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

30 stycznia 2026

Streszczenie

Celem pracy było zaprojektowanie i zaimplementowanie akceleratora graficznego o architekturze fixed-pipeline, realizującego wybrany podzbiór funkcjonalności OpenGL ES 1.1 Common-Lite. Projekt wykorzystuje układ FPGA Intel Cyclone V oraz język opisu sprzętu Amaranth HDL. Zaimplementowany potok graficzny obejmuje: transformacje geometryczne, rasteryzację trójkątów z perspektywiczną interpolacją, system oświetlenia (ambient, diffuse), testy głębokości i szablonu oraz operacje mieszania kolorów. Akcelerator zintegrowano z systemem SoC poprzez magistralę Wishbone i interfejs CSR. Projekt zawiera również kompletne środowisko testowe oraz aplikacje demonstracyjne, które weryfikują poprawność działania zaimplementowanych funkcjonalności.

The aim of this work was to design and implement a fixed-pipeline graphics accelerator that realizes a selected subset of OpenGL ES 1.1 Common-Lite functionality. The project uses an Intel Cyclone V FPGA and the Amaranth HDL hardware description language. The implemented graphics pipeline includes: geometric transformations, triangle rasterization with perspective-correct interpolation, lighting system (ambient, diffuse), depth and stencil tests, and color blending operations. The accelerator was integrated with the SoC system through the Wishbone bus and CSR interface. The project also includes a complete testing environment and demonstration applications that verify the correctness of the implemented functionalities.

Spis treści

Rozdział 1.

Wprowadzenie

1.1. Motywacja

Renderowanie grafiki trójwymiarowej w czasie rzeczywistym stanowi kluczową funkcjonalność współczesnych systemów komputerowych — od urządzeń mobilnych, przez komputery osobiste, po zaawansowane stacje robocze. Tradycyjnie zadanie to realizowane jest przez wyspecjalizowane procesory graficzne (GPU) wykonane w technologii ASIC (Application-Specific Integrated Circuit). Implementacja akceleratora graficznego w układzie FPGA (Field-Programmable Gate Array) stanowi alternatywne podejście, które łączy elastyczność konfigurowalnego sprzętu z możliwością pełnej kontroli nad architekturą i zużyciem zasobów.

Projekty oparte na FPGA znajdują szczególne zastosowanie w systemach wbudowanych, gdzie wymagania dotyczące zużycia energii, kosztu i możliwości dostosowania są często równie istotne jak wydajność obliczeniowa. Ponadto, implementacja GPU w FPGA oferuje doskonałe możliwości edukacyjne, pozwalając na głębokie zrozumienie działania potoku graficznego na poziomie sprzętowym.

1.2. Cel pracy

Głównym celem niniejszej pracy było zaprojektowanie i zaimplementowanie akceleratora graficznego o architekturze fixed-pipeline, realizującego wybrany podzbiór funkcjonalności specyfikacji OpenGL ES 1.1 Common-Lite. Fixed-pipeline oznacza architekturę, w której kolejność i rodzaj operacji graficznych są ustalone i nie mogą być modyfikowane przez programy użytkownika (shadery), w przeciwieństwie do współczesnych GPU o programowalnym potoku graficznym.

W ramach pracy zaimplementowano następujące elementy potoku graficznego:

- transformacje geometryczne wierzchołków,

- rasteryzację trójkątów z perspektywiczną interpolacją,
- wybrane elementy systemu oświetlenia (ambient, diffuse),
- testy głębokości i szablonu (depth/stencil tests),
- operacje mieszania kolorów (blending).

Projekt obejmuje również zbudowanie kompletnego środowiska testowego oraz aplikacji demonstracyjnych weryfikujących poprawność działania akceleratora.

1.3. Struktura pracy

Praca składa się z następujących rozdziałów:

Rozdział 2 przedstawia podstawy teoretyczne grafiki komputerowej oraz potoku renderowania 3D, w tym transformacje geometryczne, rasteryzację oraz techniki cieniowania.

Rozdział 3 opisuje zastosowane technologie i narzędzia, ze szczególnym uwzględnieniem języka opisu sprzętu Amaranth HDL oraz platformy sprzętowej DE1-SoC.

Rozdział 4 zawiera szczegółowy opis architektury zaprojektowanego akceleratora graficznego, wraz z opisem poszczególnych modułów potoku.

Rozdział 5 prezentuje szczegóły implementacji kluczowych komponentów systemu, z naciskiem na rozwiązania specyficzne dla platformy FPGA.

Rozdział 6 opisuje metodologię testowania oraz środowisko uruchomieniowe, w tym aplikacje demonstracyjne.

Rozdział 7 podsumowuje osiągnięte wyniki oraz przedstawia możliwe kierunki rozwoju projektu.

Rozdział 2.

Podstawy teoretyczne

2.1. Potok renderowania grafiki 3D

Rendering grafiki trójwymiarowej w czasie rzeczywistym opiera się na złożonym procesie przekształcania reprezentacji trójwymiarowej sceny w obraz dwuwymiarowy wyświetlany na ekranie. Proces ten realizowany jest przez *potok graficzny* (graphics pipeline) — sekwencję operacji przekształcających dane wierzchołków i prymitywów geometrycznych w piksele obrazu.

Klasyczny potok graficzny fixed-pipeline, na którym opiera się specyfikacja OpenGL ES 1.1 [1], składa się z następujących głównych etapów:

1. **Przetwarzanie wierzchołków** (Vertex Processing) — transformacja współrzędnych wierzchołków z przestrzeni modelu przez przestrzenie: świata, widoku, obcinania (clip space) do znormalizowanych współrzędnych urządzenia (Normalized Device Coordinates, NDC).

2. **Cieniowanie wierzchołków** (Vertex Shading) — obliczanie koloru wierzchołków na podstawie modelu oświetlenia, właściwości materiału i parametrów źródeł światła.

3. **Przycinanie** (Clipping) — odrzucanie lub obcinanie fragmentów prymitywów znajdujących się poza obszarem widoczności (frustum).

4. **Rasteryzacja** (Rasterization) — przekształcanie prymitywów geometrycznych (np. trójkątów) w zbiór fragmentów odpowiadających poszczególnym pikselom obrazu.

5. **Przetwarzanie fragmentów** (Fragment Processing) — wykonywanie testów głębokości, szablonu, mieszania kolorów i zapisywanie finalnych wartości pikseli do bufora ramki.

2.2. Transformacje geometryczne

2.2.1. Przestrzenie współrzędnych

W grafice komputerowej wierzchołki przechodzą przez szereg transformacji między różnymi przestrzeniami współrzędnych:

- **Przestrzeń modelu** (Model Space) — lokalna przestrzeń współrzędnych obiektu 3D.
- **Przestrzeń świata** (World Space) — wspólna przestrzeń dla wszystkich obiektów w scenie.
- **Przestrzeń widoku** (View/Eye Space) — przestrzeń względem kamery obserwatora.
- **Przestrzeń obcinania** (Clip Space) — współrzędne jednorodne po zastosowaniu macierzy projekcji.
- **NDC** (Normalized Device Coordinates) — znormalizowane współrzędne w zakresie $[-1, 1]$.
- **Przestrzeń ekranu** (Screen Space) — współrzędne w pikselach na ekranie.

2.2.2. Macierze transformacji

Transformacje między przestrzeniami realizowane są przy użyciu mnożenia przez macierze 4×4. Kluczowe macierze to:

- **Macierz model-widok** (M_{mv}) — przekształca wierzchołki z przestrzeni modelu do przestrzeni widoku.
- **Macierz projekcji** (M_p) — przekształca współrzędne z przestrzeni widoku do przestrzeni obcinania, realizując projekcję perspektywiczną lub ortogonalną.
- **Macierz model-widok-projekcja** ($M_{mvp} = M_p \cdot M_{mv}$) — pełna transformacja wierzchołka.

Dla wektora jednorodnego wierzchołka $\mathbf{v} = [x, y, z, 1]^T$ transformacja ma postać:

$$\mathbf{v}_{clip} = M_{mvp} \cdot \mathbf{v}$$

Transformacje wektorów normalnych wymagają użycia odwrotności transpozycji macierzy model-widok:

$$\mathbf{n}_{view} = (M_{mv}^{-1})^T \cdot \mathbf{n}$$

2.3. Model oświetlenia Phonga

Model oświetlenia Phonga [7] to empiryczny model oświetlenia lokalnego, który aproksymuje oddziaływanie światła z powierzchnią obiektu. Składa się z trzech składowych:

2.3.1. Oświetlenie ambient

Symuluje rozproszone światło otoczenia, niezależne od kierunku i położenia źródła światła:

$$I_{ambient} = k_a \cdot I_a$$

gdzie k_a to współczynnik odblaskowości materiału, a I_a to intensywność światła otoczenia.

2.3.2. Oświetlenie diffuse

Symuluje rozpraszanie światła na matowej powierzchni zgodnie z prawem Lamberta:

$$I_{diffuse} = k_d \cdot I_l \cdot \max(\mathbf{n} \cdot \mathbf{l}, 0)$$

gdzie k_d to współczynnik rozpraszania, I_l to intensywność światła, $\mathbf{n}$ to wektor normalny powierzchni, a $\mathbf{l}$ to wektor kierunku do źródła światła.

2.3.3. Oświetlenie specular

Symuluje odbicia zwierciadlane (w niniejszym projekcie niezaimplementowane):

$$I_{specular} = k_s \cdot I_l \cdot \max(\mathbf{r} \cdot \mathbf{v}, 0)^{\alpha}$$

gdzie k_s to współczynnik odbicia, $\mathbf{r}$ to wektor odbity, $\mathbf{v}$ to wektor kierunku do obserwatora, a α to wykładnik połysku.

W zaimplementowanym akceleratorze wykorzystano składowe ambient i diffuse, obliczane w przestrzeni widoku dla każdego wierzchołka, a następnie interpolowane na powierzchni trójkąta podczas rasteryzacji.

2.4. Rasteryzacja trójkątów

Rasteryzacja to proces określania, które piksele ekranu są pokryte przez prymityw geometryczny [1]. Dla trójkątów wykorzystuje się następujące metody:

2.4.1. Funkcje krawędziowe

Dla trójkąta o wierzchołkach $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$ definiuje się trzy funkcje krawędziowe:

$$E_i(x, y) = (x - x_i)(y_{i+1} - y_i) - (y - y_i)(x_{i+1} - x_i)$$

Piksel (x, y) należy do wnętrza trójkąta, jeśli wszystkie trzy funkcje krawędziowe mają ten sam znak (zgodny z orientacją trójkąta).

2.4.2. Współrzędne barycentryczne

Współrzędne barycentryczne $(\lambda_0, \lambda_1, \lambda_2)$ określają, w jakim stopniu dane wierzchołki wpływają na ostateczną wartość atrybutów. Dla punktu (x, y):

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

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

Współrzędne te służą do interpolacji atrybutów (koloru, głębokości, współrzędnych tekstury) na powierzchni trójkąta.

2.4.3. Interpolacja perspektywiczna

W przypadku projekcji perspektywicznej, liniowa interpolacja w przestrzeni ekranu nie jest poprawna. Aby uzyskać poprawne wartości atrybutów, należy:

1. Pomnożyć atrybuty przez odwrotność współrzędnej w $(1/w)$ w wierzchołkach.
2. Wykonać liniową interpolację tak przygotowanych wartości oraz samej wartości $1/w$.
3. Podzielić wyinterpolowane wartości atrybutów przez wyinterpolowaną wartość $1/w$.

Matematycznie, dla atrybutu A:

$$A(x, y) = \frac{\sum_{i=0}^{2} \lambda_i \cdot A_i / w_i}{\sum_{i=0}^{2} \lambda_i / w_i}$$

2.5. Testy głębokości i szablonu

2.5.1. Test głębokości (Z-buffer)

Test głębokości (depth test) służy do rozwiązywania problemu widoczności, czyli określania, które fragmenty znajdują się najbliżej obserwatora i powinny być widoczne. Dla każdego fragmentu o współrzędnej głębokości z porównywana jest wartość z aktualną wartością w buforze głębokości z_{buffer}. Fragment przechodzi test,

jeśli:

$$z \text{ op } z_{buffer}$$

gdzie op to operator porównania (np. LESS, LEQUAL, GREATER).

Jeśli fragment przejdzie test, jego wartość z zastępuje wartość w buforze (jeśli zapis jest włączony).

2.5.2. Test szablonu (Stencil buffer)

Bufor szablonu (stencil buffer) służy do selektywnego maskowania pikseli i realizacji złożonych efektów renderowania (np. cienie, odbicia, kontury). Test szablonu porównuje wartość referencyjną s_{ref} z wartością w buforze szablonu s_{buffer} przy użyciu operatora porównania i maski:

$$(s_{ref} \,\&\, \text{mask}) \text{ op } (s_{buffer} \,\&\, \text{mask})$$

W zależności od wyniku testu szablonu i testu głębokości, wartość w buforze szablonu może być modyfikowana według określonych operacji (KEEP, ZERO, REPLACE, INCR, DECR, INVERT).

2.6. Mieszanie kolorów (Alpha blending)

Alpha blending umożliwia renderowanie obiektów przezroczystych lub półprzezroczystych poprzez mieszanie koloru fragmentu (C_{src}) z kolorem już znajdującym się w buforze ramki (C_{dst}). Operacja ta opisywana jest wzorem:

$$C_{final} = C_{src} \cdot f_{src} \text{ op } C_{dst} \cdot f_{dst}$$

gdzie:

- f_{src}, f_{dst} — współczynniki blendingu (np. α_{src}, $1 - \alpha_{src}$),
- op — operator blendingu (np. ADD, SUBTRACT, REVERSE_SUBTRACT).

Typowe ustawienie dla przezroczystości to:

$$C_{final} = C_{src} \cdot \alpha_{src} + C_{dst} \cdot (1 - \alpha_{src})$$

Rozdział 3.

Technologie i narzędzia

3.1. OpenGL ES 1.1 Common-Lite

OpenGL ES [1] to specyfikacja API grafiki 3D przeznaczona dla systemów wbudowanych. Wersja 1.1 definiuje fixed-pipeline, czyli potok graficzny o ustalonej strukturze, w przeciwieństwie do nowszych wersji obsługujących programowalne shadery.

Profil *Common-Lite* stanowi uproszczoną wersję profilu Common, przeznaczoną dla platform o ograniczonej mocy obliczeniowej, które nie posiadają sprzętowej obsługi arytmetyki zmiennoprzecinkowej. W profilu tym operacje matematyczne wykonywane są w arytmetyce stałoprzecinkowej (fixed-point).

Kluczowe cechy OpenGL ES 1.1 Common-Lite obejmują:

- obsługę transformacji geometrycznych macierzami,
- model oświetlenia Phonga z wieloma źródłami światła,
- teksturowanie 2D,
- rasteryzację trójkątów, linii i punktów,
- testy głębokości i szablonu,
- mieszanie kolorów (alpha blending),
- arytmetykę stałoprzecinkową.

Niniejszy projekt implementuje wybrany podzbiór tej specyfikacji, koncentrując się na kluczowych funkcjonalnościach potoku graficznego.

3.2. Język opisu sprzętu Amaranth HDL

Amaranth HDL (wcześniej znany jako nMigen) to nowoczesny język opisu sprzętu wbudowany w język Python. W przeciwieństwie do klasycznych języków HDL takich jak Verilog czy VHDL, Amaranth wykorzystuje możliwości języka wysokiego poziomu (Python) do generowania i kompozycji modułów sprzętowych.

3.2.1. Główne cechy Amaranth

- **Generatywność** — możliwość programowego tworzenia złożonych struktur sprzętowych przy użyciu pętli, funkcji i klas Pythona.
- **Modularność** — system komponentów i sygnatur (`wiring.Component`, `wiring.Signature`) ułatwia budowę hierarchicznych projektów.
- **Biblioteka standardowa** — gotowe komponenty takie jak FIFO (`SyncFIFO`, `AsyncFIFO`), strumienie (`stream`), narzędzia synchronizacji między domenami zegarowymi.
- **Symulacja** — zintegrowane środowisko symulacyjne umożliwia testowanie projektów bez konieczności syntezy sprzętowej.
- **Interoperacyjność** — możliwość eksportu do Verilog/SystemVerilog oraz integracji z istniejącymi projektami.

3.2.2. Struktura projektu w Amaranth

Podstawową jednostką w Amaranth jest `Module`, który grupuje logikę kombinacyjną (`m.d.comb`) i synchroniczną (`m.d.sync`). Moduły mogą być zagnieżdżane i łączone za pomocą systemu portów i sygnałów.

Przykładowy moduł w Amaranth:

```
from amaranth import *
from amaranth.lib import wiring
from amaranth.lib.wiring import Component, In, Out

class Adder(Component):
    a: In(unsigned(32))
    b: In(unsigned(32))
    result: Out(unsigned(33))

    def elaborate(self, platform):
        m = Module()
        m.d.comb += self.result.eq(self.a + self.b)
        return m
```

Listing 3..1: Przykład modułu Amaranth

3.3. Platforma sprzętowa DE1-SoC

Projekt został zaimplementowany i przetestowany na płycie deweloperskiej DE1-SoC firmy Terasic, wyposażonej w układ FPGA Intel Cyclone V SoC.

3.3.1. Specyfikacja układu Cyclone V SoC

Intel Cyclone V SoC to hybryda łącząca:

- **FPGA** — programowalna logika z 32 070 elementów logicznych (ALM), 4 065 280 bity pamięci wbudowanej (M10K i MLAB), 87 bloków DSP (jedno mnożenie 27x27, dwa 18x18 lub trzy 9x9).
- **HPS** (Hard Processor System) — dwurdzeniowy procesor ARM Cortex-A9 o taktowaniu 800 MHz, 1 GB pamięci DDR3, kontrolery pamięci i interfejsów peryferyjnych.

3.3.2. Zasoby płyty DE1-SoC

Płyta DE1-SoC oferuje:

- wyjście VGA (DAC 4-bit na kanał RGB),
- 64 MB pamięci SDRAM podłączonej do FPGA,
- 1 GB pamięci DDR3 współdzielonej z HPS,
- interfejsy GPIO, Ethernet, USB,
- obsługę kart SD do bootowania systemu Linux.

Architektura SoC pozwala na ścisłą integrację logiki programowalnej (akcelerator graficzny) z procesorem ARM (system operacyjny, aplikacje).

3.4. Integracja z systemem — SoC i magistrale

3.4.1. Magistrala Wishbone

Magistrala Wishbone to otwarty standard magistrali systemowej, powszechnie stosowany w projektach open-source i SoC. Specyfikacja definiuje synchroniczny interfejs master-slave z sygnałami:

- `adr` — adres,

- `dat_w`, `dat_r` — dane do zapisu/odczytu,
- `we` — sygnał zapisu (write enable),
- `sel` — wybór bajtów,
- `cyc`, `stb` — sygnały cyklu i aktywacji transferu,
- `ack` — potwierdzenie transakcji.

W projekcie magistrala Wishbone służy do komunikacji akceleratora z pamięcią oraz systemem HPS poprzez mostek AXI–Wishbone.

3.4.2. Magistrala Avalon

Avalon to zastrzeżony standard magistrali firmy Intel, wykorzystywany w środowisku Qsys/Platform Designer. Magistrala Avalon Memory-Mapped (Avalon-MM) umożliwia dostęp do pamięci i rejestrów sterujących.

W projekcie wykorzystano:

- **Avalon-MM Master** — do dostępu akceleratora do pamięci (bufory wierzchołków, indeksów, ramki, głębokości).
- **Avalon-MM Slave** — do rejestrów CSR (Control and Status Registers), dostępnych z poziomu procesora ARM.

3.4.3. Interfejs CSR

Moduł `GraphicsPipelineAvalonCSR` definiuje mapę rejestrów sterujących akceleratorem. Oprogramowanie z poziomu systemu Linux może odczytywać i zapisywać te rejestry przez mapowanie pamięci (`/dev/mem`), konfigurując parametry renderowania:

- adresy i formaty buforów (wierzchołki, indeksy, ramka, głębokość, szablon),
- topologię prymitywów (TRIANGLE_LIST, TRIANGLE_STRIP, TRIANGLE_FAN),
- macierze transformacji,
- parametry oświetlenia,
- ustawienia testów głębokości, szablonu, mieszania kolorów,
- parametry viewport i scissor.

3.5. Arytmetyka stałoprzecinkowa

Ze względu na charakter platformy FPGA oraz zgodność ze specyfikacją OpenGL ES 1.1 Common-Lite, projekt wykorzystuje arytmetykę stałoprzecinkową [**?**] zamiast zmiennoprzecinkowej.

3.5.1. Formaty reprezentacji

W projekcie zastosowano następujące formaty:

- **Q13.13** — 26-bitowy format ze znakiem: 13 bitów całkowitych, 13 bitów ułamkowych. Używany do reprezentacji pozycji, normalnych, macierzy transformacji.
- **Q1.17** — 18-bitowy format: 1 bit całkowity, 17 bitów ułamkowych. Używany do współrzędnych barycentrycznych, głębokości, znormalizowanych wektorów.
- **UQ0.9** — 9-bitowy format bez znaku: 0 bitów całkowitych, 9 bitów ułamkowych. Używany do kanałów koloru (wartości w zakresie $[0, 1)$).

3.5.2. Operacje arytmetyczne

Amaranth HDL automatycznie zarządza szerokością sygnałów w operacjach arytmetycznych. W projekcie zdefiniowano własne typy stałoprzecinkowe, które integrują się z systemem typów Amaranth i zapewniają automatyczną obsługę:

- dodawania i odejmowania (z poszerzeniem wyniku),
- mnożenia (wykorzystanie bloków DSP FPGA),
- nasycania (saturate) — ograniczanie wartości do zakresu docelowego typu.

Bloki DSP układu Cyclone V umożliwiają efektywne realizowanie mnożeń stałoprzecinkowych, co jest kluczowe dla wydajności transformacji geometrycznych i cieniowania. Wybrane szerokości sygnałów zostały dostosowane do optymalnego wykorzystania tych bloków (mnożenie Q13.13 działające na 27-bitowych operandach, UQ1.17 na 18-bitowych, a UQ0.9 na 9-bitowych).

Rozdział 4.

Architektura akceleratora

4.1. Ogólny przegląd architektury

Zaprojektowany akcelerator graficzny PixelForge realizuje kompletny potok renderowania fixed-pipeline zgodny z podzbiorem specyfikacji OpenGL ES 1.1 Common-Lite. Architektura jest w pełni potokowa, co oznacza, że różne etapy przetwarzania mogą działać równolegle nad różnymi danymi, maksymalizując przepustowość systemu.

Potok składa się z następujących głównych modułów (w kolejności przetwarzania):

1. Index Generator — generator indeksów wierzchołków
2. Input Topology Processor — procesor topologii wejściowej
3. Input Assembly — asembler danych wejściowych
4. Vertex Transform — transformacja wierzchołków
5. Vertex Shading — cieniowanie wierzchołków
6. Primitive Clipper — przycinanie prymitywów
7. Perspective Divide — dzielenie perspektywiczne
8. Triangle Prep — przygotowanie trójkątów
9. Triangle Rasterizer — rasteryzacja trójkątów (wiele równoległych jednostek)
10. Depth/Stencil Test — test głębokości i szablonu
11. Swapchain Output — wyjście do bufora ramki

Moduły połączone są interfejsami strumieniowymi (`amaranth.lib.stream`), a między nimi znajdują się bufory FIFO synchroniczne lub asynchroniczne, które:

- izolują moduły od siebie, ułatwiając osiąganie wymagań czasowych,
- obsługują zmienną latencję poszczególnych etapów (np. przycinanie może generować różną liczbę trójkątów),
- umożliwiają pracę w różnych domenach zegarowych.

4.2. Podział na domeny zegarowe

Potok graficzny podzielony jest na dwie główne domeny zegarowe w celu optymalizacji wydajności i spełnienia wymagań czasowych:

- **Domena wierzchołkowa/prymitywów** — obejmuje moduły od Index Generator do Triangle Prep. Moduły te wykonują złożone obliczenia na wierzchołkach (transformacje macierzowe, cieniowanie, przycinanie). Mogą działać przy niższej częstotliwości zegarowej, gdyż ich przepustowość nie jest krytyczna dla ogólnej wydajności.
- **Domena rasteryzacji/fragmentów** — obejmuje moduły od Triangle Rasterizer do Swapchain Output. Moduły te operują na pikselach i stanowią wąskie gardło wydajnościowe (fill-rate limited). Działają przy wyższej częstotliwości zegarowej, aby maksymalizować liczbę przetwarzanych pikseli na sekundę.

Połączenie między domenami realizowane jest poprzez asynchroniczne FIFO (`AsyncFIFO`), które bezpiecznie przekazują dane między różnymi zegarami.

4.3. Moduły przetwarzania wierzchołków

4.3.1. Index Generator

Moduł `IndexGenerator` generuje strumień indeksów wierzchołków na podstawie skonfigurowanego bufora indeksów. Obsługuje formaty:

- `U8` — 8-bitowe indeksy bez znaku,
- `U16` — 16-bitowe indeksy bez znaku,
- `U32` — 32-bitowe indeksy bez znaku,
- `NOT_INDEXED` — tryb bezindeksowy (generowanie sekwencji 0, 1, 2, ...).

Moduł odczytuje dane z pamięci poprzez interfejs Wishbone Master, obsługując różne szerokości odczytu w zależności od formatu indeksów. Rozpoczęcie generowania indeksów następuje po impulsie sygnału `start`.

4.3.2. Input Topology Processor

Moduł `InputTopologyProcessor` przetwarza strumień indeksów zgodnie z wybraną topologią prymitywów:

- `TRIANGLE_LIST` — każde trzy kolejne indeksy tworzą niezależny trójkąt,
- `TRIANGLE_STRIP` — kolejne trójkąty współdzielą dwa wierzchołki,
- `TRIANGLE_FAN` — wszystkie trójkąty mają wspólny pierwszy wierzchołek.
- `LINE_LIST` — każde dwa kolejne indeksy tworzą niezależną linię,
- `LINE_STRIP` — kolejne linie współdzielą jeden wierzchołek.
- `POINTS` — każdy indeks reprezentuje pojedynczy punkt.

Moduł obsługuje również funkcję *primitive restart* — specjalną wartość indeksu, która przerywa bieżący strip/fan i rozpoczyna nowy, oraz *base vertex offset* — wartość dodawaną do każdego indeksu.

4.3.3. Input Assembly

Moduł `InputAssembly` pobiera dane atrybutów wierzchołków z pamięci na podstawie strumienia indeksów. Każdy atrybut (pozycja, normalna, kolor, współrzędne tekstury) może być skonfigurowany jako:

- **atrybut z bufora** — pobierany z pamięci pod adresem: $\text{base} + \text{index} \times \text{stride}$,
- **atrybut stały** — ta sama wartość dla wszystkich wierzchołków.

Obecnie moduł obsługuje jedynie format Q16.16 dla atrybutów w pamięci (dane są przechowywane jako 32-bitowe wartości ze znakiem). Dane są odczytywane z pamięci poprzez magistralę Wishbone.

4.3.4. Vertex Transform

Moduł `VertexTransform` wykonuje transformacje geometryczne wierzchołków przy użyciu macierzy 4×4 (dla pozycji i współrzędnych tekstury) oraz 3×3 (dla normalnych). Transformacje obejmują:

- transformację pozycji do przestrzeni widoku: $\mathbf{p}_{view} = M_{mv} \cdot \mathbf{p}_{model}$,
- transformację pozycji do przestrzeni obcinania: $\mathbf{p}_{clip} = M_{p} \cdot \mathbf{p}_{view}$,

- transformację normalnej do przestrzeni widoku: $\mathbf{n}_{view} = M_{norm} \cdot \mathbf{n}_{model}$, gdzie $M_{norm} = (M_{mv}^{-1})^T$.

Cały układ wykorzystuje tylko jeden układ mnożący, co pozwala na oszczędność zasobów sprzętowych kosztem wydłużenia latencji przetwarzania wierzchołka. Nie jest to jednak problemem, gdyż istnieją inne etapy potoku o znacznie wyższej latencji (np. rasteryzacja).

4.3.5. Vertex Shading

Moduł `VertexShading` oblicza kolor wierzchołka na podstawie modelu oświetlenia Phonga. Implementuje składowe:

- **Ambient** — $C_{ambient} = k_{ambient} \cdot I_{ambient}$,
- **Diffuse** — $C_{diffuse} = k_{diffuse} \cdot I_{light} \cdot \max(\mathbf{n} \cdot \mathbf{l}, 0)$.

Moduł obsługuje konfigurowalną liczbę kierunkowych źródeł światła. Obliczenia wykonywane są w przestrzeni widoku, zgodnie ze specyfikacją OpenGL ES 1.1. Kolor wierzchołka obliczony tutaj będzie interpolowany (z użyciem perspektywicznej interpolacji) podczas rasteryzacji.

4.4. Moduły przetwarzania prymitywów

4.4.1. Primitive Clipper

Moduł `PrimitiveClipper` implementuje algorytm przycinania Sutherlanda-Hodgmana, który przycina trójkąty do bryły obcinania (frustum). Bryła ta w przestrzeni obcinania definiowana jest sześcioma płaszczyznami:

$$-w \leq x \leq w, \quad -w \leq y \leq w, \quad -w \leq z \leq w$$

Dla każdego trójkąta algorytm iteracyjnie przycina go względem kolejnych płaszczyzn. Wynikiem może być:

- brak trójkąta (cały poza frustum),
- ten sam trójkąt (cały wewnątrz),
- wiele trójkątów (częściowe przecięcie — powstają nowe wierzchołki w miejscach przecięć z płaszczyznami).

Ten moduł, ze względu na iteracyjną naturę algorytmu oraz próbę minimalizacji zużycia zasobów, jest potencjalnie wąskim gardłem potoku, ale trójkąty wychodzące poza frustum są stosunkowo rzadkie, więc ogólna wydajność jest ograniczana tylko w patologicznych przypadkach.

4.4.2. Perspective Divide

Moduł `PerspectiveDivide` wykonuje dzielenie perspektywiczne, czyli przekształcenie współrzędnych jednorodnych z przestrzeni obcinania do znormalizowanych współrzędnych urządzenia (NDC):

$$x_{ndc} = \frac{x_{clip}}{w}, \quad y_{ndc} = \frac{y_{clip}}{w}, \quad z_{ndc} = \frac{z_{clip}}{w}$$

Dodatkowo obliczana jest wartość $1/w$, która będzie potrzebna do perspektywicznej interpolacji atrybutów podczas rasteryzacji.

Dzielenie realizowane jest poprzez obliczenie odwrotności w, a następnie mnożenie współrzędnych przez tę odwrotność.

Moduł obliczający odwrotność (`FixedPointInv`) zamienia dzielnik do dziedziny [1, 2) poprzez odpowiednie przesunięcie bitowe, bierze początkowe przybliżenie z tablicy LUT, a następnie poprawia wynik za pomocą jednej iteracji metody Newtona-Raphsona. Całość używa jedynie jeden blok DSP o szerokości takiej, jak wejście.

Po tej operacji współrzędne są w zakresie $[-1, 1]$, zamieniamy je do zakresu $[0, 1]$ dla ułatwienia dalszych obliczeń:

$$x'_{ndc} = \frac{x_{ndc} + 1}{2}, \quad y'_{ndc} = \frac{y_{ndc} + 1}{2}, \quad z'_{ndc} = \frac{z_{ndc} + 1}{2}$$

Dzięki temu możemy użyć formatu UQ1.17 do reprezentacji współrzędnych NDC, co pozwoli na efektywniejsze wykorzystanie zasobów sprzętowych w dalszych etapach potoku.

4.4.3. Triangle Prep

Moduł `TrianglePrep` zbiera trzy wierzchołki trójkąta (w przestrzeni NDC) i przygotowuje dane do rasteryzacji:

1. **Transformacja viewport** — przekształcenie współrzędnych NDC $[0, 1]$ do współrzędnych ekranu w pikselach:

   $$x_{screen} = x'_{ndc} \cdot \text{width} + x_{offset}$$

   $$y_{screen} = y'_{ndc} \cdot \text{height} + y_{offset}$$

   Każdy wierzchołek otrzymuje swój zestaw współrzędnych ekranowych (x_{screen}, y_{screen}). Niech $(x_0, y_0), (x_1, y_1), (x_2, y_2)$ będą współrzędnymi ekranowymi trzech wierzchołków trójkąta.

2. **Obliczenie pola powierzchni** trójkąta (dwukrotność pola):

   $$A = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

3. **Face culling** — na podstawie znaku pola powierzchni i konfiguracji (front-face, cull-mode) decyzja, czy trójkąt powinien być renderowany. Odrzucane są trójkąty zwrócone tyłem (back-facing) lub przodem (front-facing), zgodnie z konfiguracją.
4. **Obliczenie odwrotności pola** $(1/A)$ — wykorzystywane do normalizacji współrzędnych barycentrycznych.
5. **Obliczenie bounding box** — prostokąt ograniczający trójkąt w pikselach, ograniczony dodatkowo przez viewport i scissor rectangle.

Wynikiem jest struktura `TriangleContext`, która zawiera wszystkie dane potrzebne do rasteryzacji.

4.5. Rasteryzacja i przetwarzanie fragmentów

4.5.1. Triangle Rasterizer

Moduł `TriangleRasterizer` jest najbardziej złożonym elementem potoku. Realizuje rasteryzację trójkąta, czyli określenie, które piksele są pokryte przez trójkąt, oraz obliczenie interpolowanych atrybutów dla każdego fragmentu.

Architektura modułu:

- **Generator pikseli** — przechodzi po bounding box trójkąta, generując współrzędne (x, y) kolejnych pikseli.
- **Distributor** — rozdziela strumień pikseli do wielu równoległych procesorów fragmentów.
- **Procesory fragmentów** (wiele równoległych jednostek) — dla każdego piksela:
  1. obliczają wartości funkcji krawędziowych (edge functions),
  2. sprawdzają, czy piksel jest wewnątrz trójkąta (wszystkie funkcje mają ten sam znak),
  3. obliczają liniowe współrzędne barycentryczne,
  4. interpolują liniowo głębokość z,
  5. obliczają perspektywicznie poprawne współrzędne barycentryczne (z użyciem $1/w$),
  6. interpolują atrybuty (kolor, współrzędne tekstury) perspektywicznie.
- **Recombiner** — zbiera wyniki z procesorów fragmentów i przekazuje je dalej w kolejności, zapewniając zgodność z semantyką OpenGL (fragmenty muszą być przetwarzane zgodnie z kolejnością trójkątów, z których pochodzą).

Zastosowanie wielu równoległych procesorów fragmentów pozwala na ukrycie latencji dzielenia (potrzebnego do obliczenia współrzędnych perspektywicznych) poprzez przetwarzanie wielu pikseli jednocześnie.

4.5.2. Depth/Stencil Test

Moduł `DepthStencilTest` dla każdego fragmentu:

1. Odczytuje aktualną wartość głębokości i szablonu z pamięci (bufor depth/stencil w formacie D16_X8_S8, 16 bitów głębokości, 8 bitów szablonu).
2. Wykonuje test szablonu zgodnie z konfiguracją (operator porównania, wartość referencyjna, maska).
3. Wykonuje test głębokości (jeśli test szablonu się powiódł).
4. Aktualizuje wartość w buforze szablonu zgodnie z operacjami SFAIL, ZFAIL, ZPASS.
5. Aktualizuje wartość w buforze głębokości (jeśli test głębokości się powiódł i zapis jest włączony).
6. Przekazuje fragment dalej (do Swapchain Output) tylko wtedy, gdy oba testy się powiodły.

Moduł obsługuje dwa zestawy konfiguracji — dla trójkątów front-facing i back-facing, zgodnie ze specyfikacją OpenGL.

4.5.3. Swapchain Output

Moduł `SwapchainOutput` wykonuje ostatni etap potoku:

1. Dla fragmentu, który przeszedł testy głębokości/szablonu, odczytuje aktualny kolor z bufora ramki.
2. Wykonuje operację blendingu (mieszania kolorów) zgodnie z konfiguracją:

   $$C_{out} = C_{src} \cdot f_{src} \text{ op } C_{dst} \cdot f_{dst}$$

   gdzie C_{src} to kolor fragmentu, C_{dst} to kolor w buforze, f_{src}, f_{dst} to współczynniki, a op to operator (ADD, SUBTRACT, REVERSE_SUBTRACT).
3. Zapisuje wynikowy kolor do bufora ramki.

Moduł wykorzystuje format koloru UQ0.9 (9-bitowy bez znaku) dla każdego kanału RGB i A, co odpowiada zakresowi $[0, 1)$ i pozwala na efektywne wykorzystanie małych bloków DSP (9×9 bitów).

4.6. Interfejs sterowania — CSR

Moduł `GraphicsPipelineCSR` oraz jego wariant z interfejsem Avalon (`GraphicsPipelineAvalonCSR`) zapewniają dostęp do rejestrów sterujących akceleratorem z poziomu procesora ARM.

Rejestry podzielone są na grupy funkcjonalne:

- **Index configuration** — adres bufora indeksów, liczba indeksów, format, sygnał start.
- **Input topology** — topologia (LIST/STRIP/FAN), primitive restart, base vertex.
- **Vertex attributes** — dla każdego atrybutu (pozycja, normalna, kolor, współrzędne tekstury): adres bazowy, stride, offset, flaga czy używać bufora czy stałej wartości.
- **Transformation matrices** — macierze model-view (4×4), projekcji (4×4), normalnych (3×3), tekstur (4×4 dla każdej jednostki teksturującej).
- **Lighting** — dla każdego źródła światła: kierunek, kolory ambient/diffuse/specular.
- **Material** — kolory ambient/diffuse/specular materiału, połysk.
- **Framebuffer** — adresy buforów koloru, głębokości, szablonu; formaty; rozdzielczość.
- **Primitive assembly** — front-face, cull-mode, polygon-mode.
- **Viewport/Scissor** — parametry transformacji viewport, prostokąt scissor.
- **Depth/Stencil** — konfiguracje testów głębokości i szablonu (operatory, maski, wartości referencyjne, operacje aktualizacji).
- **Blending** — włączenie blendingu, współczynniki, operator.
- **Status** — flagi busy dla poszczególnych etapów potoku, pozwalające na synchronizację (oczekiwanie na zakończenie renderowania).

Oprogramowanie może odczytywać status potoku, aby określić, kiedy bezpiecznie jest zmieniać konfigurację lub rozpoczynać nowe wywołanie rysowania. Statusy `busy` wskazują, które etapy wciąż przetwarzają dane:

- `busy_ia` — Input Assembly (bezpieczna zmiana vertex/index buffers po wyzerowaniu),
- `busy_vs` — Vertex Shading (bezpieczna zmiana macierzy/oświetlenia po wyzerowaniu),

- `busy_prep` — Triangle Prep (bezpieczna zmiana viewport/scissor po wyzerowaniu),
- `busy_all` — cały potok (bezpieczna zmiana adresu bufora ramki po wyzerowaniu).

Rozdział 5.

Implementacja

5.1. Struktura projektu

Projekt zorganizowany jest w strukturę modułową, wykorzystującą możliwości języka Python i Amaranth HDL. Główne katalogi:

- `gpu/` — moduły akceleratora graficznego w Amaranth HDL:
  - `input_assembly/` — moduły generowania indeksów, topologii i asemblera wierzchołków,
  - `vertex_transform/` — transformacje geometryczne,
  - `vertex_shading/` — cieniowanie wierzchołków,
  - `rasterizer/` — przycinanie, dzielenie perspektywiczne, przygotowanie i rasteryzacja trójkątów,
  - `pixel_shading/` — testy głębokości/szablonu, blending, wyjście do framebuffer,
  - `utils/` — narzędzia pomocnicze (arytmetyka stałoprzecinkowa, interfejsy magistral, matematyka),
  - `pipeline.py` — główny moduł łączący cały potok.
- `tests/` — testy jednostkowe i integracyjne w pytest,
- `quartus/` — projekt Intel Quartus (Qsys i projekt na DE1-SoC),
- `software/` — aplikacje demonstracyjne i narzędzia diagnostyczne w C dla ARM Linux:
  - `src/` — kod źródłowy aplikacji,
  - `include/` — nagłówki (mapa CSR, mapa pamięci SoC, API PixelForge),
- `tools/` — skrypty pomocnicze (generowanie nagłówków C z map CSR).

5.2. Generacja kodu sprzętowego

Główny moduł `gpu/pipeline.py` definiuje klasę `GraphicsPipelineAvalonCSR`, która instancjonuje wszystkie moduły potoku i łączy je interfejsami strumieniowymi oraz rejestrami CSR.

Elaboracja projektu (konwersja z Amaranth do SystemVerilog) odbywa się poprzez wywołanie:

```
python -m gpu.pipeline
```

które generuje:

- `graphics_pipeline_avalon_csr.sv` — plik SystemVerilog z całym potokiem graficznym,
- `graphics_pipeline_csr_map.json` — mapa rejestrów CSR (nazwy, adresy, szerokości).

Plik `.sv` jest następnie integrowany z projektem Qsys jako komponent IP.

5.3. Integracja z Qsys

W środowisku Intel Quartus Prime wykorzystano narzędzie Qsys (Platform Designer) do budowy systemu SoC. Projekt `soc_system.qsys` zawiera:

- **HPS** — Hard Processor System (ARM Cortex-A9, kontrolery pamięci DDR3, peryferyjne),
- **PixelForge GPU** — zaimplementowany akcelerator graficzny (`graphics_pipeline_avalon_csr`):
  - interfejs Avalon-MM Slave (CSR) — podłączony do magistrali HPS,
  - interfejsy Avalon-MM Master (vertex, depth/stencil, color) — podłączone do kontrolera pamięci SDRAM lub DDR3.
- **VGA Pixel Buffer DMA** — kontroler DMA przekazujący dane z bufora ramki do przetwornika DAC VGA.
- **PLL** — generatory zegarów dla różnych części systemu (domena wierzchołków, domena fragmentów, VGA pixel clock).
- **Mosty i magistrale** — mosty AXI-to-Avalon, Avalon-to-Wishbone, magistrale łączące komponenty.

Po wygenerowaniu systemu w Qsys, projekt kompilowany jest w Quartus, generując plik konfiguracyjny `.sof` (SRAM Object File), który następnie konwertowany jest do formatu `.rbf` (Raw Binary File) umożliwiającego bootowanie z karty SD.

5.4. Kluczowe decyzje implementacyjne

5.4.1. Wykorzystanie bloków DSP

Układ Cyclone V SoC posiada 87 bloków DSP, które wspierają:

- mnożenie 9×9 bitów (dla kolorów),
- mnożenie 18×18 bitów (dla współrzędnych NDC/ekranu),
- mnożenie 27×27 bitów (dla transformacji wierzchołków).

W implementacji wykorzystano te bloki do realizacji operacji arytmetyki stałoprzecinkowej. Po dzieleniu perspektywicznym dane są konwertowane z formatu Q13.13 (wymagającego bloków 27×27) do formatu Q1.17 (wymagającego bloków 18×18), co podwaja efektywność użycia DSP w etapach rasteryzacji.

5.4.2. Równoległe procesory fragmentów

Rasteryzacja jest najbardziej czasochłonnym etapem potoku. Aby zwiększyć przepustowość, zaimplementowano wiele równoległych procesorów fragmentów. Każdy procesor może niezależnie sprawdzać, czy piksel należy do trójkąta, i obliczać interpolowane atrybuty.

Liczba procesorów jest parametrem konfigurowalnym. W przykładzie użyto 8 procesorów, co pozwala na osiągnięcie zadowalającej wydajności przy rozsądnym zużyciu zasobów FPGA.

5.4.3. Bufory FIFO

Między każdym etapem potoku umieszczono bufory FIFO o głębokości 256 elementów (domyślnie). Pozwala to na:

- ukrycie opóźnień związanych z dostępem do pamięci,
- obsługę zmiennej latencji (np. przycinanie może generować więcej trójkątów),
- ułatwienie spełnienia wymagań czasowych (timing closure) poprzez izolację sygnałów.

5.4.4. Dostęp do pamięci

Akcelerator posiada cztery niezależne interfejsy Wishbone Master do dostępu do pamięci:

- `index_bus` — odczyt bufora indeksów (Index Generator),
- `vertex_bus` — odczyt buforów wierzchołków i indeksów (Input Assembly),
- `depthstencil_bus` — odczyt/zapis bufora głębokości i szablonu (Depth/Stencil Test),
- `color_bus` — odczyt/zapis bufora ramki (Swapchain Output).

Rozdzielenie magistral zapobiega konfliktom dostępu i maksymalizuje przepustowość pamięci. W implementacji Qsys wszystkie cztery magistrale są podłączone do wspólnego kontrolera pamięci, który arbitruje dostęp.

5.5. Oprogramowanie systemowe

5.5.1. Środowisko uruchomieniowe

System uruchomieniowy na płycie DE1-SoC oparty jest na Linuksie dla ARM. Obraz systemu (`disk.img`) zawiera:

- bootloader U-Boot,
- jądro Linux skonfigurowane dla DE1-SoC,
- system plików z minimalnym środowiskiem użytkownika (BusyBox),
- plik `.rbf` z konfiguracją FPGA, ładowany automatycznie przy starcie.
- testowe aplikacje demonstracyjne i biblioteka PixelForge.

Po uruchomieniu systemu użytkownik loguje się jako `root` (bez hasła) i uzyskuje dostęp do aplikacji demonstracyjnych.

5.5.2. Biblioteka PixelForge

Oprogramowanie zawiera bibliotekę `libpixelforge.a`, która enkapsuluje niskopoziomowy dostęp do rejestrów CSR i zapewnia API do:

- mapowania rejestrów CSR i pamięci VRAM przez `/dev/mem`,
- konfiguracji potoku graficznego (ustawianie macierzy, parametrów oświetlenia, buforów),
- inicjowania wywołań rysowania (draw calls),
- oczekiwania na zakończenie renderowania,
- zarządzania pamięcią VRAM (alokator bufora ramki, wierzchołków, głębokości).

5.5.3. Aplikacje demonstracyjne

Projekt zawiera kilka aplikacji demonstracyjnych:

- **pixelforge_demo** — minimalna aplikacja rysująca dwa trójkąty (nałożone na siebie), demonstrująca podstawowy przepływ danych.
- **demo_cube** — rysowanie obracającego się kolorowego sześcianu, demonstrująca transformacje geometryczne i interpolację kolorów.
- **demo_depth** — cztery sześciany obracające się w przestrzeni 3D, które czasami się przysłaniają, demonstrująca test głębokości.
- **demo_alpha** — obracający się wachlarz przezroczystych kwadratów z alfa-blendingiem (SRC_ALPHA / ONE_MINUS_SRC_ALPHA) oraz jeden oświetlający (additive-blending - SRC_ALPHA / ONE), demonstrując operacje blendingu kolorów.
- **demo_obj** — wczytywanie i renderowanie modeli w formacie Wavefront OBJ (np. kule, wielościany) z oświetleniem diffuse oraz opcjonalnym efektem konturu za pomocą bufora szablonu.
- **dump_gpu_csr** — narzędzie diagnostyczne wyświetlające zawartość wszystkich rejestrów CSR akceleratora.
- **dump_vga_dma** — narzędzie diagnostyczne wyświetlające rejestry kontrolera VGA DMA.

Aplikacje budowane są przy użyciu cross-compilera ARM (np. `arm-linux-gnueabihf-gcc`) i uruchamiane na płycie DE1-SoC.

5.5.4. Instalacja i uruchomienie

Wymagania

Do budowania i testowania projektu wymagane są:

- Python 3.10+,
- Amaranth HDL,
- pytest,
- Intel Quartus Prime (dla syntezy FPGA),
- ARM cross-compiler (`arm-linux-gnueabihf` lub kompatybilny) dla budowania aplikacji oprogramowania.

Instalacja

Projekt instaluje się poprzez:

```
git clone https://github.com/qbojj/PixelForge.git
cd PixelForge
pip install -e ".[dev]"
```

Wykonywanie testów

Testy jednostkowe i integracyjne uruchamiane są za pomocą pytest:

```
# wszystkie testy automatyczne (bez inspekcji wizualnej)
pytest

# wszystkie testy
pytest --run-slow
```

Wszystkie testy przechodzą pomyślnie (91 testów jednostkowych).

Budowanie dla FPGA

Proces budowania dla FPGA obejmuje:

1. Elaboracja projektu Amaranth HDL:
   `python -m gpu.pipeline`
2. Regeneracja mapowania rejestrów CSR i nagłówków C:

```
    python tools/generate_csr_headers.py \\
        --input graphics_pipeline_csr_map.json \\
        --output software/include/pixelforge_csr.h
```

3. Otwarcie projektu Quartus (`quartus/soc_system.qpf`),
4. Regeneracja systemu w Platform Designer (`quartus/soc_system.qsys`),
5. Kompilacja projektu (synteza, umieszczenie, routing),
6. Konwersja pliku `.sof` na `.rbf` do zaprogramowania FPGA:

```
    quartus_cpf -c soc_system.sof soc_system.rbf
```

7. Wgranie pliku `.rbf` na pierwszą partycję karty SD.

Budowanie aplikacji demonstracyjnych

Aplikacje demonstracyjne buduje się w katalogu `software/`:

```
cd software
export CROSS_COMPILE=arm-linux-gnueabihf-
make
```

Pliki wykonywalne można skopiować na kartę SD i uruchamiać na płycie DE1-SoC. Dostarczony jest do tego skrypt automatyzujący kopiowanie aplikacji i dostarczonych plików `.obj`.

```
sudo make DESTDIR=<katalog_docelowy> install
```

Rozdział 6.

Testowanie i weryfikacja

6.1. Metodologia testowania

Projekt obejmuje wielopoziomową strategię testowania:

1. **Testy jednostkowe modułów** — każdy moduł potoku testowany jest niezależnie w środowisku symulacyjnym Amaranth. Testy weryfikują poprawność obliczeń (np. transformacji macierzowych, interpolacji) dla różnych danych wejściowych.
2. **Testy integracyjne potoku** — pełny potok testowany jest z użyciem syntetycznych danych wejściowych (np. prosty trójkąt), następnie generowany jest obraz w formacie .ppm, do inspekcji wizualnej.
3. **Testy sprzętowe na FPGA** — akcelerator uruchomiony na płycie DE1-SoC renderuje różne sceny 3D, weryfikując zarówno poprawność funkcjonalną, jak i wydajność.

Testy realizowane są za pomocą frameworka `pytest` oraz wbudowanego symulatora Amaranth.

6.2. Test renderowania trójkąta

Test integracyjny `test_render_triangle.py` weryfikuje cały potok, renderując prosty trójkąt:

1. Konfiguracja symulowanej pamięci z danymi wierzchołków (pozycje, kolory).
2. Ustawienie macierzy transformacji (model-view, projekcja).
3. Wywołanie rysowania (start signal).

4. Odczyt bufora ramki po zakończeniu renderowania.
5. Generowanie pliku obrazu do inspekcji wizualnej.

Test wykorzystuje wbudowany symulowany kontroler pamięci Wishbone, który emuluje zachowanie rzeczywistej pamięci.

6.3. Weryfikacja na sprzęcie

Po syntezie i wgraniu bitstreamu do FPGA, wykonano szereg testów sprzętowych:

6.3.1. Test podstawowy — trójkąt

Aplikacja `pixelforge_demo` renderuje dwa nałożone na siebie trójkąty o interpolowanych kolorach wierzchołków. Test weryfikuje:

- poprawność rasteryzacji i interpolacji kolorów,
- poprawność zapisu do bufora ramki,
- poprawność wyświetlania na wyjściu VGA.

6.3.2. Test rotacji — sześcian

Aplikacja `demo_cube` renderuje obracający się sześcian. Test weryfikuje:

- poprawność transformacji w czasie (animacja),
- poprawność rysowania wielu trójkątów,
- wizualną płynność animacji (frame rate).

6.3.3. Test bufora głębokości

Aplikacja `demo_depth` renderuje cztery sześciany obracające się w przestrzeni 3D, które czasami się przysłaniają, demonstrująca test głębokości. Test weryfikuje:

- poprawność testu głębokości (przednie obiekty zasłaniają tylne),
- poprawność zapisu i odczytu bufora głębokości,
- brak artefaktów (z-fighting, niepoprawna kolejność).

6.3.4. Test oświetlenia i bufora szablonu

Aplikacja `demo_obj` wczytuje model 3D (kula) i renderuje go z oświetleniem diffuse. Opcjonalnie włącza efekt konturu za pomocą bufora szablonu (dwuprzebiegowe renderowanie). Test weryfikuje:

- poprawność obliczeń oświetlenia (model Phonga),
- poprawność testu szablonu,
- poprawność operacji na buforze szablonu (INCR, KEEP, REPLACE),
- wizualną jakość efektu konturu.

6.3.5. Test mieszania kolorów

Aplikacja `demo_alpha` renderuje obracający się wachlarz przezroczystych kwadratów z alfa-blendingiem (SRC_ALPHA / ONE_MINUS_SRC_ALPHA) oraz jeden oświetlający (additive-blending - SRC_ALPHA / ONE), demonstrując operacje blendingu kolorów. Test weryfikuje:

- poprawność mieszania kolorów,
- poprawność braku hazardów (nowe trójkąty widzą zmodyfikowane wartości bufora ramki).

6.4. Metryki wydajności

Zaimplementowany akcelerator osiąga następujące parametry wydajnościowe na płycie DE1-SoC:

- Dla testów `pixelforge_demo` i `demo_cube`, akcelerator osiąga stabilne 60 FPS przy rozdzielczości 640×480,
- Dla testu `demo_obj` z dostarczonymi modelami (około 500–1000 trójkątów), akcelerator osiąga stabilne 60 FPS przy rozdzielczości 640×480,
- Dla testu `demo_depth` akcelerator widocznie traci na wydajności, gdy kostki zajmują dużą część ekranu przez zbliżanie się do kamery.

Dla modelu kuli (960 trójkątów) w `demo_obj`, akcelerator osiąga stabilne 60 FPS, co potwierdza, że implementacja jest fill-rate limited (wydajność zależy od liczby rysowanych pikseli, a nie liczby trójkątów).

6.5. Zużycie zasobów FPGA

Po syntezie projektu w Intel Quartus Prime dla układu Cyclone V 5CSEMA5F31C6, zużycie zasobów wynosi:

Tabela 6.1: Zużycie zasobów FPGA

Zasób	**Użyte**	**Dostępne**
Elementy logiczne (ALM)	31,372	32,070
Pamięć wbudowana (bit)	634,097	4,065,280
Bloki DSP	75	87

Projekt wykorzystuje w znacznym stopniu dostępne zasoby FPGA, pozostawiając niewiele marginesu na dalsze rozszerzenia bez przenoszenia na większy układ.

Rozdział 7.

Podsumowanie i perspektywy rozwoju

7.1. Osiągnięte cele

W ramach niniejszej pracy udało się zrealizować wszystkie założone cele:

1. **Zaprojektowanie i zaimplementowanie potoku graficznego fixed-pipeline** — zbudowano kompletny akcelerator graficzny obejmujący transformacje wierzchołków, cieniowanie, przycinanie, rasteryzację i przetwarzanie fragmentów.
2. **Implementacja kluczowych funkcjonalności OpenGL ES 1.1 Common-Lite**:
   - transformacje geometryczne (macierze model-view, projekcji),
   - rasteryzacja trójkątów z perspektywiczną interpolacją,
   - oświetlenie Phonga (ambient, diffuse),
   - testy głębokości i szablonu,
   - mieszanie kolorów (alpha blending).
3. **Integracja z systemem SoC** — akcelerator został zintegrowany z procesorem ARM poprzez magistralę Wishbone/Avalon i interfejs CSR, umożliwiając konfigurację i sterowanie z poziomu systemu Linux.
4. **Weryfikacja działania** — zbudowano kompletne środowisko testowe (testy jednostkowe, integracyjne) oraz aplikacje demonstracyjne, które potwierdzają poprawność i użyteczność akceleratora.

Projekt PixelForge stanowi w pełni funkcjonalny akcelerator graficzny 3D, zdolny do renderowania scen w czasie rzeczywistym na platformie FPGA.

7.2. Wnioski

7.2.1. Amaranth HDL jako narzędzie projektowania sprzętu

Język Amaranth HDL okazał się niezwykle efektywnym narzędziem do projektowania złożonych systemów cyfrowych. Możliwość wykorzystania abstrakcji języka Python (klasy, funkcje, generatory) znacznie przyspieszyła proces tworzenia modułów i ułatwiła ich testowanie. System `wiring` oraz biblioteka `stream` zapewniły spójne interfejsy między modułami, redukując błędy integracyjne.

7.2.2. Arytmetyka stałoprzecinkowa

Zastosowanie arytmetyki stałoprzecinkowej zamiast zmiennoprzecinkowej pozwoliło na efektywne wykorzystanie bloków DSP układu FPGA oraz uproszczenie logiki. Odpowiedni dobór formatów (Q13.13, Q1.17, UQ0.9) dla różnych etapów potoku zoptymalizował zużycie zasobów sprzętowych, zachowując wystarczającą precyzję obliczeń.

7.2.3. Równoległość na poziomie fragmentów

Kluczowym elementem osiągnięcia zadowalającej wydajności było zrównoleglenie przetwarzania fragmentów. Użycie wielu procesorów fragmentów pozwoliło na ukrycie latencji dzielenia perspektywicznego i znaczące zwiększenie fill-rate.

7.2.4. Ograniczenia wydajnościowe

Akcelerator jest fill-rate limited — wydajność zależy głównie od liczby rysowanych pikseli, a nie liczby trójkątów. Dla rozdzielczości 640×480 i typowych scen osiągnięto 60 FPS. Zwiększenie rozdzielczości wymaga dalszej optymalizacji.

7.3. Prace pokrewne

Zagadnienie implementacji potoku graficznego na platformach FPGA było już wcześniej podejmowane w literaturze.

Przykładowo, praca [8] przedstawia akcelerator 3D o architekturze zbliżonej do fixed-pipeline, implementowany na FPGA, z naciskiem na algorytmy rasteryzacji i zarządzania pamięcią. Praca ta nie obejmuje transformacji geometrycznych ani oświetlenia, koncentrując się głównie na etapie rasteryzacji.

Praca magisterska [9] opisuje implementację prostego akceleratora graficznego na platformie FPGA, z obsługą transformacji wierzchołków, obcinania i rasteryzacji

lini. Jednakże nie posiada możliwości wyświetlania trójkątów, ani obsługi cienowania fragmentów.

Ta praca wyróżnia się tym, że implementuje kompletny potok, który może bez udziału procesora wykonać pełną komendę rysowania dla trójkątów, od pobrania danych z pamięci, przez transformacje, rasteryzację, aż po zapis do bufora ramki.

7.4. Możliwe rozszerzenia

Projekt stanowi solidną podstawę do dalszego rozwoju. Możliwe kierunki rozszerzeń obejmują:

7.4.1. Teksturowanie

Dodanie obsługi tekstur 2D zgodnie ze specyfikacją OpenGL ES 1.1. Wymaga to:

- modułu próbkowania tekstur (texture sampling) z buforów w pamięci,
- filtrowania (nearest, bilinear),
- obsługi różnych formatów tekstur (RGB, RGBA, paletyzowane).

7.4.2. Oświetlenie specularne

Rozszerzenie modułu `VertexShading` o składową specular modelu Phonga, co pozwoli na renderowanie błyszczących powierzchni.

7.4.3. Obsługa linii i punktów

Obecnie akcelerator obsługuje jedynie prymitywy typu trójkąt. Dodanie rasteryzacji linii (algorytm Bresenhama) i punktów rozszerzy możliwości renderowania.

7.4.4. Guard-band clipping

Zaimplementowanie techniki guard-band clipping, która redukuje konieczność przycinania dla większości trójkątów, poprawiając wydajność.

7.4.5. Mipmapping

Obsługa mipmap dla tekstur, poprawiająca jakość renderowania obiektów w różnych odległościach i redukująca aliasing.

7.4.6. Większe rozdzielczości

Dostosowanie systemu do wyższych rozdzielczości (np. 1280×720, 1920×1080) poprzez zwiększenie przepustowości pamięci i liczby procesorów fragmentów.

7.4.7. TBR (Tile-Based Rendering)

Implementacja techniki Tile-Based Rendering, która dzieli scenę na mniejsze kafelki, co pozwala na efektywniejsze zarządzanie pamięcią i redukcję nadmiarowego renderowania. Pozwoliło by to na oddzielenie fazy rasteryzacji od magistrali pamięci, co by pozwoliło na osiągnięcie częstotliwości wyższych niż maksymalna częstotliwość magistrali pamięci.

7.4.8. Programowalne shadery

Ostatecznym rozszerzeniem byłoby wprowadzenie programowalnych shaderów (vertex shader, fragment shader), co przybliżyłoby architekturę do współczesnych GPU. Wymagałoby to zaprojektowania procesora wektorowego w sprzęcie oraz backendu kompilującego shadery do kodów maszynowych.

7.4.9. Antyaliasing

Dodanie technik antyaliasingu (np. MSAA - Multi-Sample Anti-Aliasing) w celu poprawy jakości renderowanych obrazów poprzez redukcję ząbkowania krawędzi.

7.5. Wkład edukacyjny

Projekt PixelForge ma istotną wartość edukacyjną, demonstrując:

- pełny cykl projektowania systemu cyfrowego — od specyfikacji, przez implementację w HDL, syntezę, aż po weryfikację na sprzęcie,
- zastosowanie języka wysokiego poziomu (Python/Amaranth) do opisu sprzętu,
- integrację logiki programowalnej (FPGA) z procesorem (SoC),
- praktyczną implementację algorytmów grafiki komputerowej na poziomie sprzętowym.

Kod projektu został udostępniony jako open-source, co umożliwia innym studentom i inżynierom naukę oraz dalszy rozwój akceleratora.

7.6. Słowa końcowe

Niniejsza praca dowodzi, że implementacja akceleratora graficznego w układzie FPGA jest możliwa i praktyczna, nawet przy użyciu stosunkowo niedrogiej platformy deweloperskiej. Zastosowanie nowoczesnego języka opisu sprzętu (Amaranth HDL) oraz podejścia modułowego pozwoliło na efektywne zarządzanie złożonością projektu i osiągnięcie funkcjonalnego, wydajnego systemu graficznego.

Akcelerator PixelForge stanowi przykład, jak elastyczność FPGA może być wykorzystana do budowy wyspecjalizowanych systemów obliczeniowych, oferując pełną kontrolę nad architekturą i optymalizacją zasobów — co jest szczególnie istotne w zastosowaniach wbudowanych i edukacyjnych.

Bibliografia

[1] Khronos Group, *OpenGL ES 1.1 Specification*, https://registry.khronos.org/OpenGL/specs/es/1.1/es_full_spec_1.1.pdf, 2008.

[2] Amaranth HDL Documentation, https://amaranth-lang.org/docs/amaranth/latest/, 2026.

[3] Intel Corporation, *Cyclone V Device Handbook*, https://www.intel.com/programmable/technical-pdfs/683375.pdf, 2023.

[4] Terasic Technologies, *DE1-SoC User Manual*, http://www.ee.ic.ac.uk/pcheung/teaching/ee2_digital/de1-soc_user_manual.pdf, 2013.

[5] OpenCores, *Wishbone B4 Specification*, https://opencores.org/cdn/downloads/wbspec_b3.pdf, 2010.

[6] Intel Corporation, *Avalon Interface Specifications*, https://docs.altera.com/r/docs/683091/22.3/avalon-interface-specifications/introduction-to-the-avalon-interface-specifications, 2020.

[7] Phong, B. Illumination for computer generated pictures. *Seminal Graphics: Pioneering Efforts That Shaped The Field, Volume 1*. pp. 95-101 (1998), https://doi.org/10.1145/280811.280980.

[8] Eric Nadeau, Skyler Whorton, *FPGA-Based Graphics Acceleration* https://digital.wpi.edu/pdfviewer/ws859h34g, 2010.

[9] Kibret Abebe *Design of 3D Graphics Accelerator Code for FPGA* https://www.scribd.com/document/489506912/Kibret-Abebe-1-pdf, 2011.