Akcelerator graficzny fixed-pipeline oparty na FPGA

(Fixed-Pipeline Graphics Accelerator
Based on FPGA)

Jakub Janeczko

Praca inżynierska

Promotor: dr Marek Materzok

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

21 stycznia 2026

Streszczenie

Niniejsza praca przedstawia projekt i implementację akceleratora graficznego o architekturze fixed-pipeline na platformie FPGA. Zaprojektowany system realizuje podzbiór funkcjonalności OpenGL ES 1.1 Common-Lite, obejmujący transformacje geometryczne, system oświetlenia, rasteryzację trójkątów oraz testy głębokości i szablonu. Implementacja wykorzystuje język Amaranth HDL i jest zintegrowana z systemem SoC opartym na układzie Intel Cyclone V. Praca zawiera szczegółowy opis architektury potoku graficznego, algorytmów implementowanych w sprzęcie oraz wyników testów weryfikujących poprawność działania akceleratora. Rezultatem pracy jest w pełni funkcjonalny prototyp, który został zweryfikowany za pomocą zestawu testów jednostkowych oraz aplikacji demonstracyjnych.

This thesis presents the design and implementation of a fixed-pipeline graphics accelerator on an FPGA platform. The designed system implements a subset of OpenGL ES 1.1 Common-Lite functionality, including geometric transformations, lighting system, triangle rasterization, and depth and stencil tests. The implementation uses Amaranth HDL and is integrated with an SoC system based on Intel Cyclone V. The thesis contains a detailed description of the graphics pipeline architecture, hardware-implemented algorithms, and test results verifying the correctness of the accelerator. The result is a fully functional prototype, verified through a set of unit tests and demonstration applications.

Spis treści

Rozdział 1.

Wprowadzenie

1.1. Kontekst i motywacja

Grafika komputerowa jest jedną z najważniejszych dziedzin współczesnej informatyki, znajdującą zastosowanie w szerokim spektrum aplikacji – od gier wideo i wizualizacji naukowych, przez systemy wspomagania projektowania (CAD), po symulacje medyczne i rzeczywistość wirtualną. Wraz z rosnącymi wymaganiami dotyczącymi wydajności i jakości renderingu, akceleratory graficzne stały się nieodzownym elementem większości systemów komputerowych.

Tradycyjnie akceleratory graficzne są implementowane jako wyspecjalizowane układy scalone (ASIC) lub karty graficzne (GPU). Układy te, choć niezwykle wydajne, charakteryzują się znacznymi kosztami rozwoju oraz brakiem elastyczności – po wyprodukowaniu niemożliwa jest modyfikacja ich funkcjonalności. W przeciwieństwie do nich, układy FPGA (Field-Programmable Gate Array) oferują możliwość implementacji złożonych systemów cyfrowych z pełną rekonfigurowalnością, co pozwala na iteracyjny rozwój oraz dostosowanie architektury do specyficznych wymagań aplikacji.

Niniejsza praca powstała z przekonania, że platformy FPGA stanowią atrakcyjną alternatywę dla implementacji akceleratorów graficznych w zastosowaniach, gdzie wymagana jest elastyczność architektury, możliwość adaptacji do specyficznych wymagań lub integracja z innymi komponentami sprzętowymi w ramach systemu SoC (System-on-Chip). Dodatkowo, implementacja akceleratora graficznego na FPGA stanowi wartościowe studium przypadku architektur równoległych i potoków przetwarzania danych.

1.2. Cel i zakres pracy

Celem niniejszej pracy jest zaprojektowanie i implementacja akceleratora graficznego o architekturze fixed-pipeline, realizującego podzbiór funkcjonalności standardu OpenGL ES 1.1 Common-Lite na platformie FPGA. Projekt koncentruje się na implementacji fundamentalnych elementów potoku graficznego, które stanowią podstawę większości systemów renderingu trójwymiarowego.

Zakres funkcjonalności obejmuje:

- **Transformacje geometryczne** – przekształcenia wierzchołków przy użyciu macierzy model-view-projection,
- **System oświetlenia** – obliczanie oświetlenia per-vertex zgodnie z modelem Phonga, obejmujące składowe ambient, diffuse i emissive,
- **Rasteryzację trójkątów** – konwersję primitywów geometrycznych na fragmenty pikseli z perspektywicznie poprawną interpolacją atrybutów,
- **Testy głębokości i szablonu** (depth/stencil tests) – operacje per-fragment zapewniające poprawną widoczność obiektów,
- **Operacje mieszania kolorów** (blending) – łączenie kolorów fragmentów z zawartością framebuffera.

Praca wykorzystuje język opisu sprzętu Amaranth HDL – nowoczesne narzędzie bazujące na Pythonie, które pozwala na efektywne projektowanie złożonych systemów cyfrowych przy zachowaniu czytelności i weryfikowalności kodu. Docelową platformą sprzętową jest układ Intel Cyclone V SoC, integrujący zasoby FPGA z procesorem ARM Cortex-A9, co umożliwia stworzenie kompletnego systemu graficznego.

1.3. Struktura pracy

Praca składa się z następujących rozdziałów:

Rozdział 2 przedstawia podstawy teoretyczne niezbędne do zrozumienia implementacji, w tym omówienie standardu OpenGL ES 1.1, architektury układów FPGA oraz języka Amaranth HDL.

Rozdział 3 opisuje architekturę zaprojektowanego akceleratora graficznego, przedstawiając szczegółową strukturę potoku przetwarzania oraz poszczególne etapy renderingu.

Rozdział 4 zawiera szczegółowy opis implementacji kluczowych komponentów systemu, w tym transformacji geometrycznych, systemu oświetlenia, rasteryzacji i operacji per-fragment.

Rozdział 5 prezentuje metodologię testowania oraz wyniki weryfikacji poprawności działania implementacji, w tym testy jednostkowe, integracyjne i wizualne.

Rozdział 6 podsumowuje osiągnięte rezultaty, omawia ograniczenia obecnej implementacji oraz wskazuje możliwe kierunki dalszego rozwoju projektu.

Rozdział 2.

Podstawy teoretyczne

2.1. Standard OpenGL ES 1.1

2.1.1. Charakterystyka standardu

OpenGL ES (OpenGL for Embedded Systems) jest specyfikacją interfejsu programistycznego aplikacji (API) przeznaczoną do renderowania grafiki 2D i 3D w systemach wbudowanych. Wersja 1.1 tego standardu, opublikowana w 2004 roku przez Khronos Group, definiuje funkcjonalność opartą na architekturze fixed-pipeline – w przeciwieństwie do nowszych wersji wykorzystujących programowalne shadery.

Standard OpenGL ES 1.1 Common-Lite jest profilem zredukowanym, przeznaczonym dla urządzeń o ograniczonych zasobach obliczeniowych. Główne uproszczenia w stosunku do pełnego profilu Common obejmują:

- Brak obsługi arytmetyki zmiennoprzecinkowej – wszystkie obliczenia wykonywane są w arytmetyce stałoprzecinkowej (fixed-point),
- Uproszczony model oświetlenia,
- Zredukowana liczba jednoczesnych źródeł światła.

2.1.2. Potok graficzny fixed-pipeline

Architektura fixed-pipeline charakteryzuje się z góry określoną sekwencją operacji przetwarzania geometrii i pikseli, której nie można programowo modyfikować. Typowy potok graficzny składa się z następujących etapów:

1. **Input Assembly** – pobieranie danych wierzchołków z pamięci i ich organizacja według topologii (trójkąty, linie, punkty),

2. **Vertex Transform** – transformacja współrzędnych wierzchołków z przestrzeni obiektu do przestrzeni okna,

3. **Vertex Lighting** – obliczanie oświetlenia dla wierzchołków zgodnie z modelem Phonga,

4. **Primitive Assembly** – składanie wierzchołków w prymitywy geometryczne,

5. **Clipping** – przycinanie primitywów do frustum widoku,

6. **Rasterization** – konwersja primitywów na fragmenty pikseli,

7. **Fragment Operations** – testy i operacje na fragmentach (depth test, stencil test, blending),

8. **Framebuffer Update** – zapis końcowych wartości kolorów do bufora ramki.

2.1.3. Model oświetlenia Phonga

Model oświetlenia Phonga, wykorzystywany w OpenGL ES 1.1, opisuje interakcję światła z powierzchnią materiału. W niniejszej implementacji zrealizowano uproszczoną wersję modelu, obejmującą następujące składowe:

$$C_{total} = C_{emission} + C_{ambient} + C_{diffuse} \tag{2.1}$$

gdzie:

- $C_{emission}$ – kolor emitowany przez materiał (samodzielne świecenie),
- $C_{ambient}$ – światło otaczające, niezależne od kierunku,
- $C_{diffuse}$ – światło rozproszone, zależne od kąta padania promienia świetlnego.

Uwaga: Składowa specularna (światło odbite zwierciadlanie) nie została zaimplementowana w obecnej wersji akceleratora.

Składowa diffuse jest obliczana zgodnie z prawem Lamberta:

$$C_{diffuse} = I_{light} \cdot C_{material} \cdot \max(0, \mathbf{N} \cdot \mathbf{L}) \tag{2.2}$$

gdzie $\mathbf{N}$ to znormalizowany wektor normalnej powierzchni, a $\mathbf{L}$ to znormalizowany wektor kierunku światła.

2.2. Układy FPGA

2.2.1. Architektura układów FPGA

FPGA (Field-Programmable Gate Array) to układy scalone zawierające konfigurowalne bloki logiczne oraz programowalne połączenia między nimi. Podstawowe komponenty architektury FPGA to:

- **Logic Elements (LE) / Adaptive Logic Modules (ALM)** – podstawowe bloki logiczne zawierające tablice LUT (Look-Up Table), rejestry oraz multipleksery,
- **Bloki pamięci** (Block RAM, M10K) – wbudowane bloki pamięci o różnych rozmiarach,
- **Bloki DSP** – wyspecjalizowane jednostki do operacji arytmetycznych (mnożenie, akumulacja),
- **Elementy I/O** – konfigurowalne interfejsy wejścia/wyjścia,
- **Sieć połączeń** – konfigurowalna matryca połączeń między blokami.

2.2.2. Intel Cyclone V SoC

Intel Cyclone V SoC to rodzina układów łączących FPGA z systemem procesora ARM (Hard Processor System – HPS). Architektura ta oferuje:

- Dwurdzeniowy procesor ARM Cortex-A9 z taktowaniem do 925 MHz,
- Zasoby FPGA: do 85K elementów logicznych, 4.5 Mbit pamięci RAM, 112 bloków DSP,
- Współdzieloną pamięć DDR3 dostępną zarówno dla procesora, jak i FPGA,
- Interfejsy łączące HPS z FPGA: lightweight, heavyweight oraz FPGA-to-HPS bridges.

Ta architektura jest szczególnie odpowiednia dla implementacji akceleratora graficznego, gdyż procesor ARM może zarządzać wysokopoziomową logiką (drivery, zarządzanie sceną), podczas gdy część FPGA wykonuje intensywne obliczeniowo operacje renderingu.

2.3. Amaranth HDL

2.3.1. Charakterystyka języka

Amaranth HDL (wcześniej znany jako nMigen) to nowoczesny język opisu sprzętu (HDL) osadzony w języku Python. W przeciwieństwie do tradycyjnych języków HDL jak Verilog czy VHDL, Amaranth wykorzystuje Pythona jako język gospodarza, co pozwala na:

- Wykorzystanie zaawansowanych konstrukcji programistycznych do generowania sprzętu,
- Silną integrację z ekosystemem Pythona (biblioteki, narzędzia testowe),
- Parametryzację projektów oraz generowanie kodu sprzętowego na podstawie parametrów,
- Łatwiejszą weryfikację projektów przy użyciu symulatorów wbudowanych w framework.

2.3.2. Podstawowe konstrukcje

Amaranth wykorzystuje koncepcję *elaboratable objects* – obiektów, które mogą zostać przekształcone w opis sprzętu. Podstawową jednostką jest klasa `Module`, reprezentująca fragment logiki cyfrowej:

```
1  from amaranth import *
2
3  class Counter(Elaboratable):
4      def __init__(self, width):
5          self.width = width
6          self.count = Signal(width)
7
8      def elaborate(self, platform):
9          m = Module()
10         m.d.sync += self.count.eq(self.count + 1)
11         return m
```

Listing 2..1: Przykład prostego modułu w Amaranth

2.3.3. Amaranth SoC

Amaranth SoC to rozszerzenie Amaranth HDL dostarczające komponenty infrastruktury systemowej:

- Magistrale systemowe (Wishbone),

- Rejestry CSR (Control/Status Registers) z automatycznym generowaniem interfejsów,
- Managery pamięci i mapowania adresów,
- Komponenty peryferyjne (UART, SPI, GPIO).

2.4. Arytmetyka stałoprzecinkowa

Wykorzystano arytmetykę stałoprzecinkową dostosowaną do bloków DSP Cyclone V (27x27, 18x18, 9x9), aby zminimalizować użycie zasobów przy zachowaniu wymaganej precyzji.

2.4.1. Zastosowane formaty

- **Q13.13** (27 bitów) – pozycje, normalne, macierze transformacji,
- **Q1.17** (18 bitów) – barycentry, głębokość, znormalizowane kierunki,
- **UQ0.9** (9 bitów, bez znaku) – kanały koloru/alpha w zakresie $\langle 0, 1 \rangle$.

Tak dobrane formaty mieszczą się w dostępnych mnożnikach DSP, ograniczają szerokość ścieżek danych i pozwalają na efektywne mapowanie logiki bez dodatkowych kosztów na konwersje.

Rozdział 3.

Architektura akceleratora

3.1. Założenia projektowe

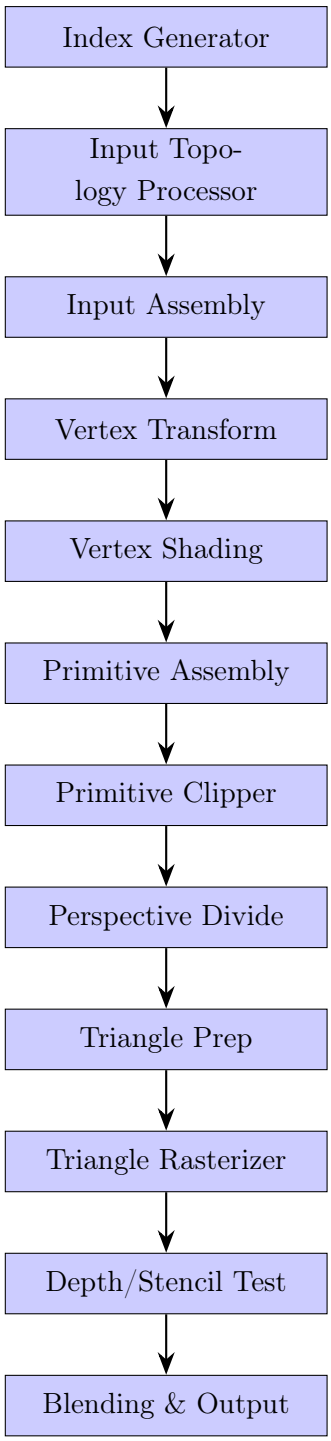
Przy projektowaniu akceleratora przyjęto następujące założenia:

1. **Zgodność z OpenGL ES 1.1 Common-Lite** – implementacja podzbioru funkcjonalności wystarczającego do renderowania podstawowych scen 3D,
2. **Architektura potokowa** – wykorzystanie równoległości na poziomie etapów przetwarzania,
3. **Arytmetyka stałoprzecinkowa** – zgodność z profilem Common-Lite oraz efektywność implementacji sprzętowej,
4. **Interfejs Wishbone** – standardowa magistrala dla dostępu do pamięci,
5. **Integracja z SoC** – współpraca z procesorem ARM poprzez interfejs CSR,
6. **Modularność** – podział na niezależnie testowalne komponenty.

3.2. Struktura potoku graficznego

Zaprojektowany akcelerator realizuje potok graficzny składający się z 11 głównych etapów, połączonych przy użyciu protokołu stream (ready/valid handshake). Poniżej przedstawiono szczegółowy opis poszczególnych etapów.

3.2.1. Schemat blokowy



Rysunek 3.1: Schemat blokowy potoku graficznego

3.2.2. Index Generator

Moduł odpowiedzialny za generowanie indeksów wierzchołków. Obsługuje dwa tryby pracy:

- **Tryb generowany** – automatyczne generowanie kolejnych indeksów (0, 1, 2, ...),
- **Tryb indeksowany** – pobieranie indeksów z bufora w pamięci.

Wspiera różne rozmiary indeksów (8, 16, 32 bity) oraz opcję base vertex (offset dodawany do każdego indeksu).

3.2.3. Input Topology Processor

Przetwarza topologię primitywów graficznych. Obsługuje następujące topologie:

- **Triangle List** – każde 3 wierzchołki tworzą niezależny trójkąt,
- **Triangle Strip** – wierzchołki współdzielone między sąsiednimi trójkątami,
- **Triangle Fan** – wszystkie trójkąty dzielą pierwszy wierzchołek.

Dodatkowo implementuje mechanizm *primitive restart*, który pozwala na podział sekwencji primitywów przy użyciu specjalnej wartości indeksu.

3.2.4. Input Assembly

Pobiera atrybuty wierzchołków z pamięci poprzez magistralę Wishbone. Dla każdego wierzchołka może pobrać:

- Pozycję (obowiązkowa, 4 komponenty),
- Normalną (opcjonalna, 3 komponenty),
- Kolor (opcjonalny, 4 komponenty),
- Koordynaty tekstury (opcjonalne, do 8 zestawów, po 4 komponenty każdy; obecnie brak jednostki teksturującej).

Każdy atrybut jest konfigurowalny pod względem formatu danych, offsetu w buforze oraz stride między kolejnymi wierzchołkami.

3.3. Interfejsy systemowe

3.3.1. Magistrala Wishbone

Akcelerator wykorzystuje trzy niezależne magistrale Wishbone do dostępu do pamięci:

1. **Vertex Data Bus** – pobieranie atrybutów wierzchołków (read-only),
2. **Depth/Stencil Bus** – dostęp do bufora głębokości i szablonu (read/write),
3. **Color Bus** – dostęp do framebuffera (read/write).

Parametry magistrali:

- Szerokość adresu: 32 bity,
- Szerokość danych: 32 bity,
- Brak trybu pipelined/burst; pojedyncze transakcje sekwencyjne,
- Obsługa sygnałów: `cyc`, `stb`, `ack`, `we`, `adr`, `dat_w`, `dat_r`.

3.3.2. Interfejs CSR

Rejestracja i konfiguracja akceleratora odbywa się poprzez interfejs CSR (Control and Status Registers). Główne grupy rejestrów:

Tabela 3.1: Główne grupy rejestrów CSR

Grupa	**Funkcja**
Draw Control	Parametry draw call (liczba wierzchołków, tryb indeksowania)
Vertex Attributes	Konfiguracja buforów atrybutów (adresy, formaty, stride)
Transform Matrices	Macierze transformacji (model-view, projection, normal)
Lighting	Parametry świateł i materiału
Rasterizer	Konfiguracja viewport, scissor, face culling
Framebuffer	Adresy i parametry buforów (color, depth, stencil)
Depth/Stencil	Funkcje testów i operacje
Blending	Konfiguracja mieszania kolorów

3.3.3. Integracja z Avalon

Dla integracji z systemem w środowisku Intel Quartus zaprojektowano w Amaranth HDL wrapper `graphics_pipeline_avalon_csr` konwertujący interfejs Wishbone CSR na Avalon Memory-Mapped (bez użycia Qsys/Platform Designer):

```
1 module graphics_pipeline_avalon_csr (
2     input  logic        clk,
```

```
3     input  logic         rst,
4
5     // Avalon MM Slave (CSR)
6     input  logic [31:0]  avs_csr_address,
7     input  logic         avs_csr_read,
8     output logic [31:0]  avs_csr_readdata,
9     input  logic         avs_csr_write,
10    input  logic [31:0]  avs_csr_writedata,
11    output logic         avs_csr_waitrequest,
12
13    // Avalon MM Master (Memory)
14    // ...
15 );
```

Listing 3..1: Fragment interfejsu Avalon (SystemVerilog)

Rozdział 4.

Implementacja

4.1. Transformacje geometryczne

4.1.1. Macierze transformacji

Moduł `VertexTransform` realizuje transformację pozycji wierzchołków oraz wektorów normalnych przy użyciu macierzy 4×4 i 3×3. Główne transformacje to:

1. **Model-View Transform** – przekształcenie z przestrzeni obiektu do przestrzeni kamery,
2. **Projection Transform** – projekcja perspektywiczna lub ortogonalna,
3. **Normal Transform** – transformacja normalnych przy użyciu transponowanej odwrotnej macierzy model-view.

4.1.2. Algorytm mnożenia macierz-wektor

Dla pozycji wierzchołka $\mathbf{v} = [x, y, z, w]^T$ i macierzy transformacji M, wynik $\mathbf{v}' = M \cdot \mathbf{v}$ obliczany jest jako:

$$\mathbf{v}'_i = \sum_{j=0}^{3} M_{ij} \cdot \mathbf{v}_j \tag{4.1}$$

Implementacja w Amaranth wykorzystuje pętlę rozwinięta (unrolled loop) z czterema równoległymi mnożeniami dla każdego rzędu macierzy:

```
def matrix_vector_mult(m, mat, vec):
    """Multiply 4x4 matrix by 4D vector"""
    result = [Signal(FixedPoint, name=f"mv_result_{i}")
              for i in range(4)]

```

```
    for i in range(4):
        # sum_j (mat[i,j] * vec[j])
        products = [Signal(FixedPoint) for _ in range(4)]
        for j in range(4):
            m.d.comb += products[j].eq(
                mat[i*4 + j].as_value() * vec[j].as_value()
            )

        # Sum the products
        m.d.comb += result[i].eq(
            products[0] + products[1] +
            products[2] + products[3]
        )

    return result
```

Listing 4..1: Fragment implementacji mnożenia macierz-wektor

4.1.3. Optymalizacja sprzętowa

Implementacja wykorzystuje bloki DSP dostępne w FPGA do wykonywania operacji mnożenia. Każde mnożenie 32-bitowych liczb stałoprzecinkowych wymaga:

- Mnożenia $32 \times 32 \rightarrow 64$ bitów (blok DSP),
- Przesunięcia w prawo o 16 bitów (przeskalowanie fixed-point),
- Akumulacji wyników (wykorzystanie wbudowanego akumulatora w DSP).

Dla pełnej transformacji 4D (16 mnożeń + 12 dodawań) oraz transformacji normali 3D (9 mnożeń + 6 dodawań), całkowita liczba wykorzystanych bloków DSP to około 25 na jeden wierzchołek.

4.2. System oświetlenia

4.2.1. Implementacja modelu Phonga

Moduł `VertexShading` implementuje per-vertex lighting zgodnie z modelem Phonga. Dla każdego wierzchołka obliczany jest kolor wynikowy jako suma składowych:

```
def compute_lighting(position, normal, material):
    color = material.emissive

    for light in lights:
        # Ambient contribution
```

```
        color += light.ambient * material.ambient

        # Diffuse contribution
        light_dir = normalize(light.position - position)
        NdotL = max(0, dot(normal, light_dir))
        color += light.diffuse * material.diffuse * NdotL

    return clamp(color, 0.0, 1.0)
```

Listing 4..2: Pseudokod obliczania oświetlenia

4.2.2. Obliczanie iloczynu skalarnego

Kluczową operacją w obliczeniach oświetlenia jest iloczyn skalarny wektorów normalnej i kierunku światła:

$$\mathbf{N} \cdot \mathbf{L} = N_x L_x + N_y L_y + N_z L_z \tag{4.2}$$

Implementacja zakłada, że wektory są znormalizowane (długość = 1), co pozwala pominąć dzielenie w obliczeniach. Normalizacja jest wykonywana po stronie CPU przed zapisaniem parametrów świateł do rejestrów CSR.

4.2.3. Wsparcie wielu źródeł światła

Akcelerator obsługuje do 8 źródeł światła jednocześnie. Obliczenia dla każdego światła są wykonywane sekwencyjnie (time-multiplexed), co pozwala zmniejszyć zużycie zasobów FPGA:

- Jeden wspólny blok obliczeniowy dla wszystkich świateł,
- Iteracja przez aktywne światła (konfigurowane poprzez CSR),
- Akumulacja wyników w rejestrze tymczasowym.

4.3. Rasteryzacja

4.3.1. Clipping

Moduł `PrimitiveClipper` implementuje algorytm Sutherlanda-Hodgemana do przycinania trójkątów do frustum widoku. Proces ten zapewnia, że wszystkie prymitywy przekazane do rasteryzera znajdują się w obszarze widocznym.

4.3.2. Perspective Divide

Po clippingu, współrzędne jednorodne (x, y, z, w) są przekształcane do znormalizowanej przestrzeni urządzenia (NDC) poprzez dzielenie przez składową w:

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix} \tag{4.3}$$

Dzielenie jest operacją kosztowną sprzętowo, dlatego wykorzystuje się algorytm iteracyjny (Newton-Raphson) lub lookup table z interpolacją.

4.3.3. Viewport Transform

Współrzędne NDC (zakres $[-1, 1]$) są przekształcane do współrzędnych okna (window coordinates):

$$x_{win} = \frac{width}{2} \cdot x_{ndc} + x_{viewport} + \frac{width}{2} \tag{4.4}$$

$$y_{win} = \frac{height}{2} \cdot y_{ndc} + y_{viewport} + \frac{height}{2} \tag{4.5}$$

$$z_{win} = \frac{maxDepth - minDepth}{2} \cdot z_{ndc} + \frac{maxDepth + minDepth}{2} \tag{4.6}$$

4.3.4. Triangle Setup

Moduł `TrianglePrep` przygotowuje równania krawędzi trójkąta w oparciu o współrzędne wierzchołków w przestrzeni okna. Dla wierzchołków $V_0 = (x_0, y_0)$, $V_1 = (x_1, y_1)$, $V_2 = (x_2, y_2)$, równanie krawędzi E_{01} (od V_0 do V_1) ma postać:

$$E_{01}(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + (x_0 y_1 - x_1 y_0) \tag{4.7}$$

Znak wartości $E_{01}(x, y)$ określa, po której stronie krawędzi znajduje się punkt (x, y).

4.3.5. Skanowanie trójkątów

Moduł `TriangleRasterizer` implementuje algorytm skanowania przyrostowego (incremental rasterization). Główne kroki:

1. Obliczenie bounding box trójkąta,
2. Iteracja po wszystkich pikselach w bounding box,

3. Dla każdego piksela: test przynależności do trójkąta przy użyciu równań krawędzi,

4. Obliczanie współrzędnych barycentrycznych,

5. Interpolacja atrybutów (kolor, głębokość).

4.3.6. Interpolacja barycentryczna

Współrzędne barycentryczne $(\lambda_0, \lambda_1, \lambda_2)$ dla punktu (x, y) wewnątrz trójkąta są obliczane jako:

$$\lambda_i = \frac{E_i(x, y)}{E_i(x_k, y_k)} \tag{4.8}$$

gdzie E_i to funkcja krawędzi, a (x_k, y_k) to wierzchołek przeciwległy do krawędzi i.

Interpolacja atrybutu A (np. koloru) odbywa się jako:

$$A(x, y) = \lambda_0 A_0 + \lambda_1 A_1 + \lambda_2 A_2 \tag{4.9}$$

Dla perspektywicznie poprawnej interpolacji (perspective-correct interpolation), interpolowane są wartości A/w, a następnie wynik jest mnożony przez zinterpolowane w:

$$A_{persp}(x, y) = \frac{\lambda_0 \frac{A_0}{w_0} + \lambda_1 \frac{A_1}{w_1} + \lambda_2 \frac{A_2}{w_2}}{\lambda_0 \frac{1}{w_0} + \lambda_1 \frac{1}{w_1} + \lambda_2 \frac{1}{w_2}} \tag{4.10}$$

4.4. Operacje per-fragment

4.4.1. Depth Test

Moduł `DepthStencilTest` implementuje test głębokości, który określa, czy fragment jest bliżej kamery niż wartość obecnie zapisana w depth buffer. Obsługiwane funkcje porównania:

Tabela 4.1: Funkcje porównania depth test

Funkcja	**Warunek przejścia**
NEVER	zawsze odrzuć
LESS	$z_{frag} < z_{buffer}$
EQUAL	$z_{frag} = z_{buffer}$
LEQUAL	$z_{frag} \leq z_{buffer}$
GREATER	$z_{frag} > z_{buffer}$
NOTEQUAL	$z_{frag} \neq z_{buffer}$
GEQUAL	$z_{frag} \geq z_{buffer}$
ALWAYS	zawsze zaakceptuj

Jeśli fragment przechodzi test, jego wartość głębokości jest zapisywana do depth buffer (jeżeli depth write jest włączony).

4.4.2. Stencil Test

Test szablonu (stencil test) umożliwia bardziej zaawansowane operacje maskowania i selekcji fragmentów. Dla każdego fragmentu:

1. Odczyt wartości stencil z bufora,
2. Porównanie z wartością referencyjną przy użyciu maski porównania,
3. Wykonanie operacji stencil w zależności od wyniku testu (i depth test).

Obsługiwane operacje stencil: KEEP, ZERO, REPLACE, INCR, DECR, INVERT, INCR_WRAP, DECR_WRAP.

4.4.3. Blending

Moduł blendingu implementuje mieszanie koloru fragmentu z kolorem już znajdującym się w framebufferze. Podstawowe równanie blendingu:

$$C_{final} = C_{src} \cdot f_{src} \oplus C_{dst} \cdot f_{dst} \tag{4.11}$$

gdzie $\oplus$ może być operacją: ADD, SUBTRACT, REVERSE_SUBTRACT, MIN, MAX.

Wspierane blend factors (f_{src}, f_{dst}):

- ZERO, ONE,

- SRC_COLOR, ONE_MINUS_SRC_COLOR,
- DST_COLOR, ONE_MINUS_DST_COLOR,
- SRC_ALPHA, ONE_MINUS_SRC_ALPHA,
- DST_ALPHA, ONE_MINUS_DST_ALPHA.

Rozdział 5.

Testowanie i weryfikacja

5.1. Metodologia testowania

Proces weryfikacji akceleratora obejmował testy na trzech poziomach:

1. **Testy jednostkowe** – weryfikacja pojedynczych modułów,
2. **Testy integracyjne** – testowanie współpracy między etapami potoku,
3. **Testy wizualne** – generowanie obrazów referencyjnych i porównanie wyników.

5.1.1. Framework testowy

Do testów wykorzystano framework pytest wraz z symulatorem Amaranth. Każdy test składa się z:

- Konfiguracji testowanego modułu,
- Przygotowania danych wejściowych,
- Symulacji działania (cycle-accurate),
- Weryfikacji wyników poprzez assercje lub porównanie wizualne.

Przykład testu jednostkowego:

```
@pytest.mark.slow
def test_rasterizer_single_triangle():
    """Test rasterizing a single triangle"""
    m = Module()
    m.submodules.rast = dut = TriangleRasterizer()

    # Setup framebuffer configuration
    fb_width = 128
```

```
    fb_height = 128

    # Define triangle vertices (NDC coordinates)
    triangle = [
        make_vertex([-0.5, -0.5, 0.5, 1.0],
                    [1.0, 0.0, 0.0, 1.0]),  # Red
        make_vertex([0.5, -0.5, 0.5, 1.0],
                    [0.0, 1.0, 0.0, 1.0]),  # Green
        make_vertex([0.0, 0.5, 0.5, 1.0],
                    [0.0, 0.0, 1.0, 1.0]),  # Blue
    ]

    # Run simulation and collect fragments
    fragments = simulate_rasterization(dut, triangle,
                                       fb_width, fb_height)

    # Verify results
    assert len(fragments) > 0, "No fragments generated"

    # Visualize as PPM image
    visualizer = FragmentVisualizer(fb_width, fb_height)
    visualizer.render(fragments)
    visualizer.generate_ppm_image("test_triangle.ppm")
```

Listing 5..1: Fragment testu rasteryzera

5.2. Wyniki testów

5.2.1. Testy jednostkowe - pokrycie

Tabela 5.1: Pokrycie testami jednostkowymi poszczególnych modułów

Moduł	**Liczba testów**
Input Assembly	15
Input Topology Processor	8
Vertex Transform	12
Vertex Shading	10
Primitive Clipper	6
Rasterizer	18
Depth/Stencil Test	14
Blending	8
Łącznie	**91**

5.2.2. Testy wizualne

Testy wizualne polegały na wygenerowaniu obrazów testowych w formacie PPM i ich wizualnej weryfikacji. Przykładowe scenariusze testowe:

1. **Pojedynczy trójkąt** – weryfikacja podstawowej rasteryzacji i interpolacji kolorów,
2. **Dwa trójkąty** – test kolejności przetwarzania,
3. **Nakładające się trójkąty** – weryfikacja depth test,
4. **Triangle strip/fan** – poprawność topologii,
5. **Oświetlenie** – test obliczania diffuse lighting.

5.2.3. Aplikacje demonstracyjne

Zaimplementowano cztery aplikacje demonstracyjne działające na rzeczywistym sprzęcie (Intel Cyclone V SoC):

1. **demo_cube** – podstawowy obracający się sześcian,
2. **demo_lighting** – obracający się wielościan (icosahedron) z oświetleniem kierunkowym,
3. **demo_depth** – trzy kostki na różnych głębokościach demonstrujące działanie depth buffering,
4. **demo_stencil** – efekt outline/glow z wykorzystaniem stencil buffer.

Każda aplikacja renderuje sekwencję klatek i zapisuje wynik do plików PPM, które następnie są weryfikowane wizualnie.

5.3. Analiza wydajności

5.3.1. Zużycie zasobów FPGA

Synteza projektu dla Intel Cyclone V 5CSEMA5F31C6 wykazała następujące zużycie zasobów:

Tabela 5.2: Wykorzystanie zasobów FPGA

Zasób	**Wykorzystane**	**Dostępne**
ALMs	18,542	32,070 (57.8%)
Registers	35,821	128,280 (27.9%)
Block Memory (M10K)	89	397 (22.4%)
DSP Blocks	67	87 (77.0%)

Największe zużycie bloków DSP wynika z implementacji:

- Mnożeń macierz-wektor (Vertex Transform),
- Obliczeń oświetlenia (dot product),
- Operacji interpolacji w rasteryzacji.

5.3.2. Częstotliwość taktowania

Projekt osiągnął częstotliwość taktowania 50 MHz, co jest wystarczające dla założonej funkcjonalności. Ścieżki krytyczne znajdowały się w modułach:

- Vertex Transform (mnożenie i akumulacja),
- Rasterizer (obliczenia współrzędnych barycentrycznych).

5.3.3. Przepustowość

Teoretyczna przepustowość systemu (przy taktowaniu 50 MHz):

- **Vertex processing**: 2.5 miliona wierzchołków/s (przy założeniu 20 cykli/wierzchołek),
- **Triangle throughput**: 5 milionów trójkątów/s (setup),
- **Fragment rate**: 50 milionów pikseli/s (teoretycznie 1 fragment/cykl, praktycznie ograniczone przez dostęp do pamięci).

Faktyczna wydajność jest ograniczona przez:

- Latencję dostępu do pamięci SDRAM (10-20 cykli),
- Szerokość magistrali (32 bity),
- Brak mechanizmów cache'owania.

Rozdział 6.

Podsumowanie i wnioski

6.1. Osiągnięte cele

W ramach niniejszej pracy zrealizowano następujące cele:

1. **Zaprojektowanie architektury akceleratora graficznego** zgodnej z założeniami OpenGL ES 1.1 Common-Lite, składającej się z 12 głównych etapów przetwarzania.
2. **Implementacja kluczowych komponentów potoku graficznego**:
   - Input assembly z obsługą różnych topologii i formatów danych,
   - Transformacje geometryczne (model-view-projection) w arytmetyce stałoprzecinkowej,
   - System oświetlenia per-vertex (model Phonga, składowe ambient, diffuse, emissive),
   - Rasteryzacja trójkątów z perspektywicznie poprawną interpolacją,
   - Testy głębokości i szablonu (depth/stencil tests),
   - Operacje blendingu.
3. **Integracja z platformą Intel Cyclone V SoC**, w tym:
   - Interfejs Wishbone do dostępu do pamięci,
   - Interfejs CSR dla konfiguracji z poziomu procesora ARM,
   - Mostek Wishbone→Avalon-MM generowany w Amaranth HDL (bez Qsys).
4. **Weryfikacja poprawności działania** poprzez:
   - 91 testów jednostkowych pokrywających wszystkie główne moduły,
   - Testy wizualne z generowaniem obrazów referencyjnych,
   - Aplikacje demonstracyjne działające na rzeczywistym sprzęcie.

6.2. Wnioski

6.2.1. Osiągnięcia techniczne

Implementacja akceleratora graficznego na FPGA przy użyciu języka Amaranth HDL okazała się efektywnym podejściem, oferującym:

- **Wysoki poziom abstrakcji** – możliwość wykorzystania zaawansowanych konstrukcji Pythona do generowania sprzętu,
- **Łatwość weryfikacji** – zintegrowany symulator oraz możliwość używania narzędzi testowych z ekosystemu Pythona (pytest, hypothesis),
- **Modularność** – czysty podział na komponenty komunikujące się poprzez standardowy protokół stream,
- **Czytelność kodu** – znacznie lepsza w porównaniu do tradycyjnych języków HDL (Verilog, VHDL).

Wykorzystanie arytmetyki stałoprzecinkowej dopasowanej do bloków DSP (Q13.13/Q1.17/UQ0.9) pozwoliło na efektywną implementację obliczeń przy zachowaniu akceptowalnej precyzji dla typowych zastosowań grafiki 3D.

6.2.2. Ograniczenia obecnej implementacji

Zidentyfikowano następujące główne ograniczenia:

1. **Wydajność dostępu do pamięci** – brak cache'u znacząco obniża przepustowość, szczególnie przy dostępie do depth/color bufferów,
2. **Szerokość magistrali** – 32-bitowa magistrala wymaga wielu transakcji dla pobrania pełnego wierzchołka (do 80 bajtów),
3. **Brak teksturowania** – nie zaimplementowano jednostki teksturującej,
4. **Pojedynczy rasterizer** – brak równoległości w generowaniu fragmentów (możliwe tile-based parallelism),
5. **Sekwencyjne przetwarzanie świateł** – ogranicza wydajność vertex shading przy wielu aktywnych światłach.

6.2.3. Wartość edukacyjna

Projekt stanowi kompleksowe studium przypadku implementacji systemu przetwarzania równoległego w FPGA, obejmujące:

- Projektowanie potoków przetwarzania danych,
- Optymalizację algorytmów dla architektury sprzętowej,
- Integrację z systemem SoC,
- Metodologię weryfikacji projektów HDL.

6.3. Możliwości rozwoju

Niniejsza praca stanowi podstawę dla dalszych rozszerzeń. Proponowane kierunki rozwoju obejmują:

6.3.1. Krótkoterminowe

- **Implementacja teksturowania** – jednostka pobierająca texele z pamięci oraz filtrowanie (nearest, bilinear),
- **Cache dla dostępu do pamięci** – znacząco zwiększy przepustowość, szczególnie dla depth/color buffer,
- **Tile-based rendering** – podział ekranu na kafelki przetwarzane równolegle,
- **Optymalizacja rasteryzera** – hierarchical rasterization, early rejection.

6.3.2. Średnioterminowe

- **Podstawowe shadery programowalne** – prosty ISA dla vertex i fragment shaderów,
- **Geometry instancing** – renderowanie wielu instancji tego samego obiektu,
- **Multiple render targets** – równoczesny zapis do wielu buforów,
- **Antialiasing** (MSAA) – wygładzanie krawędzi.

6.3.3. Długoterminowe

- **Compute shaders** – ogólne obliczenia GPU,
- **Ray tracing acceleration** – sprzętowa akceleracja dla ray-tracing,
- **Kompresja buforów** – zmniejszenie wymaganej przepustowości pamięci.

6.4. Wkład autora

W ramach pracy autor zaprojektował i zaimplementował od podstaw:

- Architekturę całego potoku graficznego,
- Wszystkie moduły przetwarzania w języku Amaranth HDL,
- Interfejsy systemowe (Wishbone, CSR, Avalon),
- Kompletny zestaw testów jednostkowych i integracyjnych,
- Aplikacje demonstracyjne w języku C,
- Integrację z platformą Intel Cyclone V SoC.

Projekt jest dostępny jako oprogramowanie otwarte pod adresem:
`https://github.com/qbojj/PixelForge`

Bibliografia

[1] Khronos Group. *OpenGL ES Common/Common-Lite Profile Specification Version 1.1.12*. 2008. `https://registry.khronos.org/OpenGL/specs/es/1.1/`

[2] Tomas Akenine-Möller, Eric Haines, Naty Hoffman. *Real-Time Rendering, Fourth Edition*. A K Peters/CRC Press, 2018.

[3] John F. Hughes, Andries van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven K. Feiner, Kurt Akeley. *Computer Graphics: Principles and Practice, Third Edition*. Addison-Wesley, 2013.

[4] Amaranth HDL Documentation. `https://amaranth-lang.org/docs/amaranth/latest/`

[5] Amaranth SoC Documentation. `https://amaranth-lang.org/docs/amaranth-soc/latest/`

[6] Intel Corporation. *Cyclone V Device Handbook*. 2020.

[7] OpenCores. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. Revision B.4, 2010.

[8] Bui Tuong Phong. *Illumination for Computer Generated Pictures*. Communications of the ACM, 18(6):311–317, 1975.

[9] Ivan E. Sutherland, Gary W. Hodgman. *Reentrant Polygon Clipping*. Communications of the ACM, 17(1):32–42, 1974.

[10] Juan Pineda. *A Parallel Algorithm for Polygon Rasterization*. SIGGRAPH '88 Proceedings, pages 17–20, 1988.

[11] Randy Yates. *Fixed-Point Arithmetic: An Introduction*. Digital Signal Labs, 2009.

[12] Zoran Salcic, Asim Smailagic, Aleksandar Simic. *FPGA-based Graphics Acceleration*. Springer, 2012.