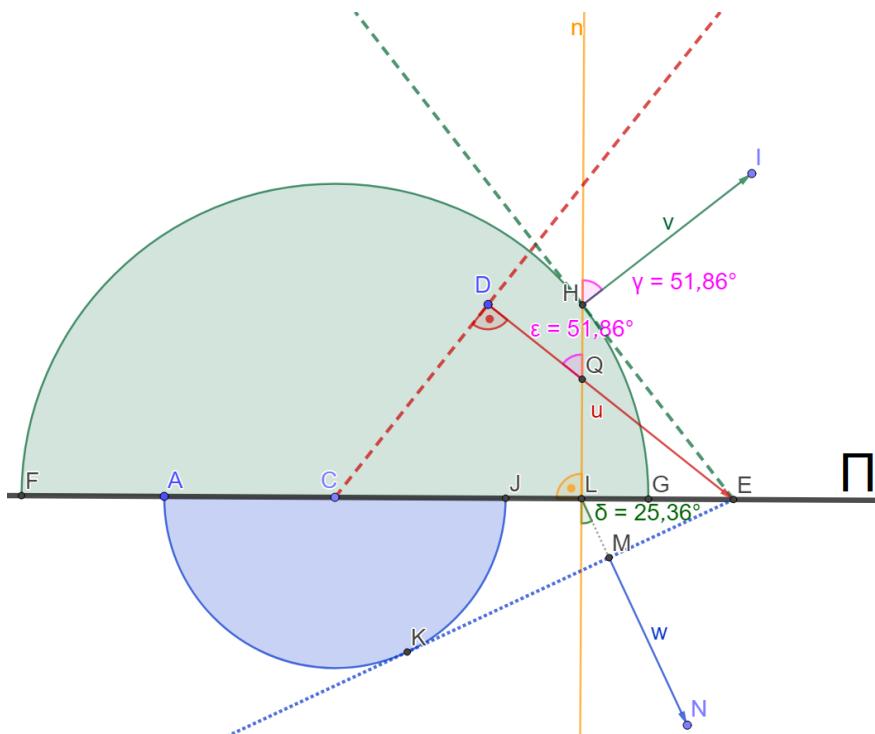
<sup>vii</sup>Do irradiancji powrócimy w momencie omawiania podstawowych wielkości radiometrycznych.



**Rys. 1.5:** Konstrukcja promieni ( $\vec{u}$ ,  $\vec{v}$ ) w przypadku a) źródła o symetrii sferycznej (np. punktowa żarówka) oraz b) fali płaskiej (np. światło słoneczne dochodzące do powierzchni Ziemi). W obu przypadkach promienie są półprostymi (ukazane schematycznie za pomocą wektorów) co zakłada brak anizotropii ośrodka

Koncepcja promieni wraz z zasadą Huygensa, która mówi, iż każdy punkt ośrodka, do którego dochodzi zaburzenie (fala) staje się nowym źródłem elementarnej fali kulistej o takiej samej częstotliwości co zaburzenie pierwotne, pozwala w łatwy sposób zobrazować prawa odbicia i załamania<sup>viii</sup>. Ważnym spostrzeżeniem jest również fakt, iż promienie padające, odbite i załamane na granicy dwóch ośrodków są współplaszczyznowe. Dzięki czemu graficzne przedstawienie tych praw ulega uproszczeniu do dwóch wymiarów przestrzennych.

<sup>viii</sup>Zasada Huygensa została podana na podstawie serii wnioskowych doświadczeń i nie mówi nic o rzeczywistych zjawiskach rozpraszania zachodzących w ośrodku na poziomie mikroskopowym. Szeroka dyskusja zjawisk rozpraszania zawarta jest w [12].



**Rys. 1.6:** Konstrukcja fali odbitej i załamanej na bazie zasady Huygensa. Fala **padająca** na granicę dwóch ośrodków  $\Pi$  pod kątem  $\epsilon$  do normalnej  $\vec{n}$  powoduje powstanie fali **odbitej**, takiej że  $\gamma = \epsilon$  oraz **załamanej**, przy czym kąt załamania  $\delta \neq \epsilon$  i zależy od relacji między prędkością propagacji fali w obu ośrodkach rozdzielonych przez  $\Pi$

Na rysunku 1.6 przedstawiona została procedura powstawania fali odbitej i załamanej w myśl zasady Huygensa. Padający front falowy opisany przez promień  $\vec{u}$  pada na płaszczyznę  $\Pi$  przechodzącą przez punkty  $A$  i  $C$  pod kątem  $\epsilon$  mierzonym względem kierunku normalnego  $\vec{n}$  do tej powierzchni. W momencie gdy fala padająca dociera do  $C$  punkt tej przestrzeni staje się źródłem nowej elementarnej fali, z tą różnicą, że szybkość jej propagacji zależy od ośrodka w którym się ona propaguje. W przypadku zilustrowanym powyżej szybkość propagacji w ośrodku 'nad' granicą wyznaczoną przez płaszczyznę  $\Pi$  jest większa niż 'pod' nią ( $v_i > v_t$ ). W związku z tym zanim front falowy fali padającej dotrze do punktu  $E$  fronty falowe powstałe w  $C$  przemieszczą się na inną odległość. Jako, że fala padająca i odbita poruszają się w tym samym ośrodku ich prędkość propagacji jest identyczna ( $v_i = v_r$ ) przez co  $|DE| = |CH|$  (inaczej mówiąc odległość frontu falowego od  $C$  po czasie  $\Delta t$  potrzebnym na dotarcie fali z  $D$  do punktu  $E$  jest równa  $|DE|$ ). Styczne poprowadzone do obu półsfer z punktu  $E$  definiują nowe fronty falowe po odbiciu i załamaniu fali od powierzchni  $\Pi$ , takie że:

$$\epsilon = \gamma, \quad (1.2.25)$$

$$\frac{\sin \epsilon}{\sin \delta} = \frac{v_i}{v_t} = \frac{n_t}{n_i} = n_w, \quad (1.2.26)$$

gdzie:

$\epsilon$  - kąt zawarty między  $\vec{n}$  a  $\vec{u}$  (kąt padania),

$\gamma$  - kąt zawarty między  $\vec{n}$  a  $\vec{v}$  (kąt odbicia),

$\delta$  - kąt zawarty między  $\vec{n}$  a  $\vec{w}$  (kąt załamania).

O ile prawo odbicia (1.2.25) jest intuicyjnym prawem, znanim z prostych rozważań mechaniki klasycznej (np. odbicie nieobracającego się krążka hokejowego od bandy), o tyle prawo załamania (1.2.26) zawiera w sobie element  $n_w = \frac{n_t}{n_i}$ , znany jako względny współczynnik załamania, równy ilorazowi współczynnika załamania w nowym materiale  $n_t$  i współczynnika załamania materiału, z którego pada promieniowanie  $n_i$ , którego zachowanie nie wynika wprost z równań Maxwella a z mikroskopowych własności obu materiałów.

Jeżeli dielektryk znajdzie się w polu oddziaływania elektrycznego, w wyniku oddziaływania z tym polem nastąpi redystrybucja ładunku w materiale. Pole to dokona separacji ładunków (para ładunków dodatniego i ujemnego tworzy dipol elektryczny), przez co pole wewnętrz materiału będzie inne niż wymuszające. Powstały moment dipolowy na jednostkę objętości (przy założeniu liniowości i izotropii ośrodka) jest zdefiniowany jako:

$$\vec{P} = (\epsilon - \epsilon_0)\vec{E} = (\epsilon_r - 1)\epsilon_0\vec{E} \quad (1.2.27)$$

i nazywany jest *polaryzacją elektryczną*. Jej wartość wyrazić można również jako iloczyn ładunku  $q_e$  oraz jego przesunięcia  $x$  z uwzględnieniem ilości  $N$  ładunków w jednostce objętości:

$$P = q_e x N \quad (1.2.28)$$

Łącząc powyższe wzory (1.2.27) oraz (1.2.28) otrzymuje się wzór na względny współczynnik przenikalności elektrycznej:

$$\epsilon_r = 1 + \frac{1}{\epsilon_0} \frac{q_e x N}{E} \quad (1.2.29)$$

Jeżeli każdy elektron ośrodka, na który pada pole  $\vec{E}$  o wartości  $E = E_0 \cos \omega t$  rozpatrzymy jako oscylator wymuszony dążący do przejścia w stan równowagi z siłą  $F = -k_e x = -m_e \omega_0^2 x$  (w przybliżeniu niewielkich odkształceń  $x$ ) otrzymamy równanie jego ruchu w postaci:

$$m_e \frac{d^2 x}{dt^2} = F_E(t) - k_e x = q_e E_0 \cos \omega t - m_e \omega_0^2 x, \quad (1.2.30)$$

którego rozwiązanie jest następujące:

$$x(t) = \frac{q_e}{m_e} \frac{1}{\omega_0^2 - \omega^2} E(t). \quad (1.2.31)$$

Podstawienie powyższego wyniku do równania (1.2.29) daje wzór na względny współczynnik przenikalności elektrycznej:

$$\epsilon_r = \epsilon_r(\omega) = 1 + \frac{N q_e^2}{\epsilon_0 m_e} \frac{1}{\omega_0^2 - \omega^2}. \quad (1.2.32)$$

Wielkość ta wiąże się z prędkością światła w danym ośrodku poprzez tzw. *współczynnik załamania*  $n$ <sup>ix</sup>

$$n = \frac{c}{v} = \sqrt{\frac{\epsilon \mu}{\epsilon_0 \mu_0}} = \sqrt{\epsilon_r \mu_r} \approx \sqrt{\epsilon_r}. \quad (1.2.33)$$

<sup>ix</sup>Zastosowane przybliżenie  $\mu_r \approx 1$  wiąże się z faktem, że wiele przezroczystych materiałów w zakresie widzialnym ma przenikalność magnetyczną niewiele różniącą się od przenikalności magnetycznej próżni.

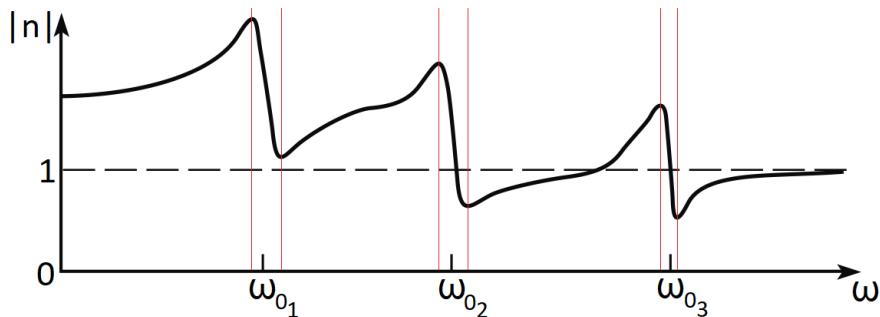
W efekcie:

$$n^2(\omega) = 1 + \frac{Nq_e^2}{\epsilon_0 m_e} \frac{1}{\omega_0^2 - \omega^2}. \quad (1.2.34)$$

Uzyskane równanie to nic innego niż *relacja dyspersji* - zależność współczynnika załamania od częstotliwości wymuszenia  $\omega$ . W ogólniejszym przypadku, dany ośrodek może posiadać wiele częstotliwości rezonansowych  $\omega_{0j}$ , którym odpowiednio podlega  $N_j$  oscylatorów w jednostce objętości. Co więcej w wyniku interakcji między oscylatorami dochodzi do dyssypacji energii. Jeśli do równania (1.2.30) zostanie dodany czynnik związany z tłumieniem proporcjonalnym do prędkości  $m_e \gamma \frac{dx}{dt}$  równanie dyspersji przekształci się do postaci<sup>x</sup>

$$n^2(\omega) = 1 + \frac{q_e^2}{\epsilon_0 m_e} \sum_{j=1}^k \frac{N_j}{\omega_{0j}^2 - \omega^2 + i\gamma_j \omega}. \quad (1.2.35)$$

Na rysunku 1.7 widać, że tak długo, jak częstotliwość  $\omega$  jest istotnie różna od dowolnej częstotliwości rezonansowej  $\omega_{0j}$  współczynnik załamania rośnie wraz ze wzrostem  $\omega$  ( $\frac{dn}{d\omega} > 0$ ) co zaobserwować można podczas rozszczepienia światła białego w pryzmacie, gdzie składowe o wyższej częstotliwości (energii) doświadczają większego ugięcia. Jest to tak zwana *dyspersja normalna*. Jednakże w zakresach bliskich częstotliwośćm rezonansowym  $\omega_{0j}$  dominujący zaczyna być czynnik tłumiący. Z uwagi na silną absorpcję zakresy te nazywa się *pasmami absorpcji* a sam proces *dyspersją anomaloną* ( $\frac{dn}{d\omega} < 0$ ).



**Rys. 1.7:** Współczynnik załamania w funkcji częstotliwości  $|n(\omega)|$  dla hipotetycznego materiału. Wąskie pasma częstotliwości wokół częstotliwości rezonansowych  $\omega_{0j}$  związane z dyspersją anomaloną zostały zaznaczone czerwonymi pionowymi liniami

Na koniec tego podrozdziału, warto również powiedzieć, że w każdym przypadku  $n(\omega)$  z równania (1.2.35) można przedstawić w postaci:

$$n(\omega) = \eta(\omega) - ik(\omega), \quad k(\omega) \in \mathbb{R}_+ \cup \{0\}, \quad (1.2.36)$$

gdzie:

$\eta(\omega)$  - rzeczywista część współczynnika załamania,

$k(\omega)$  - nieujemny współczynnik odpowiadający za tłumienie w ośrodku.

<sup>x</sup>Relację tę można wyprowadzić poprzez analogię do rozważań klasycznego wymuszonego oscylatora z tłumieniem [13].

Dla takiej notacji zależność opisująca ewolucję pola elektrycznego  $\vec{E}$  z równania (1.2.20) przybiera postać:

$$\begin{aligned}
 E_y(x, t; \omega) &= E_{0y} \cos \left[ \omega \left( t - \frac{x}{v(\omega)} \right) + \phi \right] \\
 &= E_{0y} \cos \left[ \omega \left( t - \frac{n(\omega)x}{c} \right) + \phi \right] \\
 &= E_{0y} \cos \left[ \omega \left( t - \frac{(\eta(\omega) - ik(\omega))x}{c} \right) + \phi \right] \\
 &= E_{0y} e^{-\omega k(\omega)x/c} e^{i(\omega(t-\eta(\omega)x/c)+\phi)} \\
 &= E_{0y} e^{-\omega k(\omega)x/c} \cos \left[ \omega \left( t - \frac{\eta(\omega)x}{c} \right) + \phi \right]
 \end{aligned} \tag{1.2.37}$$

Fala propaguje się w ośrodku z prędkością taką, jakby  $\eta(\omega)$  był współczynnikiem załamania. Jednakże w trakcie jej ruchu następuje wykładnicze tłumienie amplitudy oscylacji  $E_{0y} e^{-\omega k(\omega)x/c}$  zależne od  $k(\omega)$ , co odpowiada prawu Lambert-Berra. Przezroczystość ośrodka (materiału) dla danego promieniowania jest określona za pomocą głębokości wnikańia  $\lambda \equiv \alpha^{-1} \equiv (2\omega k(\omega)/c)^{-1}$ , której wartość musi być odpowiednio duża względem grubości danego ośrodka aby uznać go za przezroczysty.<sup>xi</sup>

### 1.2.1 Dystrybucja energii na granicy dwóch ośrodków - równania Fresnela

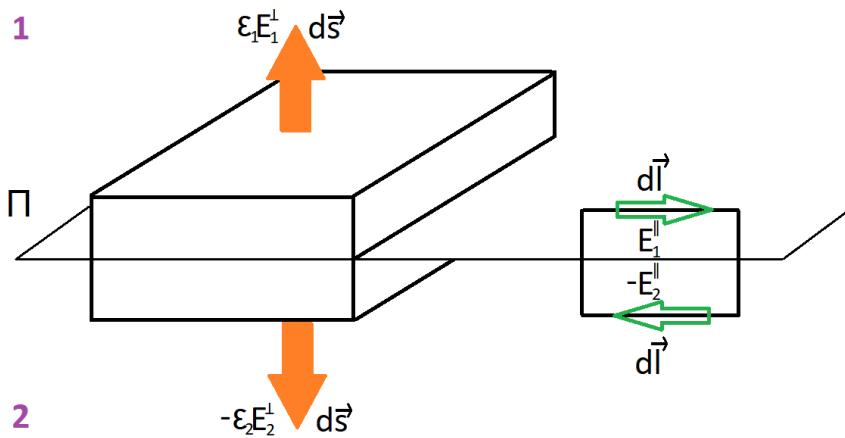
Dotychczasowe rozważania oparte na faktach doświadczalnych i fundamentalnych prawach elektromagnetyzmu pozwoliły zrozumieć podstawowe mechanizmy związane z transportem fali elektromagnetycznej w ośrodkach. Do tej pory jednak nic nie zostało powiedziane o tym, w jaki sposób energia fali padającej na barierę między dwoma ośrodkami dystrybuje się między promienie odbite i załamane. Odpowiedź na to pytanie dają wzory Fresnela.

#### Warunki ciągłości pola elektrycznego i magnetycznego

Na granicy dwóch ośrodków w każdym punkcie, do którego dociera fala elektromagnetyczna odpowiednie składowe pól elektrycznego  $\vec{E}$  i magnetycznego  $\vec{B}$  muszą zostać zachowane. Tak zwane warunki ciągłości można wyliczyć korzystając wprost z równań Maxwella (1.2.1)-(1.2.4). Musi jednak zajść modyfikacja tych równań poprzez włączenie przenikalności elektrycznej  $\epsilon$  i magnetycznej  $\mu$  pod znak całki z uwagi na fakt, iż te w takim wypadku posiadają one zależność przestrzenną.

---

<sup>xi</sup>Czynnik 2 w wyrażeniu na głębokość wnikańia  $\lambda$  bierze się z faktu, że wartość oświetlenia jest proporcjonalna do kwadratu amplitudy drgań pola elektrycznego  $\vec{E}$ .



**Rys. 1.8:** Ciągłość pola elektrycznego na granicy ośrodków. Wymiar w kierunku normalnym do  $\Pi$   $\partial h$  obu obiektów, względem których następuje całkowanie jest nieskończonym w szczególności w porównaniu z  $|d\vec{l}|$

Rysunek 1.8 ilustruje sposób, w jaki można wyprowadzić warunki ciągłości dla pola elektrycznego  $\vec{E}$  na granicy między ośrodkiem 1 a 2 wyznaczonym przez powierzchnię  $\Pi$ . Z prawa Gaussa dla elektryczności (1.2.2) przy założeniu braku swobodnego ładunku powierzchniowego na  $\Pi$ :

$$\oint_A \epsilon \vec{E} \cdot d\vec{S} = 0, \\ \epsilon_1 E_1^\perp - \epsilon_2 E_2^\perp = 0. \quad (1.2.38)$$

Wykorzystanie prawa Faraday'a (1.2.1) daje kolejny warunek (wysokość pętli  $\partial h \ll |d\vec{l}|$ ):

$$\oint_C \vec{E} \cdot d\vec{l} = - \iint_A \frac{\partial \vec{B}}{\partial t} \cdot d\vec{S}, \\ E_1^{\parallel} |d\vec{l}| - E_{1 \rightarrow 2}^{\perp} \partial h - E_2^{\parallel} |d\vec{l}| + E_{2 \rightarrow 1}^{\perp} \partial h = \frac{\partial \vec{B}}{\partial t} |d\vec{l}| \partial h, \\ E_1^{\parallel} - E_2^{\parallel} = 0. \quad (1.2.39)$$

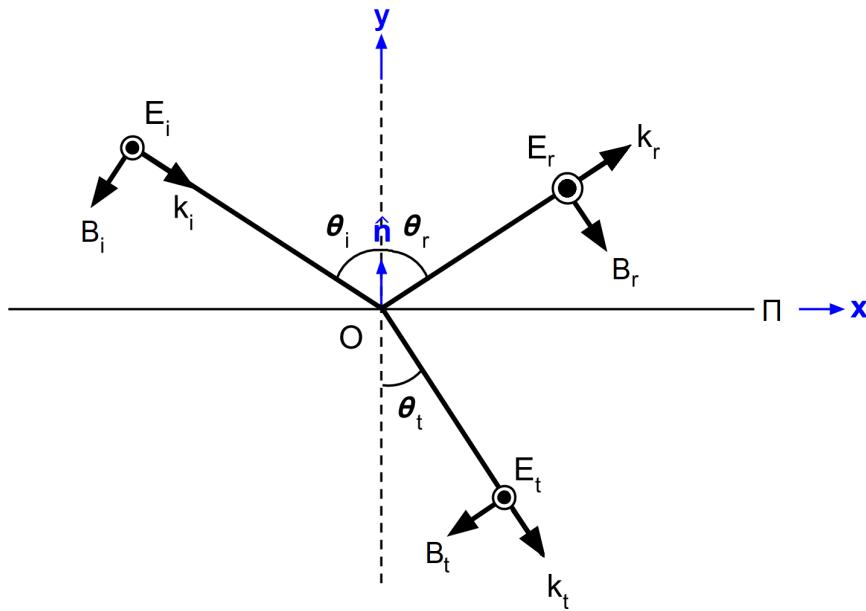
Analogiczne działania z wykorzystaniem dwóch pozostałych praw Maxwella prowadzą do zależności:

$$B_1^\perp - B_2^\perp = 0, \quad (1.2.40)$$

$$\mu_1^{-1} B_1^{\parallel} - \mu_2^{-1} B_2^{\parallel} = 0. \quad (1.2.41)$$

### Wzory Fresnela

Rozważmy falę elektromagnetyczną padającą na granicę między dwoma ośrodkami  $\Pi$  spolaryzowaną w taki sposób, że jej wektor pola elektrycznego  $\vec{E}$  oscyluje w kierunku prostopadłym do płaszczyzny padania (polaryzacja poprzeczna).



**Rys. 1.9:** Konfiguracja pola elektrycznego  $\vec{E}$  i magnetycznego  $\vec{B}$  w przypadku polaryzacji poprzecznej na granicy dwóch ośrodków wyznaczonej przez powierzchnię  $\Pi$ . Fala elektromagnetyczna pada na  $\Pi$  w punkcie  $O$  i w tym punkcie rozważane są odpowiednie wielkości, jednakże ze względu na czytelność ilustracji odpowiednie wektory zostały rozsunięte

Przy takiej konfiguracji, jak na rysunku 1.9 warunek ciągłości składowej stycznej  $\mu^{-1}\vec{B}$  jest wyrażony poprzez następujące równanie:

$$-\frac{B_i}{\mu_i} \cos \theta_i + \frac{B_r}{\mu_i} \cos \theta_r = -\frac{B_t}{\mu_t} \cos \theta_t. \quad (1.2.42)$$

Z (1.2.22), wykorzystując prawo odbicia  $\theta_i = \theta_r$  oraz wiedząc, że fala odbita propaguje się w tym samym ośrodku co fala padająca  $v_i = v_r$ :

$$\frac{1}{\mu_i v_i} (E_i - E_r) \cos \theta_i = \frac{1}{\mu_t v_t} E_t \cos \theta_t. \quad (1.2.43)$$

Pamiętając, że  $n = \frac{c}{v}$ :

$$\frac{n_i}{\mu_i} (E_i - E_r) \cos \theta_i = \frac{n_t}{\mu_t} E_t \cos \theta_t. \quad (1.2.44)$$

Teraz łącząc to z warunkiem ciągłości składowej stycznej pola elektrycznego (1.2.39):

$$E_i \left( \frac{n_i}{\mu_i} \cos \theta_i - \frac{n_t}{\mu_t} \cos \theta_t \right) = E_r \left( \frac{n_i}{\mu_i} \cos \theta_i + \frac{n_t}{\mu_t} \cos \theta_t \right), \quad (1.2.45)$$

$$2E_i \frac{n_i}{\mu_i} \cos \theta_i = E_t \left( \frac{n_i}{\mu_i} \cos \theta_i + \frac{n_t}{\mu_t} \cos \theta_t \right), \quad (1.2.46)$$

$$r_{\perp} \equiv \left( \frac{E_r}{E_i} \right)_{\perp} = \frac{\frac{n_i}{\mu_i} \cos \theta_i - \frac{n_t}{\mu_t} \cos \theta_t}{\frac{n_i}{\mu_i} \cos \theta_i + \frac{n_t}{\mu_t} \cos \theta_t}, \quad (1.2.47)$$

$$t_{\perp} \equiv \left( \frac{E_t}{E_i} \right)_{\perp} = \frac{2 \frac{n_i}{\mu_i} \cos \theta_i}{\frac{n_i}{\mu_i} \cos \theta_i + \frac{n_t}{\mu_t} \cos \theta_t}. \quad (1.2.48)$$

Wyprowadzone współczynniki  $r_{\perp}$  oraz  $t_{\perp}$  są to tzw. amplitudowe współczynniki odpowiednio odbicia oraz załamania. Postępując w sposób analogiczny jak poprzednio można również pokazać, że dla fali elektromagnetycznej spolaryzowanej w taki sposób, że wektor pola elektrycznego  $\vec{E}$  leży w płaszczyźnie padania, amplitudowe współczynniki będą wyglądały w następujący sposób:

$$r_{\parallel} \equiv \left( \frac{E_r}{E_i} \right)_{\parallel} = \frac{\frac{n_t}{\mu_t} \cos \theta_i - \frac{n_i}{\mu_i} \cos \theta_t}{\frac{n_t}{\mu_t} \cos \theta_i + \frac{n_i}{\mu_i} \cos \theta_t}, \quad (1.2.49)$$

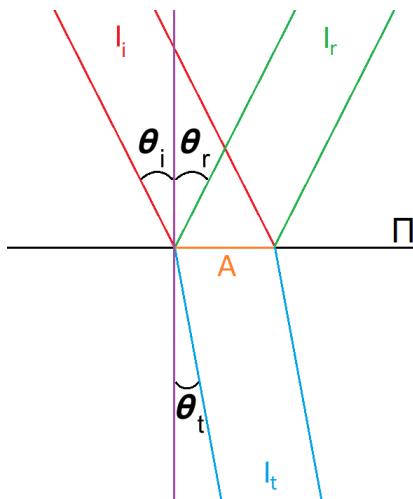
$$t_{\parallel} \equiv \left( \frac{E_t}{E_i} \right)_{\parallel} = \frac{2 \frac{n_i}{\mu_i} \cos \theta_i}{\frac{n_t}{\mu_t} \cos \theta_i + \frac{n_i}{\mu_i} \cos \theta_t}. \quad (1.2.50)$$

W przypadku gdy rozpatrywane materiały nie przejawiają cech magnetycznych tzn.  $\mu_i \approx \mu_t \approx \mu_0$ , powyższe równania ulegają znacznemu uproszczeniu.

W tym momencie znając wyrażenia na amplitudowe współczynniki odbicia i załamania można odpowiedzieć na pytanie, jaką część energii fali padającej uniesie ze sobą fala odbita, a ile załamana. Inaczej mówiąc, jaka będzie *reflektancja R* i *transmitancja T* w zadanych warunkach. W tym celu należy odwołać się do definicji oświetlenia, jako uśrednionej w czasie wartości wektora Poyntinga (1.2.24):

$$I = \frac{c\epsilon_0}{2} E_0^2. \quad (1.2.51)$$

Niech na interfejs  $\Pi$  tak jak na rysunku 1.10 pomiędzy ośrodkami pod kątem  $\theta_i$  pada wiązka promieniowania, oświetlająca pewien obszar o polu  $A$ , zaś  $I_i$ ,  $I_r$ ,  $I_t$  będą gęstościami strumienia promieniowania padającego, odbitego i załamanego.



**Rys. 1.10:** Odbicie i załamanie strumienia padającego promieniowania  $I_i$  na powierzchni  $\Pi$  (rzut na płaszczyznę padania). Strumień padającego promieniowania  $I_i$  przypada na pewną powierzchnię  $A$ . Z uwagi na fakt, iż  $\theta_i = \theta_r \neq \theta_t$  pole przekroju poprzecznego strumieni  $I_i$  oraz  $I_t$  jest różne

Wtedy moc odpowiednich wiązek będzie równa:

$$P_i = I_i A \cos \theta_i,$$

$$P_r = I_r A \cos \theta_r,$$

$$P_t = I_t A \cos \theta_t.$$

W związku z tym reflektancja  $R$  zdefiniowana jako stosunek mocy promieniowania odbitego  $P_r$  do padającego  $P_i$  będzie równa:

$$R \equiv \frac{P_r}{P_i} = \frac{I_r A \cos \theta_r}{I_i A \cos \theta_i} = \frac{I_r}{I_i} = \left( \frac{E_{0r}}{E_{0i}} \right)^2 = r^2. \quad (1.2.52)$$

W powyższym równaniu współczynniki związane z prędkością i względną przenikalnością uległy skróceniu, gdyż fala padająca i odbita poruszają się w tym samym ośrodku.

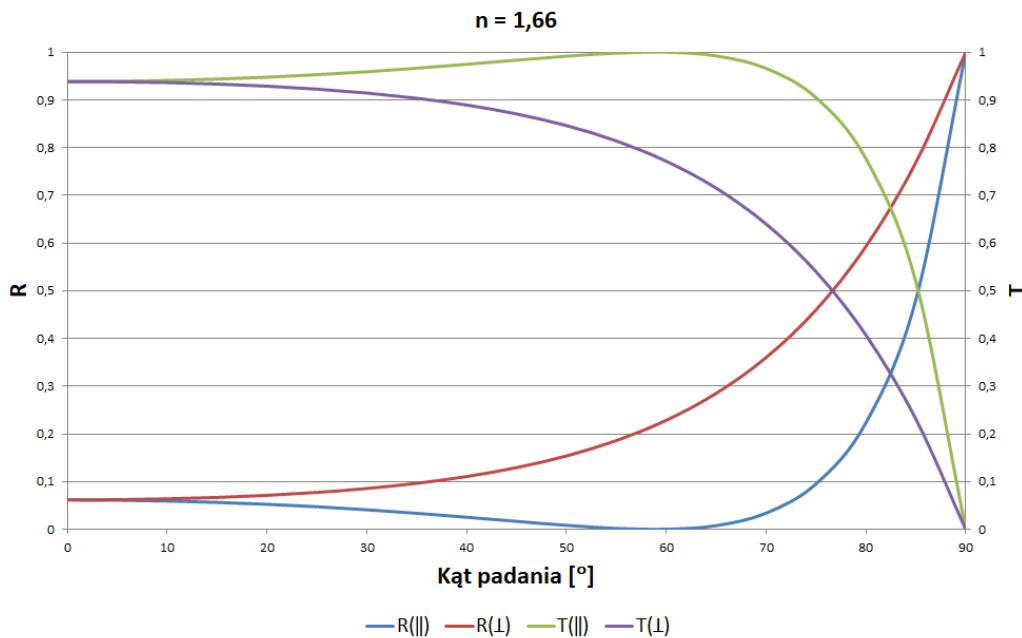
Analogicznie transmitancja  $T$ , która jest stosunkiem mocy promieniowania transmitowanego  $P_t$  do padającego  $P_i$  (zakładając materiały niemagnetyczne  $\mu = \mu_0$ ):

$$T \equiv \frac{P_t}{P_i} = \frac{I_t A \cos \theta_t}{I_i A \cos \theta_i} = \frac{\frac{v_t \epsilon_t}{2} \cos \theta_t}{\frac{v_i \epsilon_i}{2} \cos \theta_i} \left( \frac{E_{0t}}{E_{0i}} \right)^2 = \frac{\frac{1}{v_t} \cos \theta_t}{\frac{1}{v_i} \cos \theta_i} t^2 = \frac{\frac{n_t}{c} \cos \theta_t}{\frac{n_i}{c} \cos \theta_i} t^2 = \frac{n_t \cos \theta_t}{n_i \cos \theta_i} t^2. \quad (1.2.53)$$

Można pokazać, że w obu przypadkach polaryzacji pola elektrycznego  $\vec{E}$  (tj. poprzecznej  $\vec{E}_\perp$  i podłużnej  $\vec{E}_\parallel$ ) :

$$R + T = 1. \quad (1.2.54)$$

Jest to wprost wyrażenie zasady zachowania energii - w wyniku procesów rozpraszania energia przed rozproszeniem jest równa sumie energii wiązek rozproszonych.



**Rys. 1.11:** Reflektancja i transmitancja dla obu typów polaryzacji w funkcji kąta padania,  $n = 1,66 \in \mathbb{R}$ . Dla niskich kątów padania znaczącą część energii unosi fala transmitowana. Sytuacja ulega gwałtownej zmianie dopiero w okolicy kątów bliskich padaniu równoległemu do  $\Pi$  (im mniejsze  $n$ , tym zmiana zachodzi gwałtowniej)

Ogólny przypadek, w którym na ogół ośrodek wykazuje pewne tłumienie, przez co współczynnik załamania staje się zespolony (1.2.35), wymaga przepisania reflektancji i transmitancji w następujący sposób:

$$R = rr^* = |r|^2, \quad (1.2.55)$$

$$T = \frac{n_t \cos \theta_t}{n_i \cos \theta_i} tt^* = \frac{n_t \cos \theta_t}{n_i \cos \theta_i} |t|^2. \quad (1.2.56)$$

Uprośćmy teraz wyrażenia (1.2.47) oraz (1.2.49) w taki sposób, że  $\mu_i \approx \mu_t \approx \mu_0$  oraz  $n_w = \frac{n_t}{n_i} = \eta - ik$  (dla uproszczenia zapisu zależność od częstotliwości zostanie pominięta):

$$r_{\perp} = \frac{\cos \theta_i - n_w \cos \theta_t}{\cos \theta_i + n_w \cos \theta_t}, \quad (1.2.57)$$

$$r_{\parallel} = \frac{n_w \cos \theta_i - \cos \theta_t}{n_w \cos \theta_i + \cos \theta_t}. \quad (1.2.58)$$

Obliczenie reflektancji dla obu przedstawionych polaryzacji prowadzi do wyniku:

$$R_{\perp} = \frac{(\eta^2 + k^2) \cos^2 \theta_t - 2\eta \cos \theta_i \cos \theta_t + \cos^2 \theta_i}{(\eta^2 + k^2) \cos^2 \theta_i + 2\eta \cos \theta_i \cos \theta_t + \cos^2 \theta_i}, \quad (1.2.59)$$

$$R_{\parallel} = \frac{(\eta^2 + k^2) \cos^2 \theta_i - 2\eta \cos \theta_i \cos \theta_t + \cos^2 \theta_t}{(\eta^2 + k^2) \cos^2 \theta_i + 2\eta \cos \theta_i \cos \theta_t + \cos^2 \theta_t}. \quad (1.2.60)$$

W przypadku dobrego przewodnika, tzn. takiego którego przewodność elektryczna  $\sigma$  jest wysoka, prawdą jest jednak stwierdzenie, że fala przechodząca do takiego materiału będzie propagować się wzdłuż normalnej ( $\theta_t = 0$ ) do granicy dwóch ośrodków i to niezależnie od wartości kąta padania  $\theta_i$ . Stąd powyższe równania ulegną uproszczeniu (tym razem zaznaczając bezpośrednio zależność częstotliwościową):

$$R_{\perp}(\omega) = \frac{(\eta(\omega)^2 + k(\omega)^2) - 2\eta(\omega) \cos \theta_i + \cos^2 \theta_i}{(\eta(\omega)^2 + k(\omega)^2) + 2\eta(\omega) \cos \theta_i + \cos^2 \theta_i}, \quad (1.2.61)$$

$$R_{\parallel}(\omega) = \frac{(\eta(\omega)^2 + k(\omega)^2) \cos^2 \theta_i - 2\eta(\omega) \cos \theta_i + 1}{(\eta(\omega)^2 + k(\omega)^2) \cos^2 \theta_i + 2\eta(\omega) \cos \theta_i + 1}. \quad (1.2.62)$$

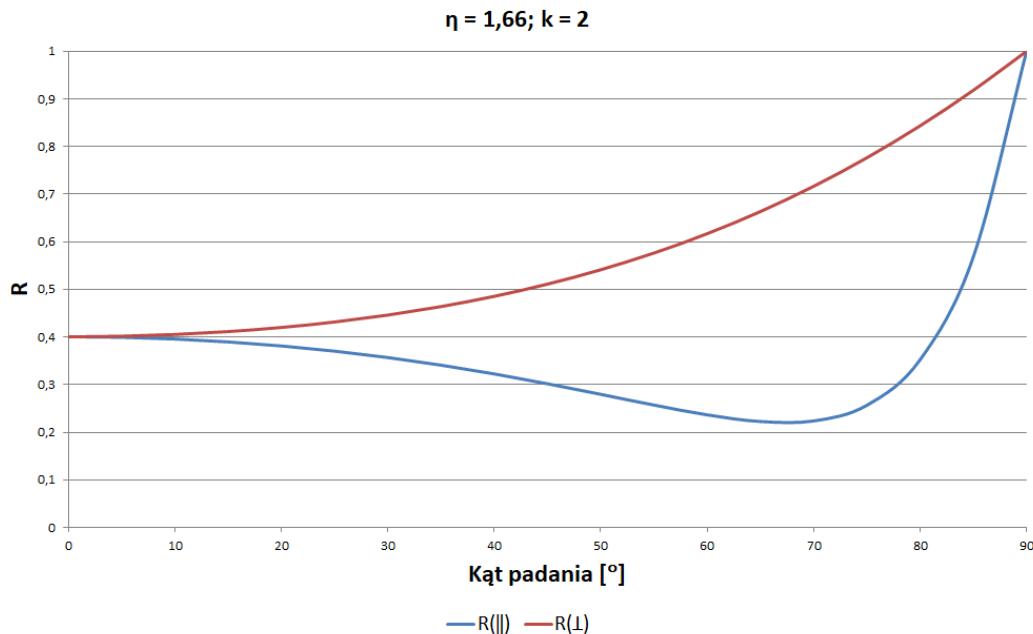
Powyższe wyprowadzone równania na reflektancję ośrodków przewodzących są zgodne z formułami podanymi bez dowodu w [14].

Na wykresie 1.12 zilustrowano zależność reflektancji dla poprzecznie i równolegle spolaryzowanej fali padającej na przewodnik dla ustalonych parametrów  $\eta = 1,66$  oraz  $k = 2$ . Reflektancja jest znacznie większa już w przypadku padania prostopadłego  $\theta_i = 0^\circ$  względem sytuacji zaprezentowanej na wykresie 1.11 - im większe  $k$ , tym materiał (przewodnik) odbija więcej promieniowania.

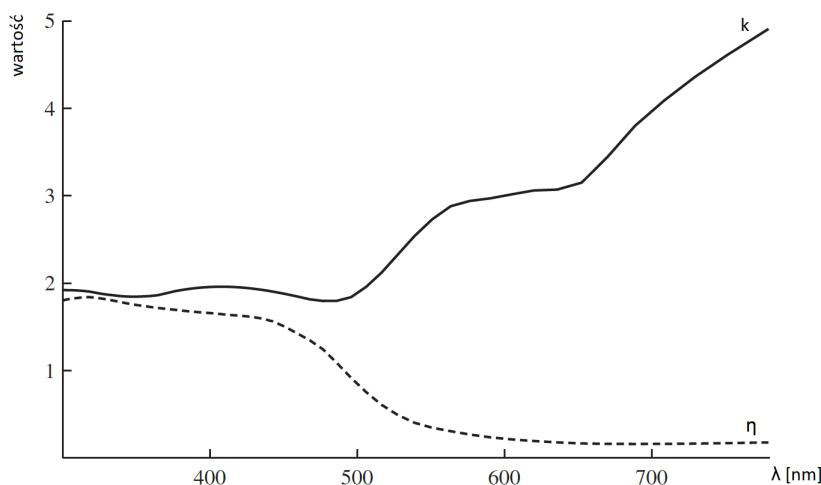
W kontekście grafiki komputerowej poprawne wyliczenie reflektancji zwłaszcza w przypadku metali nie jest prostym zadaniem. Jak pokazano we wzorze (1.2.36) oba współczynniki  $\eta$  oraz  $k$  nie są stałe dla danego materiału, tylko zależą od częstości fali padającej  $\omega$  i w przypadku większości materiałów nie jest to zależność trywialna. W wielu przypadkach można posłużyć się wartościami tablicowymi, w innych pomiary tych wartości mogą być niedostępne. Wykres 1.13 ilustruje przykładową zależność dla złota w funkcji długości fali padającej  $\lambda^{xii}$ . Jest on o tyle interesujący, że można z jego pomocą wyjaśnić, dlaczego złoto posiada swoją charakterystyczną barwę. Otóż wartość  $k$  jest niemalejąca w funkcji długości fali padającej (dla większości zakresu widzialnego) i ulega szybkiemu wzrostowi zaraz po przekroczeniu długości ok. 500 nm (w okolicach światła żółtego). Analiza wzorów (1.2.61), (1.2.62) oraz

<sup>xii</sup>Z punktu widzenia nauki znacznie bardziej wartościowy byłby wykres w funkcji częstości  $\omega$  fali padającej z uwagi na fakt, że długość fali  $\lambda$  zależy od ośrodka, w którym się ona propaguje. Cytowane źródło pokazuje jednak praktyczne (tzn. inżynierskie) podejście w kwestii tworzenia systemu generowania realistycznej grafiki - Autorom jest łatwiej w obliczeniach posługiwać się wprost długością fali.

wykresu 1.12 mówi, że dla danego kąta padania  $\theta_i$  większa wartość  $k$  skutkuje zwiększeniem reflektancji - złoto silnie selektywnie odbija światło żółte i czerwone.



**Rys. 1.12:** Reflektancja dla dobrego przewodnika dla obu typów polaryzacji w funkcji kąta padania,  $n_w = \eta + ik = 1,66 + 2i \in \mathbb{C}$ . Obecność czynnika tłumiącego sprawia, iż tylko niewielka objętość przewodnika odczuwa obecność padającej fali, dzięki czemu większa część promieniowania (w porównaniu z dielektrykami) odbija się od powierzchni nawet, gdy  $\theta_i = 0^\circ$  - jest to przyczyna tzw. połysku metalicznego



**Rys. 1.13:** Współczynniki  $\eta$  oraz  $k$  dla złota dla optycznych długości fal  $\lambda$ . Obie wartości wykazują silną zależność od długości fali  $\lambda$ . Na podstawie przebiegu ich zmienności można wyjaśnić, dlaczego metal ten posiada swój charakterystyczny kolor. Wykres zaczerpnięty z [14]

### 1.2.2 Podstawowe zagadnienia radiometrii

Radiometria zajmująca się ilościowym opisem promieniowania oraz wielkości fizycznych z nim związanych, dostarcza odpowiednich narzędzi matematycznych stanowiących podstawę do wyprowadzenia algorytmów używanych w symulacjach propagacji światła w przestrzeni. W połączeniu z zasadami wynikającymi z optyki geometrycznej oraz zależnościami wyprodukowanymi w poprzednim podrozdziale pozwala stworzyć system śledzenia promieni, zdolny do generowania fotorealistycznych obrazów.

- *Strumień natężenia/moc promieniowania*  $\Phi$  (ang. *radiant flux/power*) jest miarą emitowanej/odbitej/transmitowanej/padającej energii promieniowania  $Q$  w jednostce czasu  $t$  (mocy promieniowania) i zdefiniowana jest jako:

$$\Phi = \frac{\partial Q}{\partial t}. \quad (1.2.63)$$

- *Oświetlenie* (ang. *irradiance*), czyli powierzchniowa gęstość mocy promieniowania padającego<sup>xiii</sup>

$$E = \frac{\partial \Phi}{\partial A} = \frac{\partial^2 Q}{\partial A \partial t}, \quad (1.2.64)$$

odpowiada, jak już wspomniano wcześniej, uśrednionej w czasie wartości wektora Poingtinga  $\langle \vec{S} \rangle_T$ , wiążącego ze sobą przestrzenne zależności pola elektrycznego  $\vec{E}$  i magnetycznego  $\vec{B}$ . Wielkość ta pozwala wyjaśnić m. in. prawa odwrotności kwadratu odległości  $\frac{1}{r^2}$  oraz Lamberta:

- Jeżeli przyjmiemy, że na rysunku 1.5 a) w punkcie  $A$  znajduje się izotropowe punktowe źródło promieniowania o stałej mocy  $\phi$  to oba fronty falowe (żółty (z) i pomarańczowy (p)) będą unosić ze sobą identyczną ilość energii, jednak z uwagi na różną odległość od źródła powierzchniowa gęstość energii będzie różna:

$$\begin{aligned} \Phi_z &= \Phi_p, \\ E_z S_z &= E_p S_p, \\ 4\pi r_z^2 E_z &= 4\pi r_p^2 E_p, \\ \frac{E_z}{E_p} &= \frac{r_p^2}{r_z^2}. \end{aligned}$$

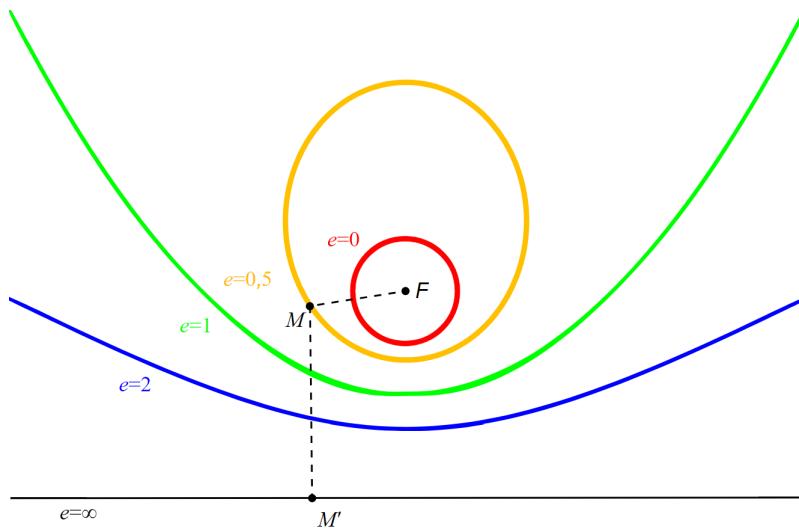
Skąd:

$$E \propto \frac{1}{r^2}. \quad (1.2.65)$$

- Kierunkowe źródło światła np. latarka ustawione w taki sposób, że promieniowanie pada prostopadle do powierzchni oświetla obszar  $A_{\perp}$  o oświetleniu  $E_1 \propto \frac{\Phi}{A_{\perp}}$ . W miarę zwiększania kąta padania  $\theta_i$  oświetlana powierzchnia zwiększa się  $A = \frac{A_{\perp}}{\cos \theta_i}$  i  $E_2 \propto \frac{\Phi}{A} = \frac{\Phi \cos \theta_i}{A_{\perp}}$ . Na tej podstawie prawo Lamberta stanowi, że:

$$E \propto \cos \theta_i. \quad (1.2.66)$$

<sup>xiii</sup>Różnica w oznaczeniu oświetlenia  $E$ , względem dotychczasowego  $I$  bierze się najprawdopodobniej z faktu, iż historycznie radiometria rozwijana była w oderwaniu od ścisłej teorii elektromagnetyzmu. W radiometrii symbolem  $I$  oznaczana jest natomiast *intensywność* (ang. *intensity*) definiowana jako miara strumienia natężenia przypadająca na jednostkowy kąt bryłowy  $I = \frac{\partial \Phi}{\partial \omega}$  [14].



**Rys. 1.14:** Zależność pola i kształtu oświetlanej powierzchni od kąta padania. Światło latarki padające na płaszczyznę tworzy na niej obszary, których brzegi są krzywymi stożkowymi. Dla padania prostopadłego  $\theta_i = 0^\circ$  brzeg ten jest okręgiem, a powierzchnia nim objęta najmniejsza. Parametr  $e$  odpowiada ekscentryczności krzywej stożkowej [15]

- *Radiancja* (ang. *radiance*), która jest najważniejszą wielkością radiometryczną z punktu widzenia techniki śledzenia promieni jest zdefiniowana następująco:

$$L = \frac{\partial^2 \Phi}{\partial A^\perp \partial \omega} = \frac{\partial^2 \Phi}{\cos \theta_i \partial A \partial \omega}, \quad (1.2.67)$$

gdzie  $\partial A^\perp$  jest rzutem  $\partial A$  na powierzchnię prostopadłą do  $\vec{\omega}$ . Wielkość ta wyraża ilość światła przypadającego na różniczkowy kąt brylowy  $\partial\omega$  i różniczkowy element powierzchni  $\partial A$ . Inaczej mówiąc jest to ilość światła rozchodzącej się w wybranym kierunku wyznaczonym przez  $\vec{\omega}$ . Jeśli ośrodek jest nietłumiący, to radiancja pozostaje stała wzduż promieni.

W danym punkcie  $\vec{p}$  na powierzchni obiektu istnieje zawsze pewien rozkład radiancji zależny od położenia oraz kierunku  $L(\vec{p}, \vec{\omega})$ . Rozkłady te definiuje się osobno dla *radiancji padającej*  $L_i(\vec{p}, \vec{\omega}_i)$  (ang. *incident radiance*) oraz *radiancji wychodzącej*  $L_o(\vec{p}, \vec{\omega}_o)$  (ang. *exitant radiance*) przy czym w ogólności:

$$L_i(\vec{p}, \vec{\omega}) \neq L_o(\vec{p}, \vec{\omega}). \quad (1.2.68)$$

Różniczkowe oświetlenie wynikające z radiancji padającej na  $\vec{p}$  z definicji (1.2.67):

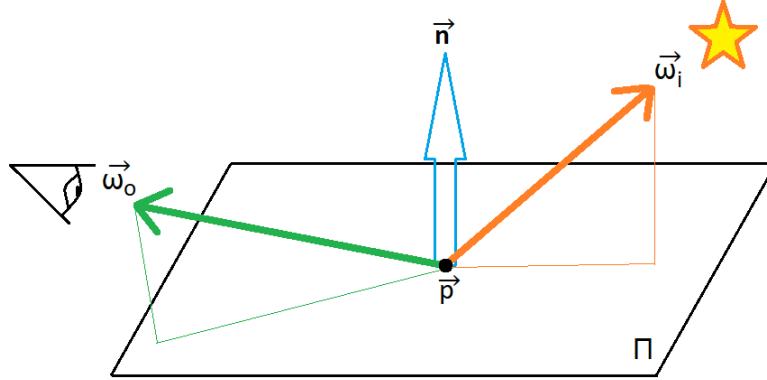
$$dE(\vec{p}, \vec{\omega}_i) = L_i(\vec{p}, \vec{\omega}_i) \cos \theta_i d\vec{\omega}_i. \quad (1.2.69)$$

Z drugiej strony wiadomo, że wskutek tego oświetlenia pewna część energii zostanie rozproszona w kierunku  $\vec{\omega}_o$ . Założenie, że mamy do czynienia z ośrodkami o liniowej odpowiedzi prowadzi do wniosku, że różniczkowa radiancja wychodząca jest proporcjonalna do różniczkowego oświetlenia:

$$dL_o(\vec{p}, \vec{\omega}_o) \propto dE(\vec{p}, \vec{\omega}_i). \quad (1.2.70)$$

Funkcja  $f_s(\vec{p}, \vec{\omega}_o, \vec{\omega}_i)$  będąca czynnikiem proporcjonalności między  $dL_o(\vec{p}, \vec{\omega}_o)$  a  $dE(\vec{p}, \vec{\omega}_i)$  jest tzw. dwukierunkową funkcją rozkładu rozproszenia (ang. *bidirectional scattering distribution function*, BSDF) i opisuje to, jak dużo padającego światła z kierunku  $\vec{\omega}_i$  jest rozpraszańe przez powierzchnię w punkcie  $\vec{p}$  w kierunku  $\vec{\omega}_o$ :

$$f_s(\vec{p}, \vec{\omega}_o, \vec{\omega}_i) = \frac{dL_o(\vec{p}, \vec{\omega}_o)}{dE(\vec{p}, \vec{\omega}_i)} = \frac{dL_o(\vec{p}, \vec{\omega}_o)}{L_i(\vec{p}, \vec{\omega}_i) \cos \theta_i d\vec{\omega}_i}. \quad (1.2.71)$$



**Rys. 1.15:** Funkcja dwukierunkowego rozkładu rozproszenia BSDF. Każdy kierunek  $\vec{\omega}$  daje się opisać poprzez parę kątów  $[\phi, \theta] \in [0, 2\pi] \times [0, \pi]$  ( $\phi$  określone jest przez rzut  $\vec{\omega}$  na  $\Pi$ , zaś  $\theta$  poprzez rzut  $\vec{\omega}$  na normalną  $\vec{n}$  w punkcie  $\vec{p}$ ), co sprawia, że BSDF jest czterowymiarową funkcją par kierunków  $(\vec{\omega}_o, \vec{\omega}_i)$  opisującą, jaka część promieniowania padającego wzdłuż  $\vec{\omega}_i$  jest rozpraszańe w kierunku  $\vec{\omega}_o$ . Ilustracja wykonana na podstawie [14]

Aby BSDF uznać za realistyczne tzn. zgodne z podstawowymi zjawiskami fizycznymi musi ona spełniać kilka warunków:

1. Jest nieujemna dla dowolnej kombinacji parametrów.
2. Nie może wprowadzać dodatkowej energii do układu tzn.:

$$\forall_{\vec{p}, \vec{\omega}_o} : \int_{S^2} f_s(\vec{p}, \vec{\omega}_o, \vec{\omega}') \cos \theta' d\vec{\omega}' \leq 1, \quad (1.2.72)$$

gdzie całkowanie odbywa się po sferze wokół punktu  $\vec{p}$ .

3. Jeżeli BSDF opisuje procesy odbicia (ang. *bidirectional reflectance distribution function*, BRDF) musi ona spełniać relację odwrotności biegu promieni (tzw. *zasada Helmholtza*):

$$f_s(\vec{p}, \vec{\omega}_o, \vec{\omega}_i) = f_s(\vec{p}, \vec{\omega}_i, \vec{\omega}_o). \quad (1.2.73)$$

Równanie opisujące wpływ rozkładu radiancji padającej na rozkład po rozproszeniu jest kluczem do generowania realistycznej grafiki komputerowej i nazywane jest *relacją rozpraszania* (ang. *scattering equation*):

$$L_o(\vec{p}, \vec{\omega}_o) = \int_{S^2} f_s(\vec{p}, \vec{\omega}_o, \vec{\omega}_i) L_i(\vec{p}, \vec{\omega}_i) \cos \theta_i d\vec{\omega}_i. \quad (1.2.74)$$

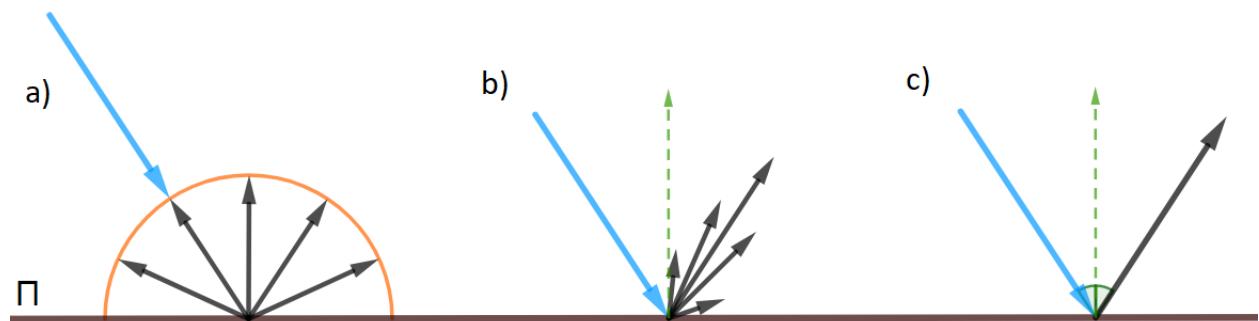
### 1.2.3 Modelowanie materiałów

Wprowadzona w poprzednim podrozdziale funkcja BSDF (1.2.71) jest tylko ogólnym wyrażeniem relacji między radiancją padającą a rozproszoną. Najogólniejsze podejście wymagałoby analizowania interakcji światła z atomami, na które ono pada. Podejście to jest na tyle nieefektywne, że przez ostatnie dekady powstały dziesiątki propozycji modeli opartych na różnych założeniach, a przez to dostosowanych do pewnego podzbioru występujących w rzeczywistości powierzchni.

Znakomita część modeli operuje zbiorem kilku intuicyjnych parametrów, które mogą odwoływać się np. do szorstkości, siły rozbłysku czy anizotropii występującej w materiale. Odpowiedni dobór tego typu modelu oraz jego parametrów w wielu przypadkach pozwala uzyskiwać przekonywujące rezultaty i to dlatego są one najczęściej i najczęściej wykorzystywane (są też atrakcyjne pod względem złożoności obliczeń). Po drugiej stronie znajdują się złożone modele teoretyczne i eksperymentalne BSDF, których celem jest perfekcyjne oddanie tego, w jaki sposób dany materiał się zachowuje. W przypadku modeli teoretycznych wiąże się to z koniecznością operowania wieloma parametrami, których znaczenie i wpływ na wygląd powierzchni może być niejasne. Eksperymentalne BSDF zaś wymagają pomiaru własności powierzchni z użyciem odpowiedniej aparatury - jest to czasochłonne, a uzyskane BSDF nadają się tylko do wykorzystania z konkretnym materiałem [16].

Wspomniano już wcześniej, że jednym z przypadków BSDF są BRDF, których zadaniem jest symulacja zjawisk odbicia. Odbicie w rzeczywistych materiałach jest wypadkową różnych mechanizmów (rysunek 1.16), których efekty można opisać jako:

- *odbicie rozproszone* (ang. *diffuse*), które jest najlepiej dostrzegalne w przypadku powierzchni matowych i w najprostszym modelu Lamberta zakłada, że padająca radiancja jest odbijana jednakowo we wszystkich kierunkach,
- *kierunkowe odbicie rozproszone* (ang. *glossy specular*) występuje, gdy powierzchnia materiału w skali mikro jest niemal idealnie gładka i odbicie zaczyna nabierać charakteru kierunkowego w kierunku bazowym, zadanym przez prawo odbicia modyfikowanym przez pewien rozkład kierunków. Szczególnym przypadkiem tego typu zachowania jest *odbicie zwierciadlane* (ang. *perfect specular*), gdy powierzchnia jest idealnie gładka, a przez to kierunek bazowy jest jedynym kierunkiem, w którym odbicie następuje.



Rys. 1.16: Rodzaje odbicia od powierzchni. Odbicie rozproszone a), kierunkowe rozproszone b) i jego szczególny przypadek odbicie zwierciadlane c)

Mając powyższe na uwadze, w grafice komputerowej stosuje się zwykle wypadkową tych zjawisk używając sumy przyczynków do radiancji wychodzącej pochodzących od odbicia rozproszonego i kierunkowego:

$$L_o(\vec{p}, \vec{\omega}_o) = L_{o_d}(\vec{p}, \vec{\omega}_o) + L_{o_s}(\vec{p}, \vec{\omega}_o). \quad (1.2.75)$$

Istnieje jeszcze dodatkowy (stały dla danego materiału) składnik radiancji wychodzącej  $L_{o_a}(\vec{p}, \vec{\omega}_o) = L_{o_a}(\vec{p})$ , a pochodzący od hipotetycznego bezkierunkowego źródła światła (ang. *ambient light*), którego celem jest zgrubne (zerowe) przybliżenie faktu, że światło, które nie oświetla bezpośrednio danego obiektu ulega wielokrotnym odbiciom w przestrzeni zanim dotrze do obserwatora:

$$L_o(\vec{p}, \vec{\omega}_o) = L_{o_a}(\vec{p}) + L_{o_d}(\vec{p}, \vec{\omega}_o) + L_{o_s}(\vec{p}, \vec{\omega}_o), \quad (1.2.76)$$

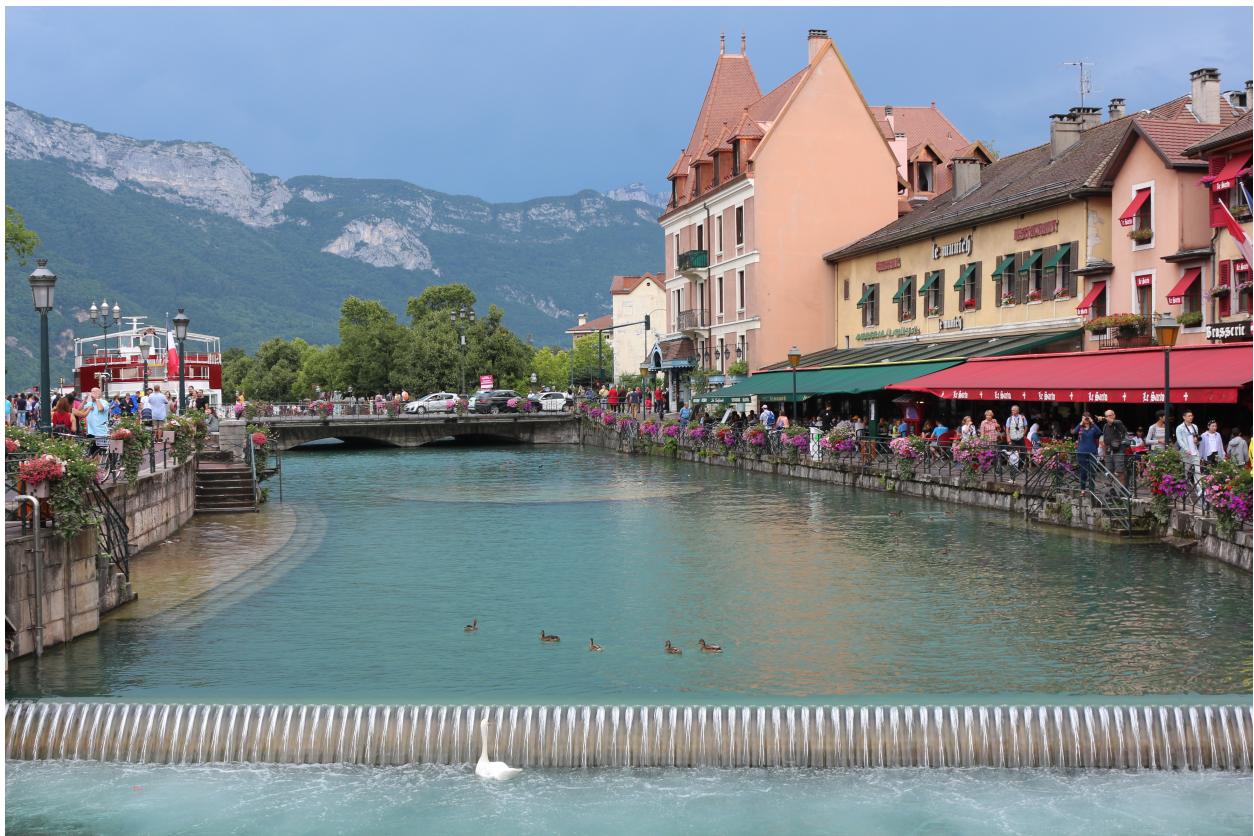
$$L_{o_a}(\vec{p}) = k_a(\vec{p})L_a, \quad (1.2.77)$$

gdzie:

$L_a$  - stała wartość radiancji wynikająca z globalnego oświetlenia,

$k_a(\vec{p}) \in [0, 1]$  - współczynnik dla danego materiału określający stopień jego interakcji ze światłem otoczenia.

Tak naprawdę jest to wyraz sztuczny, podyktowany wygodą oraz skróceniem czasu tworzenia grafiki. Śledzenie promieni, które u swoich podstaw zakłada rozważanie wielu ścieżek, po których światło może się poruszać automatycznie doświetla obiekty, które nie muszą leżeć w bezpośrednim zasięgu oddziaływania źródeł światła tworząc znacznie bardziej realistyczne rezultaty.



**Rys. 1.17:** Przykład rozproszonego odbicia kierunkowego. Powierzchnia rzeki pod wpływem ruchu i wiatru staje się nierówna, w efekcie czego zaczyna ona odbijać otoczenie w taki sposób, że daje się wyróżnić pewien kierunek główny, wokół którego istnieje pewien rozkład wektorów odbicia. Im ten rozkład jest bardziej skupiony wokół jednego kierunku, tym lepiej dostrzegalne są szczegóły odbitego świata. Fotografia ze źródeł własnych (Annecy, Francja)

### Model Lamberta światła rozproszonego

Podstawowy i najprostszy model oświetlenia powierzchni (pomijając interakcję z virtualnym światłem otoczenia) stanowiący bazę dla innych bardziej złożonych algorytmów. Bazuje on na wspomnianym wcześniej prawie kosinusów Lamberta (1.2.66) i sprawdza się dla powierzchni rozpraszających światło idealnie we wszystkich kierunkach. Dla takich materiałów odbita radiancja nie jest funkcją  $\vec{\omega}_o$  tzn.  $L_o(\vec{p}, \vec{\omega}_o) = L_o(\vec{p})$ . Mając na uwadze wprowadzone ogólne równanie rozpraszania (1.2.74) jest to możliwe tylko wtedy, gdy sama funkcja BSDF nie zależy od  $\vec{\omega}_o$  ani  $\vec{\omega}_i$  (z uwagi na zasadę odwrotności w przypadku odbicia) zatem:

$$L_o(\vec{p}) = f_s(\vec{p}) \int_{S^2} L_i(\vec{p}, \vec{\omega}_i) \cos \theta_i d\vec{\omega}_i = f_s(\vec{p}) E_i(\vec{p}). \quad (1.2.78)$$

Różniczkowy wyjściowy strumień natężenia musi być zatem równy:

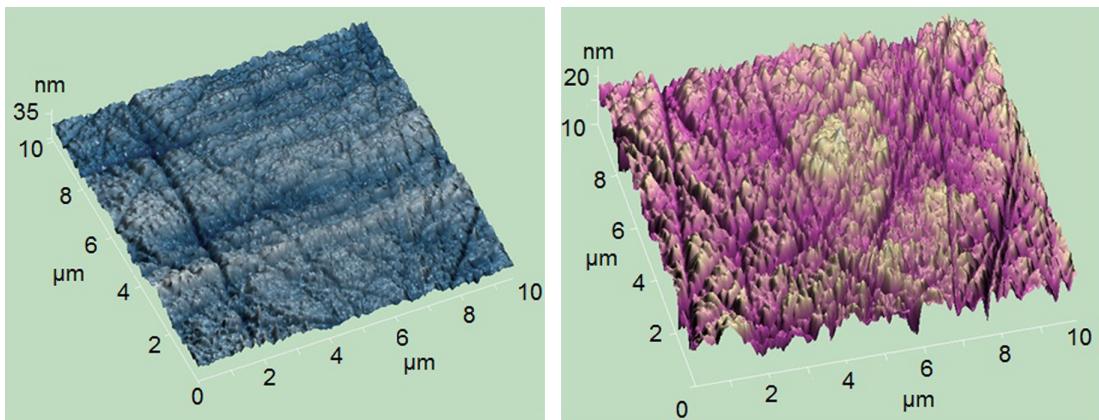
$$\begin{aligned}
 d\Phi_o &= dA \cdot \int_{S^2} L_o(\vec{p}) \cos \theta_o d\vec{\omega}_o \\
 &= dA \cdot L_o(\vec{p}) \int_{H^2} \cos \theta_o d\vec{\omega}_o \\
 &= dA \cdot L_o(\vec{p}) \int_0^{2\pi} d\phi \int_0^{\frac{\pi}{2}} d\theta_o \cos \theta_o \sin \theta_o \\
 &= dA \cdot \pi L_o(\vec{p}) \int_0^{\frac{\pi}{2}} d\theta_o \sin 2\theta_o \\
 &= dA \cdot \pi L_o(\vec{p}) \\
 &= dA \cdot \pi f_s(\vec{p}) E_i(\vec{p}) \\
 &= d\Phi_i \cdot \pi f_s(\vec{p}). \tag{1.2.79}
 \end{aligned}$$

Całkowanie po sferze wokół punktu  $\vec{p}$  zostało zredukowane do całkowania po półsferze wyznaczonej przez kierunek normalny w  $\vec{p}$ . Niech  $k_d(\vec{p}) = \frac{d\Phi_o}{d\Phi_i}$  będzie stopniem odbicia, wtedy poszukiwana funkcja BSDF (BRDF) będzie następująca:

$$f_s(\vec{p}) = \frac{k_d(\vec{p})}{\pi}. \tag{1.2.80}$$

### Model Orena-Nayara światła rozproszonego

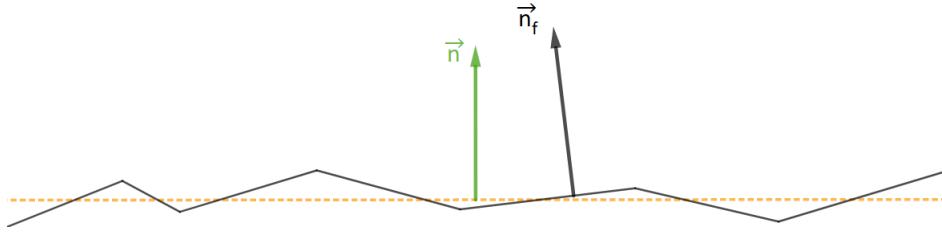
Zdecydowana większość rzeczywistych powierzchni, mimo że wydaje się być gładka w rzeczywistości nie jest. Badanie powierzchni na przykład przy pomocy mikroskopii sił atomowych (ang. *Atomic Force Microscopy*, AFM) pokazuje, że materiał w rzeczywistości posiada pewne mikronierówności - te zaś wpływają na to jak dany materiał jako całość rozprasza światło.



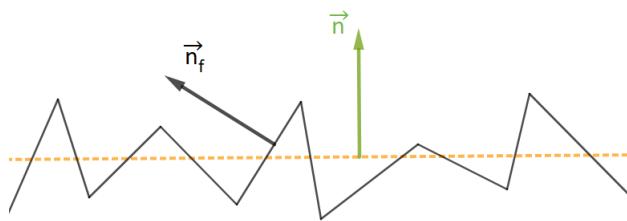
**Rys. 1.18:** Topologia rzeczywistej powierzchni uzyskana techniką mikroskopii sił atomowych (AFM) [17]. W znacznej części przypadków makroskopowo płaskie obiekty posiadają skomplikowane topologie, które między innymi wpływają na makroskopowy sposób oddziaływanego ze światłem

Dość powszechnym podejściem w symulacjach jest przybliżenie powierzchni materiału poprzez tzw. *mikrościanki* (ang. *microfacet*). Gdy oświetlona zostaje różniczkowa powierzchnia  $dA$  można przyjąć, że określona liczba mikrościanek bierze udział w procesie rozpraszania.

Wynik rozpraszania zależy będzie od wkładu poszczególnych mikrościanek w ten proces oraz funkcji określającej ich rozkład przestrzenny.



**Rys. 1.19:** Przybliżenie powierzchni w modelu mikrościanek dla niskiej szorstkości. Wektory normalne do poszczególnych mikrościanek  $\vec{n}_f$  niewiele się różnią od wypadkowego wektora normalnego  $\vec{n}$



**Rys. 1.20:** Przybliżenie powierzchni w modelu mikrościanek dla wysokiej szorstkości. Widoczna jest znaczna wariancja kierunku wektora normalnego  $\vec{n}_f$  względem wypadkowego  $\vec{n}$

Jednym z takich modeli, który bierze pod uwagę fizyczne właściwości powierzchni i modeluje światło rozproszone jest model wprowadzony i potwierdzony eksperymentalnie przez Orena i Nayara [18][19]. Twórcy przybliżyli powierzchnię poprzez zbiór V-kształtnych i symetrycznych rowków, które indywidualnie zachowują się tak, jak to wynika z modelu Lambertego, o rozkładzie orientacji opisanym poprzez rozkład Gaussa. Stopień szorstkości kontrolowany jest parametrem  $\sigma^2$  będący wariancją rozkładu. Pełen model charakteryzuje się dość wysoką złożonością obliczeniową, jednak jego twórcy na podstawie analizy wpływu poszczególnych składowych na wynik końcowy zaproponowali uproszczoną i stosowaną w praktyce formę:

$$f_s(\vec{p}, \vec{\omega}_i(\phi_i, \theta_i), \vec{\omega}_o(\phi_o, \theta_o)) = \frac{k_d(\vec{p})}{\pi} (A + B \max(0, \cos(\phi_i - \phi_o)) \sin \alpha \tan \beta), \quad (1.2.81)$$

$$A = 1 - \frac{\sigma^2}{2(\sigma^2 + 0,33)}, \quad (1.2.82)$$

$$B = \frac{0,45\sigma^2}{\sigma^2 + 0,09}, \quad (1.2.83)$$

$$\alpha = \max(\theta_i, \theta_o), \quad (1.2.84)$$

$$\beta = \min(\theta_i, \theta_o). \quad (1.2.85)$$

Mimo, że model Orena-Nayara jest klasyfikowany jako model fizyczny, jest on kontrolowany pojedynczym parametrem  $\sigma^2$ , co jest jego dużą zaletą. Co więcej, gdy  $\sigma^2 = 0$  (czyli powierzchnia jest idealnie gładka) funkcja BSDF przechodzi naturalnie w model Lambertego ( $A = 1$ ,  $B = 0$ ).

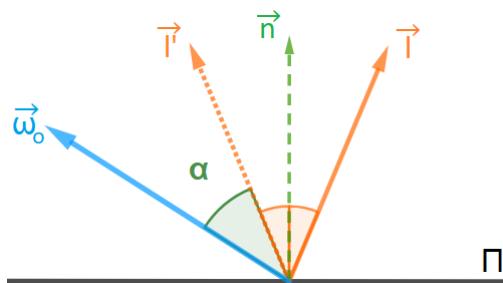
W modelu tym należy mieć na uwadze, iż kąty  $\phi_i$ ,  $\theta_i$ ,  $\phi_o$  oraz  $\theta_o$ , które podają wzajemną relację przestrzenną między  $\vec{\omega}_i$  a  $\vec{\omega}_o$  muszą być określone względem płaszczyzny  $\Pi$  definiowanej poprzez wektor normalny  $\vec{n}$  w punkcie zderzenia  $\vec{p}$ . Konieczne jest zatem stworzenie ortonormalnej bazy w  $\vec{p}$ , gdzie jednym z wersorów jest  $\vec{n}$  i dokonanie rzutowania  $\vec{\omega}_i$  i  $\vec{\omega}_o$  na wektory tej bazy w celu określenia odpowiednich kątów.

### Model Phonga oraz Blinna-Phonga dla odbić kierunkowych

Powierzchnie niematowe charakteryzują się pewnym stopniem połysku zależnym od stopnia jej szorstkości. Tak na przykład o powierzchni kontaktowej większości radiatorów można powiedzieć, że rozpraszają światło niejednakowo w różnych kierunkach i zależnym od pewnego kąta  $\alpha$ . Jednak, gdy zostaną one poddane polerowaniu, mikronierówności powierzchni zaczynają być niwelowane (szorstkość się zmniejszy) aż do uzyskania powierzchni, która zachowuje się niemal jak lustro  $\theta_r \simeq \theta_i$ . Phong Bui-Thong [20] opracował model dla takich zjawisk oparty na założeniu, że odbicie kierunkowe jest maksymalne, gdy  $\alpha = 0^\circ$  i jest szybko gasnącą funkcją tego kąta. Sam kąt mierzony jest między  $\vec{\omega}_o$  a odbiciem zwierciadlanym kierunku w stronę źródła światła  $\vec{l}$  względem normalnej w danym punkcie  $\vec{n}$  (rysunek 1.21), a za funkcję modelującą zasięg rozbłysku przyjął zależność  $\cos^e \alpha$  (rysunek 1.22):

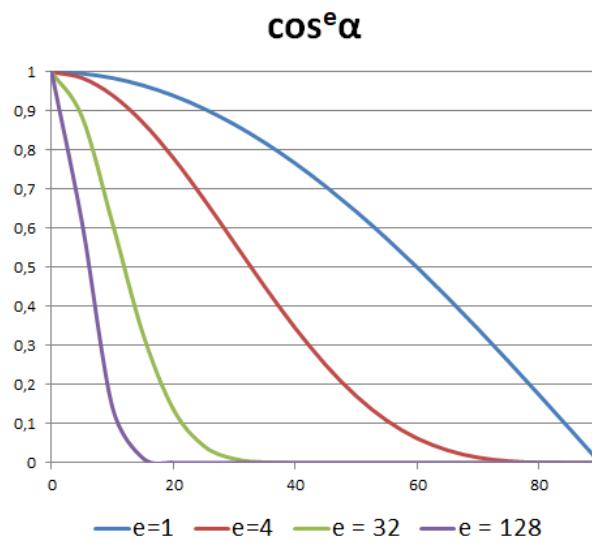
$$f_s(\vec{p}, \vec{l}, \vec{\omega}_o; e) = k_s(\vec{p}) \cos^e (\angle(\vec{l}, \vec{\omega}_o)), \quad (1.2.86)$$

gdzie  $k_s(\vec{p}) \in [0, 1]$  stanowi współczynnik odbicia w danym punkcie  $\vec{p}$ .



**Rys. 1.21:** Modelowanie odbić w modelu Phonga. Kąt  $\alpha$  definiowany jest jako kąt między  $\vec{\omega}_o$  a odbiciem względem normalnej  $\vec{n}$  do powierzchni kierunku do źródła światła  $\vec{l}$

Parametr  $e$  występujący we wzorze (1.2.86) decyduje o skupieniu odbitej wiązki wokół  $\vec{l}$  i dla powierzchni lustrzanej  $e \rightarrow \infty$ .

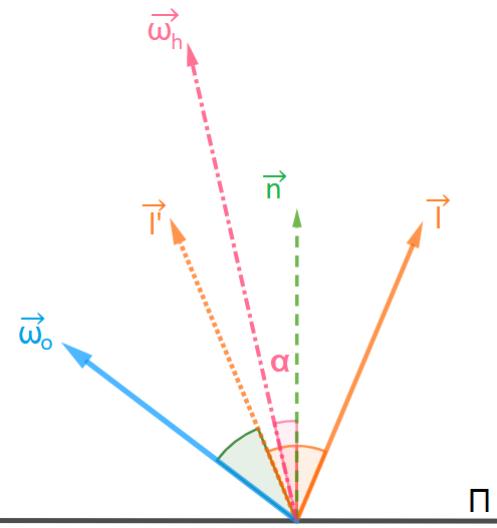


Rys. 1.22: Zależność funkcji rozbłysku od parametru skupienia  $e$ . Im większa wartość  $e$  tym rozkład staje się silniej skupiony wokół  $\alpha = 0^\circ$

Pewną modyfikacją oryginalnego modelu Phonga jest model Blinna-Phonga, który został wprowadzony w celu uniknięcia obliczania bezpośrednio odbitego wektora  $\vec{l}'$  [21] i zaproponował użycie wektora połówkowego (rysunek 1.23):

$$\vec{\omega}_h = \frac{\vec{l} + \vec{\omega}_o}{|\vec{l} + \vec{\omega}_o|}. \quad (1.2.87)$$

Wektor ten jest następnie używany w celu określenia kąta  $\alpha = \angle(\vec{n}, \vec{\omega}_h)$  i użyty w ten sam sposób jak w oryginalnym modelu Phonga (1.2.86).



Rys. 1.23: Modelowanie odbić w modelu Blinna-Phonga. Kąt  $\alpha$  definiowany jest jako kąt między  $\vec{\omega}_h = \frac{\vec{l} + \vec{\omega}_o}{|\vec{l} + \vec{\omega}_o|}$  a normalną  $\vec{n}$  do powierzchni

Oba te modele mają tę wadę, że są modelami empirycznymi tzn. nie biorą pod uwagę rzeczywistych zjawisk zachodzących w materiale. Co więcej ani model Phonga ani Blinna-Phonga nie spełnia podstawowych założeń związanych z funkcją BRDF [22] tzn. łamana jest

zasada odwracalności oraz zachowania energii. Aby temu zaradzić należy dokonać normalizacji:

$$\begin{aligned}
 1 &= c \int_{S^2} f_s(\cos \alpha; e) \cos \alpha d\vec{\omega} \\
 &= c \int_0^{2\pi} d\alpha \int_0^{\frac{\pi}{2}} d\alpha \cos^{e+1} \alpha \sin \alpha \\
 &= 2\pi c \int_0^1 u^{e+1} du \\
 &= \frac{2\pi c}{e+2}.
 \end{aligned} \tag{1.2.88}$$

Stąd uwzględniając stałą normalizacyjną  $c$  właściwa postać BSDF dla modelu Phonga i Blinna-Phonga ma postać:

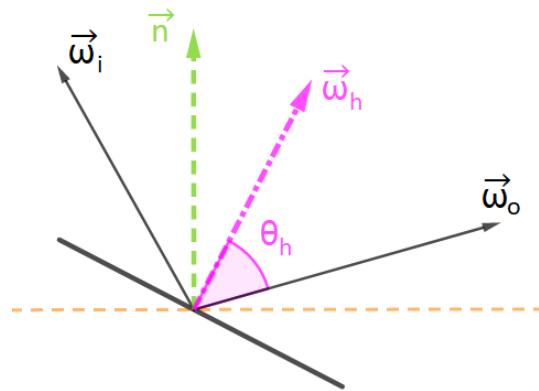
$$f_s(\vec{p}, \alpha; e) = k_s(\vec{p}) \frac{e+2}{2\pi} \cos^e \alpha. \tag{1.2.89}$$

Okazuje się jednak, że sama normalizacja nie jest wystarczająca [23]. Skoro odbicie w rzeczywistych materiałach jest wypadkową różnych mechanizmów odbicia, to aby zasada zachowania energii była spełniona, należy nałożyć ograniczenie na sumę współczynników:

$$k_d(\vec{p}) + k_s(\vec{p}) \leq 1. \tag{1.2.90}$$

### Model Torrance'a-Sparrowa dla odbić kierunkowych

Tak samo jak w przypadku światła rozproszonego poprawne modelowanie odbicia kierunkowego wymaga rozważenia powierzchni jako zbioru pewnej liczby mikrościanek o orientacji zależnej od funkcji rozkładu. Model, który przybliża powierzchnię poprzez zbiór odbijających lustrzanie mikrościanek znany jest jako model Torrance'a-Sparrowa i uznawany jest za najbardziej kompleksowy model wśród tych, które pomijają własności anizotropowe powierzchni [24][14]. Model ten zakłada, że wyłącznie mikrościanki o orientacji zgodnej z wektorem połówkowym  $\vec{\omega}_h = \frac{\vec{\omega}_o + \vec{\omega}_i}{|\vec{\omega}_o + \vec{\omega}_i|}$  uczestniczą w odbiciu zwierciadlanym.



**Rys. 1.24:** Geometria odbicia w modelu Torrance'a-Sparrowa. Podobnie jak w przypadku modelu Blinna-Phonga duże znaczenie posiada wektor połówkowy  $\vec{\omega}_h$ , który jest wektorem normalnym do mikrościanki

Jego ogólna funkcja BSDF (BRDF) jest zapisywana jako:

$$f_s(\vec{p}, \vec{\omega}_i(\phi_i, \theta_i), \vec{\omega}_o(\phi_o, \theta_o)) = \frac{D(\vec{\omega}_h)G(\vec{\omega}_o, \vec{\omega}_i)F_r(\vec{\omega}_o)}{4 \cos \theta_o \cos \theta_i}. \quad (1.2.91)$$

Składa się na nią:

$D(\vec{\omega}_h)$  - czynnik rozkładu orientacji mikrościanek, który w różnych publikacjach można być opisany poprzez różne funkcje rozkładu (np. Gaussa). W tym konkretnym przypadku posługiwać będziemy się rozkładem Blinna znanim już z dyskusji o modelu Phonga (1.2.89):

$$D(\vec{\omega}_h) = c(\vec{\omega}_h \cdot \vec{n})^e = \frac{e+2}{2\pi}(\vec{\omega}_h \cdot \vec{n})^e. \quad (1.2.92)$$

$G(\vec{\omega}_o, \vec{\omega}_i)$  - czynnik geometryczny, który wyraża, jaka część światła dla danej pary kierunków ( $\vec{\omega}_o, \vec{\omega}_i$ ) nie zostaje osłonięta w wyniku relacji między wzajemnym położeniem mikrościanek. Efekt ten obliczany jest używając wyrażenia:

$$G(\vec{\omega}_o, \vec{\omega}_i) = \min \left\{ 1, \frac{2(\vec{n} \cdot \vec{\omega}_h)(\vec{n} \cdot \vec{\omega}_o)}{\vec{\omega}_o \cdot \vec{\omega}_h}, \frac{2(\vec{n} \cdot \vec{\omega}_h)(\vec{n} \cdot \vec{\omega}_i)}{\vec{\omega}_o \cdot \vec{\omega}_h} \right\}. \quad (1.2.93)$$

$F_r(\vec{\omega}_o)$  - współczynnik odbicia obliczony zgodnie z wyprowadzonymi wzorami na współczynniki Fresnela (1.2.59)-(1.2.62) (kąt padania  $\theta_h$  mierzony jest między  $\vec{\omega}_o$  a  $\vec{\omega}_h$ ).

## Podsumowanie

Celem tego rozdziału było pokazanie podstawowych praw fizyki rządzących transporatem światła pomiędzy obiektami. Szczególnie istotne były wszystkie kroki potrzebne do wyprowadzenia wzorów Fresnela na reflektancję i transmitancję wraz z relacją rozpraszania, która wiąże ze sobą radiancję wychodzącą z radiancją padającą poprzez funkcję BSDF. Zaprezentowane zostały również przykładowe funkcje BSDF dla modelowania odbić od powierzchni (czyli funkcje BRDF). Temat funkcji BSDF jest znacznie bardziej obszerny niż zaprezentowano tutaj i więcej informacji na ten temat można znaleźć w cytowanej bibliografii. Niemniej jednak opisane zagadnienia korespondują z tym, co udało się zrealizować w części projektowej.



## Rozdział 2

---

### Układy programowalne

---

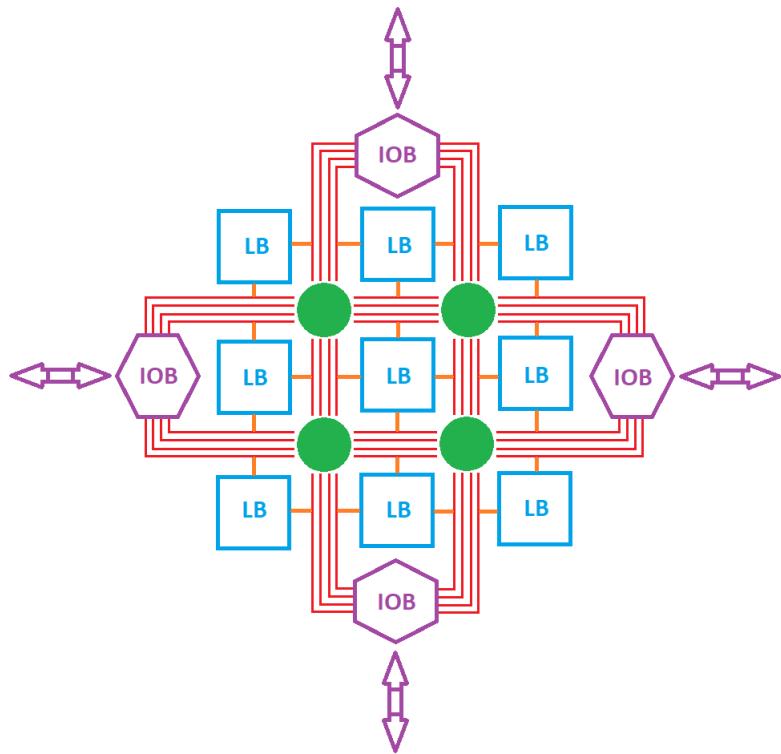
Kiedy w 1985 roku, dzięki szybko rozwijającej się litografii, wprowadzono pierwszy komercyjnie dostępny układ typu FPGA (ang. *Field Programmable Gate Array*) - układ XC2064 - bardzo szybko doceniono jego zalety wszędzie tam, gdzie wymagana jest łatwa zmiana funkcjonalności systemu przetwarzania danych. Głównym motorem rozwoju, szczególnie na początku, było powstanie i gwałtowna ekspansja Internetu i konieczność łatwego prototypowania oraz wdrażania nowych rozwiązań zwłaszcza jeśli chodzi o tworzenie przełączników i ruterów [25].

Tak duże zainteresowanie tymi układami bierze się z faktu, iż są one w łatwy sposób rekonfigurowalne, a przez to mogą realizować dowolne funkcje logiczne. W przeciwieństwie do ogólnodostępnych układów typu CPU (ang. *Central Processing Unit*), których działanie opiera się na sekwencyjnym przetwarzaniu listy rozkazów w obrębie przewidzianej przez producenta architektury instrukcji (ang. *Instruction Set Architecture*, ISA) [26] układy FPGA są konfigurowalne na poziomie sprzętowym a nie programowane. Zakupiony od producenta procesor po podaniu mu danych oraz listy instrukcji (programu) będzie je przetwarzał i na wyjściu otrzyma się oczekiwana wartość. Układ FPGA nie zrealizuje żadnej operacji mimo podania konkretnych danych wejściowych, jeśli nie zostanie skonfigurowany<sup>i</sup>.

Na konfigurowalność układów FPGA składają się wspólnie dwa czynniki - obecność bloków logicznych (ang. *logic blocks*) oraz elastycznej (tzn. konfigurowalnej) sieci połączeń między nimi (ang. *routing*) zdolnej teoretycznie wytworzyć połączenie pomiędzy dowolnymi blokami.

---

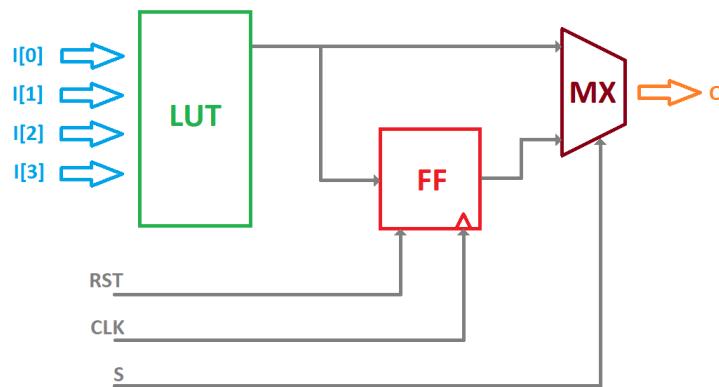
<sup>i</sup>Rozróżnienie pomiędzy konfiguracją a programowaniem powinno wybrzmieć dostatecznie mocno. Na dalszych kartach tej pracy terminy te będą używane zamiennie w kontekście układów FPGA, co jest podiktowane technologią użytą do wykonania omawianego projektu.



**Rys. 2.1:** Schemat ideowy budowy układu FPGA. Bloki logiczne (LB) połączone są między sobą oraz z blokami wejścia-wyjścia (IOB) za pomocą gęstej sieci konfigurowalnych połączeń. Bloki wejścia-wyjścia służą do ustanowienia komunikacji z urządzeniami peryferyjnymi takimi jak moduł pamięci RAM czy klawiatura

Każdy blok logiczny składa się minimalnie z trzech elementów:

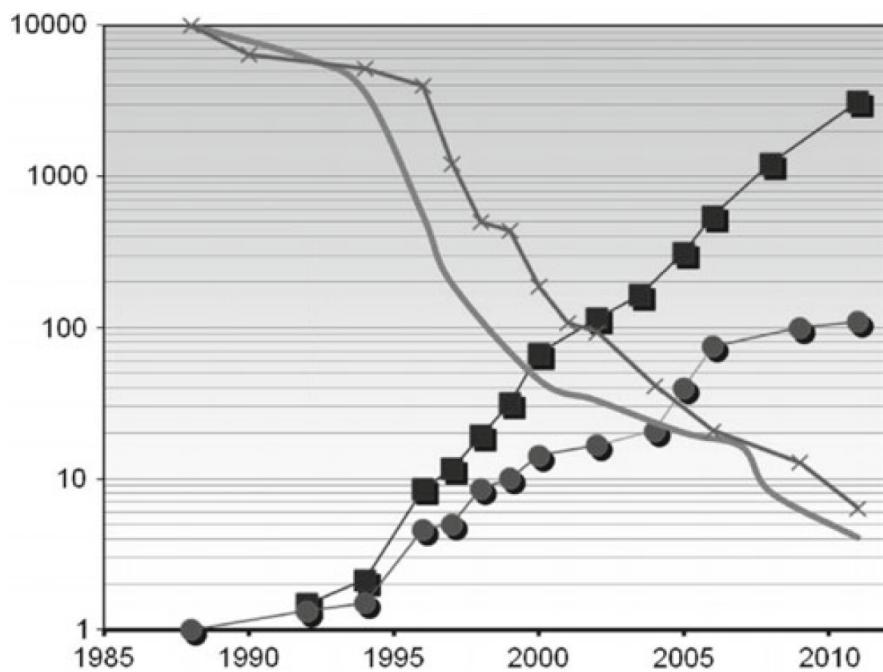
- *n*-wejściowej konfigurowalnej tablicy przeglądowej (ang. *LookUp Table*, LUT), której zadaniem jest realizacja *n*-parametrowej funkcji logicznej np.  $(\bar{A} \vee B) \wedge C$  jest 3-parametrową funkcją logiczną,
- *przerzutnika* (ang. *flip-flop*, FF) działającego jako pamięć,
- *multipleksera* (ang. *multiplexer*, MUX/MX), który dokonuje wyboru źródła sygnału, który ma zostać przekazany na wyjście bloku logicznego.



**Rys. 2.2:** Budowa podstawowego bloku logicznego w układzie FPGA. 4-wejściowy LUT realizuje ustaloną funkcję logiczną, której wynik, zależny od kombinacji sygnałów ( $I[0]:I[3]$ ), przekazywany jest do przerzutnika (FF) w celu zapamiętania. Przerzutnik działa synchronicznie z doprowadzonym sygnałem zegarowym (CLK) a jego zawartość może być zresetowana (RST). O tym, jaka wartość (O) zostanie podana na wyjściu bloku logicznego decyduje parametr (S) sterujący wyborem multipleksera (MX) pomiędzy nowym a zapamiętanym uprzednio wynikiem działania.

Pojedynczy blok logiczny sam w sobie jest dość prymitywnym elementem i samodzielnie nie jest w stanie realizować funkcji, której liczba parametrów wejściowych przekracza ilość wejść do LUT. Jednak dzięki występującej sieci połączeń<sup>ii</sup> wyjście jednego bloku logicznego może stanowić wejście drugiego. W ten sposób tworzone są znacznie bardziej rozbudowane funkcje logiczne. Wynika też z tego, że im więcej i im bardziej rozbudowane są bloki logiczne tym możliwości w zakresie komponowania funkcjonalności są szersze (wtedy jednak pojawiają się dodatkowe wyzwania konstruktorskie związane z optymalnym projektem sieci połączeń [27]).

<sup>ii</sup>W literaturze często wspomina się o blokach logicznych zanurzonych w morzu połączeń.



**Rys. 2.3:** Rozwój układów FPGA od momentu ich konstrukcji do dziś. Wartości podane na wykresie są liczone względem pierwotnego układu XC2064, który posiadał 64 programowalne bloki logiczne. Kwadratami oznaczono względną pojemność układu rozumianą poprzez ilość elementów logicznych, zamalowane koła osiągalne częstotliwości, ciągła gruba linia to cena, a linia z krzyżykami to pobierana moc. Widoczny jest znaczny postęp, który dokonał się na przestrzeni ostatnich lat [25]

Lata rozwoju układów FPGA połączone z analizą często wykorzystywanych algorytmów doprowadziły do implementacji dodatkowych elementów wchodzących w skład bloków logicznych. Są to jednostki bardziej wyspecjalizowane niż LUT, których użycie sprawia, że dany typ zadania może być wykonany znacznie szybciej i/lub jest bardziej ekonomiczne pod względem wykorzystania powierzchni układu. Tak powstały m. in:

- pamięci o niewielkiej pojemności LUTRAM,
- pamięci blokowe (ang. *block RAM, BRAM*), zdolne do przechowywania porcji danych rzędu 18 kb i wykonania do dwóch operacji dostępu w jednym cyklu zegara,
- moduły DSP dedykowane szybkiemu przetwarzaniu operacji dodawania i mnożenia.

Znajomość liczby, typu elementów składających się na układ FPGA oraz wymagań stawianych na etapie projektowania algorytmów pozwala dobrać odpowiedni układ do danych zastosowań, a przez to zoptymalizować koszty, które generowane byłyby przez elementy nie-wykorzystane. Producenci, chcąc sprostać oczekiwaniom rynkowym dostarczają całe gamy produktów, których wielkość mierzona za pomocą ilości bloków logicznych może zaczynać się na ok. 1000 bloków a kończyć na paru milionach [28].

Elastyczność w zakresie budowania funkcji logicznych posiada jednakże pewną niedogodność związaną z maksymalnymi osiągalnymi częstotliwościami pracy układów. W części przypadków (w tym omawianego systemu śledzenia promieni) samo przetworzenie sygnałów w bloku logicznym może trwać zaledwie 20-30% budżetu czasowego wynikającego z okresu

zadanego zegara. Pozostały czas jest czasem potrzebnym na propagację wyniku do kolejnego bloku logicznego poprzez dostępną w układzie konfigurowalną sieć połączeń. W efekcie uzyskiwane maksymalne częstotliwości zegara dla specyficznych zastosowań przetwarzania sygnałów (ang. *Digital Signal Processing, DSP*) mogą przekraczać 500 MHz<sup>iii</sup>, a w typowych zastosowaniach są to wartości pomiędzy 200 a 400 MHz (wartości te zależą oczywiście od technologii wykonania układu FPGA, akcelerowanego algorytmu i jakości kodu). W porównaniu do obecnych dzisiaj procesorów CPU działających coraz częściej z częstotliwościami sięgającymi 4 GHz są to wartości 10- 20-krotnie niższe. Wydawać mogłoby się zatem (tylko na podstawie porównania osiąganych częstotliwości), że nie można oczekiwać zbliżonej i wyższej wydajności od układu FPGA realizującego identyczny algorytm co odpowiadający mu procesor CPU. Nie musi być to prawda. Zależnie od typu rozwiązywanego problemu jak i umiejętności projektanta, algorytmy mogą być wykonywane o całe rzędy wielkości szybciej niż ich odpowiedniki opisane listą rozkazów CPU przy niższym zegarze oraz niższej konsumpcji energii elektrycznej. Kluczowa jest tu świadomość, iż projektant ma bezpośredni wpływ na to, jak układ zostanie skonfigurowany. Ta dowolność nierozerwalnie wiąże się z dużą odpowiedzialnością oraz stopniem doświadczenia wymaganego od projektanta.

## 2.1 Konfiguracja i języki opisu sprzętu

Pierwsze wytworzzone *układy scalone* (ang. *integrated circuit, IC*) posiadały małą złożoność, liczoną w setkach bramek logicznych, przez to charakteryzowały się niewielkim stopniem komplikacji (w rozumieniu realizowanej funkcjonalności).

W tamtych czasach odpowiedzialność za projekt i testy takiego układu leżały w gestii doświadczonego projektanta. Musiał on nie tylko w sposób ręczny wykonać projekt realizujący zadaną funkcjonalność, ale również zadbać o to, by sygnały propagujące się w układzie dochodziły do kolejnych etapów przetwarzania, wtedy gdy będą potrzebne<sup>iv</sup>. Wraz z postępem technologicznym ilość bramek logicznych, która mogła znaleźć się w pojedynczym układzie rosła, w wyniku czego pojawiła się potrzeba stworzenia narzędzi, które mogłyby wspomagać pracę na kolejnych etapach tworzenia układu elektronicznego tj.:

- tworzenia funkcjonalności,
- testów,
- syntez,
- implementacji.

W tym czasie języki programowania takie jak Pascal i C były powszechnie używane do tworzenia programów wykonywanych sekwencyjnie przez procesory zdolne do realizowania określonego zbioru instrukcji. Języki te jednakże nie mogły zostać zaadaptowane do celów opisu funkcjonalności połączonych ze sobą elementów elektronicznych (pojedynczych bramek, bloków DSP czy LUT) właśnie przez ich sekwencyjną naturę. Zintegrowane układy elektroniczne w swoim założeniu przetwarzają jednocześnie wiele sygnałów dla optymalnej

<sup>iii</sup>Tak wysokie częstotliwości, jak na układy FPGA są możliwe do uzyskania tylko dzięki obecności dedykowanych bloków DSP w strukturze układu FPGA.

<sup>iv</sup>Prędkość propagacji sygnałów jest skończona i wynika między innymi z wykorzystywanej technologii.

efektywności. Dla przykładu, jeżeli rozwiązywany problem wymaga wykonania  $n$  niezależnych od siebie operacji dodawania, najefektywniej jest zaprojektować układ tak, by na tym etapie przetwarzania znajdowało się  $n$  niezależnych od siebie sumatorów.

Potrzeba kreowania tego typu zachowań w układach elektronicznych doprowadziła do powstania *języków opisu sprzętu* (ang. *Hardware Description Languages*, HDL). Pozwoliły one nie tylko na modelowanie funkcjonalności układów elektronicznych, ale również na prowadzenie wnikliwych testów poprawności wykonania zgodnie z założeniami projektowymi dzięki środowiskom symulacyjnym. Języki te, najpopularniejsze wśród nich to Verilog oraz VHDL, zaoferowały możliwość opisu układów cyfrowych na poziomie przepływu danych pomiędzy kolejnymi rejestrami (ang. *register transfer level*, RTL). Odrębne narzędzia zaś na podstawie tego opisu (RTL) mogą dokonać ekstrakcji wymaganych elementów logicznych i połączeń między nimi [29]. W zależności od tego, jakiego typu układ jest projektowany: czy jest to *wyspecjalizowany układ scalony* (ang. *application-specific integrated circuit*, ASIC) czy układ FPGA, zespół wykstrahowanych elementów logicznych i połączeń między nimi tzw. *netlista* (ang. *netlist*) na etapie syntezы będzie inny, ale funkcjonalność zostanie zachowana. Właśnie ta cecha jest często eksplotowana przy tworzeniu układów typu ASIC, gdzie etapem pośrednim jest implementacja funkcjonalności w układzie FPGA, co pozwala przetestować w realnych zastosowaniach nowo tworzony układ, zanim ten zostanie wysłany do (kosztownej) produkcji w finalne postaci.

Implementacja w układzie FPGA jest zautomatyzowanym procesem zaczynającym się od optymalizacji wygenerowanej netlisty po to, aby realizacja zadanej funkcjonalności wymagała jak najmniej bloków logicznych z uwzględnieniem optymalizacji wpływających na szybkość przetwarzania. Ma to na celu zmniejszenie opóźnień związanych z propagacją danych, dzięki czemu możliwe jest użycie wyższych częstotliwości zegarów. Następnie bloki te łączone są ze sobą w klastry w fizycznym układzie w taki sposób, by uzyskać ich optymalny rozkład w przestrzeni układu (unikając tzw. *stłoczenia* ang. *congestion*, czyli lokalnego przeciążenia sieci połączeń) z zachowaniem jak najkrótszych odległości między nimi.

Naturalnym problemem, który tutaj występuje jest pytanie o to czy rezultaty uzyskane na drodze syntezы i implementacji są najlepsze z możliwych dla danego układu FPGA i opisu za pomocą RTL. W istocie dla dzisiejszych układów, których ilość bloków logicznych przekracza milion, a ilość dostępnych połączeń w sieci jest dziesięciokrotnie wyższa rozpatrzenie wszelkich możliwych kombinacji jest niemożliwe [30]. W związku z tym narzędzia muszą posługiwać się przybliżonymi algorytmami heurystycznymi sterowanymi za pomocą dziesiątek parametrów a i tak czas oczekiwania na finalny plik konfiguracyjny tzw. *bitstream* może przekroczyć 24 godziny (w zależności od stopnia skomplikowania RTL). Co więcej, nie ma gwarancji, że narzędzie znajdzie rozwiązanie, które spełnia wymagania czasowe dotyczące transferu sygnałów pomiędzy kolejnymi blokami - bez tego układ na pewno nie będzie działał poprawnie<sup>v</sup>. Na dodatek wpływ dwóch dowolnych parametrów sterujących syntezą i implementacją nie musi być od siebie niezależny. Dlatego producenci tych narzędzi dostarczają funkcji, które pozwalają na automatyczne dostosowanie parametrów do określonych warunków.

<sup>v</sup>Narzędzia dokonujące wyliczeń opóźnień między blokami z natury są konserwatywne, czyli ich estymacje wynikają z symulacji najgorszego możliwego przypadku. Znane są jednak przypadki, gdy narzędzia te raportowały pozytywne przejście testów opóźnień, jednak pracujące urządzenie po pewnym czasie zaczynało zachowywać się w sposób nieprzewidywalny. Błędy te najprawdopodobniej związane są ze zwiększoną szumem termicznym występującym w pracującym i rozgrzanym układzie.

czają przeważnie zestawy predefiniowanych ustawień, które według ich zapewnienia powinny działać najbardziej optymalnie. Ustawienia te można podzielić na dwie główne i z założenia wykluczające się grupy: optymalizujące projekt pod względem osiąganych częstotliwości zegara oraz ilości użytych bloków logicznych. Jednak każdy projekt, a zwłaszcza te o dużym stopniu utylizacji zasobów układu, należy traktować indywidualnie. Samodzielne poszukiwanie optymalnych parametrów przy ustalonym RTL, gdy predefiniowane ustawienia nie dają oczekiwanych rezultatów wymaga nie tylko doświadczenia, ale również szczęścia. InTime, czyli komercyjne narzędzie dostępne na rynku i rozwijane od kilku lat, pozwala poprzez użycie technik uczenia maszynowego rozwiązać większość tych problemów. Jest ono tak zaprojektowane, iż jest w stanie dokonać predykcji optymalnych parametrów syntezy i implementacji każdego projektu dla narzędzi dostarczanych przez najważniejszych producentów układów FPGA [31][32][33].

Z drugiej zaś strony, istnieje możliwość, iż stworzony opis sprzętu jest nieoptimalny dla danej architektury układu FPGA. Konieczne mogą okazać się znaczne zmiany projektowe po to, aby zmniejszyć konsumpcję zasobów (elementów składowych bloków logicznych) i/lub móc osiągnąć wymaganą częstotliwość zegara.

W związku z tym projektowanie funkcjonalności realizowanej dzięki układom FPGA jawnie się jako proces o dużym stopniu komplikacji, gdzie nawet najmniejsza zmiana na którymkolwiek etapie może wpływać na jakość rezultatu końcowego (ang. *Quality of Results*, QoR) tj. wydajność i zużycie zasobów. To czy dana implementacja jest satysfakcjonująca zależy zaś od postawionych lub narzuconych z zewnątrz wymagań.

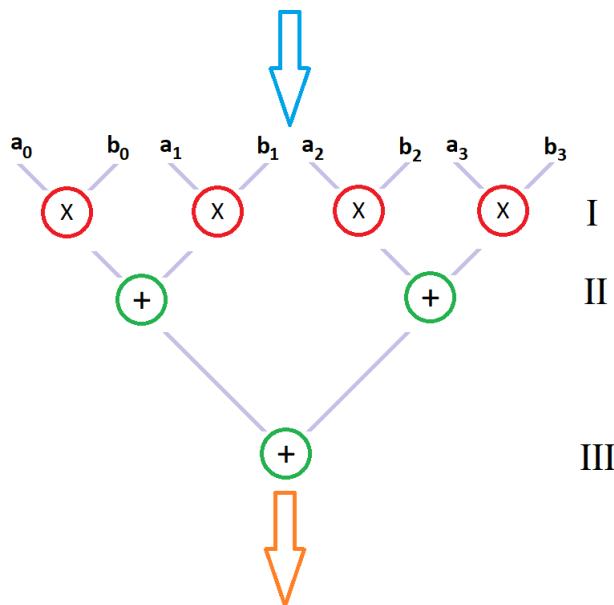
## 2.2 Nowoczesne podejście do tworzenia funkcjonalnych układów elektronicznych

W poprzednim podrozdziale wspomniano o językach opisu sprzętu, jako o narzędziu służącym do projektowania zachowania układów elektronicznych, które z natury rzeczy przetwarzają współbieżnie porcje danych. Języki te współistniały i były rozwijane równolegle z językami sekwencyjnymi służącymi do programowania procesorów, jednak z uwagi na powszechniejszy dostęp do układów typu CPU (ang. *Central Processing Unit*) znacznie bardziej rozpowszechnione zostały języki takie jak FORTRAN czy C, pozostawiając znajomość Verilog, VHDL i ich pochodne tylko dla wąskiego grona posługujących się nimi specjalistów.

W społeczności akademickiej oraz u producentów układów FPGA zrodziło się pytanie czy można udostępnić szerokiemu gronu odbiorców język, którego składnia nie odbiegałaby w znaczący sposób od znanych języków sekwencyjnych a pozwalający na przetwarzanie masowo równolegle (ang. *massively parallel*) podobnie, jak robią to HDL. Narzędzie dokonujące translacji kodu podobnego do C do jego ekwiwalentu w HDL, pozwoliłoby dotrzeć producentom układów FPGA do nowych odbiorców w znacznie prostszy sposób, a przez to zwiększyć swój zasięg i zyski. Mimo że próby stworzenia możliwie uniwersalnego narzędzia tego typu trwają już ponad 25 lat [34], dopiero niedawno stały się one wystarczająco użyteczne w realizacji postawionego im celu.

## 2.2.1 Synteza wysokiego poziomu

Proces konwersji algorytmu (rozumianego jako ciąg operacji koniecznych do wykonania, w celu otrzymania żądanego wyniku na podstawie dostarczonych danych) opisanego w języku wysokiego poziomu do jego specyfikacji na poziomie RTL nazywany jest *syntezą wysokiego poziomu* (ang. *high level synthesis*, HLS). Dokonuje ona analizy, kiedy żądane operacje mają być wykonane i wybiera odpowiednie bloki znajdujące się w układzie (pamięć, moduły DSP, przerzutniki, LUT i inne), które te operacje będą wykonywać (rysunek 2.4).



**Rys. 2.4:** Przykład optymalizacji wykonania iloczynu skalarnego  $\sum_{i=0}^3 a_i b_i$  za pomocą HLS. Iloczyn skalarny 4-elementowych wektorów wymaga wykonania 4 operacji mnożenia i 3 dodawania przy czym część operacji może zostać wykonana wspólnie. W pierwszym kroku HLS dokona alokacji 4 operatorów mnożenia (I), następnie wykonane zostaną 2 sumy częściowe (II) oraz suma końcowa (III). Takie rozłożenie wykonania operacji w czasie zapewnia maksymalną wydajność, co wiąże się również z największym zużyciem zasobów znajdujących się w układzie. Narzędzia HLS umożliwiają również takie rozłożenie operacji w czasie, by kosztem zmniejszonej wydajności oszczędzić część zasobów logicznych

Wynikowy projekt zapisany w postaci RTL, realizuje funkcjonalność opisaną w prosty i zrozumiały sposób za pomocą języka wysokiego poziomu. Projektant/programista za cenę utraty kontroli nad formą RTL zdaje się w momencie konwersji na ciąg automatycznych procesów przewidzianych przez twórcę danego HLS<sup>vi</sup>. W większości przypadków jednak samo wykrywanie operacji niezależnych, mogących być wykonanych jednocześnie w układzie jest niewystarczające, by wykorzystać pełen potencjał, jaki dają układy FPGA. Dlatego każdy HLS posiada w swojej składni pewien zestaw poleceń stanowiących wskazówkę dla procesu syntezy, w jaki sposób należy poddać konwersji daną partię kodu dla uzyskania optymalnych w danych warunkach rezultatów.

<sup>vi</sup>Szczegółowy opis tego, w jaki sposób dokonywana jest analiza algorytmu i jego konwersja do reprezentacji w postaci RTL można znaleźć w rozdziale 2 [30].

## 2.3 Xilinx i Vivado HLS

Kluczem do stworzenia narzędzia HLS, które stanie się chętnie wykorzystywane przez projektantów układów elektronicznych jest spełnienie dwóch warunków:

1. Bazowanie na dobrze znanym języku programowania jak np. C, tak aby wprowadzić jak najmniejszą ilość restrykcji w jego składni, które mogłyby stanowić ograniczenia w zakresie tworzenia funkcjonalności.
2. Dostarczenie odpowiednich rozszerzeń do języka bazowego, które udostępniałyby dostęp do optymalizacji wynikających wprost z możliwości użytkowanego sprzętu.

Narzędziem, które spełnia oba te założenia jest rozwijany przez firmę Xilinx (twórców pierwszego układu FPGA XC2064) Vivado HLS wchodzący w skład pakietu do tworzenia funkcjonalności w oparciu o układy FPGA tego producenta: Vivado Design Suite. Vivado HLS na podstawie kodu C/C++<sup>vii</sup> oraz informacji o żądanych przez użytkownika optymalizacjach, mających głównie na celu zwiększenie wydajności przetwarzania danych przez algorytm, dokonuje syntezы projektu do jego reprezentacji w VHDL i Verilog. Końcowym produktem syntezы jest *moduł IP* (ang. *Intellectual Property core*), który może stać się częścią systemu przetwarzania danych za pomocą układu FPGA firmy Xilinx.

### 2.3.1 Vivado HLS w opracowaniach

Od momentu udostępnienia użytkownikom, Vivado HLS stał się obiektem zainteresowania środowisk akademickich i inżynierskich chcących zbadać przede wszystkim poprawność syntezы kodu C/C++ oraz jakość generowanego opisu RTL w porównaniu do metod tradycyjnych. W opracowaniu [35] autorzy dokonali sprawdzenia równoważności wielu algorytmów pomiędzy kodem C a uzyskanym na jego podstawie RTL. Ponadto dostrzegli, iż badane narzędzie dokonuje wielu nietrywialnych optymalizacji, wśród których wymienić można:

- Wykrywanie propagacji oraz zwijanie wyrażeń zawierających stałe np.

```

1 ...
2 int x = 2;
3 int y = x * 4 / 5 + fun(i);
```

zostanie zoptymalizowane do postaci:

```

1 ...
2 int y = 1 + fun(i);
```

- Usuwanie wykonania operacji, które nie mają wpływu na wynik działania funkcji.
- Przenoszenie kodu niezależnego od iteracji pętli poza jej ciało.

Inni autorzy dokonali za to szczegółowych studiów przypadków implementacji często wykorzystywanych algorytmów za pomocą Vivado HLS. W publikacji [36] udowodniono poprawność stworzonego i zoptymalizowanego RTL, znajdującego zastosowanie w transporcie

---

<sup>vii</sup>Możliwe jest również wykorzystanie SystemC oraz OpenCL (Open Computing Language).

energii elektrycznej. Porównano jego wydajność i użycie zasobów względem znanego wcześniej projektu zapisanego bezpośrednio z użyciem VHDL<sup>viii</sup>. Projekt stworzony z użyciem nowego narzędzia był o ok. 10% szybszy niż jego pierwotny, jednak utylizacja zasobów była nawet dwukrotnie większa. Przydatność poszczególnych dyrektyw optymalizacyjnych udostępnianych przez Vivado HLS oraz problemy wynikające z natury analizowanych algorytmów a wpływające na stosowność tych optymalizacji została omówiona w [37]. Z kolei rozprawa [38] koncentruje się na analizie implementacji mnożenia macierzy za pomocą trzech różnych algorytmów. Autorzy pokazują w niej, jak poszczególne optymalizacje wpływają zarówno na czas wykonania jak i na wykorzystanie zasobów logicznych układu FPGA i porównują te metryki z tymi, które odpowiadają ich w pełni zoptymalizowanym wersjom stworzonym przez doświadczonych projektantów w HDL. Okazuje się, że w każdym przypadku najszybsza wersja opisana za pomocą Vivado HLS była wyraźnie wolniejsza i niemal w każdym przypadku wykorzystywała więcej zasobów docelowego układu FPGA.

Oprócz prac porównawczych, istnieją również dokumenty opisujące nietrywialne techniki optymalizacji, głównie związane z problemami występującymi w momencie konieczności iteracyjnego przetwarzania danych w pętlach [39][38]. Pokazują one wprost, w jaki sposób można radzić sobie z tego typu wyzwaniami z użyciem Vivado HLS.

Chociaż autorzy przytoczonych publikacji są zgodni, co do tego, iż pomimo zastosowania odpowiednich optymalizacji kodu, otrzymane projekty nie oferują bezwzględnie maksymalnej wydajności z niewspółmiernie dużą utylizacją bloków logicznych, nadal jednak użycie Vivado HLS jest uzasadnione w przypadku braku specjalistycznej wiedzy związanej z językami opisu sprzętu. Narzędzie to pozwala również osiągnąć satysfakcjonujące rezultaty, wymagającego średnio ok. 3-krotnie mniejszego nakładu pracy względem rozwiązania tworzonego bezpośrednio w HDL przez doświadczonego projektanta.

### 2.3.2 Podstawowe informacje związane z Vivado HLS

Firma Xilinx udostępniając użytkownikom swoich rozwiązań technologicznych Vivado HLS ułatwiała zadanie początkującym projektantom/programistom. Tworzenie algorytmu za pomocą języka C/C++ jest nie tylko łatwiejsze niż tworzenie opisu RTL, ale również powstający kod jest w zamierzeniu znacznie bardziej czytelny. Co więcej, testowanie przygotowywanych algorytmów odbywa się również w środowisku symulacyjnym C/C++, a co za tym idzie można wykorzystywać tradycyjne narzędzia do sprawdzania funkcjonalnej poprawności (*debugowania*) algorytmów na dowolnym etapie, a także zmierzyć czas wykonania algorytmu przez procesor komputera. Na dodatek Vivado HLS bierze pod uwagę, dla jakiego konkretnego układu FPGA firmy Xilinx przeprowadzana jest synteza oraz przy jakiej częstotliwości pracy. Dzięki temu narzędzie to na podstawie znanej mu szybkości wykonywania operacji dla danego FPGA dokonuje oceny, które operacje mogą zostać wykonane w tym samym cyklu zegara, a które muszą zostać rozłożone na kilka takich cykli - bardziej zaawansowane technologicznie układy będą w stanie wykonać dany algorytm szybciej i/lub z użyciem

---

<sup>viii</sup>Różnica między nimi polegała na wykorzystaniu innej reprezentacji liczbowej - projekt Vivado HLS używał liczb zmienno-, a VHDL stałopozycyjnych.

mniejszej liczby zasobów sprzętowych<sup>ix</sup>.

Atrakcyjność rozwiązania jakim jest Vivado HLS polega głównie na tym, że na pierwszy rzut oka jego składnia niewiele różni się od tej znanej z języków C/C++. Należy stworzyć funkcję główną, która zazwyczaj będzie przyjmować pewien zbiór argumentów oraz zwracać wyniki na zewnątrz modułu (argumenty te na drodze syntezы staną się portami wejścia/wyjścia tworzonego IP). Funkcja główna będzie realizować pewien algorytm wykorzystując odpowiednie konstrukcje wynikające ze składni języka bazowego i może się odwoływać do innych funkcji pomocniczych. W rzeczywistości Vivado HLS implementuje dużą część standaryzowanych konstrukcji języków C/C++ oraz typów danych, jednak istnieją pewne ograniczenia, wśród których najważniejsze to<sup>x</sup>:

- Brak wsparcia dla dynamicznego zarządzania pamięcią (polecenia `new/delete` oraz `malloc()/free()`), wynikający wprost z faktu, iż układ FPGA dysponuje ustalonymi zasobami sprzętowymi. Użycie tych zasobów musi być konkretne i znane na etapie implementacji funkcjonalności w układzie, by móc dokonać odpowiednich połączeń pomiędzy kolejnymi elementami logicznymi. O ile w przypadku tablic danych wystarczające może się okazać przyjęcie takiego rozmiaru by w danym momencie mieściły się w niej wszystkie potrzebne wartości, o tyle programiści przyzwyczajeni do *programowania zorientowanego obiektywnego* (ang. *object-oriented programming*) szybko natkną się na problemy z uwagi na:
  - brak obsługi generycznych typów danych (C++ STL) takich jak: `vector`, `map` czy `queue`,
  - ograniczone wsparcie dla polimorfizmu, które jest dopuszczalne tylko jeśli typy danych można określić w sposób statyczny (na etapie komplikacji),
  - niepełną obsługę funkcjonalności związaną z użyciem wskaźników, w szczególności możliwe jest tylko rzutowanie wskaźników pomiędzy natywnymi typami języka bazowego.
- Niedopuszczalne jest aby IP dokonywało odwołań do jakichkolwiek funkcji systemowych. Funkcjonalność realizowana poprzez stworzone IP jest sterowana tylko poprzez dane, które jest on w stanie odczytać z portów wejścia i nie posiada ono wiedzy na temat niczego innego (włączając w to inne IP, które mogą działać równolegle z nim na tym samym układzie FPGA). Stąd też np. użycie funkcji `time()` przez IP jest niedopuszczalne, można jednak stworzyć port wejściowy, przez który będzie przekazywana odpowiednia wartość z zewnątrz.
- Niemożliwe jest stosowanie funkcji rekurencyjnych (tzn. odwołujących się same do siebie) wprost<sup>xi</sup>. W miarę możliwości należy dążyć do przekształcenia algorytmu w taki

<sup>ix</sup>Oczywistą ceną za dostęp do tego typu optymalizacji jest ograniczenie stosowalności Vivado HLS tylko do układów wyprodukowanych przez firmę Xilinx.

<sup>x</sup>Spisane tutaj ograniczenia mają zastosowanie tylko i wyłącznie do ciała funkcji głównej i wszystkich funkcji wywoływanych przez nią, która zostaje poddana syntezie do RTL. W środowisku testowym C/C++ ograniczenia te nie występują.

<sup>xi</sup>Omawiany problem implementacji śledzenia promieni z użyciem ukałdu FPGA z natury rzeczy jest problemem rekurencyjnym, gdzie promienie padające na powierzchnię obiektu dzielą się na promienie odbite i załamane.

sposób, aby dało się go opisać za pomocą pętli. Alternatywnie można podjąć się emulacji tego, w jaki sposób CPU radzi sobie z przetwarzaniem rekurencyjnym z użyciem stosu [40].

Powyższe ograniczenia, a zwłaszcza te dotyczące zarządzania pamięcią oraz przetwarzania rekurencyjnego, w wielu przypadkach prowadzić będą do stworzenia od podstaw istniejących już algorytmów tradycyjnie wykonywanych przez CPU. Przy tego typu konwersji należy również przemyśleć możliwości wykorzystania potencjału przetwarzania równoległego, który może zapewnić sprzęt.

Z drugiej jednak strony producent Vivado HLS dostarcza zbiór bibliotek, które są zoptymalizowane pod kątem czasu wykonania i implementacji w układach FPGA. Ma to szcześcielne znaczenie w przypadku odwoływanego się do funkcji zawartych w bibliotekach matematycznych, gdzie udostępnione są nie tylko implementacje podstawowych funkcji takich jak np. `sin()`, `sqrt()`, ale również bardziej zaawansowanych jak `fft()` (szybka transformata Fouriera).

Użycie układów FPGA sprawia również, że programista ma dowolność w doborze i wykorzystaniu typów danych. Oprócz tradycyjnych typów jak `int`, `char`, `float` czy `double`, przechowujących standardowo 32, 8, 32 i 64 bity może on skorzystać z typów danych o dowolnej długości dla typów całkowitoliczbowych i stałopozycyjnych, jeśli tylko uzna, że takie postępowanie będzie uzasadnione - operacje arytmetyczne na mniejszych typach danych będą zwykle przeprowadzane szybciej a wykorzystanie bloków logicznych układu będzie niższe. Stałopozycyjne typy liczbowe zdefiniowane są poprzez:

- całkowitą ilość bitów, jaką zajmować będzie zmienna,
- ilość bitów przypadającą na opis całkowitej części zmiennej z uwzględnieniem bitu znaku,
- zachowanie określające sposób, w jaki dokonywane będzie zaokrąglanie,
- zachowanie w przypadku przepełnienia (czyli co się stanie, gdy wynik działania arytmetycznego nie daje się zapisać przy pomocy zadanej liczby bitów).

Typy całkowitoliczbowe o dowolnej precyzji opisane są tylko poprzez ich długość bitową wyrazoną dodatnią liczbą całkowitą. Oba rodzaje typów liczbowych o zadanej precyzji występują w wersji z i bez znaku.

W obliczu operowania typami stałopozycyjnymi o dowolnej długości dużą niedogodnością jest fakt, iż jak dotąd (stan na wersję Vivado Design Suite 2018.1) nie zostały udostępnione użytkownikom Vivado HLS zmiennoprzecinkowe typy o dowolnej precyzji, mimo iż narzędzie **System Generator** (również wchodzące w skład Vivado Design Suite), będące rozszerzeniem do środowiska **Simulink**, oferuje taką funkcjonalność pozwalając dostosować według potrzeb ilość bitów danych przypadających na eksponentę oraz mantysę [25]<sup>xii</sup>. W ręce programistów Vivado HLS zostały oddane jedynie zmiennoprzecinkowe typy zgodne ze standardem IEEE-754 [41]:

- **half**: 1 bit znaku, 5 bitów eksponenty oraz 10 bitów mantysy,

---

<sup>xii</sup>Jak się okaże własność ta będzie miała wpływ na końcowy projekt systemu śledzenia promieni.

- **float:** 1 bit znaku, 8 bitów eksponenty oraz 23 bity mantisy,
- **double:** 1 bit znaku, 11 bitów eksponenty oraz 52 bity mantisy.

Jeśli tylko stworzony moduł jest zgodny z obowiązującą składnią HLS, narzędzie dokonuje optymalnej implementacji dla ustalonego układu FPGA z uwzględnieniem optymalizacji zadanych przez użytkownika. Optymalizacje te, zwane w HLS *dyrektywami* (ang. *optimization directives*), pozwalają na modyfikację, a przez to kontrolę nad domyślnym sposobem syntezy poszczególnych partii algorytmu jak i portów wejścia/wyjścia. Wpływ wymuszenia danej dyrektywy na odpowiednie *metryki* można sprawdzić analizując informacje zawarte w wygenerowanym raporcie syntezy.

- Wykorzystanie zasobów (ang. *area*) to konserwatywne, czyli w najgorszym możliwym przypadku, zestawienie użycia zasobów sprzętowych (przerzutniki, LUT, BRAM, moduły DSP) koniecznych do realizacji zadanej w module funkcjonalności<sup>xiii</sup>.
- Opóźnienie (ang. *latency*), czyli ilość cykli zegara wymaganych do przeprowadzenia wszystkich obliczeń. W przypadku pętli jest to ilość cykli zegara, aby przetworzyć wszystkie jej iteracje.
- Opóźnienie iteracji pętli (ang. *loop iteration latency*) to ilość cykli zegara do ukończenia jednej iteracji pętli.
- Interwał (ang. *initiation interval*, II) wyraża ilość cykli zegara, jaka upłynie zanim:
  - funkcja będzie mogła przetworzyć kolejną porcję danych,
  - pętla będzie mogła przetworzyć dane w kolejnej iteracji.

Przemyślany zapis algorytmiczny rozwiązywanego problemu w połączeniu z kombinacją odpowiednich dyrektyw i analizą raportów syntezy pozwala stworzyć moduł IP o optymalnych metrykach, które uzasadniałyby wykorzystanie układu FPGA i jednocześnie, na podstawie użycia zasobów, wskazywałyby na możliwość implementacji w układzie. Należy jednak podkreślić, że Vivado HLS jedynie dokonuje transformacji algorytmu do jego reprezentacji poprzez RTL i przez to nie daje gwarancji, że proces implementacji (przeprowadzany przez odrębne narzędzie) zakończy się sukcesem tj. wygenerowaniem pliku konfiguracyjnego działającego układu.

### 2.3.3 Wykorzystanie dyrektyw optymalizacyjnych

Dyrektyny w Vivado HLS zmieniające domyślne parametry syntezy, a przez to wpływające na sposób implementacji na poziomie RTL, mają zastosowanie do:

- portów wejścia/wyjścia funkcji głównej modułu,
- funkcji,
- pętli,

---

<sup>xiii</sup>Na etapie syntezy netlisty oraz implementacji w układzie dokonywana jest seria optymalizacji, mająca na celu m. in. zmniejszenie wykorzystania zasobów, dlatego wartości estymowane i faktyczne zawsze będą się różnić.

- regionów, czyli bloku kodu zawartego pomiędzy parą odpowiadających sobie nawiasów klamrowych,
- tablic

i mogą zostać zapisane bezpośrednio w kodzie programu w postaci:

```
1 #pragma HLS <DYREKTYWA> <PARAMETRY_DYREKTYWY>
```

jak i w oddzielnym pliku `directives.tcl`. Nie ma przy tym znaczenia z punktu widzenia syntezы, który sposób definiowania dyrektyw zostanie przyjęty przez programistę (można nawet mieszać ze sobą oba sposoby). Należy tylko mieć na uwadze, że dyrektywy zapisane w kodzie mogą sprawiać problemy (tzn. wywołać niepożądane optymalizacje), gdy dany kod jest współużytkowany przez kilka osób i w różnych projektach.

Poniższe zestawienie dyrektyw obejmuje tylko część z nich, z zaznaczeniem najważniejszych ich cech, które wywierają największy wpływ na optymalizację algorytmu poddanego syntezie HLS.Więcej informacji na temat dyrektyw można odnaleźć w podręczniku użytkownika HLS [42].

#### • INTERFACE

Projekty opisywane poprzez HDL, w celu wykonywania operacji wejścia/wyjścia, wymagają posiadania tzw. portów, które działają zazwyczaj w oparciu o pewien protokół transmisji danych, który można zdefiniować używając dyrektywy `INTERFACE`. W zależności od typu danych (pojedyncza wartość, tablica, wskaźnik) oraz sposobu dostępu (wejście, wyjście, wejście/wyjście) Vivado HLS udostępnia odpowiednie protokoły transmisji - tutaj wspomniane zostaną dwa z nich.

- `AXI4-Lite` jest interfejsem, który pozwala na to, aby moduł mógł być kontrolowany poprzez CPU lub mikrokontroler. Dzięki niemu jeden interfejs może zostać wykorzystany do połączenia w grupę kilku portów (mogą to być wartości, wskaźniki, tablice), które mogą w następstwie syntezы być kontrolowane przy pomocy stworzonego sterownika (ang. *C driver*).

**Listing 2.1:** Przykład użycia interfejsu `AXI4-Lite`

```
1 int fun(int* a, int* b)
2 {
3     #pragma HLS INTERFACE s_axilite port=return bundle=BUS_A
4     #pragma HLS INTERFACE s_axilite port=a         bundle=BUS_A
5     #pragma HLS INTERFACE s_axilite port=b         bundle=BUS_A
6     return *a + *b;
7 }
```

W powyższym przykładzie wartości wskazywane przez `a`, `b` oraz wartość zwracana są obsługiwane przez ten sam interfejs `AXI4-Lite` o nazwie `BUS_A`. Wygenerowany sterownik pozwoli przy pomocy odrębnego mikrokontrolera po implementacji w układzie FPGA m. in. na:

- \* inicjalizację modułu,
- \* dostarczenie informacji dla IP, gdzie w przestrzeni adresowej znajdują się wartości, na które wskazują `a` i `b`,

- \* uruchomienie modułu i sprawdzenie czy ten zakończył swoje działanie,
  - \* odczyt wyniku działania stworzonego IP.
- **AXI4 Master**

Interfejs tego typu może zostać przypisany do portów będących tablicami i wskaźnikami, a jego cechą jest to, że pozwala dokonywać indywidualnych jak i seryjnych (ang. *burst*) transferów danych. W tym drugim przypadku wydajność transferu danych jest znacznie wyższa z uwagi na sekwencyjny odczyt/zapis względem zadanego adresu początkowego.

**Listing 2.2:** Przykład użycia interfejsu AXI4 Master z seryjnym dostępem do danych. Zmiana adresów wskazywanych umożliwiona została poprzez interfejs AXI4-Lite

```

1 void fun(int* a, int* b)
2 {
3     #pragma HLS INTERFACE s_axilite port=return bundle=BUS_A
4     #pragma HLS INTERFACE m_axi port=a           bundle=MAXI_DATA
5     #pragma HLS INTERFACE m_axi port=b           bundle=MAXI_DATA
6
7     #pragma HLS INTERFACE s_axilite port=a       bundle=BUS_A
8     #pragma HLS INTERFACE s_axilite port=b       bundle=BUS_A
9
10    const unsigned n = 100;
11    int loc_a[n];
12
13    // Wykonaj kopie danych z użyciem dostępu seryjnego
14    // SPOSOB 1
15    memcpy(loc_a, a, n * sizeof(int));
16
17    Obliczenia: for (int i = 0; i < n - 1; ++i)
18    {
19        loc_a[i] = loc_a[i] * loc_a[i + 1];
20    //    loc_a[i] = a[i] * a[i + 1];
21    }
22
23    // Zapisz nowe dane do b (rownież seryjnie)
24    // SPOSOB 2
25    Zapis: for (int i = 0; i < n; ++i)
26    {
27        #pragma HLS PIPELINE
28        b[i] = loc_a[i];
29    }
30 }
```

W powyższym przykładzie portom `a` i `b` nakazano zachowanie zgodne z protokołem `AXI4 Master` oraz poprzez interfejs `AXI4-Lite` umożliwiono sterowanie nimi z poziomu zewnętrznego mikrokontrolera. Najpierw do tablicy `loc_a` przepisana zostaje zawartość `n=50` kolejnych wartości wskazywanych przez `a` za pomocą funkcji `memcpy()` (i tylko w takim kontekście Vivado HLS umożliwia jej użycie). Następnie dokonywane są pewne operacje na tych wartościach, by w końcu zostać przepisane pod adres wskazywany przez `b`. Co ważne, również operacja zapisu wykonywana jest w trybie seryjnym - bez użycia `memcpy()` za to jako pętla z nadaną dyrektywą `PIPELINE`, gdzie dostęp do adresów następuje bezwarunkowo (dostęp do pamięci nie jest poprzedzony warunkiem logicznym) w kolejności rosnącej.

Opóźnienie takiego kodu, jak w powyższym przykładzie obliczone przez Vivado HLS 2017.4 dla układu znajdującego się na płytce ewaluacyjnej KCU116 wynosi 416 cykli (ze standardowym zegarem 100 MHz). Wystarczy jednak pominąć instrukcję transferu danych do `loc_a` i bezpośrednio w pętli `Obliczenia` odwoływać się do zawartości `a` aby dostęp do danych przestał być sekwencyjny, a opóźnienie

wzrosło do 1197 cykli zegara.

- **PIPELINE**

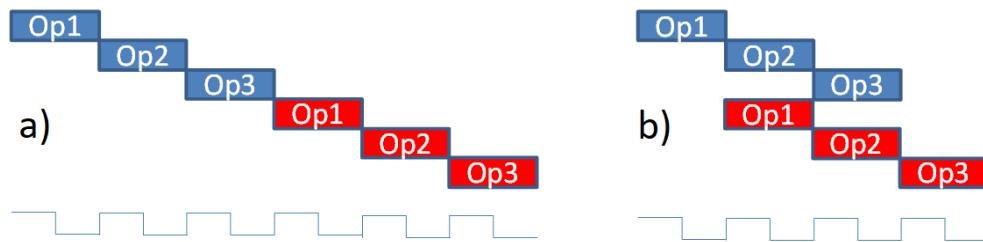
Jest to podstawowa optymalizacja mająca zastosowanie do funkcji i pętli umożliwiająca naturalną eksplorację równoległego przetwarzania z wykorzystaniem układu FPGA. Wykorzystywany jest tutaj fakt, że jeśli przetwarzanie w danej funkcji czy ciele pętli składa się z kilku następujących po sobie etapów tak jak w poniższym kodzie przykładowym, to z każdym z nich wiąże się alokacja pewnych zasobów sprzętowych.

**Listing 2.3:** Kod ilustrujący zastosowania dyrektywy PIPELINE

```

1 void fun ( . . . )
2 {
3 //#pragma HLS PIPELINE
4     Op1;
5     Op2;
6     Op3;
7 }
```

Zakładając, że każda z operacji trwa 1 cykl zegara, opóźnienie pojedynczej iteracji pętli wywołującej `fun()` to 3 cykle. W tym miejscu należy zauważyć, iż w momencie, gdy `Op1` przekaże wywołanie do `Op2` zasoby sprzętowe odpowiedzialne za realizację `Op1` stają się bezczynne, podczas gdy mogłyby zacząć przetwarzać kolejną porcję danych. W ten sposób opóźnienie wykonania  $n$ -krotnie tej funkcji zamiast  $3n$  cykli wyniosłoby tylko  $n + 2$  (interwał zoptymalizowanej wersji tej funkcji to 1 cykl).



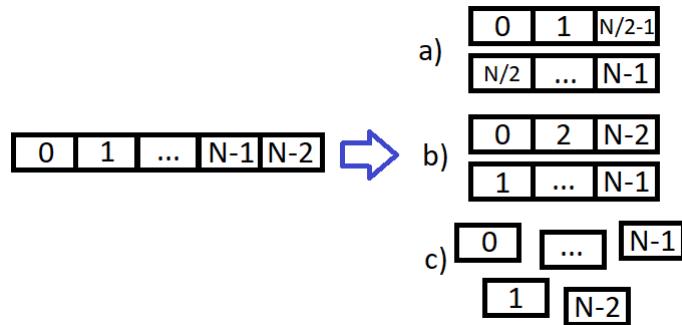
**Rys. 2.5:** Przykład działania dyrektywy PIPELINE. Niepoddana optymalizacji funkcja a) może być wywołana dopiero gdy poprzednie jej wywołanie zostanie ukończone (wykona się `Op3`). Użycie PIPELINE powoduje, że funkcja jest gotowa do przetwarzania danych w momencie gdy `Op1` z poprzedniego wywołania przekaże działanie dalej - efektem jest znaczne zwiększenie przepustowości przetwarzania

Jeśli funkcja lub pętla, którą użytkownik zamierza zoptymalizować stosując PIPELINE zawiera w swoim ciele inną pętlę lub jej hierarchię, ilość ich iteracji musi być znana w momencie kompilacji - w przeciwnym razie HLS nie będzie w stanie zastosować tej optymalizacji (patrz dyrektywa UNROLL).

- **ARRAY\_PARTITION**

Efektywność zastosowania dyrektywy PIPELINE do danej funkcji czy pętli okazuje się być często ograniczona poprzez intensywny dostęp do pamięci, w której zapisane są tablice danych. Gdy dany algorytm próbuje odwoływać się do tej samej tablicy więcej

niż dwukrotnie w tym samym cyklu zegara należy się zastanowić nad przeprojektowaniem algorytmu stanowiącego ograniczenia, gdyż układy BRAM implementujące funkcjonalność tablic umożliwiają wykonanie do dwóch operacji zapisu/odczytu danych w pojedynczym cyklu zegara. W przypadku gdy taka zmiana nie jest możliwa można się posłużyć dyrektywą **ARRAY\_PARTITION** do podziału tablicy stanowiącej ograniczenie przepustowości. Można tego dokonać na 3 sposoby zilustrowane poniższym schematem - wybór optymalnej wersji zależy od wzoru dostępu do danych znajdujących się w partycjonowanej tablicy.



**Rys. 2.6:** Sposoby podziału tablicy na mniejsze elementy. W przypadku a) pierwotna tablica zostaje podzielona na  $f = 2$  równych elementów - pierwsza otrzymuje elementy o mniejszych, a druga o większych indeksach (podział blokowy ang. *block*). Przykład b) rozmieszcza co  $f = 2$  element pierwotnej tablicy w mniejszych tablicach (podział cykliczny ang. *cyclic*). Przykład c) odpowiada rozbiciu tablicy na poszczególne rejestryst (podział całkowity ang. *complete*)

#### • UNROLL

Domyślne parametry Vivado HLS dokonują syntezy pętli w taki sposób, że istnieje tylko jeden zespół bloków logicznych odwzorowujących algorytm wykonywany przez ciało pętli, który jest używany przez wszystkie jej iteracje. Za pomocą dyrektywy **UNROLL** możliwa jest zmiana tego zachowania:

- dokonanie częściowego rozwinięcia pętli, które dokona  $f$ -krotnej replikacji logiki odpowiedzialnej za ciało pętli,
- pełne rozwinięcie wszystkich  $n$  iteracji pętli (ilość iteracji musi być znana w momencie komplikacji kodu).

W obu przypadkach rozwinięcia pętli, wszystkie zreplikowane bloki będą przetwarzać dane jednocześnie dla maksymalnej wydajności (o ile nie występują ograniczenia związane z dostępem do tablic oraz nie istnieją zależności pomiędzy wielkościami występującymi w różnych iteracjach) kosztem zwiększonej konsumpcji zasobów. Należy pamiętać, że pełne rozwinięcie pętli jest narzucone automatycznie w przypadku, gdy funkcja bądź pętla nadziedziona zostanie poddana działaniu dyrektywy **PIPELINE**. Jeśli ilość iteracji (a przez to ilość zreplikowanych bloków) nie będzie znana w momencie komplikacji, proces optymalizacji tego kodu zakończy się niepowodzeniem.

- DATAFLOW

Załóżmy, że w kodzie poddawanym syntezie HLS udaje się wyodrębnić następującą sekwencję przetwarzania:

**Listing 2.4:** Blok kodu mogący zostać poddany optymalizacji DATAFLOW

```

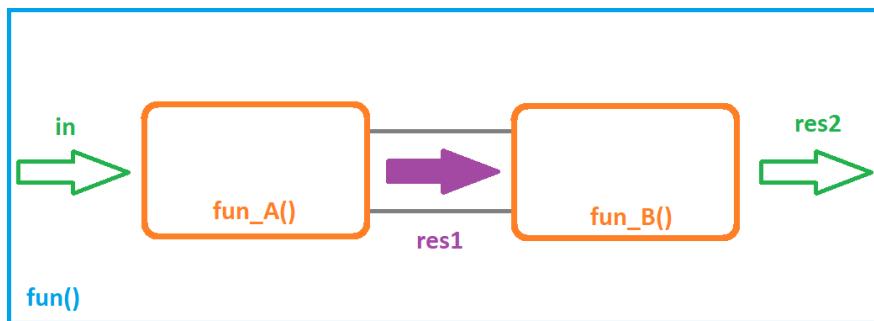
1 int fun(int in)
2 {
3 //#pragma HLS DATAFLOW
4 ...
5 fun_A(in, res1);
6 fun_B(res1, res2);
7
8 return res2;
9 }
```

Funkcja nadziedna przyjmuje pewien parametr wejściowy `in`, który jest parametrem wejściowym do `fun_A()`. Wynikiem działania `fun_A()` jest wartość `res1`, która staje się parametrem wejściowym do `fun_B()` - wynik jej działania, `res2`, zostaje zwrócony przez ciało funkcji nadziednej `fun()`. Tego typu sekwencyjne przetwarzanie danych, gdzie efekt działania jednej z funkcji (zadania) jest parametrem wejściowym kolejnej z nich, jest kandydatem do zastosowania dyrektywy DATAFLOW. Optymalizacja ta podobnie jak PIPELINE umożliwia zwiększenie wydajności algorytmu w układzie FPGA poprzez równoległe działanie wielu zadań jednocześnie - dzieje się to jednakże w oparciu o kanały transmisji danych między zadaniami.

Użycie DATAFLOW (dla funkcji i pętli) powoduje, iż pomiędzy poszczególnymi zadaniami w sekwencji tworzone są kanały buforujące dane. Te zaś, w zależności od typu danych, będą implementowane jako kolejki typu *FIFO* (ang. *first in, first out*) dla danych skalarnych, albo jako bufory typu *ping-pong* dla danych tablicowych<sup>xiv</sup>. Każde zadanie w sekwencji działa indywidualnie, jednak jest uzależnione od tego, czy bufory dostarczające do niego dane są niepuste. Takie zachowanie pozwala potencjalnie na efektywniejsze zrównoleglenie zadań względem PIPELINE, które statycznie dokonuje szeregowania wykonania operacji, jednak za cenę większego zużycia zasobów związanego z implementacją wymaganych buforów.

---

<sup>xiv</sup>Na bufor tego typu składają się dwa bufory o identycznym rozmiarze. Kiedy jeden z nich służy do odczytu danych, do drugiego dane są zapisywane. W momencie, gdy bufor, z którego były czytane dane zostaje opróżniony, role buforów ulegają zmianie.



**Rys. 2.7:** Schemat działania DATAFLOW. Pomiędzy dwoma zadaniami `fun_A()` i `fun_B()` zostaje stworzony kanał przepływu danych, dzięki któremu wartości reprezentowane przez `res1` są odpowiednio buforowane i przekazywane jako argument wejściowy kolejnego zadania. Działanie `fun_B()` rozpocznie się, gdy dane w buforze wejściowym staną się dostępne - od tej pory oba przedstawione zadania będą pracować jednocześnie

Dokonując optymalizacji DATAFLOW na danej funkcji lub pętli należy mieć na uwadze pewne obostrzenia związane z jej zastosowaniem:

- Dane wytworzone przez jedno z zadań mogą stanowić parametr wejściowy tylko jednego następnego zadania w sekwencji.
- Efekt działania danego zadania nie może zostać użyty w kolejnej iteracji przez zadanie, wykonujące się przed nim - sprzężenie (ang. *feedback*) pomiędzy kolejnymi iteracjami jest nieobsługiwane.
- Wykonanie zadania z sekwencji nie może być w żaden sposób warunkowane.
- Pętle wchodzące w skład sekwencji mogą mieć tylko jeden warunek kończący ich działanie.

#### • DEPENDENCE

Zadanie, jakie jest postawione przed Vivado HLS polega na odtworzeniu funkcjonalności opisanej poprzez algorytm za pomocą RTL w sposób, który nie narusza jego integralności tzn. dla dowolnego zbioru parametrów wejściowych moduł zaimplementowany w układzie FPGA musi dawać identyczne wyniki jak jego odpowiednik opisany kodem C/C++ uruchomiony na CPU. Z tego powodu wykrywane są zależności pomiędzy operacjami, które zachowują oryginalny ciąg przyczynowo-skutkowy. Zachowanie to w szczególności może powodować, iż przepustowość pętli z dyrektywą PIPELINE określona przez jej interwał, zostanie ograniczona przez zależności występujące pomiędzy kolejnymi iteracjami.

**Listing 2.5:** Zależność interwału pętli od występujących zależności pomiędzy kolejnymi iteracjami

```

1 ...
2 for (int i = 0; i < N - 1; ++i)
3 {
4 #pragma HLS PIPELINE
5   float a = tab[i] + tab[i + 1];
6   float b = sqrt(a);

```

```

7     tab[ i + 1] = b;
8 }
```

W przypadku takim, jak powyżej pętla nie jest w stanie przetworzyć danych z maksymalną przepustowością ( $II=1$ ) ponieważ efekt działania kolejnej operacji zależy od czasu obliczenia **b**, które przypisywane jest do **tab[i + 1]** (**tab[i]** w kolejnej iteracji).

Istnieją jednak pewne sytuacje, gdy dbanie przez HLS o zapewnienie poprawności działania jest niepożądane i prowadzić będzie do zbyt dużego ograniczenia szybkości obliczeń. Takim przykładem może być próba implementacji przy pomocy Vivado HLS procesora sterowanego listą rozkazów. W dużym uproszczeniu procesor taki przetwarza po kolei listę instrukcji, zaś każda instrukcja dokonuje pewnych operacji na rejestrach (zmiennych wewnętrznych).

**Listing 2.6:** Ilustracja problemu związanego z wykrywaniem zależności przez Vivado HLS i wpływu na wydajność

```

1 while( true )
2 {
3 #pragma HLS PIPELINE
4 ...
5 switch( inst )
6 {
7     case ADD:
8         reg = reg + 1;
9         break;
10    case NEG:
11        reg = -reg ;
12        break;
13    case SQRT:
14        reg = sqrt( reg );
15        break;
16    default:
17        break;
18    }
19 }
```

W powyższym przykładzie HLS dokonując analizy zależności pomiędzy wykorzystaniem **reg** założy konserwatywnie, że interwał przetwarzania instrukcji będzie określony poprzez instrukcję wykonującą się najdłużej (tutaj jest to funkcja **sqrt()**), gdyż tylko wtedy istnieje pewność, że moduł będzie wykonywał się w sposób poprawny. Prowadzi to do sytuacji, że nawet instrukcja pusta (realizowana przez **default**) będzie wykonywana tak długo jak **sqrt()**, co w jawnym sposób ogranicza szybkość przetwarzania. Dyrektywa **DEPENDENCE** w założeniu pozwoli wymusić, by rozważanie zależności zostało w przypadku zmiennej **reg** pominięte<sup>xv</sup>.

### • ALLOCATION

Poszukiwanie najwydajniejszego rozwiązania z wykorzystaniem dyrektyw wymienionych do tej pory może prowadzić do nadmiernej konsumpcji zasobów raportowanej

---

<sup>xv</sup>Stosując tę dyrektywę należy mieć pewność, że nie narusza to integralności algorytmu albo projektant jest w stanie poradzić sobie z ubocznymi konsekwencjami jej użycia.

po wykonaniu syntezy HLS. Dyrektywa **ALLOCATION** pozwala na ograniczenie użycia operatorów (np. dodawanie, mnożenie) oraz instancji funkcji występujących na danym poziomie hierarchii, przez co zostanie zwiększone ich współdzielenie pomiędzy operacjami.

- **INLINE**

Każda funkcja, o ile nie jest wystarczająco mała, domyślnie jest oddzielnym elementem w hierarchii wywołań a przez to alokująca zasoby sprzętowe na własny użytek. Stąd użycie zasobów może być optymalizowane tylko w obrębie danej funkcji. Użycie dyrektywy **INLINE** dla danej funkcji włącza ją do hierarchii funkcji nadzędnej, potencjalnie dając możliwość lepszego współdzielenia zasobów przez różne operacje (funkcje posiadające niewiele logiki są automatycznie włączane do hierarchii funkcji wywołującej).

### 2.3.4 Integracja stworzonego modułu IP w układzie FPGA

Stworzenie i weryfikacja poprawności działania modułu IP przy użyciu Vivado HLS i udostępnionych razem z nim narzędzi jest ważnym, ale nie ostatnim krokiem na drodze do jego działania w realnym układzie FPGA. Nowy moduł o stworzonej funkcjonalności przeważnie jest projektowany jako część większego systemu przetwarzania danych. Wiąże się to z koniecznością zaprojektowania poprawnego tzw. *schematu blokowego* (ang. *block design*) implementującego przepływ danych i kanały komunikacji między modułami. Elementy tego schematu blokowego muszą następnie zostać poddane syntezie do reprezentacji za pomocą netlisty by następnie dokonać próby jej implementacji i wygenerowania pliku konfiguracyjnego dla konkretnego urządzenia. Wszystkie te etapy wykonywane są przez narzędzie zwane *Vivado*.

#### Budowa schematu blokowego

Wraz z Vivado Design Suite dostarczana jest baza często używanych i pomocnych w rozwiązywaniu wielu problemów IP<sup>xvi</sup>. Wśród nich należy wymienić:

- Moduły będące implementacją prostych rozwiązań procesorowych (tzw. *softprocesory*) mogących pełnić funkcję mikrokontrolerów monitorujących i sterujących działaniem całego budowanego systemu. Ich wydajność oraz maksymalne częstotliwości pracy nie pozwolą wydajnie przeprowadzać skomplikowanych operacji, jednak ich zaletą jest szeroka możliwość dostrojenia ich parametrów w zakresie przewidzianym przez producenta do potrzeb użytkownika. Przykładem takiego procesora jest 32-bitowy *Microblaze*, którego podstawowa konfiguracja może być rozszerzona m. in. o [43][44]:
  - pełną obsługę obliczeń zmiennopozycyjnych pojedynczej precyzji (**float**),
  - buforowany dostęp do danych i instrukcji za pomocą pamięci podręcznej (ang. *cache*) o wybranej konfiguracji rozmiaru,
  - rozbudowany moduł do analizy błędów wykonania.

---

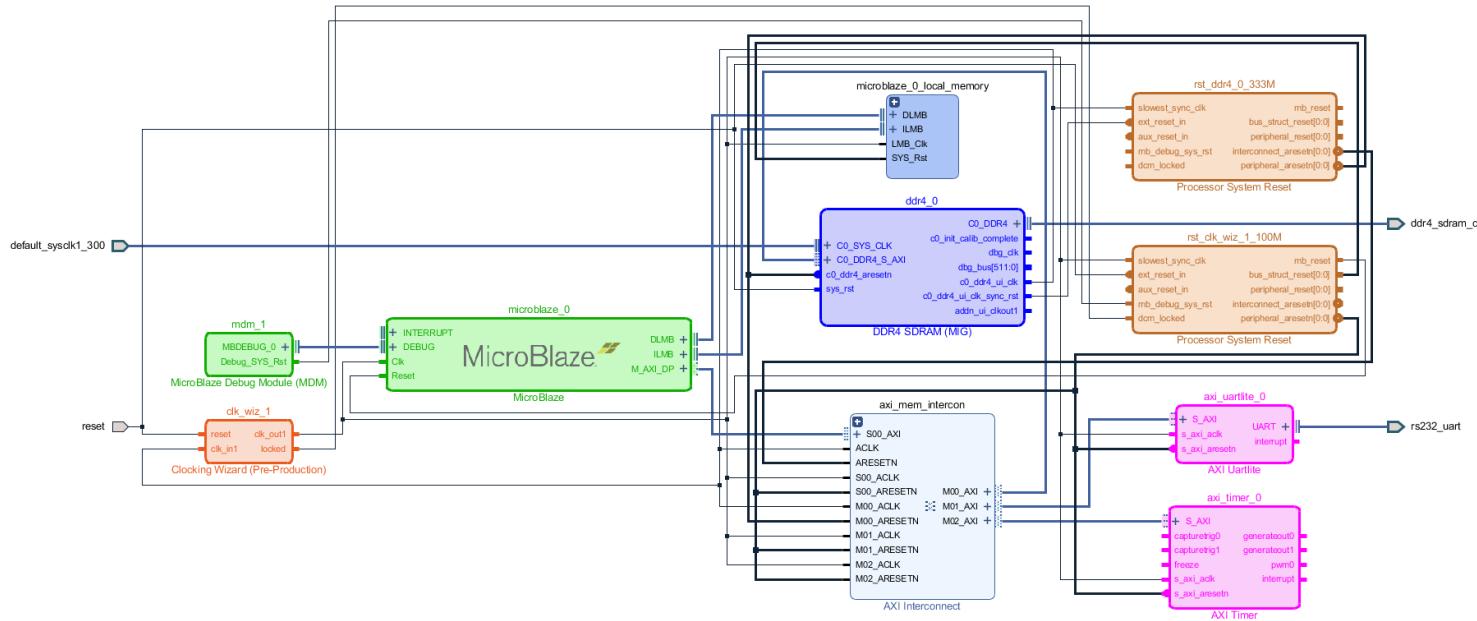
<sup>xvi</sup>Dostęp do niektórych z nich może być ograniczony przez odrębne postanowienia licencyjne.

Szczególnie istotny jest tutaj fakt, iż **Microblaze** posiada własny kompilator C/C++, który jest świadomym wybranej przez użytkownika konfiguracji tego IP. W szczególności, jeśli użytkownik zdecydował się umieścić moduł akcelerujący przetwarzanie danych zmiennopozycyjnych (ang. *floating-point processing unit*, FPU), wtedy kompilator używa dedykowanych instrukcji wszędzie tam, gdzie zajdzie potrzeba wykonania operacji na liczbach zmiennopozycyjnych (w przeciwnym razie funkcjonalność ta będzie emulowana za pomocą sekwencji operacji podstawowych).

- Kontroler pamięci zewnętrznej DDR3/DDR4 pozwalający nawiązać komunikację z zasobami mogącymi przechowywać znacznie większe ilości danych niż oferują to elementy wbudowane w FPGA.
- Układ generujący sygnały zegarowe o zadanej częstotliwości.
- Licznik czasu pozwalający mierzyć jak długo wykonywane są poszczególne zadania przez tworzony system.
- Cała rodzina IP służąca do przetwarzania, konwersji i transmisji sygnałów reprezentujących dane wizualne.

Umieszczenie potrzebnych modułów na schemacie blokowym wymaga następnie, aby połączyć ze sobą odpowiednie porty wejścia/wyjścia w taki sposób, aby cały system działał poprawnie (zobacz schemat ukazany na rysunku 2.8). Pomocny okazuje się na tym etapie dostęp do podręcznika użytkownika danego modułu, do którego uzyskuje się dostęp bezpośrednio z poziomu widoku schematu blokowego. Ponadto Vivado potrafi wykrywać niepołączone interfejsy z rodziną AXI oraz sygnały **reset** i zegara (**clock**), oferując automatyzację połączeń. Trzeba pamiętać, że modyfikacja konfiguracji modułu może prowadzić do powstania, usunięcia bądź zmiany szerokości bitowej portu/portów - w związku z tym może być wymagana interwencja ze strony użytkownika celem zaktualizowania przepływu sygnałów między modułami.

Poprawność konfiguracji połączeń można sprawdzić dzięki wbudowanemu narzędziu. Dokonuje ono walidacji czy wszystkie wymagane porty zostały połączone ze sobą oraz czy spełnione zostały wszystkie inne warunki (ang. *constraints*) wymagane do poprawnego działania projektu.



**Rys. 2.8:** Schemat blokowy prostego przykładowego systemu przetwarzania danych. Sercem systemu w tym przypadku jest mikroprocesor Microblaze wyposażony w możliwość sprawdzania poprawności przetwarzania za pomocą Microblaze Debug Module. Drugim istotnym elementem jest generator interfejsu do pamięci zewnętrznej DDR4 SDRAM - dzięki niemu możliwe jest zapisywanie i odczyt danych przez Microblaze do i z zewnętrznej względem układu FPGA pamięci RAM. Jednak, aby Microblaze mógł się komunikować z modułem zarządzającym pamięcią zewnętrzną są one połączone ze sobą za pomocą łącznika danych (AXI Interconnect). Na tej samej zasadzie z mikroprocesorem połączone są licznik czasu AXI Timer oraz moduł do komunikacji zewnętrznej AXI Uartlite (wykorzystywany najczęściej do wypisywania tekstu na terminalu). W systemie tym występują dwa główne sygnały zegarowe: pierwszy (default\_sysclk1\_300) jest sygnałem bazowym interfejsu pamięci zewnętrznej, zaś drugi generowany przez Clocking Wizard decyduje m. in o wydajności pracy mikroprocesora.

## Synteza

Narzędzia wchodzące w skład Vivado, które dokonują syntezy, przetwarzają niezależny od architektury kod RTL do postaci odwzorowującej przepływ danych z użyciem konkretnych elementów logicznych znajdujących się w układzie FPGA. Trzeba jednak zdawać sobie sprawę z tego, iż napisany w odpowiedni sposób kod RTL może ułatwić narzędziom ekstrakcję konkretnych zachowań, które będą mogły być odwzorowane bezpośrednio w wyspecjalizowanych układach np. DSP.

Powyższy problem nie występuje w przypadku tworzenia modułów z użyciem Vivado HLS. Dzięki temu, iż wie ono, dla jakiego układu FPGA dokonywana jest translacja, tworzone są takie konstrukcje w kodzie Verilog/VHDL, które wywołają syntezę optymalnej sekwencji elementów logicznych.

Mimo to nie jest prawdą, że dla danego projektu, na który składają się poszczególne moduły IP, istnieje tylko jedna reprezentacja za pomocą netlisty - jej końcowa postać zależy od parametrów syntezy. Kilka standardowych zbiorów ustawień, zwanych strategiami, dostarczanych jest przez producenta. Wśród nich są takie (`Flow_AreaOptimized_medium`, `Flow_AreaOptimized_high`), które dostosowują netlistę by zajmowała jak najmniej zasobów, ale też i takie które w zamian za zwiększenie użycia logiki układu w teorii mogą pozwolić na uzyskanie wyższych częstotliwości pracy (`Flow_PerfOptimized_high`). Dozwolone jest również samodzielnie dokonywanie zmian, a tym samym tworzenie własnych strategii syntezy [45].

## Implementacja

Proces implementacji za pomocą Vivado polega na umieszczeniu i połączeniu w układzie FPGA wytworzonej na etapie syntezy netlisty w taki sposób, aby spełnione były wszelkie warunki nałożone na projekt (dotyczące np. opóźnień sygnału, które wynikają z zastosowanej częstotliwości zegara, wyprowadzeń odpowiednich sygnałów na wybrane porty wejścia/wyjścia z układu FPGA<sup>xvii</sup>), dając w rezultacie konfigurację działającego układu o zaprojektowanej funkcjonalności. Proces ten składa się w ogólności z kilku pod-procesów, z których część jest opcjonalna, jednak zwiększa ją one prawdopodobieństwo (poprzez dokonywanie przez nie optymalizacje) spełnienia specyficznych warunków stawianych przed projektem jak np. wysokości częstotliwości przetwarzania czy założonego limitu konsumpcji energii elektrycznej [46].

Tak samo jak w przypadku syntezy, producent dostarcza kilkunastu różnych strategii implementacji pozwalających zoptymalizować projekt pod wybranym aspektem (np. poboru mocy, konsumpcji bloków logicznych, częstotliwości pracy). Można również tworzyć własne strategie.

Poziom komplikacji danego układu, rozumianego jako zespół tworzącego go modułów IP, wybrana strategia implementacji, szybkość komputera (częstotliwość procesora oraz możliwości przetwarzania współbieżnego, dostępna pojemność pamięci RAM, szybkość dysku twardego/SSD) oraz rodzaj docelowego układu FPGA wpływają będą na czas potrzebny na przeprowadzenie procesu implementacji - ten może wynosić tak kilkanaście minut jak i kilkadziesiąt godzin. Aby efekt implementacji uznać za poprawny, spełnione muszą być

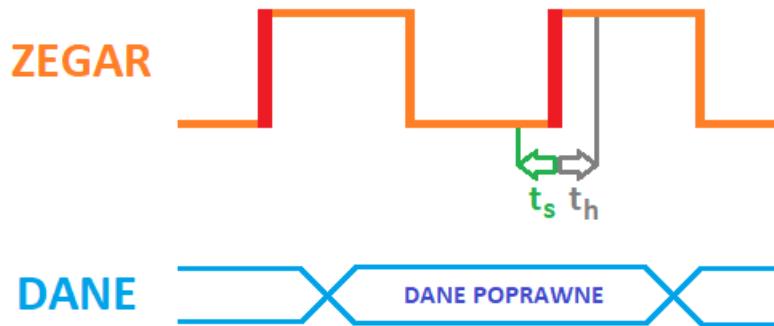
<sup>xvii</sup>Przykładem tego może być wykorzystanie zewnętrznej względem układu FPGA pamięci RAM.

następujące warunki:

- Układ FPGA musi posiadać wystarczającą ilość elementów logicznych wymaganych do implementacji projektu. Praktyczna zasada mówi o tym, aby projekt nie wymagał użycia więcej niż 70% zasobów [25].
- Sieć połączeń musi być w stanie skomunikować wszystkie potrzebne punkty (porty sygnałowe elementów logicznych) ze sobą.
- Nie mogą zostać naruszone wymagania czasowe związane z propagacją sygnałów.

Sygnały zegarowe są elementem synchronizacyjnym zapewniającym, że przesyłanie danych pomiędzy kolejnymi etapami przetwarzania odbywa się bez błędów. Kiedy sygnał reprezentujący wartość w wyniku działania danego elementu logicznego ulega zmianie, w czasie dokonywania się tej zmiany wartość jest niestabilna. Stąd wymagany jest pewien czas (ten zależny jest między innymi od układu FPGA oraz zakresu temperatur pracy), zanim będzie można użyć tej nowej wartości.

- *Czas konfiguracji*  $t_s$  (ang. *setup time*) jest to najmniejszy czas przed kolejnym zboczem zegara, kiedy sygnał musi być ustabilizowany.
- *Okres wstrzymania*  $t_h$  (ang. *hold time*) jest minimalnym czasem, przez który sygnał musi być stabilny po zboczu zegara.



**Rys. 2.9:** Czas konfiguracji i okres wstrzymania -  $t_s$  oraz  $t_h$  definiują tzw. okno czasowe, w którym sygnał musi być stabilny aby układ FPGA poprawnie realizował powierzone mu zadanie przetwarzania danych. Synchronizacja następuje względem narastającego zbocza sygnału zegarowego

W przypadku niespełnienia którejkolwiek z powyższych metryk ( $t_s$ ,  $t_h$ ), w którymkolwiek miejscu przetwarzania danych, mamy do czynienia z naruszeniem (ang. *setup violation*, *hold violation*), które najprawdopodobniej będzie wiązało się z niepoprawnie działającym układem. Raport po implementacji prezentuje dodatkowe parametry mówiące o tym:

- ile wynosi największe spóźnienie się z konfiguracją sygnału tzw. *worst negative slack* (WNS),
- o ile za krótko wynosi najgorszy z okresów wstrzymania tzw. *worst hold slack* (WHS).

Jeśli implementacja przebiegła poprawnie WNS oraz WHS są nieujemne.

## Podsumowanie

W powyższych paragrafach, dokonano zwięzłego opisu tego, czym są układy FPGA, na jakiej zasadzie działają oraz jakiego rodzaju wyzwania wiążą się z ich wykorzystaniem. Jednym z nich jest niewątpliwie sposób, w jaki dokonuje się opisu zachowania sprzętu, czyli sposobu przetwarzania danych. Zaprezentowane rozwiązań polega na wykorzystaniu syntezys wysokiego poziomu, w tym przypadku Vivado HLS, która na podstawie algorytmicznego opisu za pomocą C/C++ o zmodyfikowanych możliwościach (tj. z ograniczoną składnią i dyrektywami optymalizacyjnymi) potrafi dokonać poprawnej konwersji do języków opisu sprzętu. Niewątpliwą zaletą takiego rozwiązania jest fakt, iż nie wymaga ono specjalistycznej wiedzy oraz zwiększa produktywność programisty/projektanta. Z drugiej jednak strony, oddzielne badania i porównania sugerują, iż osiągnięcie identycznie niskiego zużycia zasobów połączonego z wysoką wydajnością, jaką zapewnia bezpośrednia konfiguracja sprzętu z użyciem VHDL czy Verilog, jest właściwie niemożliwa. Mimo to Vivado HLS wydaje się być ciekawym narzędziem, zwłaszcza do tworzenia rozwiązań prototypowych.



*Wszyscy wiedzą, że czegoś nie da się zrobić. I wtedy pojawia się ten jeden, który nie wie, że się nie da, i on właśnie to coś robi.*

przyp. Albert Einstein

## Rozdział 3

---

# Akcelerator śledzenia promieni z użyciem układu FPGA

---

### 3.1 Założenia

Wszystko, o czym wspomniano do tej pory w poprzedzających rozdziałach miało na celu stworzyć podwaliny teoretyczne, które mogą zostać wykorzystane w praktyce do budowy systemu śledzenia promieni w oparciu o układy FPGA. Projekt, którego realizacji się podjęto zakładał od samego początku:

1. Przy użyciu Vivado HLS
2. zaprojektowanie modułu do akceleracji grafiki metodą śledzenia promieni,
3. zdolnego do wizualizacji świata złożonego z prostych kształtów geometrycznych,
4. których powierzchnie zachowują się zgodnie z odpowiednimi funkcjami BSDF,
5. w czasie rzeczywistym.

Chociaż sposób wykonania projektu był zmieniany w czasie, powyższe wymagania pozostały aktualne. Warto jednak w tym momencie dokonać sprecyzowania każdego z nich.

Ad. 1. Głównym, nadzawanym celem pracy była ewaluacja możliwości Vivado HLS, jako narzędzia umożliwiającego porzucenie konfiguracji za pomocą HDL na rzecz opisu algorytmicznego w języku podobnym do C/C++. W momencie rozpoczęcia projektu znane były wyniki i wnioski z prac eksperymentalnych używających tego narzędzia w pewnych typowych zastosowaniach. Szczególnie ważna okazała się być informacja, o tym, że Vivado HLS posiada tendencję do tworzenia opisu RTL powodującego użycie stosunkowo dużej ilości elementów logicznych znajdujących się w układzie. Chcąc mieć swobodę w tworzeniu funkcjonalności należało dysponować układem FPGA o odpowiednio dużej ilości elementów logicznych, dlatego do projektu wybrano układ *Virtex 7* znajdujący się na płytce ewaluacyjnej *VC707* [47], który został później zastąpiony przez układ *Kintex UltraScale+* wchodzący w skład płytka ewaluacyjnej *KCU116* [48].



**Rys. 3.1:** Płytkę ewaluacyjną VC707. Oprócz układu FPGA znajdującego się pod widocznym na zdjęciu układem chłodzenia, płytka ewaluacyjna udostępnia do wykorzystania wiele interfejsów danych i sposobów komunikacji, wśród których wymienić można: Ethernet, HDMI, PCI Express, zewnętrzną pamięć operacyjną, programowalne przyciski czy diody LED, które mogą zostać użyte przez projektanta w realnym systemie przetwarzania danych [47]

Użycie płyt ekspansyjnych ma tę zaletę, iż wraz z nimi użytkownik otrzymuje dostęp do funkcjonalności oferowanych przez dodatkowe elementy elektroniczne zamontowane na takich płytach i mogące kooperować z układem FPGA w procesie przetwarzania danych. W kontekście omawianego projektu kluczowe okazały się być możliwość przechowywania danych w pamięci RAM oraz wykorzystanie komunikacji wizualnej zgodnej ze standardem HDMI (obie wymienione płytki posiadają porty HDMI mogące służyć jako wyjście obrazu).

**Tab. 3.1:** Porównanie podstawowych parametrów opisujących wykorzystane układy FPGA

	Elementy składowe				Proces technologiczny
	BRAM <sup>i</sup>	DSP <sup>ii</sup>	FF	LUT	
<b>VC707</b>	2060	2800	607200	303600	28 nm
<b>KCU116</b>	960	1824	433920	216960	16 nm

<sup>i</sup>Każdy element pamięci blokowej BRAM jest w stanie przechowywać 18 kb danych i umożliwia przeprowadzenie do dwóch operacji wejścia/wyjścia w pojedynczym cyklu zegara. W układach FPGA z serii UltraScale oraz UltraScale+ występuje również dodatkowy typ pamięci zwany *UltraRAM*, URAM co w zamierzeniu pozwala przechowywać większe ilości danych bezpośrednio w układzie FPGA (każdy blok URAM może pomieścić 288 kb danych), zmniejszając konieczność odwoływanego się do zewnętrznych zasobów a przez to zwiększając wydajność przetwarzania [49].

<sup>ii</sup>Moduły DSP znajdują zastosowanie w przypadku akceleracji działań arytmetycznych, zwłaszcza stałopozycyjnych [50][51]. Pozwalają na efektywniejszą implementację i uzyskiwanie wyższych częstotliwości pracy, niż gdyby te same operacje miały zostać wykonane za pomocą LUT.

Oba układy<sup>iii</sup> przedstawione w tabeli 3.1 posiadają stosunkowo dużą ilość podstawowych elementów, na bazie których można tworzyć funkcjonalność. Jednakże to VC707 jest znacznie większy pod względem dostępnych zasobów (różnice względne zaczynają się od 40% dla LUT i kończą na 115% dla pamięci BRAM). Zastanawiający w tym momencie może być zatem fakt, iż ostatecznie to układ KCU116 został wybrany do realizacji zamierzzonego projektu. Jego przewaga uwidoczniała się na etapie implementacji funkcjonalności, dzięki temu, iż jest to układ wykonany w znacznie nowszej technologii, dająccej większe możliwości podczas procesu implementacji i łączenia ze sobą elementów za pomocą sieci konfigurowalnych połączeń.

- Ad. 2. U samych podstaw przetwarzania grafiki metodą śledzenia promieni leży fakt, iż obliczenia dotyczące jednej rodziny promieni (na którą składają się wszystkie promienie powiązane z danym promieniem pierwotnym: odbite, załamane, promienie cieni) odpowiedzialnej za obliczenie koloru konkretnego piksela obrazu są niezależne od pozostałych rodzin. Wynika z tego, że w idealnym przypadku kolory wszystkich pikseli mogłyby być obliczane jednocześnie przez dedykowane dla każdego z nich układy przetwarzające. Realizacja takiego systemu w układzie FPGA graniczy aktualnie z niemożliwością, choćby ze względu na wymagane w tym celu zasoby sprzętowe. Niemniej jednak udostępniane przez układy FPGA naturalne możliwości przetwarzania jednokrotnego dają duże szanse na zwiększenie efektywności algorytmu, względem jego wersji działającej na procesorze CPU komputera.
- Ad. 3. Jedną z podstawowych cech śledzenia promieni, o której wspomniano przy okazji porównania z rasteryzacją, jest fakt, iż jeśli tylko istnieje sposób matematycznego rozwiązania przecięcia powierzchni z promieniem, to można jej użyć w procesie renderowania bez konieczności stosowania triangulacji. Podczas gdy sfera opisana przez macierz jej przekształcenia jest jednym obiektem, który należy sprawdzić na drodze promienia, triangulacja kuli wymaga setek pojedynczych trójkątów, z którymi taki test trzeba przeprowadzić. Z jednej strony zaproponowane postępowanie niweluje potrzebę stosowania struktur akcelerujących testy geometrii, z drugiej zapewnia, iż uzyskiwane punkty przecięcia będą możliwie dokładne (w zależności od precyzji obliczeń użytego typu danych).
- Ad. 4. Omawiana technika umożliwia uzyskiwanie realistycznych i interesujących obrazów dopiero wtedy, jeśli zostaną odpowiednio zastosowane różne techniki oświetlenia powierzchni. Tworzony system powinien oferować możliwość użycia różnych funkcji BSDF, które mogłyby być odpowiednio konfigurowane za pomocą parametrów podanych przez użytkownika.
- Ad. 5. Nie istnieje ścisła definicja działania programu w czasie rzeczywistym. W branży elektronicznej rozrywki (tj. gry video) przyjmuje się, że minimalna częstotliwość generowania pełnych klatek obrazu, dla zachowania złudzenia płynności, nie powinna być

<sup>iii</sup>Od tej pory, dla wygody, poprzez układ rozumieć należy tak płytę ewaluacyjną jak i rodzaj układu FPGA, który na niej się znajduje. Związane jest to z faktem, iż pełne nazwy układów FPGA mają skomplikowaną postać zależną od m.in. rodziny, technologii, wielkości czy oceny szybkości (ang. *speed grade*) układu.

niższa niż 25-30 (chociaż coraz częściej mówi się o 60). Możliwości szybkiego generowania obrazu używając śledzenia promieni zależą głównie od:

- rozdzielczości klatki obrazu,
- ilości rozpatrywanych promieni w danej rodzinie,
- ilości świateł i obiektów w scenie,
- stosowanych funkcji BSDF.

W związku z tym zachowawczo przyjęto, że jeśli tylko zaprogramowany układ będzie w stanie generować obrazy o rozdzielczości 1280 x 720 pikseli z uwzględnieniem pierwszej rodziny promieni wtórnych w czasie poniżej 0,1s, to efekt taki będzie można uznać za satysfakcyjny. W sprawdzeniu warunku dostatecznej płynności (poza standardowym pomiarem czasu wykonania) w praktyce miało pomóc użycie dostępnego na wykorzystywanych płytach kodaka i portu HDMI<sup>iv</sup>.

## 3.2 Budowa modułu śledzenia promieni

Tak postawiony problem można zasadniczo rozwiązać na dwa skrajnie różne sposoby wiążące się z zupełnie innymi wyzwaniami na etapie implementacji:

1. Stworzenie prostego układu procesorowego, pozwalającego przetwarzać podaną przez użytkownika listę rozkazów implementujących wymagany algorytm.
2. Zaprojektowanie modułu, który implementuje skończony zbiór zachowań (algorytmów), których wywołanie może być sterowane poprzez parametry wejściowe, czyli tzw. *potok przetwarzania o ustalonej funkcjonalności* (ang. *fixed pipeline*).

O ile w pierwszym przypadku nadzawanym celem jest to, aby instrukcje były wykonywane jak najszybciej, a sam układ potrzebował jak najmniej zasobów, o tyle w drugim przypadku wyzwaniem może okazać się pogodzenie ze sobą ilości oferowanych funkcji z dostępnym na układzie miejscem i koniecznością zapewnienia synchronizacji danych na etapie implementacji.

W obu przypadkach ważna jest kontrola nad utylizacją zasobów, która jest nierozerwalnie powiązana z wykorzystywanymi typami danych do przeprowadzenia obliczeń. W tabeli 3.2 zestawiono jak dużo zasobów potrzeba do implementacji operacji dodawania, mnożenia, dzielenia i pierwiastkowania w przypadku posługiwania się różnymi typami danych oraz ile cykli zegara (#) należy oczekwać na wynik w każdym przypadku. Widać, że w przypadku typów całkowitoliczbowych oraz stałopozycyjnych (`ap_fixed<W, N>`, gdzie W to ilość bitów przypadająca na pełen zapis liczby (tzw. *długość słowa*), N - bity części całkowitej ze znakiem, a pozostałe przypadają na zapis części ułamkowej) elementarne operacje tj. dodawanie i mnożenie mogą nie tylko być wykonywane w każdym cyklu zegara (# = 1), ale również wymagają niewiele zasobów układu. Mniejszy z zaprezentowanych typów stałopozycyjnych

<sup>iv</sup>Poza tym w taki sposób najłatwiej zaprezentować możliwości stworzonego systemu podczas pokazów. Zamiast operować zbiorem statycznych scen, które wcale nie muszą być efektem działającego systemu w układzie FPGA, widz otrzymuje interaktywną (np. poprzez zmianę położenia obserwatora z pomocą dostępnych na płytce przycisków dowolnego przeznaczenia) demonstrację, w której jest w stanie uwierzyć.

`ap_fixed<24, 18>` w celu wykonania mnożenia używa 1 zamiast 3 bloków DSP, co może mieć znaczenie w przypadku układów o niewielkiej ilości tychże. Z drugiej strony typy całkowitoliczbowe oraz stałopozycyjne w przypadku dwóch innych działań (a mających duże znaczenie w momencie rozwiązywania równań kwadratowych oraz normalizacji wektorów) mają albo bardzo duże opóźnienie (dzielenie), albo wymagają dużej ilości LUT (pierwiastkowanie). Chcąc dokonać implementacji pierwiastkowania liczby opisanej przez typ `ap_fixed<32, 20>` w układzie VC707 należy liczyć się z koniecznością poświęcenia ok. 1,5% wszystkich dostępnych LUT. Atrakcyjniej pod tym względem wyglądają typy zmiennoprzecinkowe, a zwłaszcza typy `half` oraz `float`. Dziwi jednak fakt, że pierwiastkowanie typu `half` wymaga znacznie więcej zasobów niż `float`, jednocześnie przechowując informację tylko o 16 bitach zamiast 32<sup>v</sup>. Tam gdzie typy całkowitoliczbowe oraz stałopozycyjne pokazywały swoją znaczną przewagę (tj. dodawanie i mnożenie), tam liczby zmiennoprzecinkowe wymagają więcej zasobów, a oczekiwanie na wynik jest dłuższy.

Wybór reprezentacji danych jest ważnym etapem tworzenia rzeczywistego systemu przetwarzania. Należy odpowiedzieć sobie na pytanie, jakie dane będą przetwarzane oraz jakie będzie ich zróżnicowanie (dynamika). Jeśli wykorzystywany jest 16 bitowy typ całkowitoliczbowy czy może dojść do sytuacji, że parametry iloczynu mogą dać wynik, który jest większy niż maksymalnie obsługiwany? Czy 8 bitów przeznaczonych na część ułamkową w zapisie stałopozycyjnym umożliwi poprawne znalezienie punktu przecięcia w każdej sytuacji, a jeśli nie to czy zwiększenie szerokości bitowej nie wpłynie negatywnie na inne metryki związane z tworzonym modelem? Jakie korzyści może przynieść zastosowanie reprezentacji zmiennoprzecinkowej, jeśli zadaniem układu jest głównie dokonywanie równolegle wielu operacji dodawania i mnożenia? Podjęcie decyzji i odpowiedzialność za nią spoczywa na projektancie.

---

<sup>v</sup>Przykład ten ilustruje poziom zaawansowania obsługi typu `half` przez Vivado HLS.

**Tab. 3.2:** Zestawienie typów danych wraz z estymacją wykorzystania poszczególnych zasobów oraz czasu wykonania (#) dla podstawowych operacji arytmetycznych. W przypadku pierwiastkowania wartości typu `half` utylizacja LUT oraz FF musiała zostać oszacowana ze względu na fakt, iż Vivado HLS włącza operacje implementujące to działanie bezpośrednio do hierarchii funkcji, w której zostało wywołane. Do sporządzenia niniejszego zestawienia użyto Vivado HLS 2017.4 dla modułu syntezowanego dla układu VC707 i działającego z częstotliwością 100 MHz

	Dodawanie				Mnożenie				Dzielenie				Pierwiastek			
	LUT	DSP	FF	#	LUT	DSP	FF	#	LUT	DSP	FF	#	LUT	DSP	FF	#
int	39	0	0	1	21	3	0	1	238	0	394	36	1416	0	317	5
ap_fixed<32, 20>	39	0	0	1	21	3	0	1	326	0	539	48	4720	0	1281	8
ap_fixed<24, 18>	24	0	0	1	40	1	0	1	224	0	370	34	2414	0	433	7
half	112	2	106	4	34	2	64	3	209	0	123	5	~950	3	~180	6
float	214	2	227	4	135	3	128	2	802	0	359	8	508	0	238	7
double	762	3	430	4	203	11	299	5	3253	0	1697	17	1912	0	1099	17

### 3.2.1 Układ procesorowy

Działanie najprostszego procesora polega na kolejnym [52]:

1. Pobraniu instrukcji z listy i jej odkodowaniu.
2. Wykonaniu operacji zgodnie z przekazanym opisem.
3. Zapisaniu wyniku.
4. Powrotu do kroku 1. ze zmienionym wskaźnikiem kolejnej instrukcji.

Dostępny dla użytkownika zbiór dozwolonych instrukcji zależy od twórcy danego procesora i określa on swobodę dotyczącą jego programowania. Dla przykładu, w przypadku gdy nie istnieje sprzętowa obsługa operacji pierwiastkowania można dokonać emulacji tej funkcjonalności poprzez odpowiednią sekwencję operacji elementarnych [53]. Przeważnie czas wykonania emulowanej funkcji będzie dłuższy niż gdyby procesor udostępniał dedykowaną do tego celu instrukcję. Z punktu widzenia wydajności ważne jest też, aby procesor był zdolny przetwarzać instrukcje z interwałem jak najbliższym 1 - wykorzystywana jest tutaj niezależność między poszczególnymi etapami przetwarzania instrukcji, która może być eksplorowana w Vivado HLS poprzez wykorzystanie dyrektywy PIPELINE - z możliwie najwyższą częstotliwością pracy.

Jednym z kierunków rozwoju architektur procesorów jest poszukiwanie takiego rozwiązania, które udostępniałoby rozsądny zestaw instrukcji<sup>vi</sup>, jednocześnie wymagając do jego implementacji jak najmniej zasobów sprzętowych. Warto w tym miejscu wspomnieć o projekcie *iDEA* [54], który został stworzony od podstaw z myślą o maksymalnym wykorzystaniu znajdujących się w układach FPGA firmy Xilinx bloków DSP. Autorzy zauważyl, że elastyczność oferowana przez bloki DSP pozwala na implementację większości operacji bezpośrednio z ich użyciem. W efekcie uzyskano 32-bitowy procesor<sup>vii</sup> mogący pracować z częstotliwością ok. 400 MHz przy tym wymagającym do jego implementacji jedynie 1 bloku DSP, 2 BRAM oraz 335 LUT<sup>viii</sup>. Porównanie tych wartości z oferowanymi przez obecne układy FPGA zasobami (tabela 3.1) prowadzi do wniosku, że procesorów takiego lub bardziej rozbudowanego typu na jednym układzie można byłoby umieścić setki - otrzymując tym samym sieć procesorów zdolnych wspólnie i stosunkowo szybko, jak na układy FPGA, przetwarzać dane.

Na bazie tej obserwacji do tej pory powstało wiele prototypowych systemów, których celem była realizacja śledzenia promieni w czasie rzeczywistym scen o dowolnej złożoności. Wartym wspomnienia jest tutaj cykl prac [55][56][57], gdzie autorzy szczegółowo przedstawiają, w jaki sposób udało im się osiągnąć zamierzony efekt poprzez bardzo dokładną analizę zależności między podsystemami: pamięci, procesorów oraz jednostek analizujących przecięcia promieni z geometrią i wykorzystywanych przez nie zasobów układu FPGA. Udoszczepniając użytkownikowi możliwość generowania promieni wtórnych do 4 pokolenia, programowanie sposobu oświetlenia powierzchni (użytkownik mógł tworzyć własne BSDF) oraz

<sup>vi</sup>Przez to sformułowanie należy rozumieć zachowanie balansu między możliwościami oddawanymi w ręce użytkownika a skomplikowaniem technologicznym.

<sup>vii</sup>Wszystkie dostępne instrukcje, za wyjątkiem mnożenia, przeprowadzane są przez omawiany procesor na 32-bitowych liczbach całkowitych. Jedynie parametry mnożenia są ograniczone do 16 bitów (wynik jest 32-bitowy), z uwagi na fakt, iż bloki DSP, w zależności od wersji, posiadają wbudowany moduł dokonujący iloczynu czynników 25(27)- i 18-bitowego w pojedynczym cyklu zegara.

<sup>viii</sup>Podane wartości dotyczą układu Virtex-6.

przetwarzania scen złożonych z milionów trójkątów rozwiązań to potrafiło generować obraz z szybkością co najmniej 1 klatki obrazu na sekundę w rozważanych przez twórców przypadkach (uwarunkowane jest to oczywiście poprzez rozdzielczość obrazu oraz złożoność sceny). Inne ciekawe rozwiązanie zostało zaproponowane w pracy [58]. Jak podnoszą sami autorzy ich system był prostszy w budowie niż ten, o którym wspomniano wcześniej, jednak eksplorował w znacznie większym stopniu wykorzystanie hierarchicznej sieci prostych procesorów arytmetycznych. Pojedynczy rdzeń (tych w układzie można było umieszczać dowolne ilości) składał się z 32 wątków zdolnych wykonywać podstawowe operacje na liczbach całkowitych oraz zmiennopozycyjnych jednak w obrębie rdzenia współdzieliły one między sobą jednostki odpowiedzialne za obliczanie iloczynów czy odwrotności liczby. Takie postępowanie było uzasadnione obserwacją, iż w kodzie programu operacje te są wykonywane odpowiednio rzadko i nie ma potrzeby, by każdy z wątków posiadał własne jednostki do przeprowadzania operacji arytmetycznych tego typu.

Cechą wspólną obu przedstawionych systemów śledzenia promieni jest to, że polegają one w znacznym stopniu na wydajności pojedynczego podsystemu przetwarzania instrukcji, który na dodatek musi być odpowiednio niewielki (używać niewiele zasobów FPGA), aby można było wspólnie przetwarzać wiele promieni. Co więcej ich twórcy podkreślają rolę języków opisu sprzętu w swobodnym kreowaniu sposobu przepływu danych w ich systemach.

### Projekt układu procesorowego z użyciem Vivado HLS

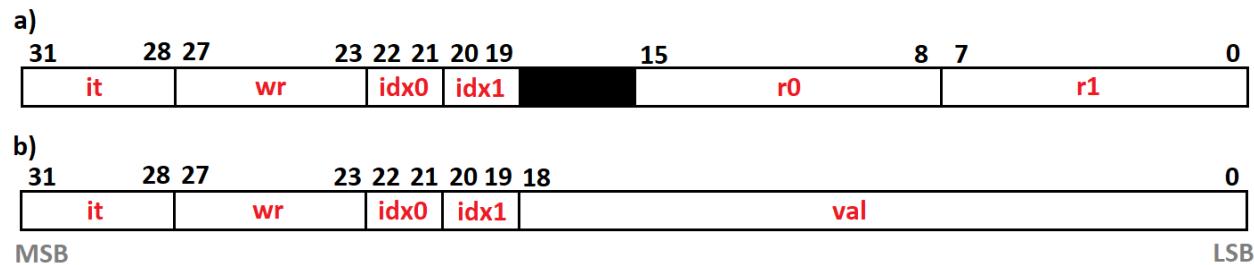
Niniejszy projekt układu procesorowego wykonano dla układu VC707 przy użyciu Vivado Design Suite 2017.2<sup>ix</sup>. Stworzony moduł IP zaprojektowano w taki sposób, aby jego uruchomienie z dostarczoną listą instrukcji oraz opcjonalnymi danymi zewnętrznymi skutkowało zapisem wartości do bufora ramki obrazu o podanych rozmiarach, stąd dla każdego piksela w pętli z zastosowaną dyrektywą DATAFLOW najpierw następuje wygenerowanie promienia o odpowiednim punkcie początkowym i kierunku, odpowiadającym rzutowaniu perspektywicznemu (`CreateRay()`), potem następuje przetworzenie ciągu instrukcji (`ProcessInstructions()`), zwieńczone zapisem obliczonej wartości (`PutColor()`) do bufora koloru. Dalszy opis będzie skoncentrowany tylko na zagadnieniach związanych z przetwarzaniem rozkazów przez `ProcessInstructions()`.

**Rejestry procesora i architektura instrukcji** Z uwagi na fakt, iż śledzenie promieni wymaga przetwarzania danych wektorowych podjęta została decyzja o tym by procesor miał do swojej dyspozycji 16 [0 : 15] wektorowych rejestrów wewnętrznych, z których każdy składa się z 4 32-bitowych elementów, co pozwoliłoby na przechowywanie większości potrzebnych danych (tablica rejestrów została całkowicie rozbita na poszczególne rejesty za pomocą `ARRAY_PARTITION`). Każda instrukcja natomiast składa się z 4 wykonywanych jednocześnie podinstrukcji, których wynikiem w przypadku działań arytmetycznych jest skalar, którego wartość może być zapisana do dowolnego rejestru wektorowego, jednakże indeks zapisu w rejestrze wektorowym jest jednakowy z indeksem podinstrukcji. Każda podinstrukcja jest opi-

---

<sup>ix</sup>Pliki projektu zostały umieszczone w repozytorium dostępnym pod adresem: [https://bitbucket.org/rtMasters/simpipeline/src/FINAL\\_SPU/](https://bitbucket.org/rtMasters/simpipeline/src/FINAL_SPU/).

sana za pomocą 32-bitowej wartości, której rozmieszczenie i znaczenie poszczególnych bitów, w zależności od typu podinstrukcji, zostało ukazane na poniższym schemacie:



Rys. 3.2: Rozmieszczenie bitów danych opisujących podinstrukcję w stworzonym procesorze

[31 : 28] - it - opisuje rodzaj podinstrukcji, która ma być wykonana,

[27 : 23] - **wr** - indeks rejestru wektorowego, do którego zostanie zapisany rezultat (indeks skalara w wektorze jest dedukowany na podstawie numeru podinstrukcji),

[22 : 21] - idx0 - indeks skalara pierwszego parametru (inne znaczenie w przypadku skoku warunkowego, patrz tabela 3.3),

[20 : 29] - `idx1` - indeks skalara drugiego parametru (inne znaczenie w przypadku skoku warunkowego, patrz tabela 3.3),

[15 : 8] - r0 - wartość liczbową ze znakiem, wskazującą na indeks rejestru wektorowego będącego pierwszym parametrem, jeśli wartość jest ujemna tzn. [15] = 1 wartość tego rejestru wzięta do obliczeń zostanie zanegowana,

[7 : 0] - r1 - wartość liczbową ze znakiem, wskazującą na indeks rejestru wektorowego będącego drugim parametrem, jeśli wartość jest ujemna tzn. [7] = 1 wartość tego rejestru wzięta do obliczeń zostanie zanegowana,

[18 : 0] - val - stała podawana przez użytkownika

Stworzony procesor potrafi przetwarzać 16 instrukcji zebranych i opisanych w tabeli 3.3. Choć przestawiony zbiór instrukcji jest niewielki, procesor jest w stanie dokonywać obliczeń przecięcia promieni z obiektami oraz dokonywać obliczeń koloru danego piksela (zwłaszcza po uwzględnieniu obecności instrukcji **PRE\_S** oraz **PRE\_D**, które dokonują inicjalizujących manipulacji na parametrach będących wstępem do obliczeń przybliżonych metodą Newtona-Raphsona).

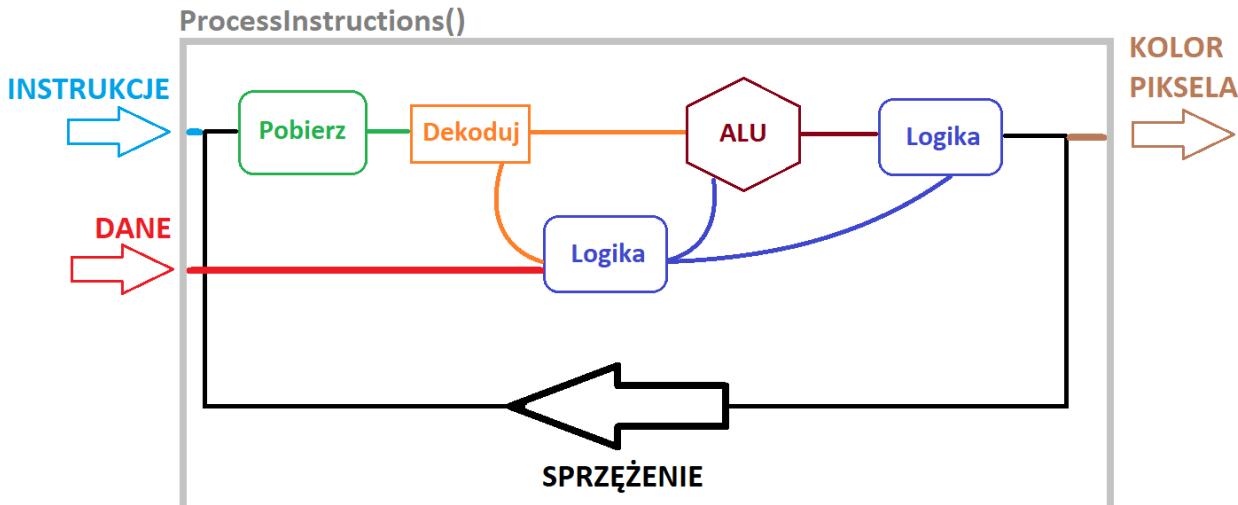
**Tab. 3.3:** Architektura instrukcji stworzonego układu procesora. Za każdym razem, gdy w tabeli używane jest **reg** następuje dostęp do tablicy wektorowych rejestrów wewnętrznych, zaś gdy **mem** dochodzi do dostępu do pamięci zewnętrznej. Warunki nakładane na instrukcje ( $i \in \{0, 1, 2, 3\}$ ) dotyczą tego, dla jakich numerów podinstrukcji można je stosować

Instrukcja	it	Operacja	Uwagi
<b>NOP</b>	<i>0000</i>	-	Instrukcja pusta
<b>LC</b>	<i>0001</i>	$\text{reg}[\text{wr}][i] = \text{val}$	
<b>LV</b>	<i>0010</i>	$\text{reg}[\text{wr}] = \text{mem}[\text{val}]$	$i \in \{2, 3\}$ Wykonuje się przed innymi podinstrukcjami. Efekt odczytu może zostać użyty przez inną podinstrukcję w tej samej instrukcji
<b>LVI</b>	<i>0011</i>	$\text{reg}[\text{wr}] = \text{mem}[\text{reg}[\text{r0}][\text{idx0}]]$	$i = 3$ Pozostałe podinstrukcje w danej instrukcji nie zostaną wykonane
<b>MOV</b>	<i>0100</i>	$\text{reg}[\text{wr}][i] = \text{reg}[\text{r0}][\text{idx0}]$	
<b>ADD</b>	<i>0101</i>	$\text{reg}[\text{wr}][i] = \text{reg}[\text{r0}][\text{idx0}] + \text{reg}[\text{r1}][\text{idx1}]$	
<b>SHR</b>	<i>0110</i>	$\text{reg}[\text{wr}][i] = \text{reg}[\text{r0}][\text{idx0}] >> \text{r1}$	
<b>MUL</b>	<i>0111</i>	$\text{reg}[\text{wr}][i] = \text{reg}[\text{r0}][\text{idx0}] \cdot \text{reg}[\text{r1}][\text{idx1}]$	
<b>DOT</b>	<i>1000</i>	$\text{reg}[\text{wr}][i] = \text{reg}[\text{r0}] \cdot \text{reg}[\text{r1}]$	Iloczyn skalarny rejestrów wektorowych zredukowanych do ich 3 pierwszych elementów
<b>ADOT</b>	<i>1001</i>	$\text{reg}[\text{wr}][i] += \text{reg}[\text{r0}] \cdot \text{reg}[\text{r1}]$	Iloczyn skalarny rejestrów wektorowych zredukowanych do ich 3 pierwszych elementów z akumulacją wyniku
<b>MADD</b>	<i>1010</i>	$\text{reg}[\text{wr}][i] += \text{reg}[\text{r0}][\text{idx0}] \cdot \text{reg}[\text{r1}][\text{idx1}]$	Iloczyn z akumulacją wyniku
<b>PRE_S</b>	<i>1011</i>	$\text{reg}[\text{wr}][0] = \text{fun1}(\text{reg}[\text{r0}][\text{idx0}])$ $\text{reg}[\text{wr}][1] = \text{fun2}(\text{reg}[\text{r0}][\text{idx0}])$	Instrukcja obliczająca wstępne parametry potrzebne, aby iteracyjnie obliczyć odwrotność pierwiastka kwadratowego
<b>PRE_D</b>	<i>1100</i>	$\text{reg}[\text{wr}][0] = \text{fun1}(\text{reg}[\text{r0}][\text{idx1}])$ $\text{reg}[\text{wr}][1] = \text{fun2}(\text{reg}[\text{r0}][\text{idx0}])$	Instrukcja obliczająca wstępne parametry potrzebne, aby iteracyjnie obliczyć iloraz dwóch liczb
<b>JMP</b>	<i>1101</i>		$i = 3$ Opuszcza wykonanie <code>unsigned(val)</code> kolejnych instrukcji

**Tab. 3.3:** Kontynuacja

<b>JMP_IF</b>	<i>1110</i>	idx0 == 0: res[idx1] == 0 idx0 == 1: res[idx1] > 0	i = 3 <b>res</b> - tablica przechowująca wyniki poprzedzających ją podinstrukcji Opuszcza wykonanie <b>unsigned(val)</b> kolejnych instrukcji, jeśli warunek uzależniony od <b>idx0</b> i wyniku konkretnej podinstrukcji wskazywanej przez <b>idx1</b> jest spełniony
<b>JMP_IFN</b>	<i>1111</i>	idx0 == 0: res[idx1] == 0 idx0 == 1: res[idx1] > 0	i = 3 <b>res</b> - tablica przechowująca wyniki poprzedzających ją podinstrukcji Opuszcza wykonanie <b>unsigned(val)</b> kolejnych instrukcji, jeśli warunek uzależniony od <b>idx0</b> i wyniku konkretnej podinstrukcji wskazywanej przez <b>idx1</b> nie jest spełniony

**Działanie procesora** Schemat przetwarzania został ukazany na rysunku 3.3.



**Rys. 3.3:** Potok przetwarzania w stworzonym procesorze. Wejście stanowi lista instrukcji oraz dane zewnętrzne. W pętli zoptymalizowanej dyrektywą PIPELINE następuje pobranie instrukcji, jej odkodowanie a następnie przetworzenie przez układy logiczne i/lub arytmetyczny (ALU). Wyniki są zapisywane do odpowiednich rejestrów i przetwarzana jest kolejna instrukcja. Po wykonaniu wszystkich instrukcji pierwszy z rejestrów wektorowych jest przyjmowany za ten, którego zawartość stanowi kolor piksela

Widać na nim (oraz co wynika z analizy dostępnych instrukcji), że jest to układ nie-skomplikowany, pozbawiony pamięci podręcznej czy możliwości zwrotnego asynchronicznego zapisu wartości poza układ za to z licznymi ograniczeniami (opisanymi w uwagach w tabeli 3.3). Rzec w tym, że Vivado HLS nie umożliwiło jego dalszej komplikacji i to przez co najmniej kilka przyczyn.

1. Z uwagi na występujące w procesorach sprzężenie zwrotne między rejestrami, Vivado HLS dobiera taki interwał pomiędzy przetwarzaniem kolejnych instrukcji, aby w każdym przypadku zapis wyniku do rejestru następował przed jego możliwym odczytem związanym z przetwarzaniem kolejnej instrukcji. Teoretycznie zachowanie się Vivado HLS w przypadku wykrycia przez nie zależności między danymi można modyfikować dzięki zastosowaniu dyrektywy **DEPENDENCE**, jednak w żaden sposób nie udało się tego wymusić, a czas syntezy HLS znacznie wzrastał (z kilku minut do kilku godzin). Gdyby ta optymalizacja działała, należałoby z teoretycznymi zależnościami między instrukcjami radzić sobie poprzez odpowiednie napisanie kodu (zmieniając kolejność instrukcji bez wpływu na algorytm czy wstawiając instrukcje puste NOP) jednak nie istniałoby sztywne ograniczenie szybkości przetwarzania.

Poleganie jedynie na statycznym osądzie HLS co do zależności między danymi wymusiło, aby operacje wykonywane przez jednostkę arytmetyczną (ALU) były wykonywane jak najszybciej. Zgodnie z informacjami zawartymi w tabeli 3.2 nadawały się do tego jedynie typy liczbowe stałopozycyjne, dla których czas wykonania operacji dodawania i mnożenia zawiera się w jednym cyklu zegara. Przyjęto zatem, że elementy skalarnie rejestrów wektorowych będą typu **ap\_fixed<32, 16>**, zaś wartości, które są wczyty-

wane bezpośrednio z przekazanej podinstrukcji (`val`) `ap_fixed<19, 13>`. Mimo takich zabiegów Vivado HLS mógł jedynie zaoferować interwał równy 3 (opóźnienie iteracji pętli wyniosło 12) przy częstotliwości zegara 100 MHz.

2. Z podobnych przyczyn jak powyżej nie udało się dokonać implementacji skoków między partiami wykonywanego programu (skoki są ważne wszędzie tam, gdzie dochodzi do warunkowego wykonania określonej porcji instrukcji). W większości procesorów, gdy następuje konieczność wykonania skoku następuje wyczyszczenie potoku przetwarzania (tzn. wszystkie instrukcje, które rozpoczęły być przetwarzane zanim znaleziono polecenie skoku są dyskredytowane, a wskaźnik na kolejną instrukcję jest ustalony w nowym miejscu). W prezentowanym rozwiązaniu teoretycznie istnieją polecenia dokonujące skoku `JMP`, `JMP_IF`, `JMP_IFN` jednak ich efektem działania jest tylko to, że wyniki przetwarzanych `unsigned(val)` instrukcji nie są zapisywane (nie można też przez to cofać się w wykonaniu instrukcji, co uniemożliwia tworzenie pętli). W rezultacie każda dodana instrukcja, na którą składają się 4 podinstrukcje, dodaje stały przyczynek do czasu wykonania równy interwałowi (3 cykle zegara dla każdego z rozpatrywanych promieni pierwotnych).
3. Implementacja powyższego procesora w strukturze układu VC707 wymaga stosunkowo dużej ilości zasobów (tabela 3.4).

**Tab. 3.4:** Wykorzystanie zasobów sprzętowych przez procesor instrukcji (`ProcessInstructions()`). Dolny wiersz przedstawia procentowe wykorzystanie zasobów układu VC707 raportowane po syntezie HLS

<b>BRAM</b>	<b>DSP</b>	<b>FF</b>	<b>LUT</b>
0	64	41457	17404
0%	2,29%	6,83%	5,73%

W dużej mierze wynika to z przyjętego przetwarzania 4 podinstrukcji jednocześnie, gdzie każda z nich może odwoływać się do dwóch dowolnych rejestrów wektorowych. Układy BRAM, które implementują funkcjonalność tablic umożliwiają do dwóch operacji dostępu w jednym cyklu zegara. Mogłoby się wydawać, że w takim razie pamięć związaną z rejestrami można podzielić na 4 i wtedy każda podinstrukcja będzie wykorzystywała dostępne dla siebie 2 operacje odczytu. Nie jest to prawda, gdyż nieznany jest schemat dostępu do danych (instrukcje wymagane przez użytkownika mogą odwoływać się dowolnie do rejestrów) stąd konieczność użycia kompletnego podziału tablicy (dyrektywa `ARRAY_PARTITION`) rejestrów na poszczególne składowe - okupione jest to znacznym wzrostem zapotrzebowania na LUT i FF. Pozornym rozwiązaniem tego problemu, byłoby przejście na tradycyjny model przetwarzania tylko jednego rozkazu przez jedną instrukcję. Takie działanie wydłużałoby tylko listę rozkazów lecz powyższe 2 problemy by pozostały.

Do tego dochodzi również fakt, iż każda instrukcja dokonuje alokacji zasobów na własne potrzeby. Nawet jeśli 2 instrukcje wykonują częściowo podobne operacje, odpowiednie zasoby nie są współdzielone między nimi. Przykładowo, jeśli ze zbioru dostępnych

instrukcji usunięte zostaną MUL oraz MADD, które wykonują podobne operacje co DOT oraz ADOT ilość modułów DSP wymaganych do implementacji procesora zmniejszy się o 16. Wynika z tego, że Vivado HLS nie jest w stanie dokonać zaawansowanych optymalizacji, potrzebnych do stworzenia procesora, prowadzących do wielokrotnego użycia zasobów na potrzeby kilku instrukcji w zaimplementowanym układzie<sup>x</sup>.

W celu sprawdzenia poprawności działania układu procesorowego, jakości wyników oraz estymacji czasu wykonania w typowym zastosowaniu stworzono prosty program testowy, którego celem było znalezienie odległości od obserwatora  $O$  znajdującego się w początku układu współrzędnych  $\vec{p}_o = [0; 0; 0]$  ustawionego w kierunku  $\vec{d}_p = [0; 0; -1]$  do najbliższego punktu znajdującej się przed nim sfery o promieniu  $R = 1$  i środku  $\vec{p}_s = [0; 0; -2, 5]$ . W ten sposób wynikiem, który się otrzymuje jest tzw. *mapa głębokości*<sup>xi</sup>. Transformacja sfery w przestrzeni była opisana poprzez macierz jej przekształcenia  $\mathbf{M}$ , a obliczeń dokonywano w przestrzeni tego obiektu<sup>xii</sup>. Wymagało to, aby promienie  $\vec{r} = \vec{p}_o + t\vec{d}$  były transformowane za pomocą macierzy odwrotnej  $\mathbf{M}^{-1}$  ( $\mathbf{MM}^{-1} = \mathbf{I}$ ) do przestrzeni sfery  $\vec{r}' = \mathbf{M}^{-1}\vec{r} = \vec{p}'_o + t'\vec{d}'$ .

Rozwiążanie równania przecięcia promienia ze sferą wymaga rozwiązania równania kwadratowego, a w szczególności obliczenia wyróżnika  $\sqrt{\Delta}$ . W tym celu posłużono się dostępną instrukcją PRE\_S, po której wywołano dwukrotnie ciąg instrukcji obliczający dwie iteracje metodą Newtona-Raphsona. Następnie obliczane były odległość od obserwatora  $t'$ <sup>xiii</sup> w przestrzeni obiektu, normalna w punkcie zderzenia  $\vec{n}'$  oraz punkt przecięcia w przestrzeni obiektu  $\vec{h}'_p$ .

Wszystkie wspomniane operacje udało się opisać za pomocą 20 instrukcji, przy czym tylko 43 podinstrukcje z 80 były różne od NOP. Związane jest to z faktem, iż iteracyjne obliczanie pierwiastka kwadratowego wymagało w tym przypadku w sumie 10 instrukcji, a każda z nich wykorzystuje nie więcej niż 2 podinstrukcje.

Program ten wykorzystano do stworzenia mapy głębokości o rozdzielcości 100 x 100 pikseli ( $10^4$  punktów obrazu), która została zaprezentowana na rysunku 3.4. Ogólny przebieg zależności jest zbieżny z oczekiwany - sfera w najbliższym dla obserwatora miejscu jest oddalona o  $x = 1, 5$  jednostki, a w najdalszym  $x = 2, 5$  (gdy promień jest niemal styczny do powierzchni sfery). Niepokojąca jest jednak zauważalna nieciągłość wartości w okolicach  $x \approx 1, 75$  od obserwatora, a związana z przyjętą precyzją obliczeń (ap\_fixed<32, 16>) i wy-

<sup>x</sup>Wniosek ten jest właściwy dla tego konkretnego projektu - w wielu mniej skomplikowanych przypadkach Vivado HLS potrafi w bardziej oszczędny sposób gospodarować zasobami układu FPGA.

<sup>xi</sup>Opisywany dalej algorytm został zapisany jako ciąg instrukcji tekstowych w pliku src/CodeParser/program.asm znajdujący się w repozytorium projektu. Instrukcje te następnie są transformowane do kodu bajtowego interpretowalnego przez procesor z użyciem src/CodeParser/PYCodeParser.py.

<sup>xii</sup>Postępowanie takie umożliwia deformację obiektu za pomocą odpowiedniej macierzy. Dodatkowo równania przecięcia, które należy rozwiązać, aby uzyskać wartość  $t$ , będącą odległością między obserwatorem a obiektem wzduż promienia, ulega uproszczeniu z postaci:

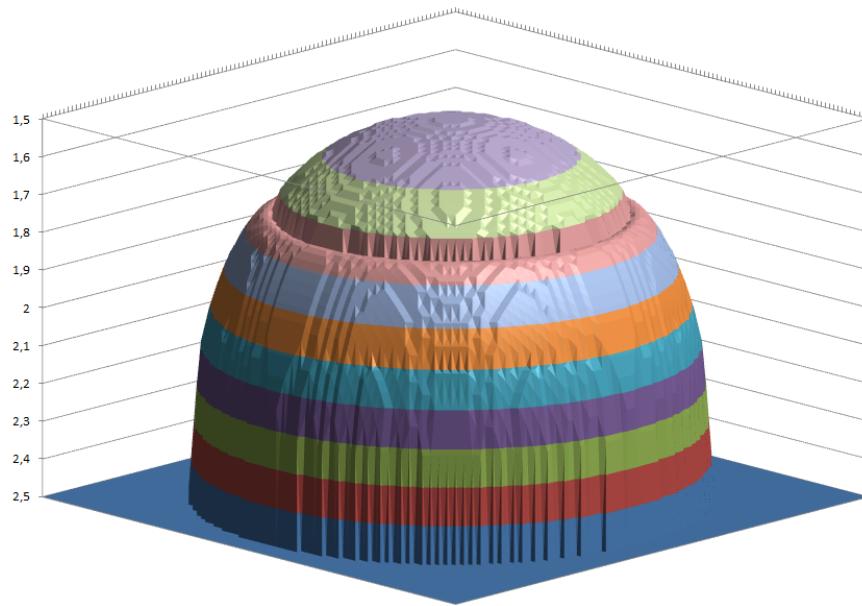
$$\sum_{i=0}^2 (r_i - p_{s_i})^2 = R^2$$

do:

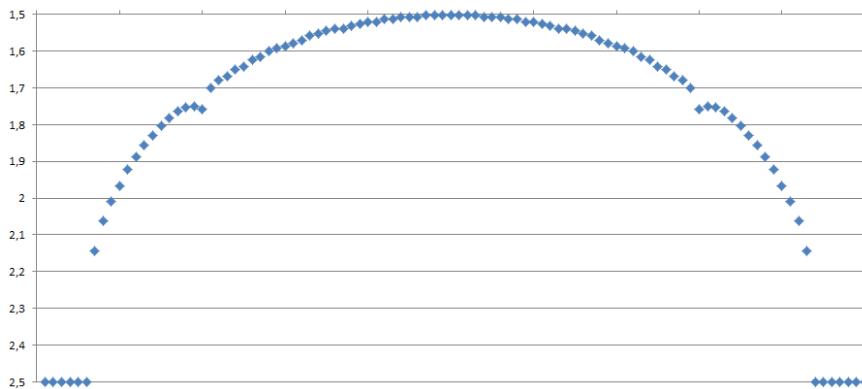
$$\sum_{i=0}^2 r'^2_i = 1.$$

<sup>xiii</sup>Z uwagi na fakt, iż kula była poddana jedynie translacji  $t = t'$ .

korzystaniem obliczeń przybliżonych użytych do obliczenia wyróżnika  $\sqrt{\Delta}$  (na rysunku 3.5 ukazany został jeden z profili odległości przechodzący przez punkt najbliższego obserwatorowi).



**Rys. 3.4:** Mapa głębokości wykonana na podstawie obliczonych przez procesor danych. Sfera o promieniu  $R = 1$  i środku w punkcie  $p_s = [0; 0; -2,5]$  w najbliższym obserwatorowi miejscu, zgodnie z oczekiwaniemi jest oddalona o wartość  $x = 1,5$



**Rys. 3.5:** Profil odległości sfery od obserwatora przechodzący przez punkt najbliższego obserwatorowi. Widoczne są nieciągłości dla odległości  $x \approx 1,75$  wynikające z precyzji obliczeń. Uzyskany profil nie jest kolisty ze względu na zniekształcenie perspektywy obserwatora

Zgodnie z tym, co powiedziano wcześniej stworzony program składał się z 20 instrukcji, gdzie każda z nich jest wykonywana z interwałem równym 3. Wykonanie takiego programu dla wspomnianej rozdzielczości ( $10^4$  pikseli) z zegarem 100 MHz (1 cykl zegara odpowiada okresowi 10 ns) zajmuje:

$$20 \cdot 3 \cdot 10^4 \cdot 10 \text{ ns} = 6 \text{ ms} \quad (3.2.1)$$

z użyciem jednego procesora. Jednakże:

1. Przedstawiony program dokonuje tylko testu z jednym obiektem. Dodanie kolejnych obiektów wiąże się z dalszymi liniami kodu, które wydłużają liniowo czas wykonania.

2. Obliczenia, jak pokazano nie charakteryzują się wysoką precyzją. Zastosowanie liczb stałopozycyjnych (wymuszone faktem, iż dyrektywa DEPENDENCE nie działała zgodnie z oczekiwaniem) nakłada warunki m. in. na skalę sceny tzn. wielkości obiektów i odległości między nimi musiałyby być porównywalne.
3. Należy zwrócić uwagę, iż program ten nie dokonuje obliczenia koloru obiektu w wyniku oddziaływania ze źródłami światła oraz rozprosień (odbicia i załamania światła).
4. Obliczenia zostały wykonane dla zaledwie ok. 1% rozdzielczości docelowej.

Nawet jeśli w strukturze układu FPGA udałoby się realnie umieścić 10 takich procesorów przetwarzających odpowiednią partię sceny i każdy z nich działałby z częstotliwością 150 MHz (taką wartość udało się uzyskać po implementacji procesora w układzie VC707), otrzymana wydajność nie pozwoliłaby na osiągnięcie zamierzonych celów związanych ze stworzeniem systemu śledzenia promieni działającego w czasie rzeczywistym. Oczywista porażka w tym przypadku wynika z poświęcenia części elastyczności, którą zapewniają języki opisu sprzętu, na rzecz łatwego w interpretacji zapisu algorytmicznego poddawanego translacji do RTL za pomocą Vivado HLS. Zgodnie z tym co przedstawiono, Vivado HLS nie nadaje się do budowy modułów wymagających dużej kontroli nad sposobem przepływu danych oraz wykonaniem algorytmu. Użycie kodu C/C++ nie pozwala również na stworzenie w jednym module zadań wykonywanych asynchronicznie - te wymagane są do implementacji pamięci podrzędnej czy struktur akcelerujących przetwarzanie przecięć promieni z geometrią sceny.

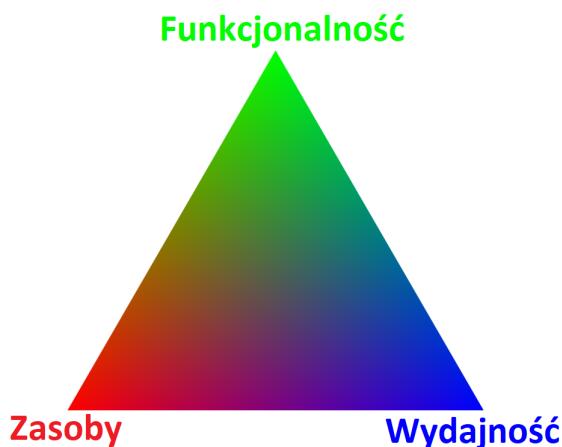
Kończąc opis tego rozwiązania, należy mieć na względzie, iż możliwość stworzenia za pomocą Vivado HLS szybkiego i małego (w sensie utylizacji zasobów układu FPGA) procesora geometrii była przez długi czas główną koncepcją, na jakiej oparty miał być system generowania obrazów metodą śledzenia promieni. Przedstawiony tutaj procesor jest najszybszym ze stworzonych, aczkolwiek wniosek ten jest prawdziwy tak długo jak do syntezy wykorzystywana jest wersja 2017.2 Vivado Design Suite. Ten sam procesor poddany syntezie z użyciem wersji 2016.2 tego narzędzia z syntezą o takich samych parametrach tworzył główną pętlę przetwarzania o interwalejącym 5, zatem wyraźnie wolniejszą. Z drugiej jednak strony nowsza wersja nie potrafiła poprawnie planować wykonania zadań w regionie objętym dyrektywą DATAFLOW (przez co jej wykorzystanie nie dawało żadnych korzyści w postaci zmniejszenia czasu wykonania) oraz nie umożliwiała wykonania testów tworzonego modułu w środowisku C/C++<sup>xiv</sup>. Z drugiej strony nowsze wersje środowiska tzn. 2017.3 i nowsze w ogóle nie potrafią przeprowadzić syntezy przygotowanego kodu (głównie ze względu na zmianę interfejsów funkcji odpowiedzialnych za obsługę liczb stałopozycyjnych, na których projekt opiera swoje działanie). Przytoczone obserwacje pokazują, że Vivado HLS ulega zmianom, nie zawsze tylko wszystkie zmiany w nim zachodzące są pożądane i aktualnie istnieje duży problem związany z wyborem optymalnej jego wersji.

### 3.2.2 Wyspecjalizowany akcelerator o ustalonej funkcjonalności

Wiedza oraz doświadczenie zdobyte podczas prac nad stworzeniem modułu śledzenia promieni opartego o przetwarzanie zestawu instrukcji miały się jednak przydać podczas pro-

<sup>xiv</sup> Wymuszało to postępowanie, takie że synteza HLS była przeprowadzana w wersji 2017.2 narzędzia, zaś testy wykonywano w wersji 2016.2.

jektowania zupełnie innego rozwiązania, opartego o zestaw funkcjonalności na stałe zakodowany w module. W porównaniu z procesorem, rozwiązanie to ma taką zaletę, że już na etapie syntezы Vivado HLS wie, jakie działania składają się na wykonanie algorytmu, a przez to jest w stanie optymalnie szeregować wykonanie operacji dla jak największej wydajności - o ile tylko program jest napisany wystarczająco starannie tzn. twórca ma świadomość, że to, co jest reprezentowane przez linie kodu będzie docelowo stanowić alokację fizycznych zasobów sprzętowych układu FPGA. Stąd od razu widać, iż dodanie każdej nowej funkcjonalności do tworzonego akceleratora, będzie wiązało się z wykorzystaniem pewnej części układu. Są to dwa wzajemnie wykluczające się czynniki warunkujące rozwój tego typu akceleratora. Trzecim jest wydajność. Omówione na początku tego rozdziału założenia narzucają, by interwał między obliczeniem koloru kolejnych pikseli był nie większy niż 10 cykli zegara przy częstotliwości pracy 100 MHz. Taki efekt można uzyskać jedynie wtedy, gdy stworzony algorytm umożliwia zastosowanie dyrektyw PIPELINE i/lub DATAFLOW w odpowiednich miejscach programu. Im wymagania względem interwału pracy bloku kodu objętego dyrektywą PIPELINE są wyższe (tzn. interwał dąży do jedności), tym Vivado HLS ma mniejsze możliwości w zakresie użycia tych samych zasobów na kolejnych etapach przetwarzania danych, w efekcie wzrasta utylizacja elementów układu FPGA.

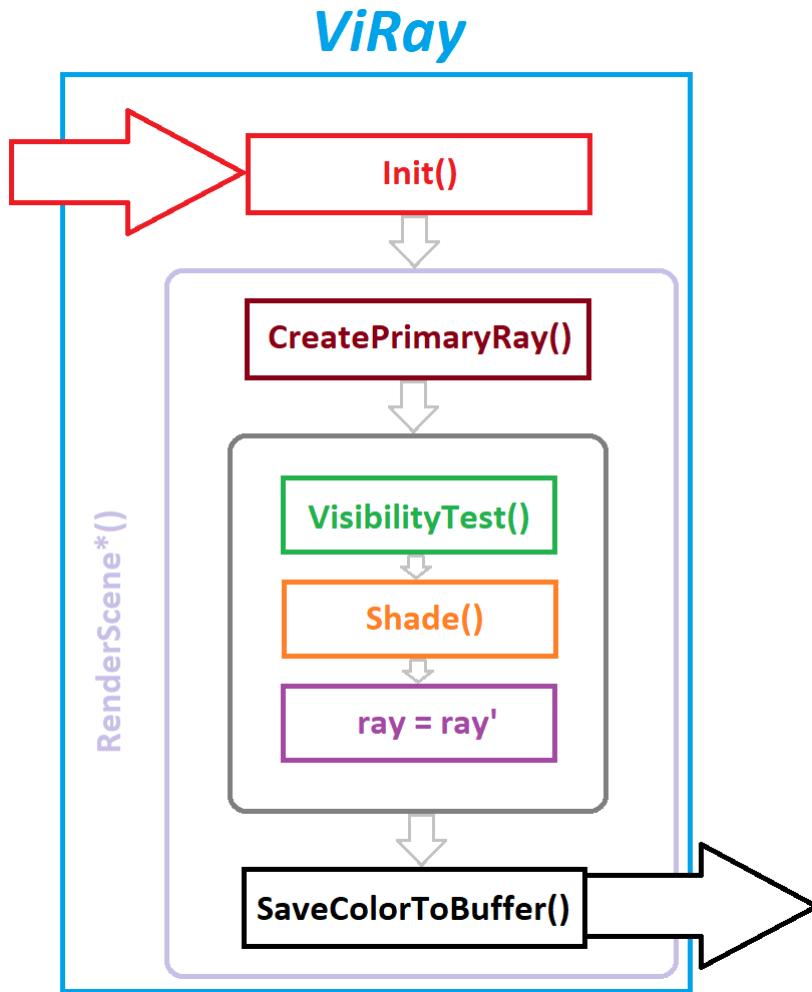


**Rys. 3.6:** Triada zależności między sprzecznymi ze sobą charakterystykami akceleratora: funkcjonalnością, wydajnością oraz utylizacją zasobów

W wyniku przeprowadzonych prac badawczo-rozwojowych powstał ViRAY, czyli modułarny system generowania obrazów metodą śledzenia promieni, implementowalny i działający w czasie rzeczywistym z użyciem układów FPGA firmy Xilinx. Jego podstawowa wersja została zoptymalizowana pod kątem układu Kintex UltraScale+ znajdującego się na płytce ewaluacyjnej KCU116, a do jego stworzenia wykorzystano Vivado Design Suite w wersji 2017.4<sup>xvxvi</sup>.

<sup>xv</sup>Nazwa systemu jest skrótwcem od ang. *Virtex ray tracer*, który został wymyślony jeszcze na samym początku prac, kiedy do dyspozycji Autora był układ Virtex 7.

<sup>xvi</sup>Pliki projektu dostępne są pod adresem: [https://bitbucket.org/rtMasters/ffcore/src/Oren-Nayar\\_correct/](https://bitbucket.org/rtMasters/ffcore/src/Oren-Nayar_correct/).



**Rys. 3.7:** Ogólna zasada działania modułu ViRAY. Dane wejściowe, które reprezentują poszczególne elementy tworzonej wirtualnej sceny są interpretowane przez funkcję `Init()` i przekazywane do głównej pętli (`RenderScene*()`), w której dokonywane są właściwe obliczenia. Dla każdego promienia pierwotnego, generowanego przez `CreatePrimaryRay()`, wykonywany jest test widoczności (`VisibilityTest()`). Jeżeli w kierunku propagacji promienia został znaleziony obiekt, przeprowadzane są obliczenia koloru piksela (`Shade()`) oraz tworzony jest odbity od powierzchni obiektu promień wtórny (`ray = ray'`), który zostanie użyty w kolejnej iteracji, celem obliczenia odbić. Po obliczeniu finalnego koloru piksela jest on zapisywany w buforze ramki (`SaveColorToBuffer()`). Cały proces zawarty w `RanderScene*()` jest kontynuowany dla wszystkich pikseli obrazu

Ogólna zasada działania stworzonego modułu z użyciem Vivado HLS została przedstawiona na rysunku 3.7. Na początku w funkcji głównej (`Init()`) dokonywana jest inicjalizacja wewnętrznych struktur danych na podstawie parametrów dostarczonych do modułu, po czym następuje przejście do głównej pętli przetwarzania (`RenderScene*()`), w której prowadzane są obliczenia dotyczące wszystkich żądanych pikseli obrazu zgodnie z dostarczonym opisem sceny. Tutaj dla każdego nowo obliczanego piksela, tworzony jest promień pierwotny (`CreatePrimaryRay()`), który jest następnie testowany (`VisibilityTest()`) na wypadek przeciecia z obiektemi. Jeśli zderzenie nastąpi, obliczany jest kolor piksela (`Shade()`) oraz generowany jest rozproszony promień wtórny względem pierwotnego (`ray = ray'`).

Ten staje się w kolejnej iteracji promieniem odpowiedzialnym za dostarczenie informacji o świetle rozproszonym. Ilość iteracji, a przez to głębokość śledzenia promieni związanych z kolorem danego piksela jest konfigurowalna. Finalny kolor piksela opisany za pomocą trzech składowych czerwonej, zielonej i niebieskiej (RGB) jest następnie zapisywany w buforze ramki (`SaveColorToBuffer()`).

Modularność ViRAY'a powoduje, że wiele kluczowych podsystemów biorących udział w tworzeniu finalnego obrazu może zostać ustalone za pomocą jednego pliku konfiguracyjnego `typedefs.h`. Z jego pomocą można m. in.:

- określić pulę rodzajów i maksymalną ilość obiektów możliwych do renderowania przez system,
- zmienić interwał między kolejnymi pikselami,
- dostosować możliwości podsystemu odpowiedzialnego za teksturowanie obiektów lub całkowicie go wyłączyć,
- wybrać, które z zaawansowanych modeli oświetlenia mają być dostępne w systemie,
- usprawnić szybkość wykonania w środowisku testowym.

Każdy parametr jest osobną dyrektywą preprocesora (`#define`), a najważniejsze z nich zostały zebrane i opisane w tabeli 3.5. Koncepcja modularności sprawia z jednej strony, iż kod programu jest dłuższy i bardziej skomplikowany poprzez zastosowanie konstrukcji typu:

```
1 #ifdef DIREKTYWA  
2     a = 2;  
3 #else  
4     a = 0.5;  
5 #endif
```

jednak posiada niezrównaną zaletę, jaką jest możliwość łatwego dostosowania parametrów modułu do aktualnych potrzeb. Pozwala to na eksplorację najbardziej optymalnych rozwiązań pod kątem tego, co zilustrowane zostało na rysunku 3.6. Na dodatek pozwala zastosować mniej wymagające algorytmy, jeśli implementacja napotyka problemy związane z wytworzeniem odpowiedniej sieci połączeń. Jest to też rozwiązanie przyszłościowe - dysponując nowszą generacją układów FPGA będzie można szybko ulepszyć charakterystykę systemu np. zmniejszając interwał czy zwiększając głębokość śledzenia promieni<sup>xvii</sup>.

---

<sup>xvii</sup> Wszelkie obrazy ukazujące aspekty działania ViRAY'a, jeśli nie podano inaczej, w tym podrozdziale pochodzą z symulacji działania rzeczywistego modułu.

**Tab. 3.5:** Opis najważniejszych parametrów sterujących zachowaniem modułu VIRAY zdefiniowanych w pliku `typedefs.h`. Wartości domyślne odpowiadają finalnej, zoptymalizowanej wersji projektu implementowalnego na płytce ewaluacyjnej KCU116 z użyciem Vivado Design Suite 2017.4

Definicja	Wartość domyślna	Opis
<code>UC_OPERATION</code>	(niezdefiniowane)	Musi zostać zdefiniowane w przypadku wykorzystywania źródeł systemu śledzenia promieni przez mikrokontroler (Microblaze)
<code>DESIRED_INNER_LOOP_II</code>	8	Żądana przez użytkownika ilość cykli zegara (interwał) pomiędzy obliczeniem kolejnych pikseli obrazu.
<code>USE_FLOAT</code>	(zdefiniowane)	Definiuje typ danych używany w trakcie obliczeń na: <code>typedef float myType</code>
<code>CORE_BIAS</code>	( <code>myType(0.001)</code> )	Niewielka dodatnia wartość rzeczywista pozwalająca uniknąć artefaktów wizualnych związanych z fałszywie pozytywnym wynikiem przecięcia, a wynikających ze skończonej dokładności obliczeń użytego typu danych
<code>MAX_DISTANCE</code>	( <code>myType(1000000)</code> )	Określa wartość przyjmowaną przez system śledzenia promieni za nieskończoność
<code>SIMPLE_OBJECT_TRANSFORM_ENABLE</code>	(zdefiniowane)	Jeśli zdefiniowane, system umożliwia jedynie uproszczoną manipulację obiektami w przestrzeni opartą o wartość skali oraz przesunięcia względem początku globalnego układu współrzędnych. W przeciwnym razie realizowane są transformacje oparte o przekształcenia macierzowe

Tab. 3.5: Kontynuacja

Definicja	Wartość domyślna	Opis
<b>SELF_RESTART_ENABLE</b>	(niezdefiniowane)	Jeśli zdefiniowane, raz uruchomiony układ będzie generował obrazy w nieskończoność. Raz odczytane dane tekstur są przechowywane - redukuje to czas potrzebny na inicjalizację pamięci tekstur (o ok. 65500 cykli), jednak uniemożliwia animowanie danych tekstury.
<b>PIXEL_COLOR_CONVERSION_ENABLE</b>	(zdefiniowane)	Dokonuje transformacji zapisu koloru do bufora ramki w taki sposób, aby możliwe było jego bezpośrednie użycie na układzie KCU116. Aby uzyskać poprawne kolory w środowisku testowym HLS, wymagane jest wyłączenie tej opcji
<b>RENDER_DATAFLOW_ENABLE</b>	(zdefiniowane)	Jeśli zdefiniowane, dokonuje implementacji głównej pętli przetwarzania z użyciem zapisu koloru pikseli do tymczasowego bufora. W przeciwnym wypadku piksele zapisywane są w trybie ciągłym do bufora ramki.
<b>AGGRESSIVE_AREA_OPTIMIZATION_ENABLE</b>	(niezdefiniowane)	Umożliwia wykorzystanie typu <code>half</code> w obliczeniach niewymagających wysokiej precyzji
<b>SPHERE_OBJECT_ENABLE</b>	(zdefiniowane)	Dodaje możliwość użycia sfer w scenie
<b>PLANE_OBJECT_ENABLE</b>	(zdefiniowane)	Dodaje możliwość użycia płaszczyzn w scenie
<b>DISK_OBJECT_ENABLE</b>	(zdefiniowane)	Dodaje możliwość użycia dysków w scenie
<b>SQUARE_OBJECT_ENABLE</b>	(zdefiniowane)	Dodaje możliwość użycia kwadratów w scenie
<b>CYLINDER_OBJECT_ENABLE</b>	(zdefiniowane)	Dodaje możliwość użycia walców w scenie
<b>CUBE_OBJECT_ENABLE</b>	(niezdefiniowane)	Dodaje możliwość użycia sześciąników w scenie
<b>CONE_OBJECT_ENABLE</b>	(zdefiniowane)	Dodaje możliwość użycia stożków w scenie
<b>AMBIENT_COLOR_ENABLE</b>	(zdefiniowane)	Światło otoczenia jako element oświetlenia powierzchni

Tab. 3.5: Kontynuacja

Definicja	Wartość domyślna	Opis
<b>DIFFUSE_COLOR_ENABLE</b>	(zdefiniowane)	Światło rozproszone jako element oświetlenia powierzchni
<b>SPECULAR_HIGHLIGHT_ENABLE</b>	(zdefiniowane)	Światło odbite zwierciadlanie jako element oświetlenia powierzchni
<b>CUSTOM_COLOR_SPECULAR_HIGHLIGHTS_ENABLE</b>	(zdefiniowane)	Umożliwia dokonanie modyfikacji koloru światła odbitego w sposób zwierciadlany za pomocą parametru materiału
<b>INTERNAL_SHADING_ENABLE</b>	(zdefiniowane)	Dokonuje inwersji normalnej $\vec{n}$ do powierzchni w punkcie zderzenia, w przypadku gdy iloczyn skalarny kierunku promienia padającego $\vec{d}$ i normalnej $\vec{n}$ w punkcie zderzenia jest nieujemny. Umożliwia obrazowanie obu stron danej powierzchni
<b>SHADOW_ENABLE</b>	(zdefiniowane)	Włącza rzucanie cieni przez obiekty znajdujące się w świecie
<b>SELF_SHADOW_ENABLE</b>	(zdefiniowane)	Włącza możliwość rzucania cieni przez dany obiekt na samego siebie
<b>FRESNEL_REFLECTION_ENABLE</b>	(zdefiniowane)	Jeśli zdefiniowane, współczynnik odbicia określany jest na podstawie wzorów Fresnella (1.2.57)(1.2.58). W przeciwnym razie, jako współczynnik odbicia zawsze brany będzie współczynnik odbicia zwierciadlanego $k_s$ danego materiału
<b>OREN_NAYAR_DIFFUSE_MODEL_ENABLE</b>	(zdefiniowane)	Włącza możliwość użycia modelu Orena-Nayara dla modelowania światła rozproszonego
<b>TORRANCE_SPARROW_SPECULAR_MODEL_ENABLE</b>	(zdefiniowane)	Włącza możliwość użycia modelu Torrance'a-Sparrowa dla modelowania światła odbitego zwierciadlanie
<b>TEXTURE_ENABLE</b>	(zdefiniowane)	Umożliwia teksturowanie powierzchni

Tab. 3.5: Kontynuacja

Definicja	Wartość domyślna	Opis
<b>TEXTURE_URAM_STORAGE</b>	(niezdefiniowane)	Jeśli zdefiniowane, instruuje narzędzia Vivado, aby do przechowywania pamięci tekstury wykorzystywane były bloki Ultra RAM. W przeciwnym razie, tekstury zapisane będą standardowo w BRAM
<b>BILINEAR_TEXTURE_FILTERING_ENABLE</b>	(zdefiniowane)	Umożliwia dokonanie filtracji tekstury dla uzyskania gładkich efektów przejścia między tekselami
<b>ADVANCED_TEXTURE_MAPPING_ENABLE</b>	(zdefiniowane)	Umożliwia poprawne teksturowanie powierzchni sferycznych i cylindrycznych
<b>FAST_INV_SQRT_ENABLE</b>	(niezdefiniowane)	Stosuje szybki algorytm obliczania odwrotności pierwiastka kwadratowego. Używane w środowisku testowym dla zmniejszenia czasu oczekiwania na wyniki
<b>FAST_DIVISION_ENABLE</b>	(niezdefiniowane)	Stosuje szybki algorytm obliczania wyniku dzielenia. Używane w środowisku testowym dla zmniejszenia czasu oczekiwania na wyniki
<b>FAST_ATAN2_ENABLE</b>	(zdefiniowane)	Jeśli włączone, używa aproksymacji wielomianowej w celu obliczenia wartości funkcji atan2(). W przeciwnym razie używana jest jej biblioteczna dokładna wersja <code>hls :: atan2()</code>
<b>FAST_ACOS_ENABLE</b>	(zdefiniowane)	Jeśli włączone, używa aproksymacji funkcji acos() opartej o tablicę przeglądową i liniową interpolację między odpowiednimi jej wartościami. W przeciwnym razie używana jest biblioteczna dokładna wersja <code>hls :: atan2()</code>

Tab. 3.5: Kontynuacja

Definicja	Wartość domyślna	Opis
<b>WIDTH</b>	((unsigned short)(1920))	Szerokość w pikselach ramki obrazu
<b>HEIGHT</b>	((unsigned short)(1080))	Wysokość w pikselach ramki obrazu
<b>WIDTH_INV</b>	(automatyczne)	Odwrotność szerokości ramki obrazu
<b>HEIGHT_INV</b>	(automatyczne)	Odwrotność wysokości ramki obrazu
<b>FRAME_ROWS_IN_BUFFER</b>	((unsigned short)(20))	Definiuje, ile pełnych wierszy obrazu będzie przechowywanych w tymczasowym buforze koloru (jeśli <b>RENDER_DATAFLOW_ENABLE</b> jest zdefiniowane). Wartość ta <u>musi</u> być dzielnikiem <b>HEIGHT</b>
<b>VERTICAL_PARTS_OF_FRAME</b>	(automatyczne)	Opisuje, ile razy tymczasowy bufor koloru będzie musiał zostać wypełniony, zanim nastąpi wygenerowanie pełnej ramki obrazu
<b>FRAME_ROW_BUFFER_SIZE</b>	(automatyczne)	Ilość elementów znajdująca się w tymczasowym buforze koloru
<b>NUM_OF_PIXELS</b>	(automatyczne)	Ilość pikseli obrazu do wyrenderowania
<b>TEXT_PAGE_SIZE</b>	((unsigned)(256 · 256))	Ilość 32-bitowych wartości dostępnych jako pamięć tekstur
<b>LIGHTS_NUM</b>	((unsigned char)2)	Maksymalna ilość światel w scenie. Pierwsze ze światel jest zawsze światłem otoczenia, pozostałe to źródła punktowe
<b>OBJ_NUM</b>	((unsigned char)8)	Maksymalna ilość obiektów, które mogą znaleźć się w generowanej scenie

Tab. 3.5: Kontynuacja

Definicja	Wartość domyślna	Opis
<b>RAYTRACING_DEPTH</b>	((unsigned char)2)	Głębokość zastosowanego śledzenia promieni. Wartość równa 1 odpowiada za śledzenie jedynie promieni pierwotnych - wartości ponad tą liczbę sprawiają, że zaczynają być rozpatrywane odpowiednie rodziny promieni wtórnego
<b>MAX_POWER_LOOP_ITER</b>	((unsigned char)10)	Maksymalna ilość iteracji wykorzystywana w algorytmie dokonującym potęgowania przez obliczanie kwadratów liczb. Jednoznacznie definiuje maksymalny możliwy wykładnik potęgi
<b>FAST_INV_SQRT_ORDER</b>	((unsigned char)2)	Ilość iteracji wykorzystywana przy szybkim obliczaniu odwrotności pierwiastka kwadratowego
<b>FAST_DIVISION_ORDER</b>	((unsigned char)2)	Ilość iteracji wykorzystywana przy szybkim obliczaniu ilorazu liczb.
<b>*PI*</b>	(różne)	Zespół stałych związanych z przetwarzaniem wartości liczby $\pi$

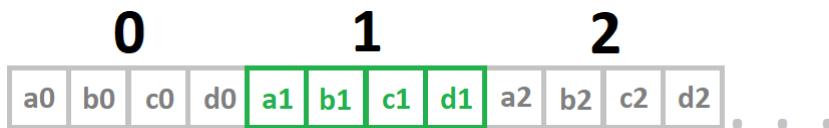
## VIRAY - funkcjonalność i najważniejsze komponenty modułu

### 1. Inicjalizacja modułu

Wejście modułu VIRAY stanowi zespół wskaźników na tablice danych i wartości, które w jednoznaczny sposób przechowują definicję sceny tzn.:

- określają transformacje oraz typy obiektów geometrycznych,
- zawierają parametry mówiące o materiale, z jakiego obiekt jest wykonany tj. BRDF, dane tekstury,
- ustanawiają parametry związane ze źródłami światła,
- definiują położenie, orientację w przestrzeni i inne podstawowe parametry kamery (obserwatora).

Dane te przychodzą do modułu w postaci 32-bitowych liczb (zmiennopozycyjnych oraz całkowitoliczbowych). W przypadku tablic, dane przez nie zawierane są zlinearyzowane tzn. każda tablica przechowuje dane o  $k$  wektorach. Każdy 32-bitowy element takiego wektora, zależnie od jego położenia w tym wektorze, posiada odpowiednie znaczenie, które musi być odpowiednio zinterpretowane (rysunek 3.8). Wymaga to zastosowania procedur, które na podstawie dostarczonych danych będą w stanie dokonać odpowiedniej inicjalizacji struktur wewnętrznych.



**Rys. 3.8:** Przykład zapisu danych w tablicach wejściowych. Elementy  $a$ ,  $b$ ,  $c$ ,  $d$  niosą odpowiednie informacje o  $i$ -tym elemencie struktury danych odpowiedzialnej za jedną z części opisu sceny

Alternatywnie, parametry wejściowe mogłyby zostać przekazane w formie tablic odpowiednich struktur, co uprościłoby proces inicjalizacji w module. W takiej sytuacji domyślnym zachowaniem Vivado HLS jest takie, że każdy element struktury staje się nowym portem danych.

```

1 typedef struct{
2     float s;
3     float t;
4 } scale;
```

W sytuacji przedstawionej powyżej w wyniku syntezы zostaną utworzone oddzielne porty dla  $s$  oraz  $t$ . Jeśli  $scale$  jest tablicą, to  $s$ ,  $t$  będą tak naprawdę portami odwołującymi się do tablic  $s[]$ ,  $t[]$ . Stąd port odpowiadający  $s[]$  musiałby wskazywać na liniowy element przechowujący kolejne wartości  $s$  tablicy struktur  $scale$ . Takie rozwiązanie, choć działa, okazało się niepraktyczne w momencie obsługi modułu za pomocą mikrokontrolera. Zgodnie z dokumentacją Vivado HLS [42] możliwa jest jednak automatyczna linearyzacja struktur, skutkująca wygenerowaniem pojedynczego portu dla struktury za pomocą dyrektywy **DATA\_PACK**. Pomimo wielu testów, skuteczność

tej dyrektywy nie została potwierdzona. Działa ona w przypadku, gdy struktura przechowuje dane całkowite oraz stałopozycyjne. Jeśli jednak zawarte są w niej wartości zmiennoprzecinkowe, odczytywane dane przez moduł nie są poprawne.

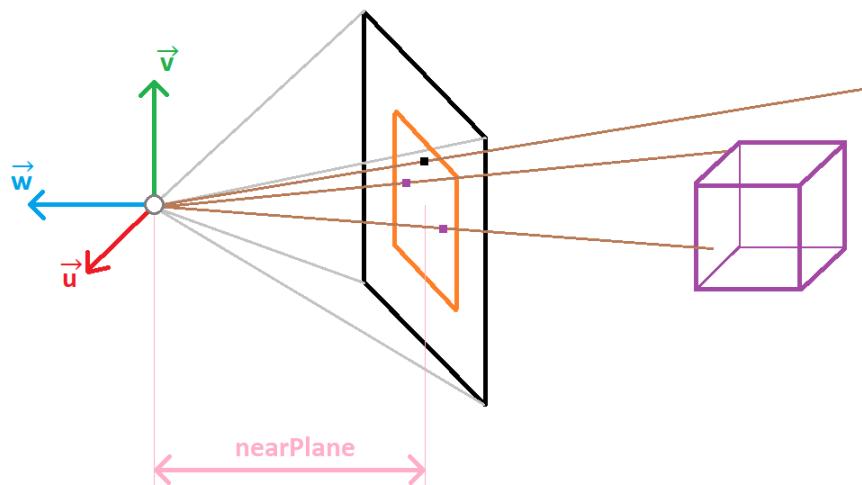
W związku z powyższym, odkodowanie parametrów wejściowych jest dokonywane bezpośrednio przez algorytm, w sposób ręczny, gdyż tylko w taki sposób można zapewnić poprawność odczytu danych przy zachowaniu rozsądnej (tj. możliwie niskiej) liczby portów.

## 2. Model kamery i parametry obrazu

Jedynym udostępnianym przez ViRAY modelem kamery jest kamera perspektywiczna o nieskończonej głębi ostrości (rysunek 3.9). Opisana jest ona poprzez następujące parametry:

- pozycję obserwatora (`eyePosition`) zadaną w globalnym układzie współrzędnych,
- ortonormalną bazę wektorów  $\vec{u}$ ,  $\vec{v}$ ,  $\vec{w}$  tworzących lokalny dla obserwatora prawoskrętny układ współrzędnych określający jego orientację w przestrzeni,
- odległość rzutni (`nearPlane`) od obserwatora w kierunku  $-\vec{w}$  definiującą pole widzenia,
- parametr odpowiedzialny za powiększenie (`zoom`) - im jest on mniejszy, tym powiększenie większe.

Parametry rzutni tj. szerokość (`WIDTH`) oraz wysokość (`HEIGHT`) ramki obrazu muszą zostać podane w momencie syntezy HLS (patrz tabela 3.5). Kiedy rozpoczyna się przetwarzanie nowego piksela obrazu, wywoływana jest funkcja `GetCameraRayForPixel()`, która przyjmuje parametr zależny od współrzędnych piksela  $[w; h] \in [0; \text{WIDTH} - 1] \times [0; \text{HEIGHT} - 1]$  - ten wpływa na kierunek propagacji nowego promienia pierwotnego.



**Rys. 3.9:** Kamera w module ViRAY. Parametry kamery tj. jej położenie, ortonormalna baza wektorów  $\{\vec{u}, \vec{v}, \vec{w}\}$ , odległość rzutni od obserwatora (`nearPlane`) oraz położenie piksela w ramce obrazu  $[w; h]$  definiują początek oraz kierunek propagacji promienia pierwotnego

### 3. Konstrukcja głównej pętli przetwarzania `RenderScene*()`

VIRAY umożliwia wykorzystanie dwóch różnych konstrukcji głównej pętli przetwarzania, które ideowo przedstawiają się następująco:

(a)

```

1 for (unsigned h = 0; h < HEIGHT; ++h)
2 {
3 #pragma HLS DATAFLOW
4
5     InnerLoop ();
6     memcpy ();
7 }
8 void InnerLoop ()
9 {
10    ...
11    for (unsigned w = 0; w < WIDTH; ++w)
12    {
13 #pragma HLS PIPELINE
14    ...
15    }
16 }
```

(b)

```

1 for (unsigned n = 0; n < NUM_OF_PIXELS; ++n)
2 {
3 #pragma HLS PIPELINE
4    ...
5     memcpy ();
6 }
```

W przypadku (a) przetwarzanie pikseli obrazu zostało rozłożone na dwie pętle. Zewnętrzna dokonuje zmiany pionowego indeksu piksela `h`, zaś w wewnętrznej (znajdującej się w funkcji `InnerLoop()`) z zastosowaną dyrektywą `PIPELINE` następuje obliczenie kolorów pełnego wiersza pikseli przy czym wszystkie piksele z wiersza zapisywane są do tymczasowego bufora pikseli, który jest w całości opróżniany do bufora ramki na koniec iteracji pętli zewnętrznej. Zachowanie takie pozwoliło wydzielić ciało zewnętrznej pętli jako region, który może zostać objęty dyrektywą `DATAFLOW`. Dane, czyli zawartość bufora tymczasowego, przepływają pomiędzy dwoma zadaniami tj. obliczeniami, a zapisem do bufora ramki. Kiedy kończy się przetwarzanie danych w pętli wewnętrznej, dane przekazywane są do operacji zapisu `memcpy()` i w tym samym momencie wewnętrzna pętla może znowu rozpoczęć pracę. Problem w takim przypadku, jest taki, iż kolejne obliczenia w pętli wewnętrznej nie stanowią kontynuacji dopiero co zakończonej funkcji `InnerLoop()` tzn. zanim zostanie wyliczony kolor pierwszego piksela w nowym wierszu minie czas równy opóźnieniu iteracji pętli wewnętrznej, który w finalnej wersji VIRAY'a wynosi 1331 cykli zegara. Biorąc pod uwagę, iż ramka obrazu w jakości Full HD ma wysokość równą  $1,43 \cdot 10^6$  nadmiarowych

cykli zegara<sup>xviii</sup>. Przy standardowym okresie zegara równym 10 ns, czas tracony tylko z tego powodu to 14,3 ms.

Twórcy Vivado HLS przewidzieli występowanie takich sytuacji w tego typu konstrukcjach wprowadzając modyfikator `rewind` do dyrektywy `PIPELINE` stosowanej w pętlach. W zamierzeniu sprawia on, że pętla wykonywać będzie się bez przerw na ponowne rozpoczęcie działania, jednak na etapie tworzenia VIRAY'a nie udało się takiego zachowania wymusić.

Najprostsze możliwe rozwiązanie kwestii nadmiarowych cykli zegara, które można włączyć poprzez niezdefiniowanie `RENDER_DATAFLOW_ENABLE` w pliku konfiguracyjnym `typedefs.h`, zostało przedstawione ideoowo w przykładzie (b). Implementuje ono przetwarzanie pikseli w pojedynczej pętli, której ciało zostało objęte dyrektywą `PIPELINE`. Indeksy piksela w ramce obrazu  $[w; h]$  są obliczane na podstawie aktualnej wartości iteratora `n`, a kolor obliczonego piksela jest zapisywany do bufora ramki bezpośrednio po jego obliczeniu. Eliminuje to nie tylko problem z wielokrotnym rozpoczęciem pętli, ale również usuwa konieczność przechowywania wartości kolorów pikseli w tymczasowym buforze oszczędzając elementy BRAM.

Wyprzedzając w pewien sposób kolejność prezentacji poszczególnych elementów składowych pełnego systemu śledzenia promieni należy powiedzieć, iż rozwiązanie (b) jest nieimplementowalne w układzie KCU116 z częstotliwością pracy uzasadniającą jego użycie<sup>xix</sup>.

W związku z powyższym zaproponowano, aby zmniejszyć w przypadku (a)częstość wywoływania `InnerLoop()`. Pętla wewnętrzna każdorazowo musiałaby przetwarzać więcej niż jeden wiersz bufora ramki, a wielkość bufora tymczasowego uległaby odpowiedniemu zwiększeniu. Ilość cykli zegara, które są potrzebne na przeprowadzenie obliczeń wszystkich pikseli ramki obrazu dana jest w przybliżeniu poniższą zależnością:

$$f(x) = \frac{\text{HEIGHT}}{x} [IL + (\text{WIDTH} \cdot x - 1) \cdot c] + \text{WIDTH} \cdot x = f_1(x) + f_2(x), \quad (3.2.2)$$

gdzie:

$x$  - ilość wierszy obrazu obliczanych w jednym wywołaniu `InnerLoop()`,

$IL$  - opóźnienie iteracji pętli wewnętrznej,

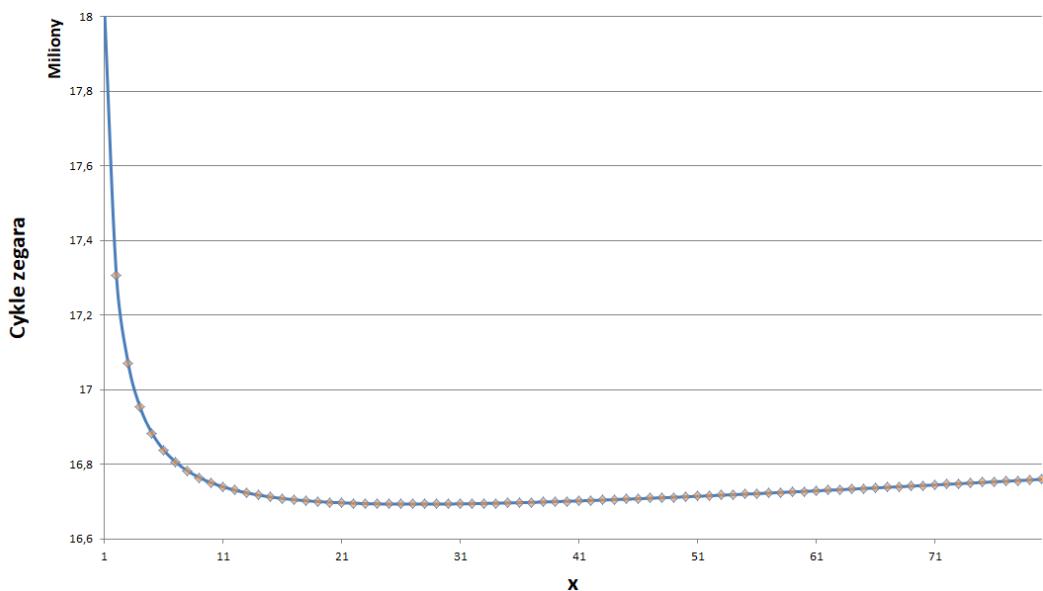
$c$  - interwał pętli wewnętrznej (`DESIRED_INNER_LOOP_II`).

Pierwszy składnik sumy  $f_1(x)$  to przyczynek do całkowitego opóźnienia związanego z obliczaniem pikseli obrazu, zaś  $f_2(x)$  wyraża ilość cykli zegara wymaganą na zapis do bufora ramki ostatniej porcji danych przechowywanych w buforze tymczasowym (wykorzystywany jest tryb seryjnego zapisu danych). Dla parametrów domyślnych systemu śledzenia promieni (patrz tabela 3.5) wykres funkcji  $f(x)$  prezentuje się następująco:

<sup>xviii</sup>Nadmiarowa ilość cykli wynikająca z każdorazowego rozpoczęcia funkcji `InnerLoop()` wynosi:

$$1331 - \text{DESIRED\_INNER\_LOOP\_II} = 1331 - 8 = 1323$$

<sup>xix</sup>Rozwiązanie typu (a), nawet pomimo nadmiarowych cykli zegara, uzyskiwało znacznie krótsze czasy wykonania z uwagi na odpowiednio wyższe częstotliwości pracy.



**Rys. 3.10:** Zależność ilości cykli potrzebnych, aby wykonać główną pętlę algorytmu w funkcji ilości wierszy obrazu  $x$  liczonych przy jednym wywoaniu funkcji implementującej pętlę wewnętrzną `InnerLoop()`

Z wykresu 3.10 widać, że znaczne oszczędności czasu wykonania można uzyskać już dla relatywnie niewielkich wartości  $x$ . Minimum globalne  $f(x)$  można zaś wyliczyć z zależności:

$$\frac{df(x)}{dx} = -\frac{\text{HEIGHT}(IL - c)}{x^2} + \text{WIDTH} \equiv 0,$$

$$x = \sqrt{\frac{\text{HEIGHT}(IL - c)}{\text{WIDTH}}} = 27,28 \approx 27.$$

Przyjęcie  $x = 27$  pozwala zaoszczędzić  $1,326 \cdot 10^6$  cykli zegara względem wersji niezoptymalizowanej ( $x = 1$ ). Trzeba tylko mieć na uwadze, że wielkość bufora tymczasowego w bitach zależy wprost proporcjonalnie od  $x$ :

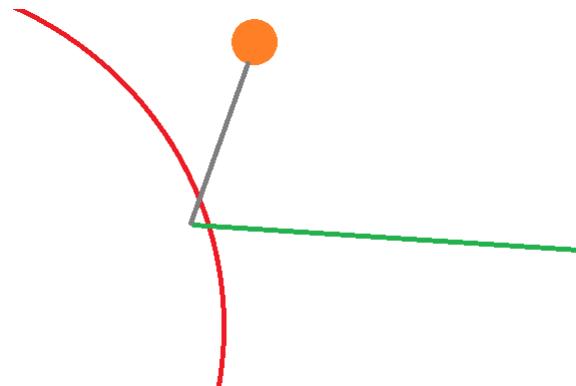
$$\text{WIDTH} \cdot x \cdot 32 \text{ b.} \quad (3.2.3)$$

Stąd w finalnej wersji przyjęto, że `FRAME_ROWS_IN_BUFFER`  $\equiv x = 20$  - taka wartość sprawia, że czas wykonania głównej pętli algorytmu `RenderScene*`() mierzony poprzez liczbę cykli jest jedynie o 5082 cykle gorszy od minimalnego. Dokonując samodzielnego zmian parametru `FRAME_ROWS_IN_BUFFER` należy zapewnić, aby podana wartość była dzielnikiem `HEIGHT`.

#### 4. Przetwarzanie geometrii sceny

Znalezienie przecięcia promienia z najbliższą powierzchnią jest sednem całej metody śledzenia promieni. VlRAY dokonuje testu przecięcia każdego promienia ze wszystkimi obiektami w scenie za pomocą funkcji `PerformHits()` oraz `PerformShadowHits()`. Nie jest to na pewno najbardziej optymalny sposób. Rzeczywiste systemy śledzenia promieni stosują struktury akcelerujące wykrywanie przecięć dzięki wielokrotnym podziałom przestrzeni na mniejsze podprzestrzenie. A czkolwiek zgodnie z przyjętymi założeniami renderowane obiekty nie są złożone z siatek trójkątów, dzięki czemu każdy dostępny kształt geometryczny jest jednym obiektem w scenie.

Test przecięcia polega na rozwiązyaniu równania typu  $f(\vec{r}) = f(\vec{o} + t\vec{d}) = 0$ , którego rozwiązaniem jest dodatnia liczba  $t \in [\text{CORE\_BIAS}; \text{MAX\_DISTANCE}]$  (patrz tabela 3.5). Zastosowanie **CORE\_BIAS** wynika z faktu skończonej precyzji obliczeń, które domyślnie wykonywane są za pomocą typu **float** (**USE\_FLOAT**).



**Rys. 3.11:** Wykorzystanie **CORE\_BIAS** w teście przecięcia. Z uwagi na precyzję obliczeń test przecięcia **promienia** z **powierzchnią** może skutkować tym, że obliczony punkt znajdzie się pod **powierzchnią** zamiast na niej. W takiej sytuacji, promienie wtórne wychodzące z tego punktu będą dawać fałszywie pozytywne testy przecięcia **promienia** z **powierzchnią**, chyba że zostanie zastosowany parametr tolerancji **CORE\_BIAS**

Obliczenia dotyczące przecięcia promienia z powierzchniami przeprowadzane są w układzie współrzędnych związanym z obiektem. Ma to taką zaletę, że ogólne równania przecięcia  $f(\vec{r}) = 0$  mogą zostać odpowiednio uproszczone oraz pozwala to przekształcać obiekty w nietrywialny sposób. W tym celu należy dokonać transformacji promienia do przestrzeni związanej z rozpatrywanym obiektem takiej, że:

$$\vec{r}' = \mathcal{T}^{-1} \{ \vec{r} \}, \quad (3.2.4)$$

gdzie:

$\vec{r}'$  - promień w lokalnej przestrzeni obiektu,

$\mathcal{T}^{-1}$  - transformacja odwrotna względem transformacji  $\mathcal{T}$  opisującej przekształcenie obiektu we współrzędnych globalnych.

Najogólniejsza forma podanego przekształcenia wymaga wykorzystania do opisu transformacji obiektów  $\mathcal{T}$  macierzy **M** postaci:

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & t_1 \\ m_{21} & m_{22} & m_{23} & t_2 \\ m_{31} & m_{32} & m_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (3.2.5)$$

Jej odwrotność  $\mathbf{M}^{-1}$  dokonuje transformacji promienia do przestrzeni obiektu w następujący sposób:

$$\vec{r}' = \begin{cases} \vec{o}' = \mathbf{M}^{-1} \vec{o} \\ \vec{d}' = \mathbf{M}_{3 \times 3}^{-1} \vec{d} \end{cases}, \quad (3.2.6)$$

przy czym  $\mathbf{M}_{3 \times 3}^{-1}$  jest macierzą złożoną tylko z elementów  $m_{ij}$  macierzy  $\mathbf{M}^{-1}$ .

Rozwiążanie równania  $f(\vec{r}') = 0$  pozwala obliczyć punkt przecięcia  $\vec{p}'$  oraz normalną  $\vec{n}'$  do powierzchni w tym punkcie w przestrzeni obiektu, które do dalszych obliczeń muszą zostać przetransformowane do globalnego układu współrzędnych za pomocą macierzy transformacji  $\mathbf{M}$  obiektu.

Z uwagi na stopień komplikacji, który uniemożliwia implementację w układzie FPGA, VlRAY standardowo (**SIMPLE\_OBJECT\_TRANSFORM\_ENABLE** jest zdefiniowane w pliku **typedefs.h**) umożliwia jedynie dokonywanie transformacji wyrównanych do osi globalnego układu współrzędnych. Oznacza to, że skalowanie oraz orientacja obiektów w przestrzeni jest ograniczona i nie może zachodzić w inny sposób niż wzdłuż wersorów globalnego układu<sup>xx</sup>.

Funkcja odpowiedzialna za odnalezienie najbliższego obiektu została wyposażona w możliwość testowania promieni z kulami, cylindrami, stożkami, płaszczyznami, dyskami i kwadratami. W pierwszych trzech przypadkach należy rozwiązać równanie kwadratowe ze względu na odległość  $t$ :

$$\begin{aligned} at^2 + bt + c &= 0, \\ t_{1,2} &= \frac{-b \pm \sqrt{\Delta}}{2a}, \end{aligned}$$

przy czym  $t = \min(t_1, t_2) \geq \text{CORE\_BIAS}$ , zaś współczynniki  $a, b, c$  zależą od równania opisującego daną powierzchnię:

- Sfera o jednostkowym promieniu  $R = 1$  i początku w  $\vec{o}_o = [0, 0, 0]$

$$\begin{aligned} (\vec{r}' - \vec{o}'_o)^2 &= R^2 \quad \Rightarrow \quad \vec{r}'^2 - 1 = 0, \\ a &= \vec{d}' \cdot \vec{d}', \\ b &= 2\vec{d}' \cdot \vec{o}', \\ c &= \vec{o}' \cdot \vec{o}' - 1. \end{aligned}$$

- Cylinder o osi obrotu wokół wersora  $\vec{y}'$ , jednostkowym promieniu  $R = 1$  i początku w  $\vec{o}'_o = [0, 0, 0]$

$$\begin{aligned} (\vec{r}'_{xz} - \vec{o}'_o)^2 &= R^2 \quad \Rightarrow \quad r'_x^2 + r'_z^2 - 1 = 0, \\ a &= d'_x^2 + d'_z^2, \\ b &= 2(d'_x o'_x + d'_z o'_z), \\ c &= o'_x^2 + o'_z^2 - 1. \end{aligned}$$

---

<sup>xx</sup>Paradoksalnie taka reprezentacja transformacji w znaczny sposób wpływa na przetwarzanie informacji o przecięciu, w skutek czego kod stał się bardziej skomplikowany i trudniejszy w interpretacji.

- Stożek o promieniu podstawy  $R = 1$ , wysokości  $h = 1$  oraz osi obrotu wokół wersora  $\vec{y}'$

$$\left(\frac{hr'_x}{R}\right)^2 - (r'_y - h)^2 + \left(\frac{hr'_z}{R}\right)^2 = 0 \quad \Rightarrow \quad r'^2_x - (r'_y - 1)^2 + r'^2_z = 0,$$

$$a = d'^2_x - d'^2_y + d'^2_z,$$

$$b = 2(d'_x o'_x - d'_y o'_y + d'_y + d'_z o'_z),$$

$$c = o'^2_x - o'^2_y + 2o'_y - 1 + o'^2_z$$

Pozostałe trzy obiekty tj. płaszczyzna, dysk oraz kwadrat wymagają rozwiązań równania:

$$(\vec{r}' - \vec{o}'') \cdot \vec{n} = 0,$$

które w przypadku gdy w układzie związanym z obiektem  $\vec{o}'' = [0, 0, 0]$ , a  $\vec{n}' = [0, 1, 0]$  redukuje się do postaci:

$$t = -\frac{o'_y}{d'_y}$$

Określenie czy promień trafił w dysk albo kwadrat wymaga dodatkowo użycia informacji o lokalnym punkcie przecięcia:

$$\vec{p}' = \vec{o}' + t\vec{d}' \tag{3.2.7}$$

Jeżeli spełniony jest warunek:

- $p'^2_x + p'^2_z \leq 1$  - promień trafił w dysk,
- $|p'_x| \leq 1 \wedge |p'_z| \leq 1$  - promień trafił w kwadrat.

Przyjęcie takich parametrów dla powyższych powierzchni sprawia, że w lokalnym układzie współrzędnych obiektu:

$$\vec{p}' \in [-1, 1]^3.$$

Fakt ten jest wykorzystywany podczas teksturowania<sup>xxi</sup>.

Funkcjonalność związana ze znalezieniem najbliższego obiektu na drodze promienia jest realizowana przez funkcje `PerformHits()` i `PerformShadowHits()`. Pierwsza z nich jest wywoływana dla promieni pierwotnych i rozproszonych, zaś druga dla promieni cienia, sprawdzających czy między źródłem światła a znalezionym punktem na obiekcie nie znajduje się jakakolwiek przeszkoda. Utylizacja zasobów układu FPGA związana z pojedynczą instancją obu tych funkcji przedstawiona została w tabeli 3.6.

**Tab. 3.6:** Wykorzystanie zasobów sprzętowych układu KCU116 przez instancje funkcji sprawdzające przecięcie promienia z geometrią raportowane po wykonaniu syntezy HLS

	<b>BRAM</b>	<b>DSP</b>	<b>FF</b>	<b>LUT</b>
<code>PerformHits()</code>	0 (0%)	109 (5,96%)	58496 (13,48%)	19136 (8,82%)
<code>PerformShadowHits()</code>	0 (0%)	107 (5,87%)	22887 (5,27%)	16121 (7,43%)

<sup>xxi</sup>Oczywiście przypadkiem specjalnym jest płaszczyzna, dla której  $\vec{p}' \in \mathbb{R}^3$ .

Tak wysokie wymagania tychże funkcji względem układu FPGA związane są z faktem, iż:

- funkcje te zostały poddane działaniu dyrektywy PIPELINE z interwałem równym 1 dla uzyskania maksymalnej wydajności obliczeń, przez co Vivado HLS ma ograniczone możliwości współdzielenia zasobów odpowiedzialnych za wykonanie poszczególnych operacji,
- każdy dostępny typ obiektu geometrycznego wymaga dokonania zespołu operacji arytmetycznych koniecznych do pełnego rozwiązania testu przecięcia, które są dla niego charakterystyczne<sup>xxii</sup>

To jednak nie wszystko. Ilość instancji tychże funkcji<sup>xxiii</sup> w ViRAY'u zależy od:

- maksymalnej ilości obiektów w scenie OBJ\_NUM,
- interwału między kolejnymi pikselami obrazu DESIRED\_INNER\_LOOP\_II,
- głębokości śledzenia promieni RAYTRACING\_DEPTH,
- maksymalnej ilości światel w scenie LIGHTS\_NUM.

W najprostszym przypadku, gdy OBJ\_NUM = DESIRED\_INNER\_LOOP\_II, a promieniami rozproszonymi są jedynie promienie odbite, liczba instancji dana jest poprzez:

$$\begin{aligned}\#PerformHits() &= \text{RAYTRACING\_DEPTH}, \\ \#PerformShadowHits() &= \text{RAYTRACING\_DEPTH} \cdot (\text{LIGHTS\_NUM} - 1).\end{aligned}$$

Z tego właśnie powodu ViRAY w finalnej wersji nie dokonuje obliczeń związanych z promieniami załamanimi, a jedynie rozproszonymi i cienia przy ilości światel w scenie ograniczonej do dwóch (jedno otaczające i jedno punktowe). W ten sposób obie te funkcje łącznie konsumują 32,5% LUT oraz 37,5% FF układu KCU116 (wg. raportu po syntezie HLS).

W tym miejscu należy również podnieść bardzo istotną kwestię związaną z wielkością RAYTRACING\_DEPTH. Aby mieć możliwość generowania informacji o odbiciach powstających na powierzchni musi być spełniona następująca zależność:

$$\text{RAYTRACING\_DEPTH} \geq 2.$$

Podczas prób implementacji wstępnych wersji ViRAY'a na układzie VC707 możliwe było zastosowanie jedynie RAYTRACING\_DEPTH = 1, podczas gdy KCU116 pozwoliło osiągnąć RAYTRACING\_DEPTH = 2. Właśnie ten fakt przesądził o zmianie platformy docelowej, pomimo teoretycznie gorszych parametrów związanych z ilością dostępnych elementów (tabela 3.1).

---

<sup>xxii</sup>Chociaż kule, walce i stożki łączy to, iż wymagają rozwiązania równania kwadratowego, to współczynniki  $a, b, c$  tego równania muszą być dla nich obliczone w różny sposób.

<sup>xxiii</sup>W zaimplementowanym module funkcja realizowana jest za pomocą zespołu elementów logicznych. Liczba instancji stanowi, ile identycznych zespołów znajdzie się w układzie.

### 5. Cieniowanie - funkcja `Shade()`

Zadaniem funkcji cieniącej `Shade()` jest określenie koloru piksela w przestrzeni barw RGB odpowiadającego znalezionemu punktowi przecięcia  $\vec{p}$  z obiektem zgodnie z właściwą funkcją BRDF. Proces ten dokonuje się na podstawie:

- położenia punktu  $\vec{p}$  względem źródeł światła,
- zastosowanych BRDF dla określenia materiału,
- informacji o teksturze, która ma zostać nałożona na obiekt.

Poniższe równanie będące uogólnieniem równania (1.2.76) opisuje, w jaki sposób liczona jest radiancja w danym punkcie z uwzględnieniem wszystkich źródeł światła dla  $k$ -tego promienia (operator \* oznacza mnożenie  $i$ -tych składowych obu operandów ze sobą) [59]:

$$\begin{aligned} L_{o_k}(\vec{p}, \vec{\omega}_o) \equiv L_o(\vec{p}, \vec{\omega}_o) &= k_a(\vec{p})c_a(\vec{p}) * L_a \\ &+ \sum_{i=1}^{\text{LIGHTS\_NUM}} k_d(\vec{p})c_d(\vec{p}) * L_d(\vec{p}, \vec{\omega}_o) \cdot c_i \cdot s_i \cdot s_{cf_i} \\ &+ \sum_{i=1}^{\text{LIGHTS\_NUM}} k_s(\vec{p})c_s(\vec{p}) * L_s(\vec{p}, \vec{\omega}_o) \cdot c_i \cdot s_i \cdot s_{cf_i}, \end{aligned} \quad (3.2.8)$$

gdzie:

- $k_a(\vec{p}), k_d(\vec{p}), k_s(\vec{p}) \in [0, 1]$  - współczynniki określające udział światła otoczenia, rozproszzonego i odbitego zwierciadlanie w tworzeniu radiancji,
- $c_a(\vec{p}), c_d(\vec{p}), c_s(\vec{p}) \in [0, 1]^3$  - kolor RGB materiału w świetle otaczającym, rozproszonym i odbitym zwierciadlanie,
- $L_a, L_d(\vec{p}, \vec{\omega}_o), L_s(\vec{p}, \vec{\omega}_o)$  - radiancja związana ze światłem otoczenia, rozproszonym i odbitym zwierciadlanie,
- $c_i = \frac{\max\left(\vec{n} \cdot \frac{\vec{p} - \vec{l}_i}{|\vec{p} - \vec{l}_i|}, 0\right)}{|\vec{p} - \vec{l}_i|^2}$  - współczynnik związany z kwadratowym zanikiem oraz orientacją względem kierunku padania światła ( $\vec{l}_i$  - położenie źródła światła),
- $s_i \in \{0, 1\}$  - wynik testu zasłaniania (`ShadowVisibilityTest()`)  $i$ -tego źródła światła przez obiekty sceny, jeśli  $s_i = 0$  punkt  $\vec{p}$  jest w cieniu,
- $s_{cf_i} \in [0, 1]$  - współczynnik determinujący czy punkt  $\vec{p}$  znajduje się w stożku światelnym źródła umieszczonego w  $\vec{l}_i$ .

Natomiast całkowita radiancja w danym punkcie  $L_{o_t}(\vec{p}, \vec{\omega}_o)$  wynikająca z wielokrotnych rozproszeń (odbić) promieni dana jest wzorem:

$$L_{o_t}(\vec{p}, \vec{\omega}_o) = \sum_{k=1}^{\text{RAYTRACING\_DEPTH}} \left( \prod_{l=1}^k r_{l-1} \right) L_{o_k}(\vec{p}, \vec{\omega}_o), \quad r_0 = 1, \quad (3.2.9)$$

gdzie:

$r_{l-1}$  jest współczynnikiem odbicia od powierzchni, na której  $l$ -ty promień ma swój początek (dla promieni pierwotnych  $k = 1$ :  $r_0 = 1$ ). Jeśli powierzchnia nie jest lustrzana, współczynnik ten obliczany jest za pomocą równań Fresnela z wykorzystaniem następujących uproszczeń:

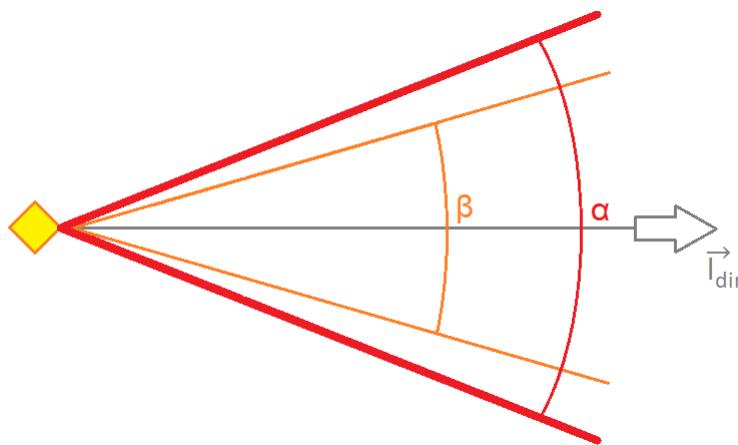
- światło jest zawsze całkowicie niespolaryzowane, przez co współczynnik odbicia  $r_{l-1}$  może zostać obliczony jako średnia arytmetyczna współczynników odbicia dla dwóch skrajnych polaryzacji tj.  $R_{\parallel}$  oraz  $R_{\perp}$ <sup>xxiv</sup>,
- współczynnik załamania światła  $n$  jest wartością stałą dla danego materiału, zatem nie zależy od częstotliwości promieniowania.

W scenie renderowanej za pomocą ViRAY'a, oprócz globalnego bezkierunkowego źródła światła otaczającego, można umieszczać *punktowe źródła światła* wysyłające promieniowanie w określonym przez użytkownika stożku świetlnym - ten zadany jest przez kierunek główny  $\vec{l}_{dir}$ , oraz kosinus połowy kąta rozwarcia  $\cos \frac{\alpha}{2}$  (rysunek 3.12). Dodatkowy kąt  $\beta$  decyduje o kosinusie  $\cos \frac{\beta}{2} \geq \cos \frac{\alpha}{2}$ , którego przekroczenie będzie wpływać na przybieranie przez  $s_{cf_i}$  wartości pośrednich między 0 a 1:

$$s_{cf_i} = \max \left( \min \left( \frac{\max \left( \vec{n} \cdot \frac{\vec{p} - \vec{l}_i}{|\vec{p} - \vec{l}_i|}, 0 \right) - \cos \frac{\alpha}{2}}{\cos \frac{\beta}{2} - \cos \frac{\alpha}{2}}, 1 \right), 0 \right) \quad (3.2.10)$$

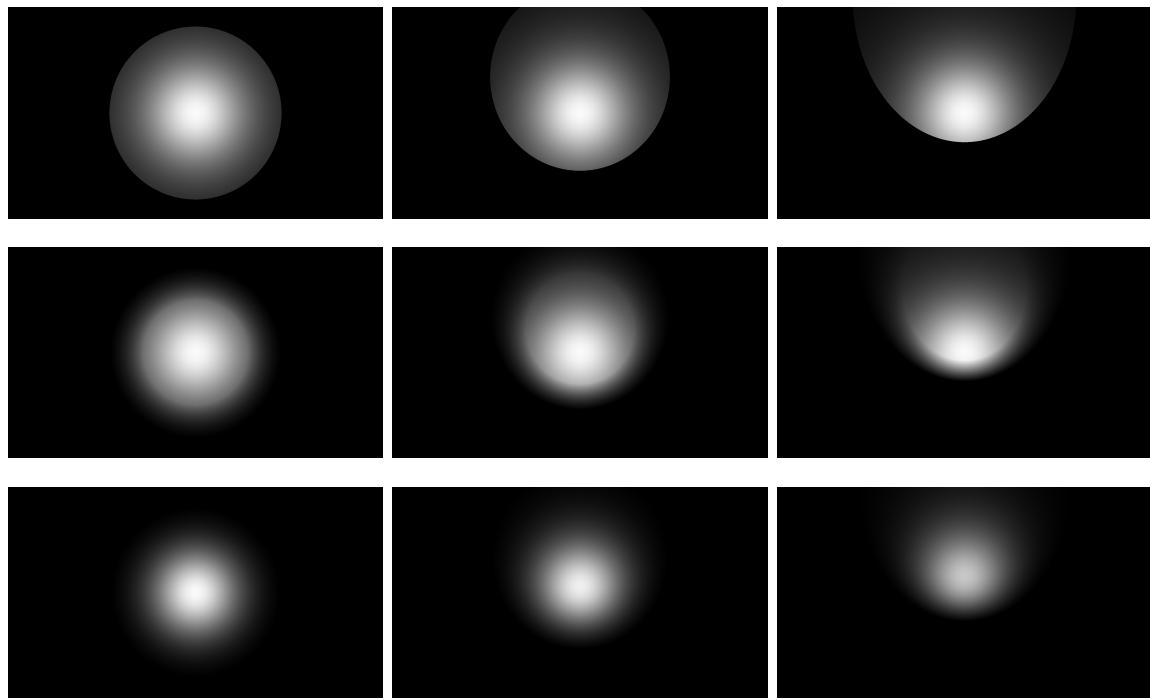
Wpływ położenia płaszczyzny względem kierunku światła  $\vec{l}_{dir}$  oraz przyjętego kąta  $\frac{\beta}{2}$  na jej oświetlenie prezentuje rysunek 3.13.

Naturalnie punktowe źródło światła jest tylko przybliżeniem rzeczywistego zachowania, ponieważ rzeczywiste źródła emitują promieniowanie z pewnej skończonej powierzchni, czego przejawem jest rozmycie granic cieni.



**Rys. 3.12:** Punktowe źródło światła w module ViRAY. Światło jest emitowane w kierunku danym przez  $\vec{l}_{dir}$  w stożku o kącie rozwarcia  $\alpha$ . Pełna intensywność występuje jedynie w mniejszym stożku o kącie rozwarcia  $\beta < \alpha$

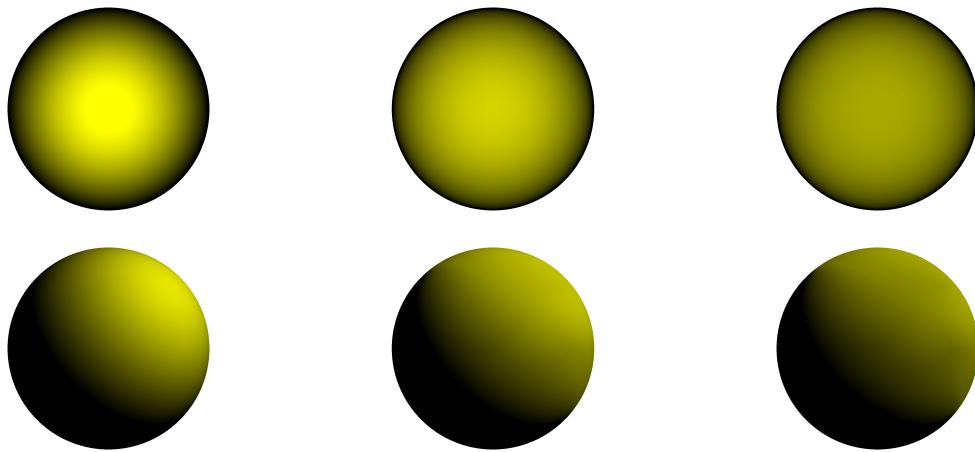
<sup>xxiv</sup>Jest to często stosowane uproszczenie, spotykane nawet w zaawansowanych systemach dokonujących śledzenia promieni.



**Rys. 3.13:** Oświetlenie płaskiej powierzchni przez punktowe źródło światła. Obrazy w kolejnych kolumnach odpowiadają różnemu ustawieniu tego samego źródła względem płaszczyzny, zaś wiersze różnią się wartością  $\cos \frac{\beta}{2}$  - idąc od góry 0,7; 0,85; 1,0 ( $\cos \frac{\alpha}{2} = 0,7 = \text{const}$ ). W przypadku, gdy  $\cos \frac{\beta}{2} \neq \cos \frac{\alpha}{2}$ , krawędź między obszarem oświetlonym a nieoświetlonym staje się rozmyta. Nieciągłości przejścia między kolorami widoczne najbardziej w przypadku  $\cos \frac{\beta}{2} = 0,85$  nie są wynikiem błędu algorytmu, a własności przetwarzania obrazu przez mózg człowieka - efekt ten zwany jest *pasmami Macha* [60]

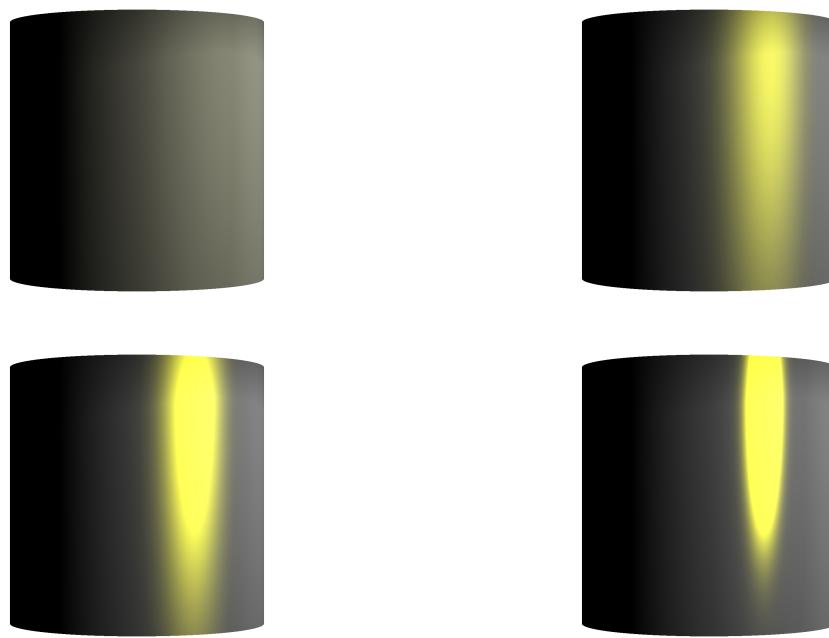
Część światła padającego na obiekt, zgodnie z równaniem (3.2.8) może się na nim rozproszyć lub odbić w sposób zwierciadlany<sup>xxv</sup>. W pierwszym przypadku rozproszenie to jest przybliżane przez model Orena-Nayara (1.2.81), którego przypadkiem granicznym, gdy  $\sigma^2 = 0$ , jest model Lamberta (1.2.80). Wpływ parametru szorstkości  $\sigma^2$  na oświetlenie powierzchni został zobrazowany na rysunku 3.14. Na jego podstawie można stwierdzić, iż wraz ze wzrostem  $\sigma^2$  rozkład oświetlenia staje się bardziej równomierny, a maksimum jasności ma mniejszą wartość.

<sup>xxv</sup>Przypadek kierunkowego odbicia rozproszonego nie jest rozpatrywany przez ViRAY.



**Rys. 3.14:** Oświetlenie powierzchni odbijającej światło całkowicie w sposób rozproszony zgodnie z modelem Orena-Nayara. Górnny wiersz odpowiada sytuacji, gdy kierunek obserwacji  $\vec{d}$  oraz kierunek światła  $\vec{l}_{dir}$  są identyczne i przechodzą przez środek oświetlanej kuli. W dolnym wierszu wspomniane kierunki nie pokrywają się. Kolejne kolumny (od lewej do prawej) odpowiadają różnym wartościom współczynnika szorstkości materiału  $\sigma^2 \in \{0; 0,25; 1\}$ , wraz ze wzrostem którego następuje coraz większe zrównanie jasności oświetlanej powierzchni

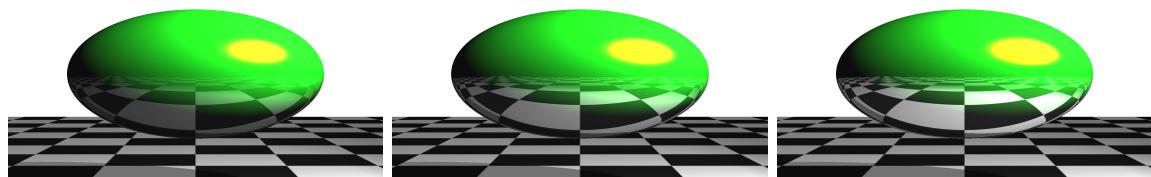
Drugim z komponentów oświetlenia jest odbicie zwierciadlane, które w VlRAY'u można modelować za pomocą modelu Blinna-Phonga (1.2.89) bądź też Torrance'a-Sparrowa (1.2.91). Decyzja o rezygnacji z modelu Phonga na rzecz Blinna-Phonga podyktowana została tym, że model Blinna-Phonga oraz Torrance'a-Sparrowa współdzielą ze sobą obliczenia związane z czynnikiem rozkładu orientacji mikrościanek wokół wektora połówkowego  $D(\vec{\omega}_h)$  (1.2.92), przez co implementacja tego pierwszego nie wiąże się z wykorzystaniem dodatkowych zasobów układu FPGA. Wpływ parametru skupienia rozbłysku  $e \in \mathbb{N}$  w modelu Blinna-Phonga został zaprezentowany na przykładzie oświetlenia powierzchni walcowej (rysunek 3.15).



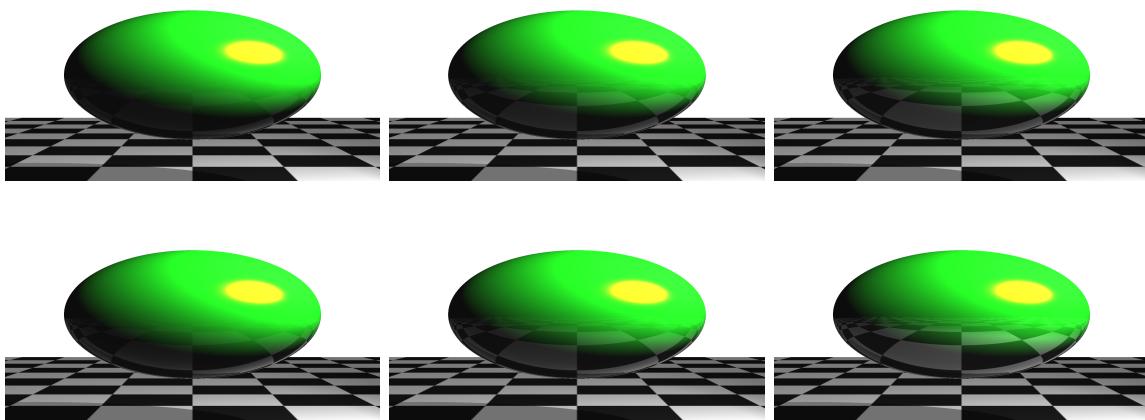
**Rys. 3.15:** Rola parametru rozbłysku  $e$  w modelu Blinna-Phonga. Idąc od lewej do prawej z góry na dół parametr  $e$  przyjmuje następujące wartości: 0, 16, 32, 128. Z uwagi na fakt, iż kolory zapisywane są w postaci 24-bitowych wartości, gdzie każda z trzech składowych RGB reprezentowana jest przez 8 bitów, na obrazach dostrzec można efekty związane z obcięciem wartości do 255, które objawiają się jako obszary rozbłysku o jednakowym kolorze.

Analizę działania modelu Torrance'a-Sparrowa najłatwiej dokonać w połączeniu ze sprawdzeniem, jak różne wartości współczynnika załamania  $\eta$  (oraz absorpcji  $k$  dla przewodników) wpływają na zdolność odbijania promieni od powierzchni. Bierze się to z faktu, iż model Torrance'a-Sparrowa ingeruje bezpośrednio w wartość współczynnika odbicia  $r_{l-1}$ , który zależy tu od przestrzennej relacji między wektorem połówkowym  $\vec{w}_h$  a kątem  $\vec{\omega}_o$  (w przeciwieństwie do typowej zależności między wektorem normalnym  $\vec{n}$  a  $\vec{\omega}_o$ ).

Poniższe obrazy uzyskano dla wartości współczynnika skupienia  $e = 128$  dla modelu Blinna-Phonga oraz  $e = 1024$  w przypadku modelu Torrance'a-Sparrowa - różnica ta opiera się na obserwacji, że model Torrance'a-Sparrowa generuje znacznie słabsze rozbłyski, dla takich samych wartości  $e$ .



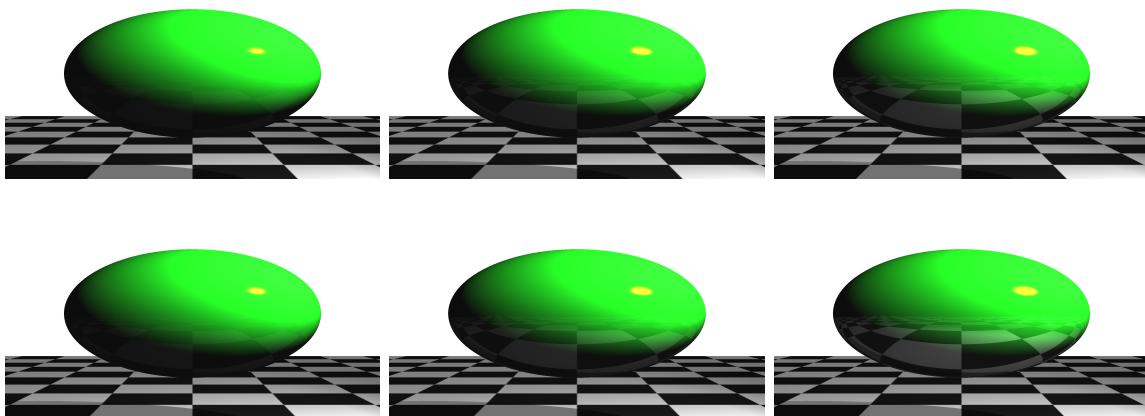
**Rys. 3.16:** Powierzchnie odbijające w sposób lustrzany w modelu Blinna-Phonga. Współczynnik odbicia  $r_{l-1}$  jest stały, niezależny od kąta padania na płaszczyznę elipsoidy i równy współczynnikowi  $k_s \in \{0, 2; 0, 4; 0, 6\}$  ( $k_s$  rośnie od lewej do prawej)



**Rys. 3.17:** Powierzchnie odbijające zgodnie ze wzorami Fresnela w modelu Blinna-Phonga. Obrazy w górnym wierszu odpowiadają dielektrykom ( $k = 0$ ) o współczynniku załamania (od lewej do prawej)  $\eta \in \{1, 33; 1, 66; 2, 00\}$ . Dla najmniejszego ze współczynników załamania  $\eta = 1, 33$  odbicia są słabo zauważalne, jednak  $r_{l-1}$  rośnie bardzo szybko, gdy promień padający staje się styczny do powierzchni. W dolnym wierszu przedstawione zostały przewodniki o  $k \in \{0, 2; 0, 6; 1, 0\}$  i  $\eta = 1, 33$ , których powierzchnie są znacznie bardziej odbijające niż w przypadku dielektryka o tym samym  $\eta$

Zastosowanie modelu Torrance'a-Sparrowa na rysunku 3.18 zamiast modelu Blinna-Phonga z odbiciem Fresnela (rysunek 3.17) wprowadza dwie główne zmiany w obrazie:

- (a) Siła rozbłysku w prawej górnej części elipsoidy jest mniejsza i należy stosować większy współczynnik skupienia  $e$ .
- (b) Współczynnik odbicia, ze względu na sposób jego określenia w modelu Torrance'a-Sparrowa między wektorem połówkowym  $\vec{\omega}_h$  a  $\vec{\omega}_o$ , nie rośnie tak szybko wraz ze wzrostem kąta padania.



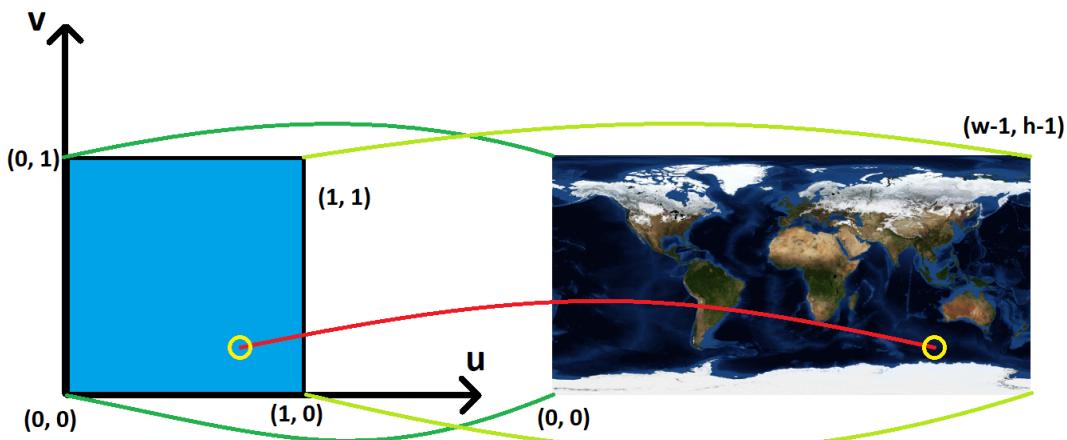
**Rys. 3.18:** Powierzchnie odbijające w modelu Torrance'a-Sparrowa. Obrazy w górnym wierszu odpowiadają dielektrykom ( $k = 0$ ) o współczynniku załamania (od lewej do prawej)  $\eta \in \{1, 33; 1, 66; 2, 00\}$ . W dolnym wierszu przedstawione zostały przewodniki o  $k \in \{0, 2; 0, 6; 1, 0\}$  i  $\eta = 1, 33$

## 6. Teksturowanie powierzchni

Teksturowanie powierzchni pozwala uatrakcyjnić wizualnie obiekty bez konieczności zwiększenia ich złożoności geometrycznej poprzez nadanie punktom znajdującym się na obiekcie odpowiednich kolorów<sup>xxvi</sup>. ViRAY pozwala nadać obiekowi pożądaną teksturę (kilka różnych obiektów może współdzielić tę samą teksturę), której dane zapisane są w postaci mapy bitowej albo mapy szarości. W tym drugim przypadku, odcień szarości  $mix \in [0, 1]$  decyduje o stopniu mieszania między dwoma wybranymi przez użytkownika kolorami:

$$\text{color} = \text{color1} \cdot mix + \text{color2} \cdot (1 - mix).$$

Problem teksturowania sprowadza się do nałożenia dwuwymiarowej mapy pikseli  $w \times h$  na powierzchnię obiektu trójwymiarowego, który może wprowadzać zniekształcenie w jej odwzorowaniu. Z tego powodu współrzędne na mapie pikseli transformowane są liniowo do przestrzeni  $[u, v] \in [0, 1]^2$  (rysunek 3.19), zaś współrzędne punktu przecięcia promienia z obiektem w przestrzeni obiektu  $\vec{p}' \in [-1, 1]^3$  poddawane są innej (w ogólnym przypadku nieliniowej) transformacji do przestrzeni  $[u, v]$ .



Rys. 3.19: Transformacja pozycji piksela z przestrzeni tekstuury  $[u, v] \in [0, 1]^2$  do współrzędnych  $[u, v]$  w przestrzeni obiektu. Tekstura na podstawie [61]

Najprostszym sposobem transformacji współrzędnych punktu  $\vec{p}'$  do  $[u, v]$  jest transformacja liniowa zwana *mapowaniem prostokątnym*:

$$u = \frac{p'_x + 1}{2},$$

$$v = \frac{p'_z + 1}{2}.$$

Dostępność dodatkowych trybów mapowania w ViRAY'u jest sterowana poprzez przełącznik **ADVANCED\_TEXTURE\_MAPPING\_ENABLE**. Definicja ta daje dostęp do odwzorowania

<sup>xxvi</sup>Tak naprawdę tekstura może być odpowiedzialna nie tylko za nadanie konkretnego koloru obiekto wi, ale może również modulować wektory normalne, współczynniki załamania oraz wiele innych parametrów mających wpływ na ostateczny odbiór obiektu przez obserwatora.

nia *cylindrycznego*:

$$u = \frac{\phi}{2\pi} = \frac{\operatorname{arc} \operatorname{tg} \frac{p'_x}{p'_z}}{2\pi},$$

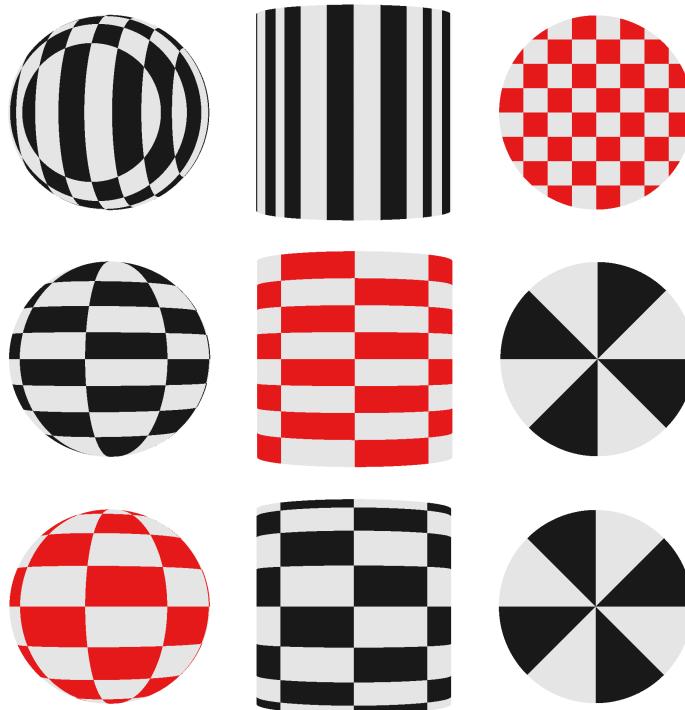
$$v = \frac{p'_y + 1}{2}$$

oraz *sferycznego*:

$$u = \frac{\phi}{2\pi} = \frac{\operatorname{arc} \operatorname{tg} \frac{p'_x}{p'_z}}{2\pi},$$

$$v = 1 - \frac{\theta}{\pi} = 1 - \frac{\operatorname{arc} \cos p'_y}{\pi}.$$

Wpływ trzech wymienionych powyżej trybów na finalne odwzorowanie tekstury na powierzchni różnych obiektów został przedstawiony na rysunku 3.20.



**Rys. 3.20:** Efekt nałożenia identycznej tekstury na różne typy obiektów w funkcji rodzaju zastosowanego mapowania. Od góry: mapowanie prostokątne, cylindryczne i sferyczne

Z zastosowaniem odwzorowania cylindrycznego oraz sferycznego wiąże się konieczność obliczenia odpowiednich kątów na podstawie współrzędnych punktu  $\vec{p}'$  z użyciem funkcji cyklotometrycznych. Wykorzystanie w tym celu wprost funkcji `hls::atan2()` oraz `hls::acos()` dostarczanych wraz z Vivado HLS rodzi jednak kłopoty związane z użyciem przez nie dużej ilości zasobów układu (tabela 3.7), co w praktyce mogłoby oznaczać niemożliwość implementacji odwzorowania cylindrycznego oraz sferycznego w końcowym systemie śledzenia promieni. Chcąc temu zaradzić ViRAY może korzystać z obliczeń przybliżonych w celu znalezienia aproksymacji wartości funkcji `atan2()` oraz `acos()`. Są to odpowiednio `ViRayUtils::Atan2()` oraz `ViRayUtils::Acos()`, które włączane są poprzez definicje `FAST_ATAN2_ENABLE` oraz `FAST_ACOS_ENABLE`.

**Tab. 3.7:** Raportowane wykorzystanie zasobów sprzętowych układu KCU116 przez poszczególne implementacje funkcji cyklometrycznych. Funkcje biblioteczne `hls::atan2()` oraz `hls::acos()` dające najdokładniejsze wyniki wymagają dużej ilości zasobów do ich fizycznej implementacji (zwłaszcza FF oraz LUT). Zaproponowane implementacje przybliżone `ViRayUtils::Atan2()` oraz `ViRayUtils::Acos()` wymagają o wiele mniej logiki układu, zachowując przy tym dokładność wystarczającą podczas teksturowania obiektów

	<b>BRAM</b>	<b>DSP</b>	<b>FF</b>	<b>LUT</b>
<code>hls::atan2()</code>	0	2	12713	16742
<code>hls::acos()</code>	0	50	12022	8038
<code>ViRayUtils::Atan2()</code>	0	5	1153	2064
<code>ViRayUtils::Acos()</code>	1	8	920	1785

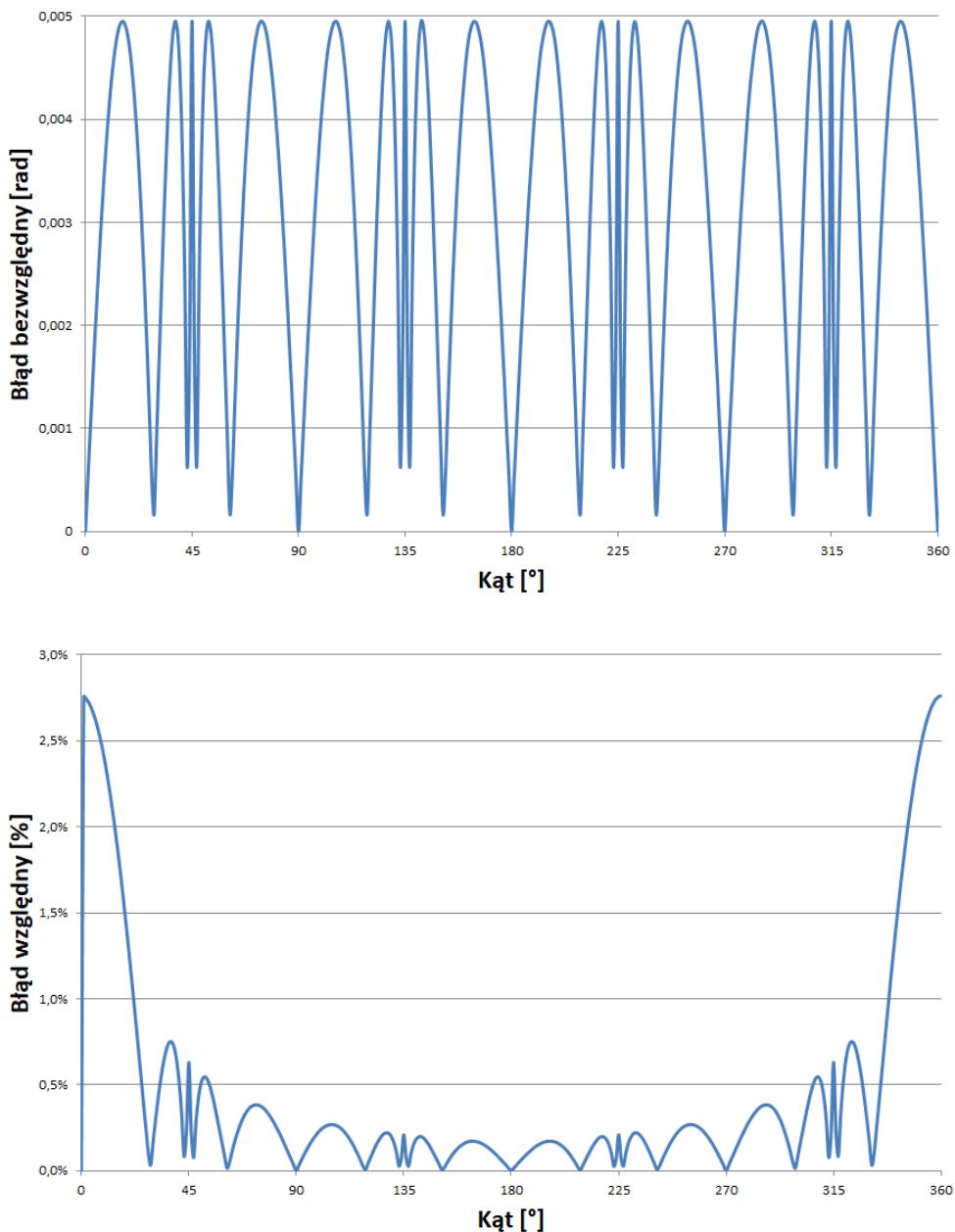
- `ViRayUtils::Atan2()`

Funkcja ta opiera swoje działanie na następującej obserwacji [62]:

$$\arctg(z) \approx 0,9724z - 0,1919z^3, \quad z = \frac{y}{x} \in [-1, 1] \quad (3.2.11)$$

oraz tożsamości:

$$\arctg(z) = \frac{\pi}{2} - \arctg(z^{-1}), \text{ gdy } |z| > 1 \quad (3.2.12)$$

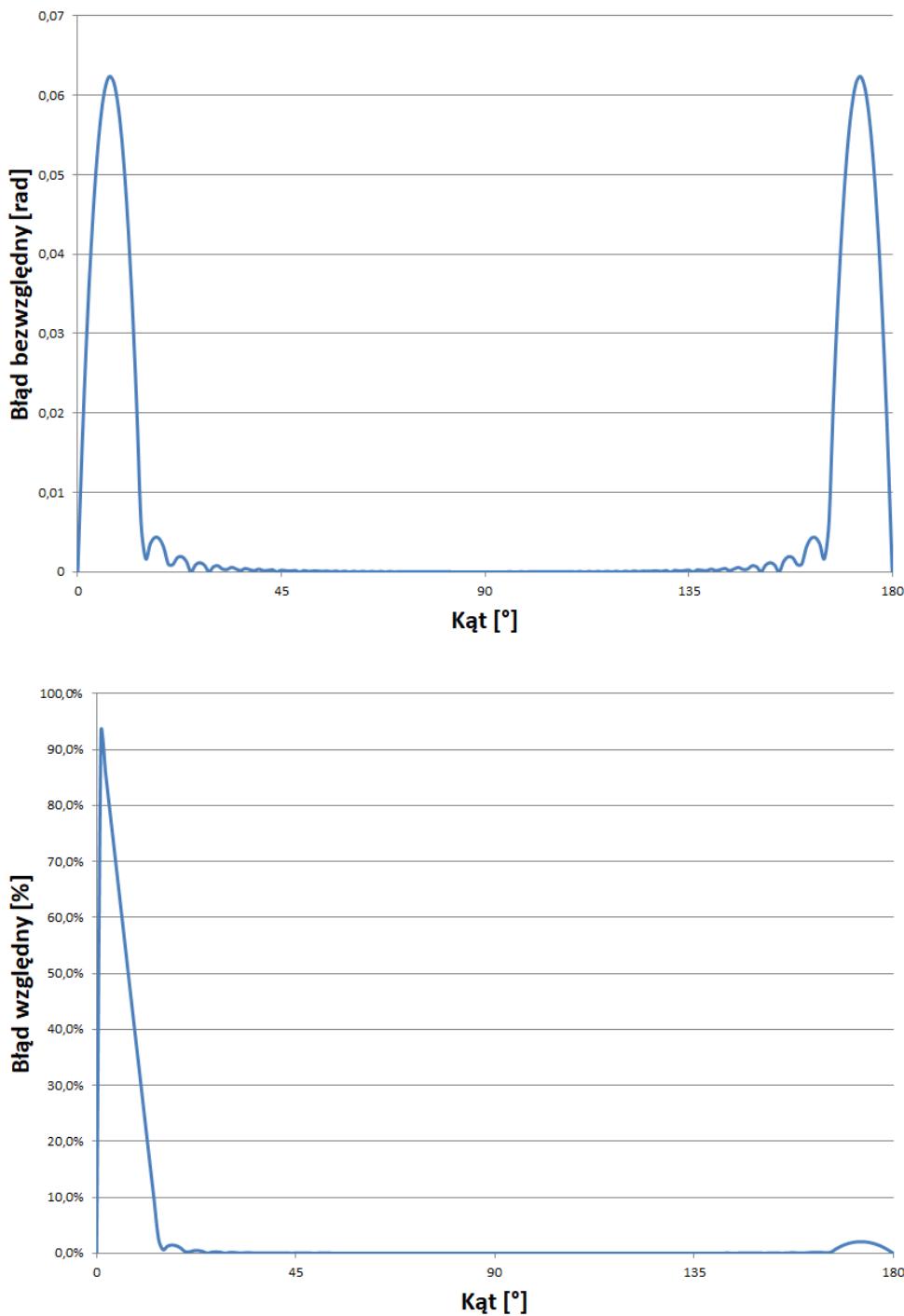


**Rys. 3.21:** Błąd bezwzględny oraz względny generowany przez przybliżoną implementację funkcji `atan2()`. Maksymalny błąd wynosi 0,00496 rad, co przekłada się w najgorszym przypadku na odstępstwo od dokładnego wyniku równe 2,8 %. Różnice te praktycznie nie wpływają na sposób odwzorowania tekstury na powierzchni

- `ViRayUtils::Acos()`

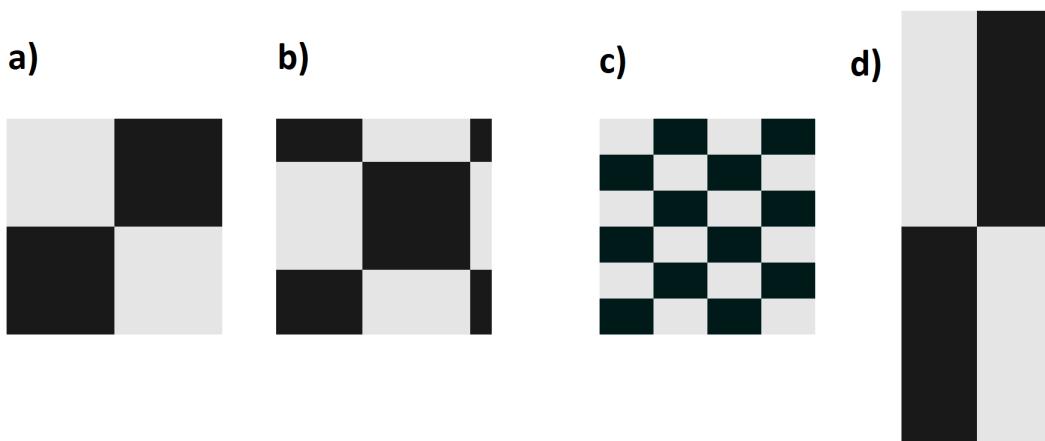
W tym przypadku rozwiążanie przybliżone oparte jest o tablicę 33 wartości, będących dokładnymi wynikami funkcji `acos()` dla równo oddalonych od siebie parametrów  $x_i \in [0, 1]$ . Dla parametru wejściowego  $x'$  następuje określenie najbliższych mu parametrów  $x_i$  oraz  $x_{i+1}$ , dla których znane są dokładne wartości funkcji. Następnie dokonywana jest interpolacja liniowa pomiędzy tymi wartościami, zależna od odległości  $x'$  od  $x_i$  oraz  $x_{i+1}$ . W przypadku, gdy  $x' < 0$  wykorzystywana jest tożsamość:

$$\arccos(x') = \pi - \arccos(|x'|). \quad (3.2.13)$$



**Rys. 3.22:** Błąd bezwzględny oraz względny generowany przez przybliżoną implementację funkcji `acos()`. Maksymalny błąd wynosi 0,0624 rad, który występuje w okolicach 0 oraz  $\pi$  radianów, a związane jest to z faktem, iż funkcja `acos()` przejawia tam największe odstępstwa od liniowości ( $\frac{d(\arccos(x))}{dx} = \frac{1}{\sqrt{1-x^2}} = -\frac{1}{x\sqrt{1-x^2}}$ ). Mimo tej jednej niedogodności, zastosowane przybliżenie sprawdza się podczas teksturowania obiektów

Algorytmy dokonujące teksturowania pozwalają również dokonywać translacji tekstuury oraz jej skalowania poprzez powielanie. Możliwości te zostały zilustrowane na rysunku 3.23.

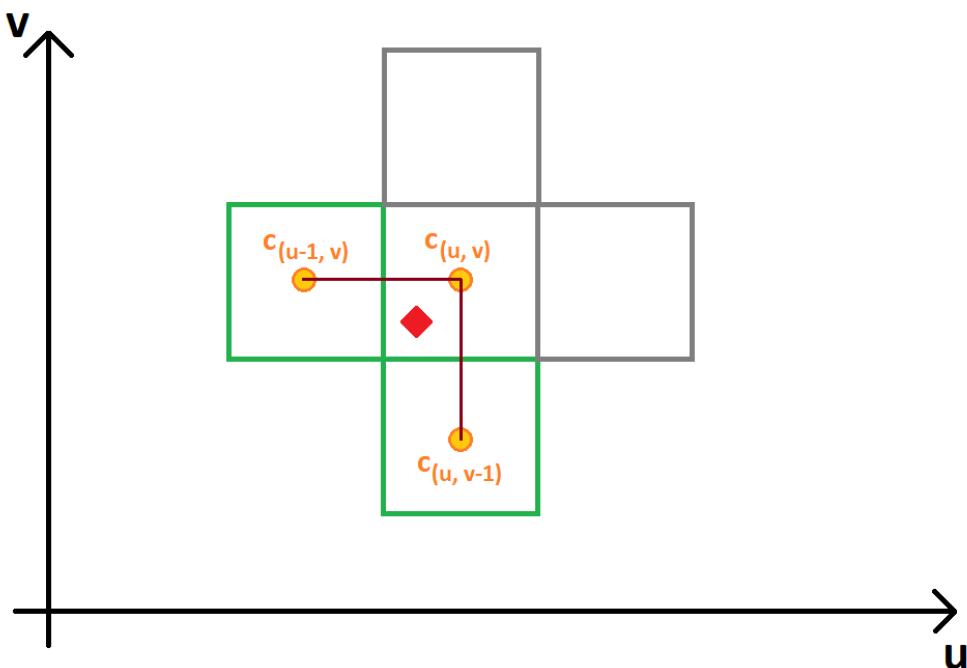


**Rys. 3.23:** Translacja oraz skalowanie tekstuury. a) Kwadrat z oryginalną tekstrurą. b) Tekstura została przesunięta o 0,1 jednostki w lewo oraz 0,2 jednostki w dół. c) Oryginalna tekstrura została przeskalowana w taki sposób, że została powielona dwukrotnie w kierunku  $u$  oraz trzykrotnie w kierunku  $v$  (możliwe są również niecałkowite wartości skali). d) Oryginalny obiekt a) został poddany skalowaniu co wpłynęło również na deformację tekstuury

Tekstury w ViRAY'u są całkowicie przechowywane w pamięci BRAM układu FPGA<sup>xxvii</sup>. Fakt ten wymusza stosowanie tekstur o odpowiednio niskiej rozdzielczości (standardowo ViRAY udostępnia miejsce na  $2^{16}$  32-bitowych wartości, patrz TEXT\_PAGE\_SIZE). W takim wypadku obraz nałożony w sposób bezpośredni na powierzchnię bez odpowiedniej filtracji zostanie poszarpany z wyraźnymi krawędziami pomiędzy pikselami tekstuury. Użycie przełącznika BILINEAR\_TEXTURE\_FILTERING\_ENABLE w pliku konfiguracyjnym za cenę rozmycia szczegółów tekstuury pozwala wyeliminować wyraźne granice pomiędzy pikselami (rysunek 3.24).

<sup>xxvii</sup>Właściwość ta wynika z faktu, iż Vivado HLS nie pozwala dokonywać asynchronicznych, względem potoku przetwarzania w module, operacji odczytu danych z pamięci zewnętrznej.

Możliwe jest również poinstruowanie Vivado HLS, aby pamięć tekstuur znalazła się w układach UltraRAM za pomocą definicji TEXTURE\_URAM\_STORAGE. Rozwiążanie to jest jednakże niezalecane z uwagi na fakt, iż bloki UltraRAM są zlokalizowane w układzie FPGA tzn. nie są tak bardzo rozproszone po całym układzie jak bloki BRAM, w wyniku czego na etapie implementacji dochodzi do nadmiernego stłoczenia, czyli lokalnego przeciążenia sieci połączeń, co prowadzi do niepowodzenia implementacji.



**Rys. 3.24:** Algorytm filtracji biliniowej tekstury. Zaznaczony na czerwono punkt  $p'$  na powierzchni obiektu odpowiada środkowemu pikselowi tekstury, jednak nie wypada on w samym centrum obszaru (tzw. *centroid*  $c_{(u,v)}$ ). W kierunku  $u$  oraz  $v$  znajdują się najbliższe piksele tekstury przylegające do piksela centralnego i najbliższe  $p'$  (w tym przypadku  $c_{(u-1,v)}$  oraz  $c_{(u,v-1)}$ ). Kolor tekstury w  $p'$  obliczony zostaje na podstawie kolorów tychże pikseli i ich wag, zależnych od odległości  $p'$  od centroidów wzduł osi  $u$  oraz  $v$ , jako interpolacja liniowa w kierunku  $u$  oraz  $v$

#### 7. Zapis finalnego koloru piksela - `SaveColorToBuffer()`

Zanim finalny kolor piksela zostanie zapisany w buforze ramki, zmienoprzecinkowy wektor  $\overrightarrow{color}$  przechowujący informację o kolorze zostaje poddany konwersji do 3 8-bitowych wartości R, G, B za pomocą prostej transformacji:

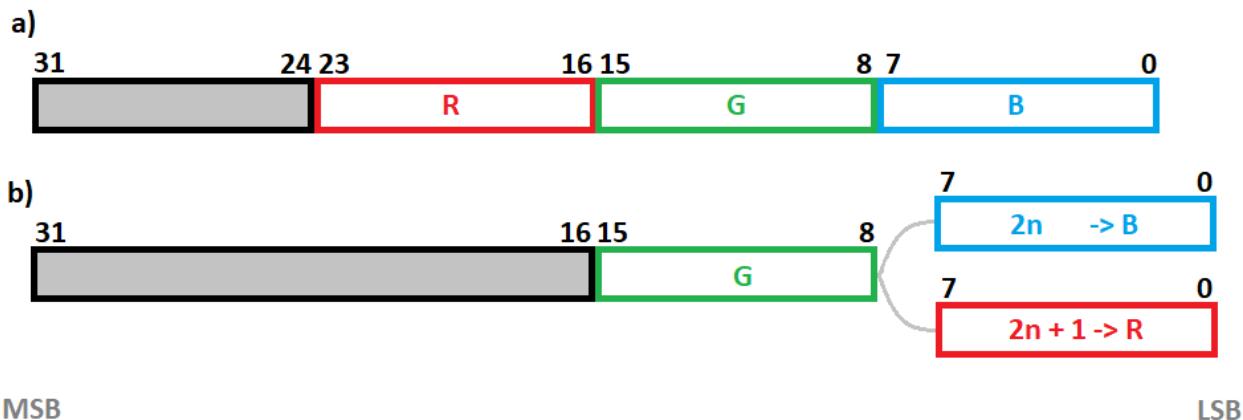
```

1 unsigned tempColor [3];
2 for (unsigned i = 0; i < 3; ++i)
3 {
4     // R: i = 0
5     // G: i = 1
6     // B: i = 2
7     if (color [i] > float (1.0)) color [i] = float (1.0);
8     tempColor [i] = unsigned(color [i] * 255.0f);
9 }
```

Sprawdzenie warunku w ciele pętli jest konieczne po to, aby wyjściowe wartości składowych koloru nie przekroczyły maksymalnej wartości możliwej do zapisania za pomocą 8 bitów. Konsekwencją tego rozwiązania są tzw. *przepalenia*, czyli obszary, gdzie intensywność którejkolwiek ze składowych w wektorze  $\overrightarrow{color}$  są większe niż 1 (patrz komentarz pod rysunkiem 3.15).

Otrzymane w ten sposób wartości są następnie pakowane w jedną 32-bitową wartość reprezentującą finalny kolor piksela przy czym proces ten jest kontrolowany poprzez

definicję `PIXEL_COLOR_CONVERSION_ENABLE`, co zostało pokazane na poniższym schemacie.



**Rys. 3.25:** Upakowanie danych koloru RGB w 32-bitowej wartości w buforze ramki. Przypadek a) odpowiada sytuacji, gdy `PIXEL_COLOR_CONVERSION_ENABLE` nie zostało zdefiniowane - wtedy każdy 32-bitowy element bufora ramki przechowuje informacje o wszystkich trzech składowych koloru. W sytuacji b), gdy `PIXEL_COLOR_CONVERSION_ENABLE` zostało zdefiniowane, każdy 32-bitowy element bufora ramki niesie ze sobą informację o składowej zielonej G - informacje o składowych niebieskiej B i czerwonej R zapisywane są naprzemiennie w kolejnych elementach (dany piksel posiada albo informację o kanale B albo R), przy czym jako pierwsza występuje składowa B ( $n = 0$ )

Trybu zapisu, który implikuje definicja `PIXEL_COLOR_CONVERSION_ENABLE`, został stworzony po to, aby na płytce ewaluacyjnej KCU116 w pełnym systemie przetwarzania danych możliwe było bezpośrednie wykorzystanie danych bufora ramki dostarczanych przez VIRAY bez konieczności stosowania dodatkowych modułów dokonujących odpowiedniej konwersji danych.

VIRAY - wydajność oraz estymacja wymaganych zasobów

Dla parametrów modułu ViRAY zgodnych z danymi zebranymi w tabeli 3.5, okresu zegara równego 3,3 ns oraz układu KCU116 raport po syntezie HLS (w wersji 2017.4) przedstawia się następująco:

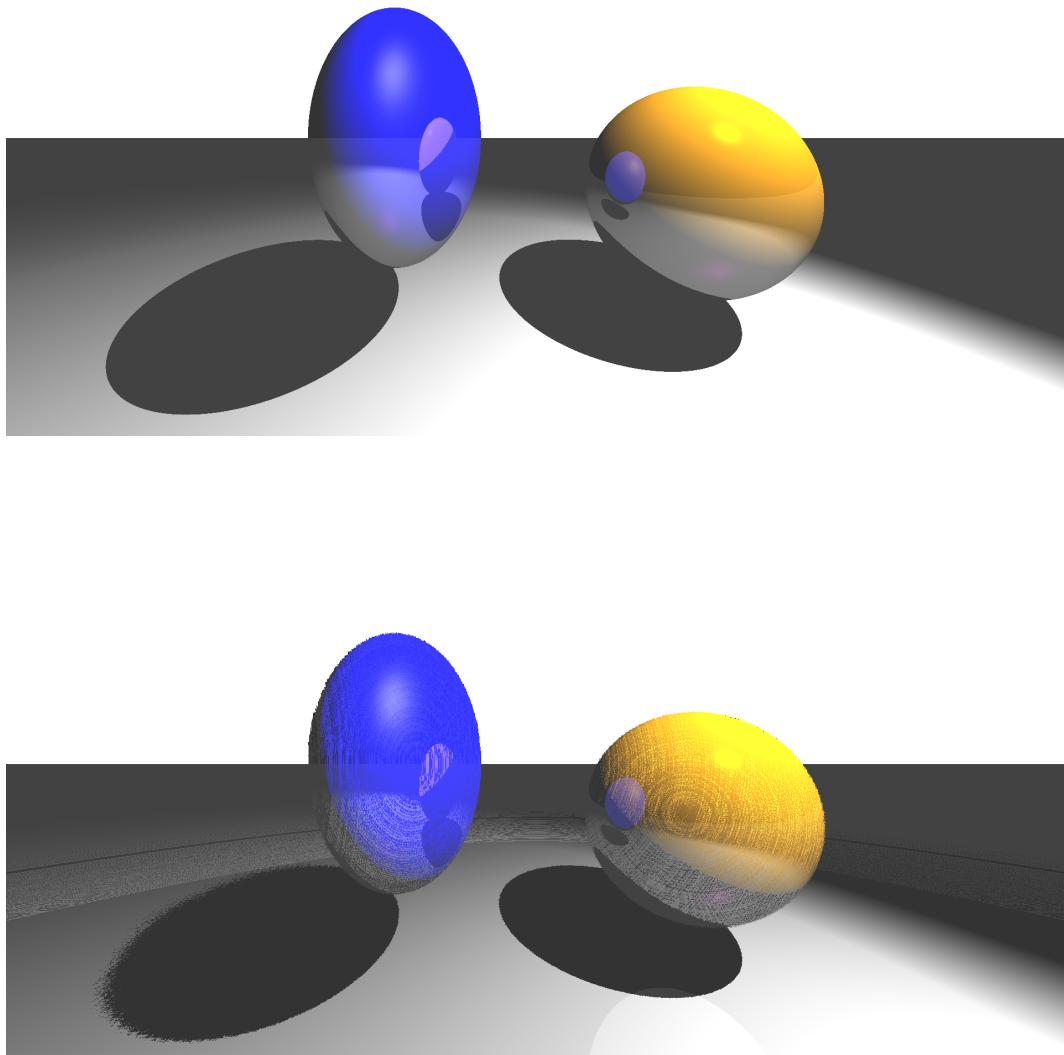
**Tab. 3.8:** Wykorzystanie zasobów układu KCU116 oraz szybkość finalnej wersji modułu ViRAY. Dolny wiersz odpowiada sytuacji, gdyby zamiast 32-bitowych liczb zmiennoprzecinkowych `float` do obliczeń w całym module wykorzystać typ połówkowej precyzji `half`. Ostatnia kolumna (#) prezentuje ilość cykli zegara potrzebnych na wykonanie wszystkich zadań związanych z tworzeniem pojedynczej ramki obrazu w rozdzielczości 1920 x 1080 pikseli (Full HD) oraz przeliczenie tej wartości na czas przetwarzania w milisekundach

	<b>BRAM</b>	<b>DSP</b>	<b>FF</b>	<b>LUT</b>	#
<code>float</code>	164 (17,1%)	911 (50,0%)	348999 (80,4%)	173126 (79,8%)	16765083 (55,3 ms)
<code>half</code>	170 (17,7%)	835 (45,8%)	176873 (40,8%)	111854 (51,6%)	16734245 (55,2 ms)

Podane powyżej wartości odnoszą się do ramki obrazu 1920 x 1080 pikseli (ok. 2,07 mln pikseli), generowanej przez moduł, który jest implementowalny w fizycznym układzie FPGA. Wynika z nich, iż wymagania dotyczące minimalnej płynności (tj. 100 ms na ramkę obrazu) oraz rozdzielczości (1 mln pikseli) zostały zrealizowane. Ponadto czas generowania obrazu jest stały i zadany przez zastosowaną konfigurację. Jest to niewątpliwie zasługa poczynionych optymalizacji algorytmów oraz uproszczeń, dzięki którym możliwe jest generowanie kolejnych pikseli obrazu z interwałem równym 8 cykli zegara (`DESIRED_INNER_LOOP_II`).

W tabeli 3.8 została również umieszczona informacja dotycząca sytuacji, gdyby wszystkie obliczenia ViRAY przeprowadzały z użyciem typu `half` zamiast `float`. Mimo iż raportowana przez Vivado HLS utylizacja zasobów jest tylko estymacją, różnice są na tyle duże, iż można założyć, że zaoszczędzone elementy układu FPGA można byłoby przeznaczyć do implementacji dodatkowych funkcjonalności bądź zmniejszenia interwału między pikselami. Z drugiej strony takie zmniejszenie precyzji nie pozostawałoby bez wpływu na jakość generowanej informacji wizualnej (rysunek 3.26).

W tym miejscu należy wytknąć twórcom Vivado HLS niekonsekwencję związaną z dostępnymi w Vivado HLS typami danych. Skoro typy całkowitoliczbowe oraz stałopozytywne mogą zostać zdefiniowane przez użytkownika w dowolny sposób, z dowolną ilością bitów, niezrozumiałym jest dostęp do jedynie podstawowych typów zmiennoprzecinkowych tj. `half`, `float` oraz `double`. Obecność typów zmiennoprzecinkowych dowolnej precyzji o rozmiarze pomiędzy 16 a 32 bitami pozwoliłaby przeprowadzić dodatkowe eksperymenty i zoptymalizować ViRAY w jeszcze większym stopniu niż możliwe jest to obecnie.



**Rys. 3.26:** Porównanie obrazów wygenerowanych przez wczesną wersję modułu ViRAY. Ta sama scena została wygenerowana z użyciem pełnej (góra) oraz połówkowej (dół) precyzji obliczeń zmiennoprzecinkowych. W tym drugim przykładzie uwidaczniają się artefakty wynikające ze zbyt niskiej precyzji obliczeń, przy czym zmiana wartości parametru CORE\_BIAS nie dawała zauważalnej poprawy rezultatów

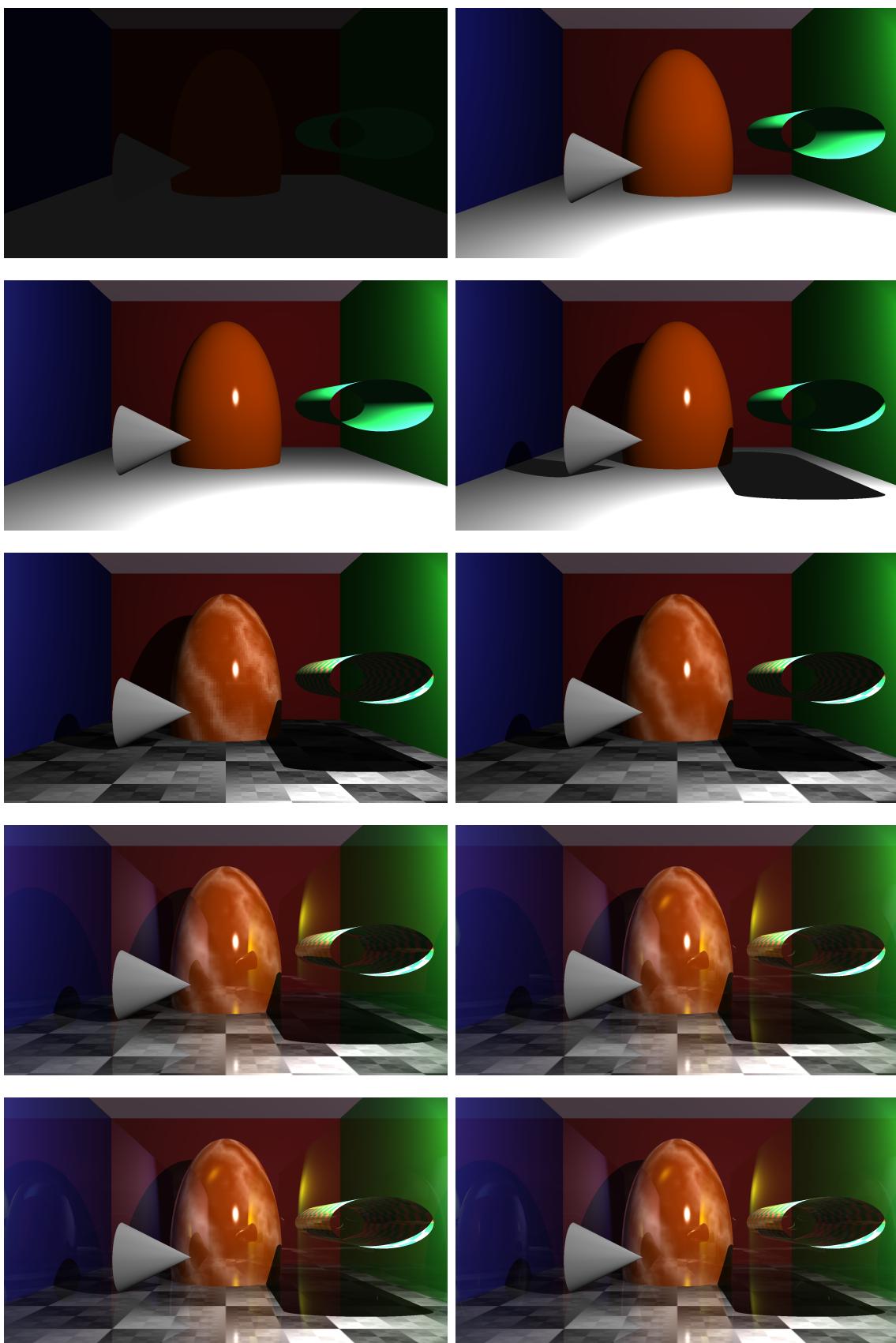
#### ViRAY - wpływ zastosowanej konfiguracji na złożoność sceny

Przedstawione do tej pory obrazy tworzone za pomocą ViRAY'a demonstrowały tylko pewne elementy funkcjonalności finalnego modułu ważne do zilustrowanie danego zagadnienia. Poniższa seria wygenerowanych obrazów (rysunek 3.27) nie tylko pokazuje działanie ViRAY'a w bardziej złożonej scenie, ale również pozwala poznać wpływ większości kluczowych parametrów na jej ostateczny wygląd. Wszystkie te obrazy, tak samo jak reszta w tym podrozdziale, są wynikiem działania środowiska symulacyjnego, które powstawały dzięki włączaniu kolejnych funkcjonalności, a ich numery zaczynają się od 1 i rosną od lewej do prawej, z góry na dół.

1. Scena oświetlona tylko poprzez globalne światło otoczenia **AMBIENT\_COLOR\_ENABLE**.
2. Włączenie światła rozproszonego **DIFFUSE\_COLOR\_ENABLE**.
3. Dodanie czynnika **SPECULAR\_HIGHLIGHT\_ENABLE** związanego z generowaniem rozbłyśków zwierciadlanych.
4. Umożliwienie rzucania cieni przez obiekty **SHADOW\_ENABLE**.
5. Nałożenie tekstur na obiekty **TEXTURE\_ENABLE**.
6. Filtracja biliniowa tekstur **BILINEAR\_TEXTURE\_FILTERING\_ENABLE** - sprawia, że tekstuра na centralnym obiekcie staje się znacznie gładzsza.
7. Rozpoczęcie generowania odbić na obiektach **RAYTRACING\_DEPTH = 2** - konfiguracja domyślna.
8. **RAYTRACING\_DEPTH = 3**.
9. **RAYTRACING\_DEPTH = 4**.
10. **RAYTRACING\_DEPTH = 10**.

Największy skok w jakości uzyskiwany jest w momencie, gdy zaczynają być generowane cienie (4), które pozwalają znacznie łatwiej zorientować się w przestrzennej relacji między obiektami a światłem oraz po włączeniu możliwości generowania promieni rozproszonych (7). Dalsze zwiększanie parametru związanego z głębokością śledzenia promieni dodaje kolejne detale na powierzchniach obiektów, jednak każdy kolejny promień wnosi coraz mniejszy wkład do końcowego efektu, gdyż  $0 \leq r_{l-1} \leq 1$  (patrz równanie (3.2.9)), w wyniku czego różnice między obrazami dla **RAYTRACING\_DEPTH = 4** oraz **RAYTRACING\_DEPTH = 10** są niewielkie.

Wartym uwagi parametrem jest również czas, w jakim wygenerowany został obraz (7). Ten odpowiada konfiguracji domyślnej V-Ray'a dostosowanej do działania na układzie KCU116 w czasie rzeczywistym. Na komputerze wyposażonym w procesor Intel Core i5 4690K działającym z częstotliwością 3,9 GHz, czas oczekiwania na wynik symulacji, będącej wykonywalnym jednowątkowym programem napisanym w języku C++, wynosi ok. 67,6 s, czyli jest 3 rzędy wielkości dłuższy, niż gdyby ta sama scena była generowana bezpośrednio w układzie FPGA za pomocą stworzonego modułu ( $\frac{67,6 \text{ s}}{55,3 \text{ ms}} \approx 1222$ ).



Rys. 3.27: Wpływ konfiguracji na poziom komplikacji generowanego obrazu. Umożliwienie generowania cieni, a następnie włączenie obliczeń związanych z tworzeniem odbić na powierzchniach najbardziej wpływają na realizm przedstawionej sceny.

### 3.3 Implementacja modułu VIRAY w układzie FPGA

Zaprezentowany w poprzednim podrozdziale moduł VIRAY jest w pełni funkcjonalny w momencie, gdy zostanie zaimplementowany w układzie FPGA oraz zajdą jednocześnie dwa warunki:

1. Będzie istniała możliwość dostarczenia do niego poprawnych danych opisujących scenę, która ma zostać wygenerowana.
2. Efekty pracy modułu, czyli finalne ramki obrazu, będą wizualnie weryfikowalne.

Na podstawie tychże wytycznych został zaprojektowany pełen system przetwarzania danych, którego uproszczona wersja schematu blokowego została przedstawiona na schemacie 3.28.

Za kontrolę działania omawianego systemu oraz dostarczenie poprawnych danych do VIRAY'a odpowiada program w języku C++ wykonywany przez mikrokontroler **Microblaze**. Został on skonfigurowany w taki sposób aby mógł wykonywać sprzętowo, a nie poprzez emulację, wszystkie operacje zmiennoprzecinkowe pojedynczej precyzji. Dodatkowo jego pamięć na dane oraz instrukcje została rozszerzona do 256 kB każda i nie korzysta on z pamięci podręcznej. Do dyspozycji mikrokontrolera zostały oddane również dodatkowe moduły jak:

- UART, dzięki któremu możliwe jest odbieranie i wysyłanie komunikatów tekstowych podawanych przez wyjście/wejście standardowe.
- Licznik czasu pozwalający przeprowadzić diagnostykę czasu wykonania operacji.
- GPIO, które w przypadku tego konkretnego systemu daje informacje na temat statusu wciśnięcia 5 przycisków zlokalizowanych w prawym dolnym rogu płytki KCU116.

Dzięki obecności kontrolera pamięci DDR4 **Microblaze** (oraz pozostałe moduły) jest w stanie odwoływać się do puli zewnętrznej względem układu FPGA pamięci RAM, w której mogą być przechowywane większe porcje informacji takie jak finalny obraz wygenerowany przez VIRAY.

Obraz ten jest pobierany w trybie ciągłym z pominięciem udziału mikrokontrolera za pomocą modułu VDMA (ang. *Video Direct Memory Access*, bezpośredni dostęp do pamięci wideo)<sup>xxviii</sup>, synchronizowany względem zegara wynikającego z docelowego formatu (ten zadany jest przez rozdzielcość aktywnego rozmiaru ramki oraz częstotliwość odświeżania obrazu) a następnie podany na zewnątrz układu FPGA (VTC, ang. *Video Timing Controller; Video Out*).

Przetworzone w zaprezentowany sposób dane wizualne trafiają na wejście periferyjnego układu ADV7511 znajdującego się na płytce ewaluacyjnej KCU116, a którego zadaniem jest translacja otrzymanych danych, wedle ustalonych przez użytkownika parametrów pracy, do informacji zgodnej ze standardem HDMI, dzięki czemu finalny obraz może być zaobserwowany po podłączeniu monitora do dostępnego wyjścia obrazu. Parametry pracy układu ADV7511 [63] muszą zostać ustalone przez **Microblaze** za pośrednictwem szeregowej szyny transmisji danych IIC [64].

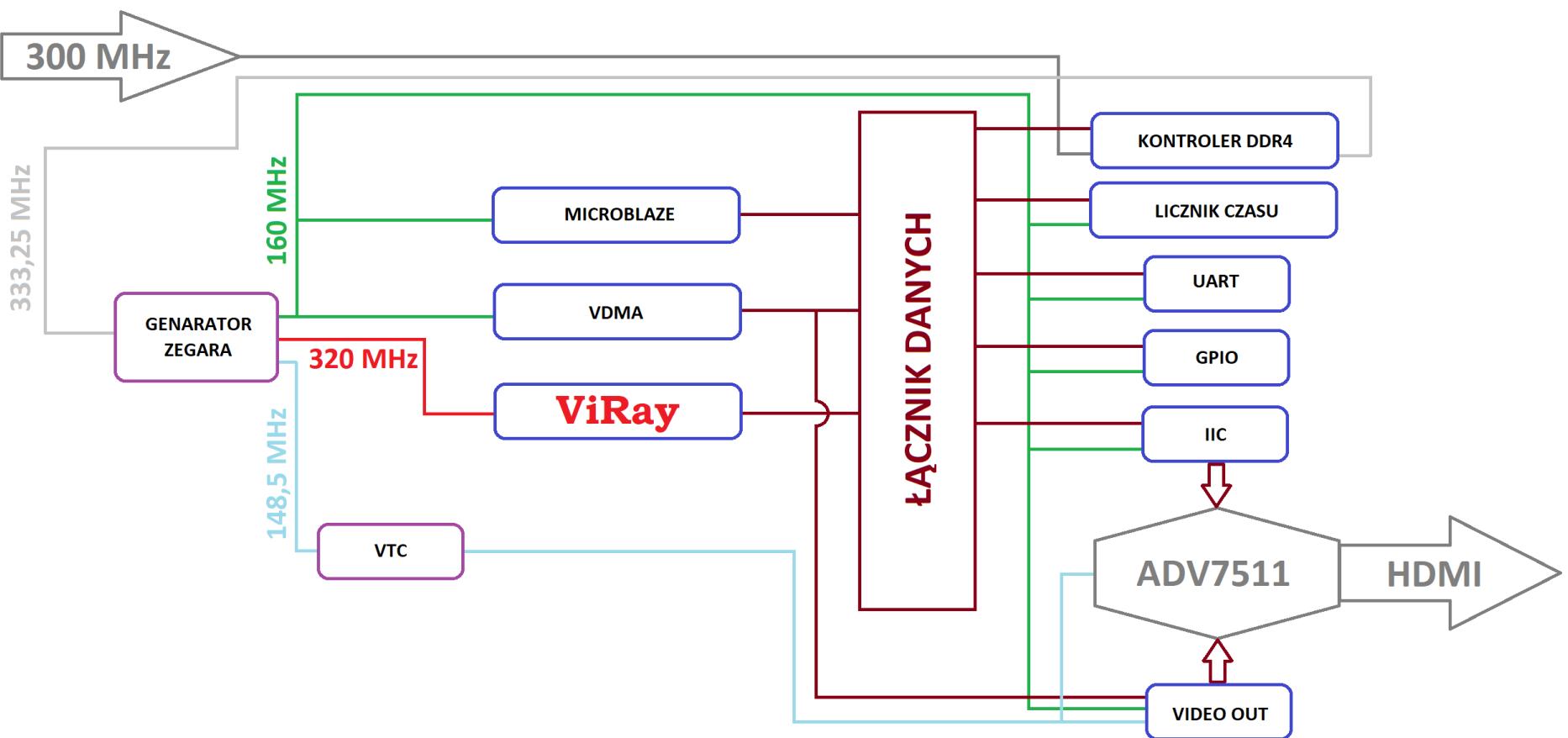
<sup>xxviii</sup>Rola mikrokontrolera w tym konkretnym zadaniu sprowadza się jedynie do skonfigurowania parametrów VDMA oraz uruchomienia jego działania.

Z układem ADV7511 będącym częścią płytka ewaluacyjnej KCU116 wiąże się obecność dyrektywy `PIXEL_COLOR_CONVERSION_ENABLE` w finalnej wersji ViRAY'a. Konstruktorzy tejże płytki postanowili dokonać redukcji możliwych do wykorzystania linii sygnałowych [48] wchodzących do układu ADV7511 z 36 do 18<sup>xxix</sup>. W efekcie w jednym cyklu zegara możliwe jest przesłanie do 18 bitów danych, jednak pełna informacja o kolorze wymaga szyny o szerokości co najmniej  $3 \cdot 8 \text{ b} = 24 \text{ b}$ . Definicja `PIXEL_COLOR_CONVERSION_ENABLE` redukuje to wymaganie do 16 bitów (32-bitowa wartość reprezentująca kolor piksela zostaje obcięta do jej 16 najmłodszych bitów, gdyż najstarsze bity nie niosą użytecznej informacji), a brakujące wartości B/R pikseli zostają w pewnym zakresie zrekonstruowane przez układ ADV7511<sup>xxx</sup>.

---

<sup>xxix</sup>Podane ograniczenie nie występuje w płytce VC707.

<sup>xxx</sup>Artefakty mogą się pojawiać na kontrastowych krawędziach między obiektami.



Rys. 3.28: Uproszczony schemat blokowy pełnego systemu przetwarzania danych wykorzystujący stworzony moduł do śledzenia promieni VIRay

Stworzony system wykorzystuje 5 głównych sygnałów zegarowych:

1. Systemowy zegar o częstotliwości 300 MHz, który jest podstawą działania kontrolera pamięci DDR4.
2. Zegar o częstotliwości 333,25 MHz, wytwarzany domyślnie przez kontroler pamięci i wykorzystywany jako baza do wygenerowania pozostałych częstotliwości.
3. Zegar 160 MHz, który wpływa na szybkość działania i komunikacji większości modułów (w szczególności Microblaze oraz VDMA).
4. Główny moduł ViRAY pracuje z częstotliwością 320 MHz.

Domyślnie kod ViRAY'a jest poddawany syntezie HLS z uwzględnieniem zegara o okresie 3,3 ns (ok. 303 MHz). Mimo to, moduł ten może zostać z powodzeniem zaimplementowany, gdy częstotliwość jego pracy wynosi 320 MHz. Dzięki temu, teoretycznie wyznaczony czas wygenerowania pełnej klatki maleje o 2,93 ms i wynosi 52,4 ms, dając ok. 19 pełnych klatek obrazu 1920 x 1080 na sekundę. Liczby te można również przedstawić w innej powszechnie spotykanej postaci, mówiącej o tym, ile milionów promieni ViRAY jest w stanie przetworzyć w ciągu sekundy (`RAYTRACING_DEPTH = 2` daje 4 promienie na 1 piksel obrazu):

$$4 \cdot \frac{1920 \cdot 1080}{52,4 \text{ ms}} = 158,29 \cdot 10^6 \frac{1}{\text{s}} \quad (3.3.1)$$

Co ciekawe, jeśli ViRAY zostanie poddany syntezie Vivado HLS bezpośrednio dla częstotliwości 320 MHz, stworzonego w ten sposób modułu nie da się zaimplementować w układzie FPGA z częstotliwością pracy równą 320 MHz.

5. Zegar synchronizujący dane obrazu o częstotliwości 148,5 MHz.

ViRAY został skonfigurowany domyślnie w taki sposób, by generować obrazy o rozdzielczości 1920 x 1080 pikseli. Jednakże ramka obrazu HDMI poza danymi pikseli zawiera również dane dodatkowe (w tym synchronizacyjne) dołączane do obrazu przez `Video Out`, w efekcie pełna ramka obrazu w podanej częstotliwości składa się z 2200 poziomych oraz 1125 pionowych linii<sup>xxxxi</sup>. Chcąc zapewnić częstotliwość odświeżania obrazu równą 60 Hz zegar synchronizacyjny musi mieć wartość:

$$2200 \cdot 1125 \cdot 60 \text{ Hz} = 148,5 \text{ MHz}$$

Syntezę pełnego systemu przeprowadzono z użyciem strategii `Vivado Synthesis Defaults`, a do implementacji `Performance_ExplorePostRoutePhysOpt` (z uwagi na podejmowane przez Vivado w tym przypadku próby optymalizacji struktury, w celu umożliwienia osiągnięcia wysokich częstotliwości pracy). W tabeli 3.9 przedstawione zostały parametry charakteryzujące cały system po implementacji oraz utylizację zasobów przez ViRAY po syntezie do reprezentacji za pomocą netlisty.

<sup>xxxxi</sup>Ilość dodatkowych linii zależy od przyjętej rozdzielczości aktywnego obrazu i można je sprawdzić dla popularnych częstotliwości w oknie konfiguracyjnym modułu `Video Timing Controller`.

**Tab. 3.9:** Wykorzystanie zasobów sprzętowych układu KCU116 przez zaprojektowany system przetwarzania danych. Środkowy wiersz dotyczy pełnego systemu ze wszystkimi modułami po pomyślnej implementacji, zaś dolny samego modułu ViRAY po syntezie do postaci netlisty. Raportowane po syntezie wykorzystanie zasobów przez ViRAY jest znacznie niższe niż sugerują to estymowane wartości podawane przez Vivado HLS

	<b>BRAM</b>	<b>DSP</b>	<b>FF</b>	<b>LUT</b>	<b>WNS [ns]</b>	<b>WHS [ns]</b>
<b>System</b>	430 (44,79%)	919 (50,38%)	238201 (54,90%)	153922 (70,94%)	0,032	0,006
<b>ViRAY</b>	232 (24,17%)	911 (49,95%)	194557 (44,84%)	130077 (59,95%)	-	-

Pierwszą rzeczą, którą należy zauważyć jest to, iż cały system przetwarzania, zawierający oprócz ViRAY'a jeszcze wiele innych modułów, wykorzystuje mniejszą ilość FF oraz LUT niż sam ViRAY po syntezie HLS (tabela 3.8). Dzieje się tak dlatego, iż Vivado HLS dokonuje estymacji wykorzystania elementów FPGA w przypadku najbardziej pesymistycznym. Procesy, które zachodzą w wyniku syntezы modułów do postaci netlisty oraz wywołana kolejno implementacja dokonują wielu różnych optymalizacji logiki, które zależą od wybranych strategii. Ten sam projekt zaimplementowany za pomocą innej strategii da inne wyniki. W szczególności narzędzie mogłoby nie być w stanie zapewnić by nie występuły naruszenia (WNS/WHS stałyby się ujemne), albo wręcz przeciwnie stworzona struktura byłaby na tyle optymalna, iż możliwe byłoby podniesienie częstotliwości zegara. Z uwagi na wysoki poziom komplikacji projektu oraz mnogość dostępnych parametrów syntezы i implementacji poszukiwanie bezwzględnie najlepszych ustawień byłoby czasochłonne i w gruncie rzeczy nieuzasadnione<sup>xxxii</sup>.

Całkowita estymowana moc energii elektrycznej pobieranej przez KCU116 w trakcie pracy przedstawionego systemu wynosi 13,444 W z czego 95% to moc dynamiczna, czyli związana z przełączaniem stanu tranzystorów. Wartość ta w porównaniu z mocą potrzebną do zasilenia procesorów współczesnych komputerów osobistych jest niewielka<sup>xxxiii</sup>, zwłaszcza gdy weźmie się pod uwagę uzyskiwany przyrost wydajności rzędu  $10^3$ .

### 3.4 Oprogramowanie mikrokontrolera

Zaletą procesora Microblaze jest nie tylko elastyczność pod względem tworzenia konfiguracji, ale przede wszystkim fakt, iż można go programować tak jak zwykły procesor za po-

<sup>xxxii</sup>Najważniejszym zadaniem na tym etapie było zapewnienie poprawności działania całego systemu. Błędy popełnione w projekcie schematu blokowego skutkują niepoprawnym działaniem systemu lub jego części i są trudno wykrywalne, czego doświadczono dodając funkcjonalność związaną z transmisją obrazu. Na komputerze z procesorem Intel Core i5 4690K oraz 16 GB pamięci operacyjnej przeprowadzenie syntezы oraz implementacji trwa niemal 6 godzin, co w znacznym stopniu wpływa na produktywność i szybkość wdrażania wszelkich poprawek.

<sup>xxxiii</sup>Oszacowanie mocy energii elektrycznej pobieranej przez procesor nie jest łatwym zadaniem. Producenci posługują się tzw. *współczynnikiem TDP* (ang. *Thermal Design Power*), który informuje jedynie o średniej mocy promieniowania termicznego emitowanego przez procesor w określonych przez producenta warunkach.

mocą języka C/C++. Fakt ten jest intensywnie wykorzystywany przez napisany sterownik, dzięki któremu następuje nawiązanie poprawnej komunikacji ze wszystkimi modułami znajdującymi się w projekcie, przygotowanie danych oraz obsługa zdarzeń<sup>xxxiv</sup>.

Najważniejszym elementem sterownika decydującym o pracy ViRAY'a jest obiekt klasy `MyWorld`, która dziedziczy po klasie `WorldDescription`. Zadaniem klasy bazowej jest ustalenie kontraktu między sterownikiem a modułem śledzenia promieni dotyczący tego, gdzie w przestrzeni adresowej znajdują się dane, które będą interpretowane przez ViRAY jako odpowiednie dane opisujące scenę (tj. kamera, obiekty, światła, materiały, tekstury) oraz jaki adres odpowiadać ma początkowi danych opisujących finalne piksele obrazu. W tym celu wykorzystywany jest sterownik generowany automatycznie przez Vivado HLS, udostępniający funkcje odpowiedzialne m. in. za przypisanie odpowiednich adresów, które będą wykorzystane przez porty, dla których zastosowano interfejs `AXI4-Lite`.

Ponadto klasa `WorldDescription` udostępnia klasie potomnej zestaw narzędzi pozwalających w łatwy sposób definiować poszczególne elementy związane z konfiguracją m. in. światła, obiektów i ich materiałów. Dzięki temu użytkownik w intuicyjny sposób może dokonywać zmian wymaganych w danej chwili parametrów i nie jest zmuszony martwić się osobiste tym, w jaki sposób dane dotyczące obiektów należy upakować w pamięci, gdyż dzieje się to automatycznie.

**Listing 3.1:** Przykład konfiguracji obiektu `floor`. Za pomocą odpowiednich funkcji `Set*`() można dokonać zmiany wartości dowolnych parametrów interpretowanych przez ViRAY()

```

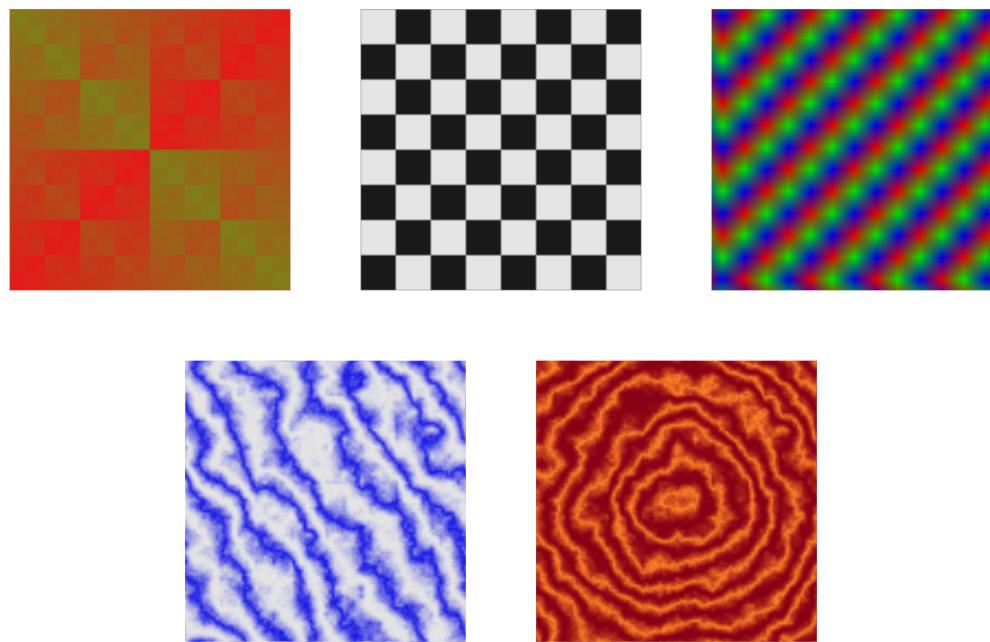
1 floor . SetObjType( ObjectType :: DISK );
2 floor . GetTransformHandler() . SetTranslation( vec3( 0.0f , -2.0f , 0.0f ) );
3 floor . GetTransformHandler() . SetOrientation( vec3( 0.0f , 1.0f , 0.0f ) );
4 floor . GetTransformHandler() . SetScale( vec3( 20.0f ) );
5
6 floor . GetMaterialHandler() . SetPrimaryDiffuseColor( vec3( 0.9f ) );
7 floor . GetMaterialHandler() . SetSecondaryDiffuseColor( vec3( 0.1f ) );
8 floor . GetMaterialHandler() . SetTextureScale( vec3( 6.0f ) );

```

Dane tekstur generowane są za pomocą instancji klasy typu `CTextureHelper` znajdującej się wewnątrz `WorldDescription`, która umożliwia tworzenie tekstur na podstawie 5 dostępnych algorytmów (rysunek 3.29) oraz przypisanie jej do obiektów z żądanym trybem mapowania - 4 z nich (`XOR`, `CHECKERBOARD`, `MARBLE` oraz `WOOD`) generując mapy szarości, na podstawie których dokonywane jest mieszanie między dwoma kolorami wybranymi przez użytkownika (`PrimaryDiffuseColor`, `SecondaryDiffuseColor`) zaś ostatnia `RGB_DOTS` tworzy prostą mapę bitową, w której kolejne piksele mają kolory podstawowe: czerwony, zielony, niebieski.

---

<sup>xxxiv</sup>Pliki tego projektu znajdują się w repozytorium [https://bitbucket.org/rtMasters/ffcore\\_uc\\_kintex/commits/branch/KCU116\\_VIRAY\\_HDMI\\_FULL\\_SYSTEM](https://bitbucket.org/rtMasters/ffcore_uc_kintex/commits/branch/KCU116_VIRAY_HDMI_FULL_SYSTEM). Do poprawnego działania są również wymagane pliki ViRAY'a z definicją `UC_OPERATION`.



**Rys. 3.29:** Typy tekstur mogące zostać wygenerowane przez sterownika. Od lewej do prawej z góry na dół: XOR, CHECKERBOARD, RGB\_DOTS, MARBLE, WOOD. Wygląd tekstur może być konfigurowany za pomocą opcjonalnych parametrów (dotyczy zwłaszcza dwóch ostatnich przykładów, których wygląd zależy od parametrów rozkładów liczb pseudolosowych)

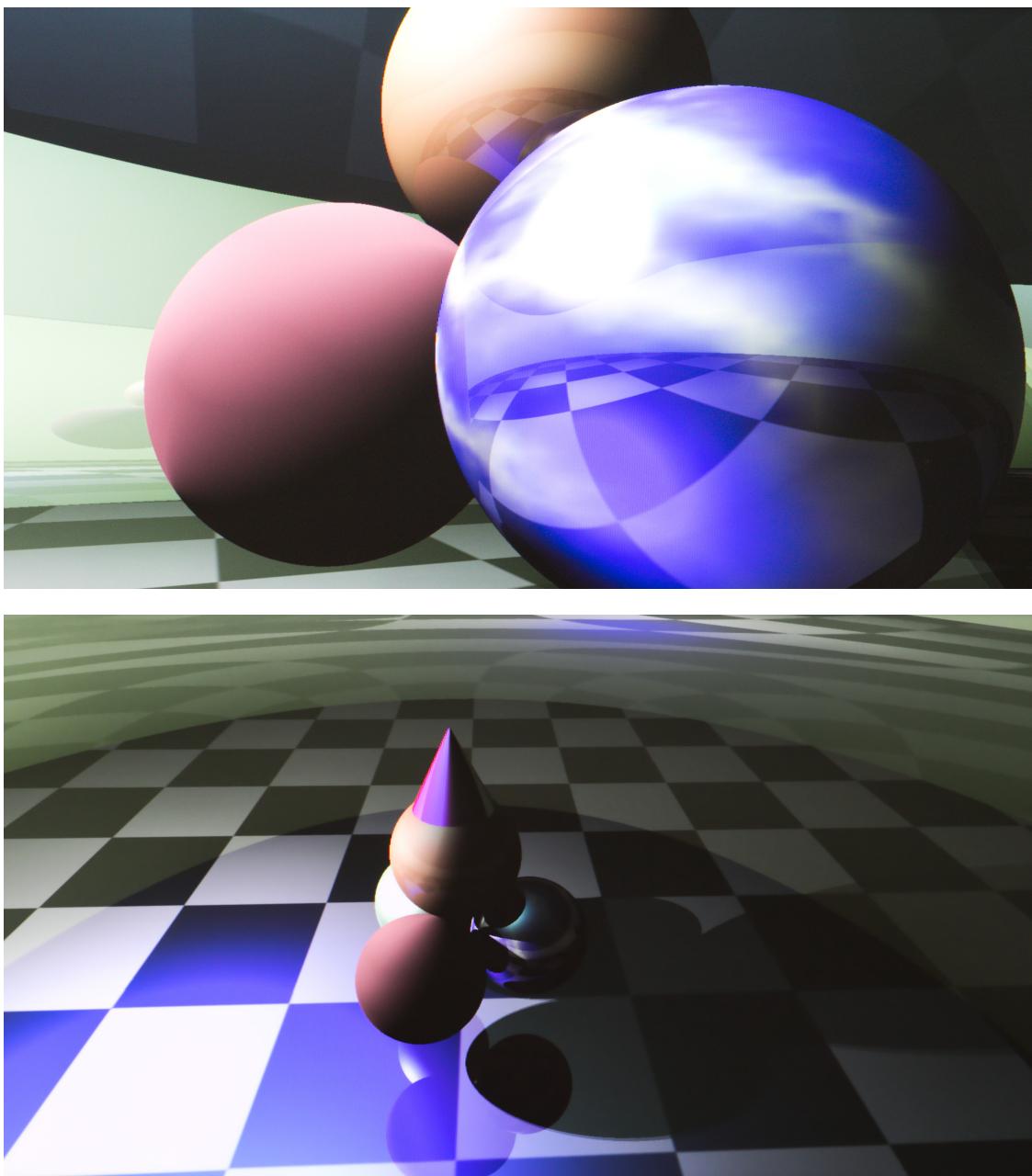
W domyślnej konfiguracji po

- zapisaniu poprzez magistralę IIC konfiguracji układu ADV7511 dostosowanej do formatu przesyłanych danych wizualnych,
- ustawieniu parametrów i uruchomieniu transferu danych za pomocą VDMA,
- zapisaniu danych dotyczących generowanej sceny trójwymiarowej,

VIRAY zostaje uruchomiony w trybie samopowtarzanym. Dzieje się to za pomocą sekwencji wywołań funkcji `XViraymain_EnableAutoRestart()` oraz `XViraymain_Start()`, a sam sterownik przechodzi do nieskończonej pętli głównej, w której dokonuje aktualizacji sceny. Aktualizacja ta wynika:

- z czasu  $\Delta t$ , który upływał między kolejnymi iteracjami pętli, a który jest liczony dzięki obecności w systemie licznika cykli zegara,
- możliwej akcji użytkownika, który mógł dokonać naciśnięcia przycisków obsługiwanych przez moduł GPIO.

W przykładowej scenie, która jest realizowana przez kod znajdujący się w repozytorium, upływ czasu pozwala dokonać zmian położenia źródła światła, przesuwać teksturę podłożą czy zmieniać rozmiary obiektów zgodnie z przyjętym algorytmem. Naciśnięcie przycisków na płytce KCU116 jest zaś traktowane, jako chęć zmiany położenia lub orientacji obserwatora w przestrzeni pozwalając ustawić kamerę w pożądanym miejscu. Poniższe ilustracje przedstawiają pracę opisywanego systemu.



**Rys. 3.30:** Wygenerowane w czasie rzeczywistym klatki animacji otrzymane dzięki wykorzystaniu techniki śledzenia promieni akcelerowanej przez stworzony z użyciem Vivado HLS moduł ViRAY. Finalna wersja systemu pozwala generować interaktywne obrazy o rozdzielczości 1920 x 1080 pikseli ze stałą szybkością 19 klatek na sekundę

## Podsumowanie

Przedstawiony rozdział stanowi najważniejszą część prezentowanej pracy. Wykorzystując wiedzę teoretyczną zebraną w dwóch poprzednich rozdziałach podjęte zostały próby stworzenia systemu, którego celem byłaby akceleracja tworzenia obrazów metodą śledzenia promieni. Okazało się, iż na obecnym etapie rozwoju Vivado HLS nie jest możliwe stworzenie rozsądniego pod względem funkcjonalności, szybkości oraz użytych zasobów rozwiązania opartego o sieć wyspecjalizowanych procesorów arytmetycznych.

Z powodzeniem natomiast zaprojektowano moduł spełniający postawione przed projektem wymagania i charakteryzujący się tym, że jego funkcjonalność jest stała i kontrolowana poprzez skończony zbiór ściśle określonych parametrów wpływających na odbiór obiektów znajdujących się w generowanej scenie. W ten sposób Vivado HLS wiedząc, w jaki konkretny sposób dane są przetwarzane na kolejnych etapach, było w stanie zastosować odpowiednie optymalizacje. Efekt końcowy wymagał włożenia wiele wysiłku i poczynienia rozmaitych kompromisów opisanych w tym rozdziale po to, aby uzyskać moduł o optymalnej funkcjonalności, który na dodatek mógłby zostać zaimplementowany w układzie FPGA.

Kluczowym czynnikiem wpływającym na skomplikowanie akceleratora jest reprezentacja danych, na których wykonywane są operacje. VIRAY domyślnie wykorzystuje typ **float** chociaż istnieje przypuszczenie (znajdujące potwierdzenie w cytowanej literaturze [56]), iż gdyby tylko Vivado HLS pozwalało stosować typy zmiennopozycyjne o dowolnej ilości bitów, VIRAY mógłby być bardziej funkcjonalny przy jednoczesnym zachowaniu wystarczającej precyzji obliczeń.

Ponadto, gdyby nie dostęp do najnowocześniejszego układu FPGA śledzenie promieni rozproszonych nie byłoby możliwe, a przez to obrazy nie różniłyby się znaczco od tych, które można stworzyć na drodze rasteryzacji. Mimo tego i tak trzeba było wykonać wiele ustępstw w stosunku do funkcjonalności (m. in. ograniczyć swobodę umieszczania obiektów w przestrzeni) oraz nie uniknięto problemów, które wynikały z naprawienia drobnych błędów w algorytmie np.:

- poprawienie obliczania reflektancji materiałów przewodzących wymusiło zmniejszenie częstotliwości zegara modułu z 335 MHz do 320 MHz,
- implementacja modelu BRDF Orena-Nayara do samego końca nie była pewna w finalnej wersji VIRAY'a, gdyż dodanie odpowiedniego kodu związanego z obliczeniem właściwego współczynnika całkowicie uniemożliwiło implementację w układzie FPGA nawet dla niskich częstotliwości pracy.

Mimo wielu napotkanych przeciwności, finalny moduł VIRAY został zintegrowany w pełnym systemie przetwarzania danych, kontrolowanym przez mikroprocesor **Microblaze**. Wyniki działania VIRAY'a w postaci animacji są generowane z szybkością 19 klatek obrazu na sekundę w rozdzielczości 1920 x 1080 pikseli i można je zobaczyć podłączając monitor lub telewizor do źródła HDMI znajdującego się na wykorzystanej płytce ewaluacyjnej KCU116.

\* \* \*

Stworzony moduł stanowi tylko bardzo uproszoną implementację śledzenia promieni, którą można poddawać dalszym modyfikacjom.

Zgodnie z założeniami, obiekty w scenie opisane są poprzez odpowiednie równania - różne dla każdego obiektu. Sprawia to, że funkcje badające przecięcia promieni z obiektami wymagają dużych ilości elementów logicznych do ich implementacji. Pod tym względem, świat złożony z trójkątów jest znacznie bardziej atrakcyjny, gdyż wystarczyłoby jedynie sprawdzać przecięcia z jednym typem obiektów. W takim przypadku należałoby stworzyć dodatkowo funkcję/moduł przyspieszający testy przecięcia, gdyż poddane triangulacji obiekty

mogą składać się z tysięcy trójkątów. Taka funkcjonalność musiałaby nie tylko błyskawicznie dokonywać testów geometrii, ale również zarządzać tysiącami obiektów, których dane nie mogą znaleźć się w całości w pamięci blokowej BRAM układu FPGA. Jak się okazuje, wynika z tego pojawienie się problemów, związanych z użyciem Vivado HLS, podobnych do tych, które zostały odkryte podczas opracowywania rozwiązania procesorowego.

Ciekawym dodatkiem mógłby być też pośredni zapis kolorów pikseli do zmienopozycyjnego bufora o znacznie większej dynamice tonalnej (ang. *high dynamic range*, HDR) po to, aby na etapie konwersji do obrazu o niskiej, 8-bitowej, dynamice (ang. *low dynamic range*, LDR) móc lepiej odwzorować detale powierzchni i przeciwdziałać przepaleniom kolorów.

Przyglądając się cytowanej literaturze traktującej o budowie kompleksowego rozwiązania przeprowadzającego symulację transportu światła w przestrzeni [14][59] już w pierwszych rozdziałach prezentowane są podstawy związane z metodą Monte-Carlo, która wykorzystywana jest do implementacji rozkładu promieni w przestrzeni. Koncepcja ta, w niniejszej pracy, ze względu na ograniczone zasoby sprzętowe układów FPGA, została całkowicie pominięta. Wykorzystując metodę Monte-Carlo można m. in. wygładzać granice między obiektami (tzw. *antialiasing*), symulować skończoną głębię ostrości kamery, dodawać źródła światła o skończonej powierzchni czy umożliwić generowanie rozproszonych odbić kierunkowych. Wadą tej metody jest jednak konieczność rozpatrywania znacznie większej liczby promieni, aby uzyskać kolor danego piksela. Aktualnie powstają nowe technologie oparte o uczenie maszynowe, których celem jest zmniejszenie liczby wymaganych promieni dla osiągnięcia tych samych efektów, co tradycyjnie [65][66]. Mimo tych wysiłków, do tej pory wykorzystanie śledzenia promieni w zastosowaniach czasu rzeczywistego pozostaje w sferze badań największych korporacji zajmujących się tym problemem od dekad.

---

## Podsumowanie

---

Celem niniejszej pracy była ewaluacja przydatności w realnych zastosowaniach projektowych narzędzia jakim jest Vivado HLS. Narzędzie to umożliwia osobom znającym tradycyjne sekwencyjne języki programowania rozpoczęcie pracy z układami konfigurowalnymi FPGA bez konieczności znajomości tajników języków opisu sprzętu. Jak się przekonano, nieznajomość języków opisu sprzętu wcale nie zwalnia z konieczności zrozumienia, w jaki sposób zbudowane są układy FPGA oraz skąd może wynikać ich wyższość nad standardowymi układami procesorowymi:

*Lacking such knowledge, it is a big challenge for software engineers to design for FPGA even with the state-of-art HLS tools.<sup>xxxv</sup>[30]*

Jako że opis algorytmiczny problemu za pomocą języka C/C++ jest istotnie różny od konfiguracji sprzętowej z użyciem Verilog czy VHDL, Vivado HLS dokonuje automatycznie szereg transformacji, na działanie których programista ma tylko ograniczony wpływ za pomocą dyrektyw optymalizacyjnych. Z użyciem wielu kluczowych dla wydajności dyrektyw jak PIPELINE czy DATAFLOW wiąże się szereg koniecznych do spełnienia warunków, które przeważnie nie będą spełnione od razu, a dopiero po znacznej modyfikacji algorytmu.

Z powodu narzucanych ograniczeń oraz dość niewielkiego wpływu na ostateczny kształt opisu w postaci RTL nie można powiedzieć, iż Vivado HLS zastąpi pracę doświadczonych projektantów układów elektronicznych. Mogą oni być nim jednak zainteresowani podczas tworzenia i testowania rozwiązań prototypowych, gdyż tworzenie modułów w języku zbliżonym do C jest znacznie szybsze i łatwe w testowaniu.

Nie da się natomiast ukryć, iż grupą docelową użytkowników Vivado HLS są wszyscy ci, którzy chcieliby przekonać się o potencjale, który posiadają układy FPGA, a nie potrafią bądź też nie chcą uczyć się od podstaw języków opisu sprzętu. W ten sposób Vivado HLS jawi się jako cudowne narzędzie, dzięki któremu wiele problemów może zostać rozwiązanych z użyciem układów FPGA znacznie szybciej niż tradycyjnie. Ponadto dodanie bądź też zmiana użytych dyrektyw może w całkowity sposób zmienić zapis syntezowanego modułu za pomocą VHDL czy Verilog, co umożliwia szybką i łatwą eksplorację rozwiązań. Użycie metod tradycyjnych, w celu osiągnięcia podobnego efektu, wymagałoby wielu godzin pracy związanych z reorganizacją kodu i jego testowaniem.

---

<sup>xxxv</sup>Brak takiej wiedzy, stanowi duże wyzwanie dla programistów chcących tworzyć projekty dla układów FPGA, nawet pomimo posługiwania się najnowocześniejszymi narzędziami przeprowadzającymi syntezę wysokiego poziomu. Tłumaczenie własne.

Problemem, którego rozwiązania podjęto się z użyciem Vivado HLS, było generowania realistycznej grafiki metodą śledzenia promieni, która opiera się o prawa rządzące propagacją światła w przestrzeni. Metoda ta polega na badaniu ścieżek, po których poruszają się fotony, zanim trafią do obserwatora tworząc finalny obraz.

Pierwsze próby rozwiązania tego problemu oparte były na fakcie, że obliczenia dla wszystkich ścieżek mogą być przetwarzane masowo równolegle. Z użyciem Vivado HLS stworzono procesor, którego zadaniem było jak najszybsze przetwarzanie instrukcji opisujących algorytm śledzenia promieni. Poza tym musiał on wykorzystywać jak najmniej zasobów układu FPGA po to, aby takich procesorów pracujących jednocześnie można było umieścić w nim jak najwięcej. Okazało się, iż swoboda projektowania funkcjonalności, którą zapewniają języki opisu sprzętu, a która wymagana jest w przypadku tworzenia wydajnego procesora, jest niemożliwa do odtworzenia z użyciem Vivado HLS.

Ostatecznie śledzenie promieni zrealizowano w postaci akceleratora o ustalonej funkcjonalności ViRAY. Z jednej strony rozwiązanie takie pozwoliło Vivado HLS pokazać, iż akceleracja jest osiągalna, gdyż cały algorytm przetwarzania jest zapisany wprost od początku do końca (w przeciwieństwie do procesora, w którym nie wiadomo, jakie instrukcje zostaną kolejno wykonane), z drugiej jednak użytkownik stworzonego modułu może wpływać na wygląd sceny jedynie za pomocą odpowiednich parametrów (implikuje to np. iż nie można korzystać z BSDF, które nie zostały przewidziane na etapie syntezy).

Rozbudowując funkcjonalność modułu trzeba było pogodzić się ze wzrostem zapotrzebowania na elementy logiczne oraz ewentualnymi problemami z implementacją modułu w układzie FPGA związanymi z synchronizacją tak dużego modułu.

Implementowalna w układzie FPGA znajdująca się na płytce ewaluacyjnej KCU116 wersja ViRAY'a generująca obrazy ze stałą szybkością 19 klatek na sekundę w rozdzielcości Full HD pozwala:

- umieścić w scenie maksymalnie 8 obiektów, które mogą być sferą, cylindrem, stożkiem, płaszczyzną, dyskiem bądź kwadratem,
- rozpatrywać pierwszą rodzinę odbitych promieni rozproszonych,
- łączyć jedno globalne światło otoczenia oraz jedno źródło punktowe o modyfikowalnych parametrach (mogące być źródłem cieni),
- wybrać model opisujący zachowanie się powierzchni obiektu spośród modeli BRDF: Blinna-Phonga, Orena-Nayara i Torrance'a-Sparrowa,
- nakładać poddawane filtracji biliniowej tekstury na dowolny rodzaj obiektów.

Parametry te w dość dużym stopniu przewyższają pierwotne oczekiwania, a obraz generowany jest o 3 rzędy wielkości szybciej niż przez procesor komputera. Warto jednak podkreślić, iż nie przeprowadzono optymalizacji parametrów strategii oraz implementacji, który mogłyby skutkować zwiększeniem maksymalnej dopuszczalnej częstotliwości pracy ViRAY'a. W tym przypadku mogłoby pomóc skorzystanie z narzędzia InTime, jednak w momencie tworzenia projektu nie było one dostępne dla Autora.

ViRAY został włączony jako część większego systemu przetwarzania danych, w którym ważną rolę odgrywa wykorzystanie mikroprocesora Microblaze oraz sprzętowego kodaka

sygnału HDMI **ADV7511**. Dzięki temu możliwym jest animowanie sceny poprzez zmianę parametrów obiektów i oglądanie efektów tych zmian na wyświetlaczu monitora/telewizora.



---

## Bibliografia

---

- [1] Segal M., Akeley K., The OpenGL® Graphics System: A Specification, Khronos Group, 2017.
- [2] *Wprowadzenie macierzy rzutu perspektywicznego*, URL <https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/building-basic-perspective-projection-matrix>.
- [3] Wright R., N.Haemel, OpenGL Superbible: Comprehensive Tutorial and Reference (7th Edition), Addison-Wesley Professional, 2015, ISBN 978-0672337475.
- [4] *Specyfikacja procesorów graficznych nvidia*, URL <https://www.geforce.com/hardware>.
- [5] *Specyfikacja procesorów graficznych amd*, URL <https://www.amd.com/en/RX-series>.
- [6] Lauritzen A., *Summed-area variance shadow maps*, URL [https://developer.nvidia.com/gpugems/GPUGems3/gpugems3\\_ch08.html](https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch08.html).
- [7] Umenhoffer T., Patow G., Szirmay-Kalos L., *Robust multiple specular reflections and refractions*, URL [https://developer.nvidia.com/gpugems/GPUGems3/gpugems3\\_ch17.html](https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch17.html).
- [8] A. Appel, *Some techniques for shading machine renderings of solids*, AFIPS Conference Proc, 1968, URL <http://graphics.stanford.edu/courses/Appel.pdf>.
- [9] Vinkler M., *Construction of acceleration data structures for ray tracing*, Brno, 2014.
- [10] Liu X., Deng Y., Ni Y., Li Z., *Fasttree: A hardware kd-tree construction acceleration engine for real-time ray tracing*, Desing Automation & Test in Europe Conference & Exhibition, 1595–1598, 2015.
- [11] Wald I., Havran V., *On building fast kd-trees for ray tracing and on doing that in o(n log n)*, Rap. tech., SCI Institute, University of Utah, 2006.
- [12] Hecht E., Optyka, Wydawnictwo Naukowe PWN, Warszawa, 2012, ISBN 9788301171056.

- [13] Królikowski W., Rubinowicz W., Mechanika teoretyczna, Wydawnictwo Naukowe PWN, 2012, ISBN 9788301172565.
- [14] Pharr M., Humphreys G., Physically Based Rendering - From Theory to Implementation, Morgan Kaufmann, Burlington, 2 wyd., 2010, ISBN 978-0-12-375079-2.
- [15] Conic section, URL <https://upload.wikimedia.org/wikipedia/commons/b/bd/Eccentricity.svg>.
- [16] Montes R., Ureña C., *An overview of brdf models*, Rap. tech., University of Granada, 2012.
- [17] Dallaeva D., *Afm imaging and fractal analysis of surface roughness of alnepilayers on sapphire substrates*, Applied Surface Science, 312, 81–86, 2014.
- [18] Oren M., Nayar S.K., *Generalization of lambert's reflectance model*, SIGGRAPH 94, 239–246, ACM Press, 1994.
- [19] Nayar S., Oren M., *Visual appearance of matte surfaces*, Science, 267, 1153–1156, 1995.
- [20] Thong B., *Illumination for computer generated pictures*, Communications of the ACM, 311–317, 1975.
- [21] Blinn J., *Models of light reflection for computer synthesized pictures*, Computer Graphics (Siggraph), 192–198, 1977.
- [22] Lewis R., *Making shaders more physically plausible*, Fourth Eurographics Workshop on Rendering, 47–62, 1994.
- [23] Lafortune P., Willems Y., *Using the modified phong reflectance model for physically based rendering*, Rap. tech., 1994.
- [24] Meister G., Wiemker R., Monno R., Spitzer H., Strahler A., *Investigation on the torrance-sparrow specular brdf model*, 1998.
- [25] Churiwala S., Designing with Xilinx® FPGAs Using Vivado, Springer International Publishing, 2017, ISBN 978-3-319-42438-5.
- [26] Intel® 64 and ia-32 architectures software developer manuals, URL <https://software.intel.com/en-us/articles/intel-sdm>.
- [27] Compton K., Hauck S., *Reconfigurable computing: A survey of systems and software*, ACM Computing Surveys, 34, 171–210, 2002.
- [28] Xilinx 7-series product selection guide, URL <https://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf>.
- [29] Palnitkar S., Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition, Prentice Hall PTR, 2013, ISBN 0-13-044911-3.

- [30] Koch D., Hanning F., Ziener D., *FPGAs for Software Programmers*, Springer International Publishing, 2016, ISBN 978-3-319-26406-6.
- [31] Kapre N., Ng H., Teo K., Naude J., *Intime: A machine learning approach for efficient selection of fpga cad tool parameters*, ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 23–26, 2015.
- [32] Yanghua Q., Raj C., Ng H., Teo K., Kapre N., *Case for design-specific machine learning in timing closure of fpga designs*, ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 169–172, 2016.
- [33] Yanghua Q., Ng H., Kapre N., *Boosting convergence of timing closure using feature selection in a learning-driven approach*, 26th International Conference on Field Programmable Logic and Applications (FPL), 2016.
- [34] Parkinson M., Taylor P., Parameswaran S., *C to vhdl converter in a codesign environment*, VHDL International Users Forum. Spring Conference, 1994.
- [35] Leung A., Bounov D., Lerner S., *C-to-verilog translation validation*, ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2015.
- [36] Sanchez F., Mateos R., Bueno E., Mingo J., Sanz I., *Comparative of hls and hdl implementations of a grid synchronization algorithm*, 39th Annual Conference of the IEEE Industrial Electronics Society, IECON, 2013.
- [37] Georgopoulos K., Chrysos G., Malakonakis P., Nikitakis A., Tampouratzis N., Dollas A., Pnevmatikatos D., Papaefstathiou Y., *An evaluation of vivado hls for efficient system design*, ELMAR International Symposium, 2016.
- [38] Skalicky S., Wood C., Łukowiak M., Ryan M., *High level synthesis: Where are we? a case study on matrix multiplication*, International Conference on Reconfigurable Computing and FPGAs (ReConFig), 2013.
- [39] Reiche O., Özkan M., Hannig F., Teich J., Schmid M., *Loop parallelization techniques for fpga accelerator synthesis*, Journal of Signal Processing Systems, 90, 3–27, 2018.
- [40] Thomas D., *Synthesisable recursion for c++ hls tools*, 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2016.
- [41] Zuras D., Cowlishaw M., Aiken A., inni, IEEE Standard for Floating-Point Arithmetic, IEEE, 2008, ISBN 978-0-7381-5752-8.
- [42] Vivado design suite user guide: High-level synthesis, URL [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2018\\_1/ug902-vivado-high-level-synthesis.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug902-vivado-high-level-synthesis.pdf).
- [43] Quick start guide: Microblaze soft processor presets, URL [https://www.xilinx.com/support/documentation/quick\\_start/microblaze-quick-start-guide.pdf](https://www.xilinx.com/support/documentation/quick_start/microblaze-quick-start-guide.pdf).

- [44] *Microblaze processor reference guide*, URL [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2018\\_1/ug984-vivado-microblaze-ref.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug984-vivado-microblaze-ref.pdf).
- [45] *Vivado design suite user guide: Synthesis*, URL [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2018\\_1/ug901-vivado-synthesis.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug901-vivado-synthesis.pdf).
- [46] *Vivado design suite user guide: Implementation*, URL [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2018\\_1/ug901-vivado-implementation.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug901-vivado-implementation.pdf).
- [47] *Vc707 evaluation board for the virtex-7 fpga: User guide*, URL [https://www.xilinx.com/support/documentation/boards\\_and\\_kits/vc707/ug885\\_VC707\\_Eval\\_Bd.pdf](https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf).
- [48] *Kcu116 evaluation board: User guide*, URL [https://www.xilinx.com/support/documentation/boards\\_and\\_kits/kcu116/ug1239-kcu116-eval-bd.pdf](https://www.xilinx.com/support/documentation/boards_and_kits/kcu116/ug1239-kcu116-eval-bd.pdf).
- [49] *Ultrascale architecture memory resources: User guide*, URL [https://www.xilinx.com/support/documentation/user\\_guides/ug573-ultrascale-memory-resources.pdf](https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf).
- [50] *7 series dsp48e1 slice: User guide*, URL [https://www.xilinx.com/support/documentation/user\\_guides/ug479\\_7Series\\_DSP48E1.pdf](https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf).
- [51] *Ultrascale architecture dsp slice: User guide*, URL [https://www.xilinx.com/support/documentation/user\\_guides/ug579-ultrascale-dsp.pdf](https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf).
- [52] Patterson D., Hennessy J., Computer Organization and Design: RISC-V Edition, Morgan Kaufmann, 2017, ISBN 9780128122761.
- [53] Moroza L., Walczyk C., Hrynchyshyna A., Holimath V., Cieśliński J., *Fast calculation of inverse square root with the use of magic constant – analytical approach*, Applied Mathematics and Computation, 316, 245–255, 2018.
- [54] Cheah H., Fahmy S., Maskell D., *idea: A dsp block based fpga soft processor*, International Conference: Field-Programmable Technology (FPT), 2012.
- [55] Schmittler J., Woop S., Wagner D., Paul W., Slusallek P., *Realtime ray tracing of dynamic scenes on an fpga chip*, ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2004.
- [56] Woop S., Schmittler J., Slusallek P., *Rpu: a programmable ray processing unit for real-time ray tracing*, ACM Transactions on Graphics (TOG), 24, 434–444, 2005.
- [57] Woop S., Brunvand E., Slusallek P., *Estimating performance of a ray-tracingasic design*, IEEE Symposium on Interactive Ray Tracing, 2006.
- [58] Spjut J., Kensler A., Kopta D., Brunvand E., *Trax: A multicore hardware architecture for real-time ray tracing*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 28, 1802–1815, 2009.

- [59] Suffern K., Ray Tracing from the Ground Up ., A K Peters/CRC Press, Wellesley, Massachusetts, 2007, ISBN 978-1568812724.
- [60] Lotto R., Williams S., Purves D., *An empirical basis for mach bands*, Proc Natl Acad Sci U S A, 1999.
- [61] *Tekstura ziemi*, URL <https://eoimages.gsfc.nasa.gov/images/imagerecords/73000/73580/world.topo.bathy.200401.3x5400x2700.png>.
- [62] *How to find a fast floating-point atan2 approximation*, URL <https://www.dsprelated.com/showarticle/1052.php>.
- [63] *Adv7511 programming guide*, URL <http://www.analog.com/en/products/audio-video/hdmidvi-transmitters/adv7511.html>.
- [64] Valdez J., Becker J., *Understanding the i2c bus*, Rap. tech., Texas Instruments, 2015.
- [65] *Gtc 2018 keynote with nvidia ceo jensen huang*, URL <https://www.youtube.com/watch?v=95nphvtVf34>.
- [66] *Nvidia rtx and frameworks ray tracing technology demonstration*, URL <https://www.youtube.com/watch?v=tjf-1BxpR9c>.

