Projekt SDUP

Generowanie grafiki trójwymiarowej z wykorzystaniem układu SoC Zynq

Wojciech Mrzygłód, Jan Wołowiec

Repozytorium git projektu

Repozytorium znajduje się pod adresem https://github.com/NVi5/ZynqSDUP

1 Cel, Algorytm(y), wzory

Celem projektu jest zaimplementowanie w FPGA prostego pipeline graficznego. Projektowany system będzie otrzymywał z procesora dane w postaci tablicy wierzchołków trójkątów. Po otrzymaniu danych mogą zostać na nich wykonane operacje przekształcenia w przestrzeni 3D, na przykład:

- Symetria osiowa
- Translacja
- Skalowanie
- Rotacja
- Pochylenie

Drugim krokiem będzie rasteryzacja otrzymanej w wyniku przekształceń geometrii do postaci obrazu 800x600. Otrzymany obraz zostanie zapisany w buforze ramki znajdującym się w pamięci DDR. Otrzymany obraz zostanie przesłany przez Ethernet oraz wyświetlony w przeglądarce internetowej (z powodu braku interfejsu graficznego w wykorzystywanym sprzęcie).

1.1 Wzory

Mnożenie wektora przez macierz wykorzystywane do operacji przekształceń:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \cdot \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{bmatrix} \tag{1}$$

Na przykład w przypadku translacji wektora (x, y, z) o wektor (a, b, c) przybiera ono postać:

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} \tag{2}$$

Edge functions wykorzystywane w rasteryzjacji, pozwalające wyznaczyć, czy dany punkt znajduje się wewnątrz trójkąta:

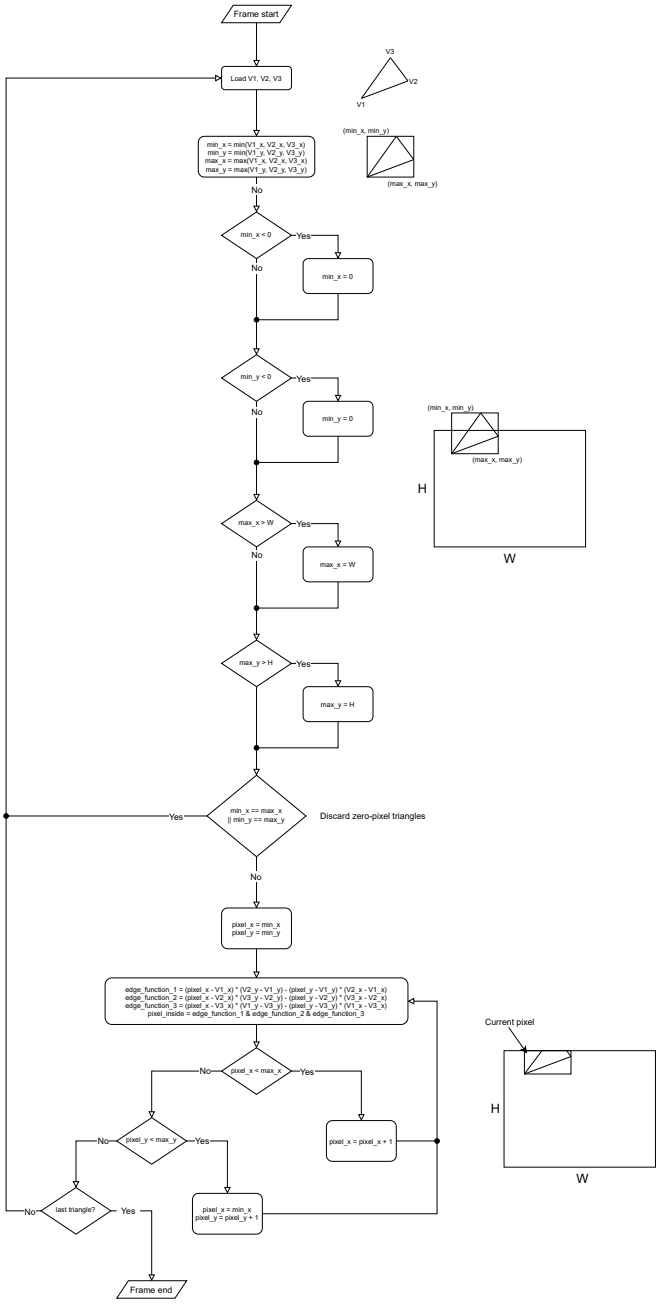
$$\begin{aligned} E_{01}(P) &= (P.x - V0.x) \cdot (V1.y - V0.y) - (P.y - V0.y) \cdot (V1.x - V0.x) \\ E_{12}(P) &= (P.x - V1.x) \cdot (V2.y - V1.y) - (P.y - V1.y) \cdot (V2.x - V1.x) \\ E_{20}(P) &= (P.x - V2.x) \cdot (V0.y - V2.y) - (P.y - V2.y) \cdot (V0.x - V2.x) \end{aligned} \tag{3}$$

1.2 Używane formaty danych

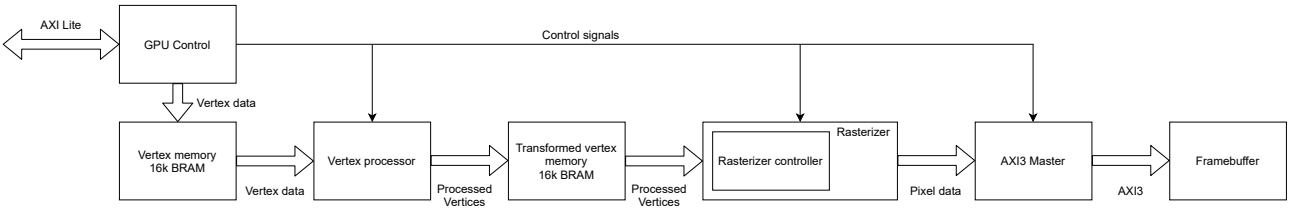
- Lista wierzchołków wyświetlanej grafiki - fixed-point Q11.7 ze znakiem
- Macierz przekształcenia - fixed-point Q11.7 ze znakiem
- Wierzchołki grafiki po transformacji - 11-bitowa liczba całkowita ze znakiem
- Kolory pikseli - 8-bit RGB232

2 Diagram algorytmu

2.1 Algorytm rasteryzacji trójkątów



3 Schemat blokowy implementowanego rozwiązania.



4 Behawioralny model algorytmu

Behawioralny model algorytmu składa się z dwóch plików. Jeden z nich symuluje moduł Vertex processor (/src/vertex_processor_behav.sv) a drugi moduł Rasterizer (/src/rasterize_behav.sv). W modelu behawioralnym pominięte zostały pamięci, kontroler oraz magistrala AXI. Wymienione moduły zostały zainstancjonowane w module testowym (/sim/behav_tb.sv). Symulacja ładuje z pliku tekstowego listę wierzchołków trójkątów składających się na wyświetlany obraz oraz dostarcza macierz przekształcenia do modułu Vertex processor, przekształcone wierzchołki wyświetlanej geometrii są rasteryzowane przez moduł Rasterizer a następnie zapisywane jako obraz w formacie BMP RGB232 pod nazwą behav_tb_image_0.bmp w katalogu /sim/results.

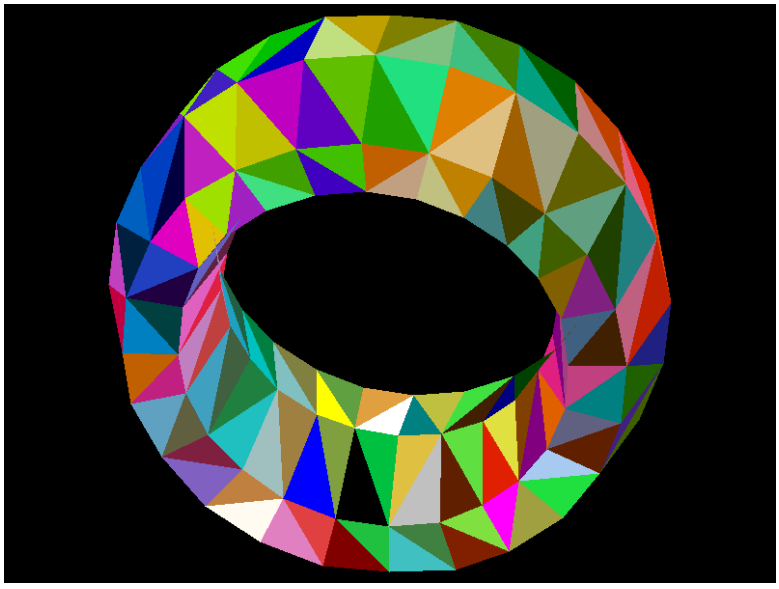
4.1 Symulacja modelu behawioralnego

W celu porównywania wyników między symulacjami każda z symulacji będzie korzystała z identycznego zestawu danych wejściowych. Wyświetlanym modelem będzie torus wygenerowany w programie Blender. Przekształceniem wyświetlanego obrazu będzie obrót względem osi OX o $\frac{\pi}{2}$ i osi OY o $\frac{\pi}{4}$. Macierz tego przekształcenia w formacie Q11.7 wygląda następująco

$$T = \begin{bmatrix} 83 & -48 & -83 & 0 \\ 34 & 118 & -34 & 0 \\ 90 & 0 & 90 & 0 \\ 0 & 0 & 0 & 128 \end{bmatrix}$$

4.2 Wynik symulacji modelu behawioralnego

Wynikiem symulacji modelu behawioralnego jest plik behav_tb_image_0.bmp zapisany w katalogu /sim/results.



Rysunek 1: Wynik symulacji modelu behawioralnego.

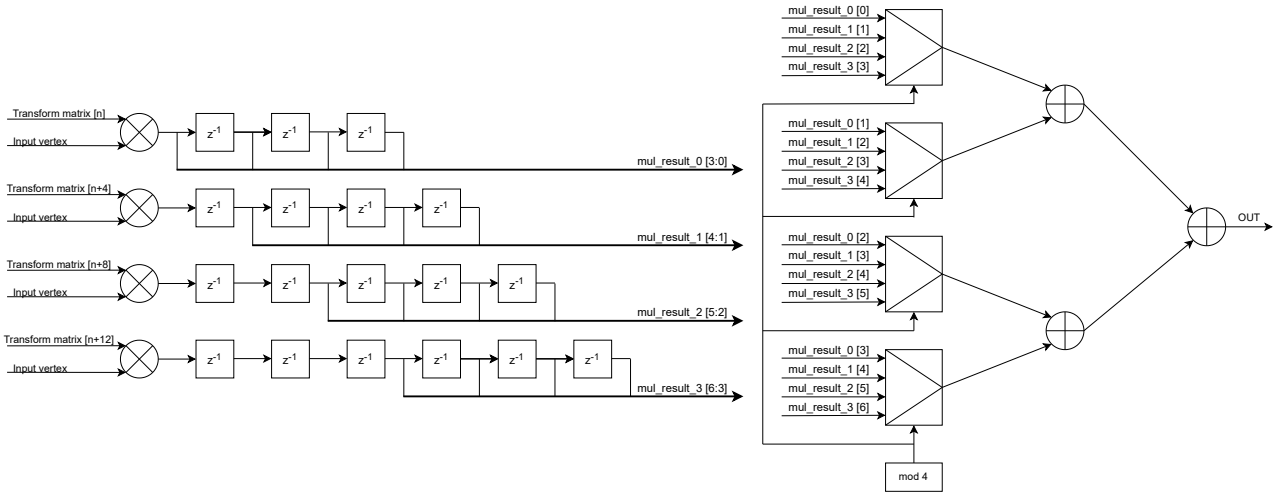
5 Implementacja modułów w postaci syntezowalnej

5.1 Wejścia/Wyjścia i dostęp do pamięci

- Lista wierzchołków do wyświetlenia jest przechowywana w pamięci Block RAM układu FPGA. Pamięć ma wymiary 16384x18.
- Wygenerowany obraz wyjściowy przechowywany będzie w pamięci RAM DDR3 układu Zynq
- Wejściowe sygnały kontrolne:
  - Elementy macierzy przekształcenia.
  - Ilość danych do wyświetlenia,
  - Sygnał start,
  - Adres bufora ramki w pamięci układu RAM DDR3 Zynq.
- Sygnały wyjściowe:
  - Adres danego piksela w buforze,
  - Kolor piksela,
  - Sygnał valid oznaczający wygenerowanie poprawnych danych gotowych do zapisania do bufora ramki,
  - sygnał frame_end oznaczający zakończenie generowania jednej klatki obrazu.

5.2 Zaimplementowane moduły

- Vertex processor (vertex_processor_rtl.sv) - moduł wykonujący transformację wierzchołków wyświetlanej geometrii. Realizuje on działanie (1) przedstawione w rozdziale 1.1. Został on zaimplementowany bazując na schemacie blokowym z Rys.2. Układ jest pipelinem o opóźnieniu 8 cykli zegara i wykorzystuje 4 bloki DSP48 pracujące w konfiguracji mnożarek 18x18.



Rysunek 2: Schemat blokowy modułu Vertex processor.

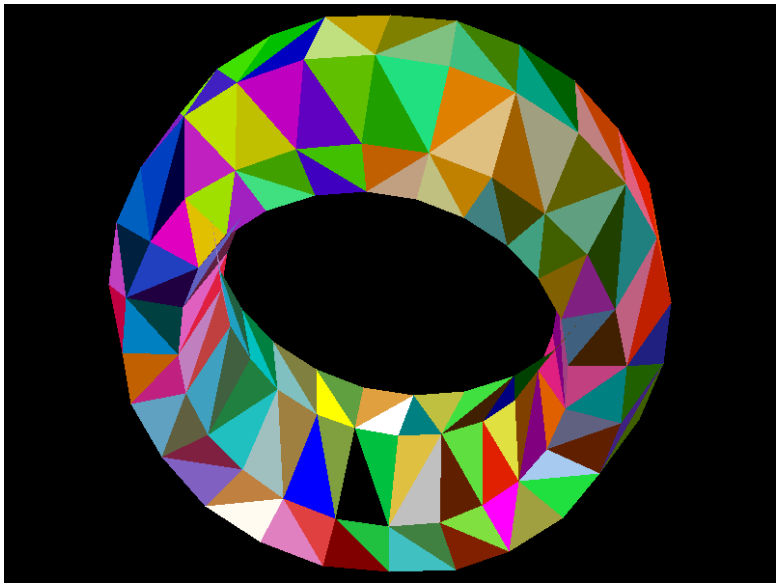
- Rasterizer control (/src/rasterizer_control.v) - moduł implementujący algorytm opisany Rys. 2.1 w postaci maszyny stanów. Na wejściu otrzymuje kolejne wierzchołki trójkątów z pamięci przechowującej dane po transformacji przez moduł Vertex processor. Po załadowaniu wierzchołków trójkąta do zrasteryzowania wyznaczane są minima i maksima w osiach X i Y obszaru wyświetlania. W kolejnym kroku moduł sprawdza czy współrzędne wierzchołków nie znajdują się poza obszarem wyświetlania. Jeżeli tak to są one odpowiednio zmieniane, tak aby ich maksima i minima mieściły się w obszarze wyświetlania. Następnie sprawdzane jest to czy trójkąt będzie miał niezerową wysokość lub szerokość. Jeżeli tak jest,

to ładowany jest kolejny trójkąt, jeżeli nie, to następuje rasteryzacja. Po zakończeniu rysowania danego trójkąta algorytm ładuje kolejny, lub kończy swoje działanie jeżeli narysowany trójkąt był ostatnim na liście.

- Rasterizer (/src/rasterizer.sv) - moduł zamieniający trzy wejściowe wierzchołki trójkątna na piksele odpowiednich kolorów. Zawiera w sobie trzy moduły które obliczają tzw. edge function opisane równaniem (3) (/src/edgeFunction.sv) - funkcje które sprawdzają po której stronie prostej stworzonej z kolejnych par wierzchołków. Jeżeli wszystkie z tych funkcji zwrócą wartość 1, to znaczy że rysowany piksel jest wewnątrz obecnie rysowanego trójkąta. Moduły te są połączone równolegle, a każdy z nich jest pipelinem o 4 cyklach opóźnienia. Każda z tych funkcji wykonuje 2 mnożenia, a więc zużycie bloków DSP48 przez ten moduł to $3 \cdot 2 = 6$.
- GPU top (/src/GPU_top.sv) - moduł instancjonujący wszystkie powyższe. Dodatkowo zawiera instancje pamięci wierzchołków, przelicza ich adresy odczytu i zapisu, generuje odpowiednie sygnały statusowe oraz kontroluje wszystkie moduły w zależności od otrzymanych sygnałów kontrolnych.

5.3 Symulacja modułu syntezowalnego

W celu przetestowania modułu w wesji syntezowalnej został stworzony testbench GPU_top_tb.sv instancjonujący moduł GPU top. Na początku testu wierzchołki trójkątów do wyświetlenia ładowane są z pliku tekstowego. następnie ustawiana jest odpowiednia ilość wierzchołków oraz sygnał start na 1 cykl zegara. Kiedy moduł GPU top ustawia sygnał valid i draw, to wartość koloru piksela wpisywana jest do zmiennej framebuffer pod odpowiednim adresem. Po wygenerowaniu jednej klatki obrazu zawartość zmiennej framebuffer jest zapisywana jako obraz BMP w katalogu /sim/results.



Rysunek 3: Wynik symulacji modelu syntezowalnego.

6 Dodanie magistral komunikacyjnych

6.1 AXI-Lite Slave

Do zapisywania danych do pamięci wierzchołków (Vertex memory), oraz do sterowania modułu i odczytywania jego statusu zostanie wykorzystana magistarala AXI-Lite Slave. Kod modułu wygenerowany został przez generator peryferiów AXI w Vivado. Wygenerowany kod należało zmodyfikować tak aby umożliwił zapis pamięci BRAM.

6.1.1 Mapa pamięci modułu

Mapa pamięci			
Nazwa obszaru	Adres	Rozmiar	Typ
transform_matrix	GPU_BASEADDR + 0x00	0x40	Reg
vertex_count	GPU_BASEADDR + 0x40	0x04	Reg
DDR_address	GPU_BASEADDR + 0x44	0x04	Reg
control	GPU_BASEADDR + 0x48	0x04	Reg
status	GPU_BASEADDR + 0x4c	0x04	Reg
Vertex_memory	GPU_BASEADDR + 0x50	0x10000	BRAM

6.2 AXI Master

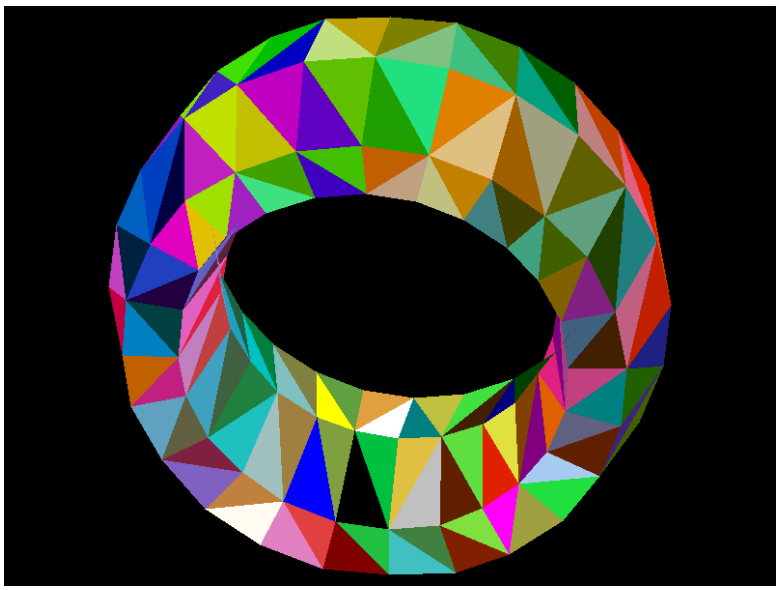
Dane wyjściowe będą zapisywane do pamięci DDR układu Zynq przez magistralę AXI3. W tym celu został stworzony moduł mastera tej magistrali podłączany bezpośrednio do wejścia Slave modułu Processing_system_7. Moduł ten do zapisu wykorzystuje transfery typu burst, w celu jak najlepszego wykorzystania dostępnego pasma transmisji pomiędzy logiką programowalną a pamięcią procesora.

7 Symulacja Microblaze

Wykorzystanie procesora Microblaze w symulacji wymagało stworzenia osobnego diagramu blokowego main_ub, zawierającego procesor oraz pozostałe wykorzystane moduły. Jedynym wyjątkiem jest blok AXI3 master, który został pominięty z racji problemów ze współpracą procesora Microblaze i pełnej magistrali AXI3 oraz zbyt małej ilości dostępnej pamięci Block RAM, aby umieścić bufor obrazu.

Aby umożliwić przetestowanie dodanego bloku AXI-Lite oraz samo w sobie działanie tej symulacji bez zewnętrznych sygnałów sterujących został napisany prosty program wyłącznie na potrzeby symulacji w sdk_ub. Program ten za pomocą magistrali AXI-Lite ustawia macierz przekształcenia oraz zapisuje sam model z listą wierzchołków do pamięci. Po zakończeniu konfiguracji ustawiany jest sygnał start.

Stworzony testbench (/src/ub_design_tb.sv) zawiera instancje dodanego schematu blokowego, zapewniając mu sygnały clock i reset, poza tym faktem działanie tego testbenchu podobne jest do testbenchu GPU_top_tb.sv. Wartość koloru pikseli zapisywana jest w zmiennej framebuffer, a po zakończeniu generowania klatki zapisywany jest on jako obraz BMP w katalogu /sim/results.



Rysunek 4: Wynik symulacji modelu z procesorem Microblaze.

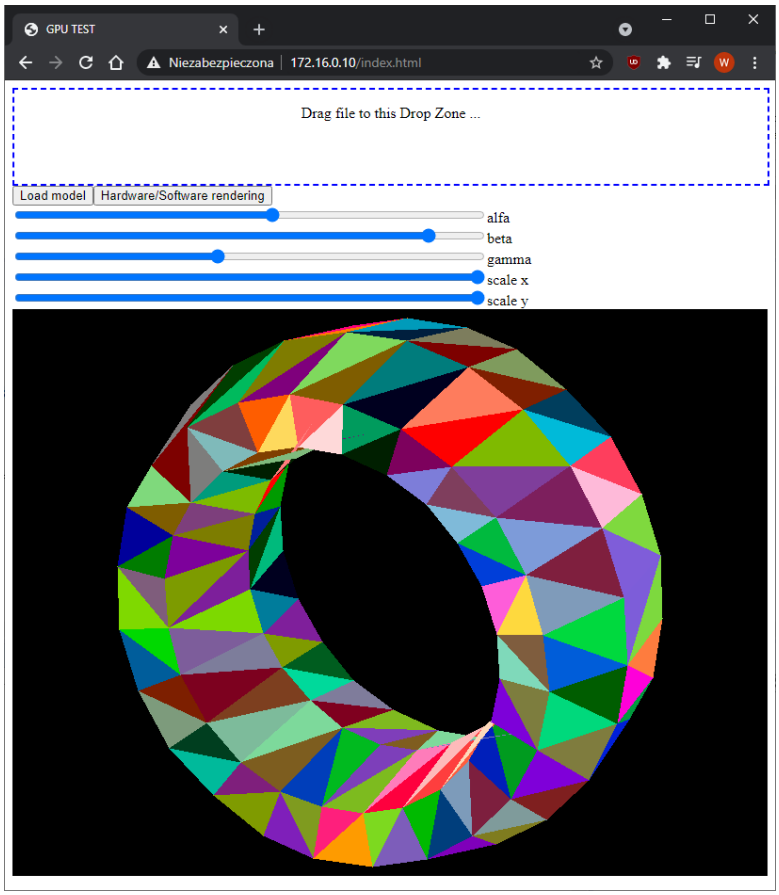
8 Driver i program na procesor ARM

Program na procesor ARM składa się z kilku elementów:

- GPU driver - prosty sterownik zaimplementowanego modułu w logice programowalnej. Zawiera definicje rejestrów i funkcje wymagane do jego obsługi.
- OpenGL Mathematics (http://glm.g-truc.net/) - biblioteka wykorzystywana do tworzenia macierzy transformacji.
- Serwer HTTP - dostarcza interfejs graficzny pod danym adresem IP.
- Serwer protokołu websocket - umożliwia transmisję obrazu i wyświetlanie go w przeglądarce internetowej, oraz transmisję plików zawierających wyświetlane modele 3d.
- Software renderer - programowa implementacja algorytmów zaimplementowanych w FPGA. Powstała w celu porównania szybkości generowania grafiki 3d programowo i sprzętowo.

8.1 Działanie programu

Po uruchomieniu program uruchamia serwery HTTP i websocket pod adresem 172.16.0.10 na portach odpowiednio 80 i 1000. Następnie do pamięci Vertex memory ładowany jest model 3d, ten sam który był wykorzystywany w symulacjach. Po otworzeniu strony pod adresem http://172.16.0.10/index.html zostaje załadowany interfejs użytkownika oraz otwierane jest połączenie protokołu websocket. Na stronie można zobaczyć obraz generowany przez zaimplementowany moduł.



Rysunek 5: Interfejs graficzny.

Za pomocą suwaków znajdujących się na stronie można obracać i skalować wyświetlany model. Po zmianie wartości jednego z nich do procesora wysyłana jest ramka zawierająca nowe parametry wyświetlania. Po

otrzymaniu takiej ramki program uruchomiony na procesorze korzystając z funkcji biblioteki OpenGL Mathematics tworzy nową macierz transformacji a następnie ustawia odpowiednie rejestry modułu w FPGA.

```
glm::mat4 Model = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, 0.0f));
Model = glm::scale(Model, glm::vec3(scale_x,scale_y,1.0f));
Model = glm::rotate(Model, alpha, glm::vec3(1.0f, 0.0f, 0.0f));
Model = glm::rotate(Model, beta, glm::vec3(0.0f, 1.0f, 0.0f));
Model = glm::rotate(Model, gamm, glm::vec3(0.0f, 0.0f, 1.0f));
glm::mat4 MVP = Model;
glm_mat4_to_fpga(MVP, GPU_TRANSFORM_MATRIX);
```

Listing 1: Przykładowy kod generujący macierz transformacji i ustawiający odpowiednie rejestry w FPGA.

8.2 Przełączanie pomiędzy implementacją sprzętową i programową

Przycisk "Hardware/Software rendering"przełącza pomiędzy programowym i sprzętowym generowaniem grafiki co może zostać użyte do oceny wydajności implementacji sprzętowej algorytmu w porównaniu do programowej.

8.3 Ładowanie plików *.obj

Program na procesor ARM może również otwierać pliki w formacie Wavefront OBJ. Po przeciągnięciu takiego pliku do niebieskiej ramki znajdującej się na stronie i naciśnięciu przycisku Load do procesora jest przesyłany wybrany plik, następnie jest on parsowany przez odpowiedni moduł programu. Ostatecznie wierzchołki geometrii opisane w pliku ładowane są do pamięci Vertex memory, co powoduje zmianę wyświetlanego obrazu.

8.4 Podwójne buforowanie

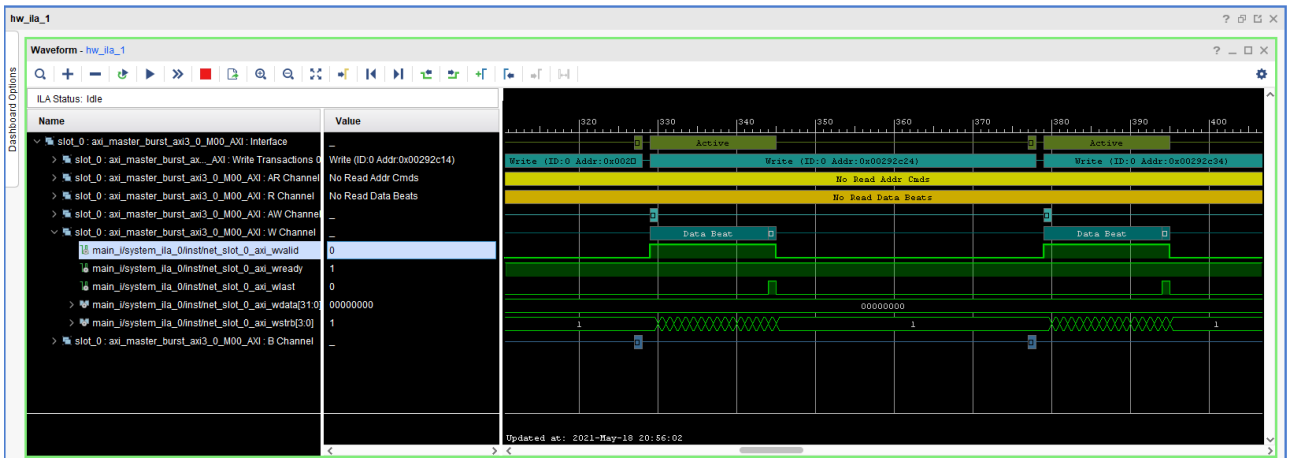
W celu wykorzystania możliwości równoległego działania procesora ARM i modułu działającego w logice programowalnej zaimplementowana została technika podwójnego buforowania. Polega ona na tym, że w pamięci istnieją dwa bufory obrazu o takim samym rozmiarze. Kiedy z jednego bufora dane są wysyłane przez Ethernet, moduł w FPGA do drugiego bufora zapisuje kolejną klatkę obrazu. Po zakończeniu transmisji pierwszej klatki adresy buforów są zamieniane i cały proces jest powtarzany. Pozwala to na zmniejszenie narzutu czasowego powodowanego przez transmisję obrazu przez sieć.

8.5 Porównanie wydajności

Typ implementacji	Wydajność [FPS]
Programowa	0.2
Sprzętowa	9
Sprzętowa + podwójne buforowanie	16
Symulacja	59
Symulacja z uwzględnieniem opóźnienia kontrolera pamięci	17

9 Problemy i ograniczenia sprzętu

- Ograniczona przepustowość pomiędzy logiką programowalną a pamięcią DDR - jak widać w tabeli porównującej wydajność implementacji wyniki uzyskane na sprzęcie różnią się od tych uzyskanych w symulacji. Największym problemem w tym przypadku jest zbyt niska przepustowość kontrolera pamięci. Po 16 cyklach zapisu potwierdzenie od kontrolera pamięci opóźnione jest o kilkadziesiąt cykli magistrali, co pokazują przebiegi zebrane za pomocą ILA (Integrated Logic Analyzer).

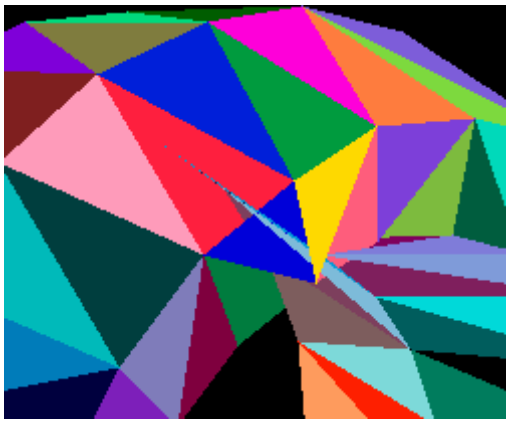


Rysunek 6: Opóźnienie potwierdzenia zapisu do pamięci na magistrali AXI.

Jak widać na powyższym rysunku na zapisanie 16 pikseli w pamięci potrzeba 51 cykli magistrali. Uwzględniając opóźnienie przy zapisie w symulacji uzyskano maksymalnie 17 klatek na sekundę. Wcześniej w symulacji było przyjęte założenie, że magistrala AXI może być wykorzystywana przez 100% czasu, stąd też dużo wyższe wyniki. Po uwzględnieniu tej zmiany wyniki pokrywają się z tymi uzyskanymi na sprzęcie.

- Ograniczona przepustowość interfejsu sieciowego - maksymalna częstotliwość odświeżania obrazu jaką można uzyskać korzystając z 100Mb/s Ethernetu to około 26 FPS przy założonej rozdzielczości i formacie kolorów. W rzeczywistości jest ona jeszcze niższa z powodu narzutu protokołów TCP i websocket.

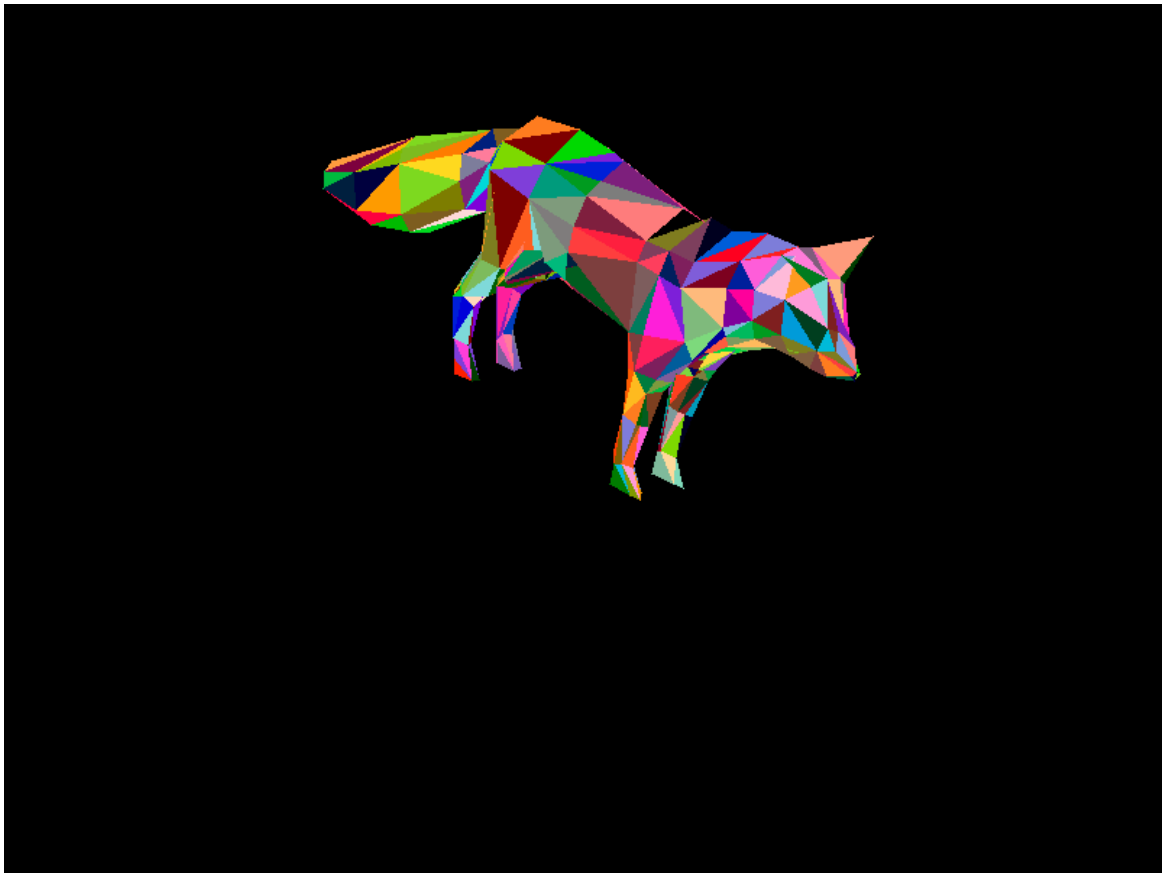
- Brak bufora Z - Trójkąty rasteryzowane są w takiej kolejności w jakiej zapisane są w pamięci. Przy wyświetlaniu nie jest brana pod uwagę współrzędna Z wierzchołków, co powoduje nadpisywanie trójkątów znajdujących się bliżej przez te znajdujące się dalej, co nie powinno mieć miejsca. Jednak implementacja bufora Z wymaga dużej ilości pamięci o krótkim czasie dostępu co jest niemożliwe do wykonania na użytej platformie sprzętowej.



Rysunek 7: Błąd wyświetlania spowodowany brakiem bufora Z.

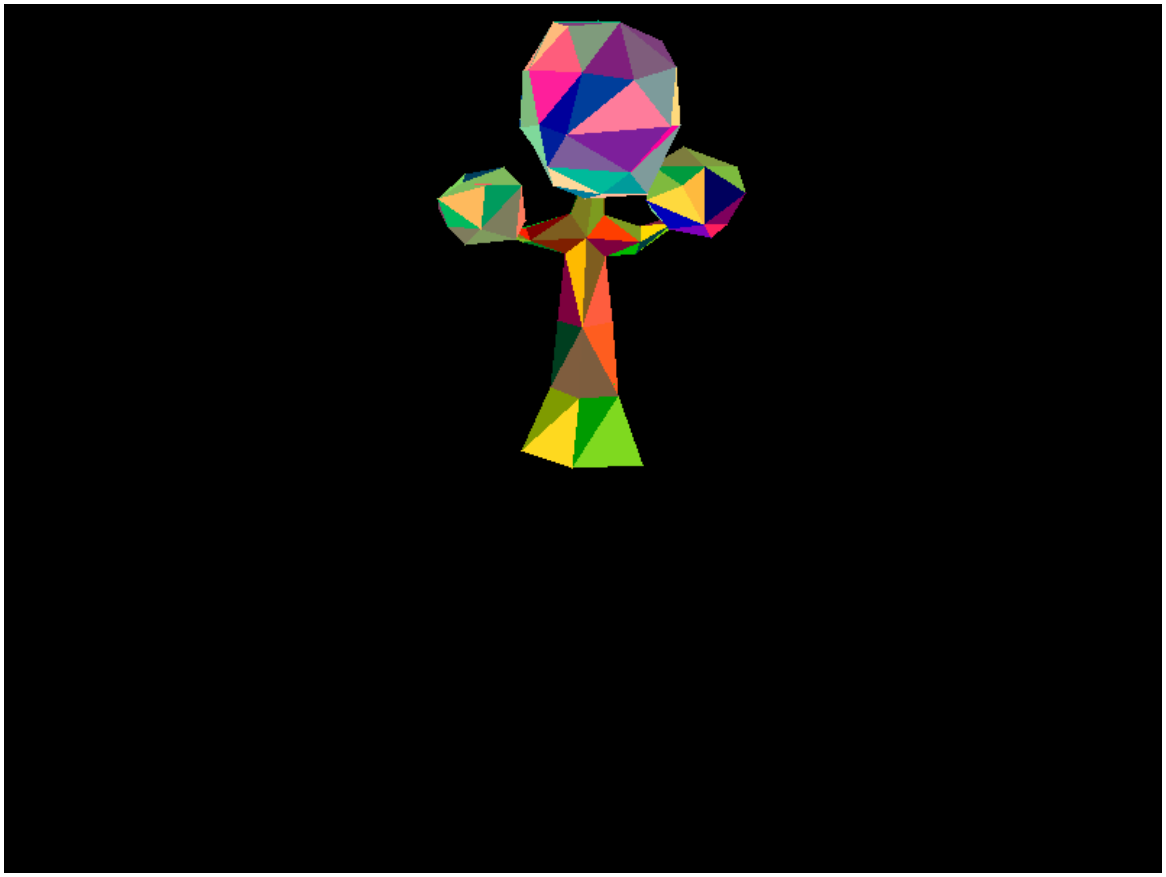
- Brak metody na obliczanie kolorów trójkątów. W obecnej implementacji ich kolory są kolejnymi ośmiobitowymi liczbami.

10 Wyświetlanie przykładowych modeli

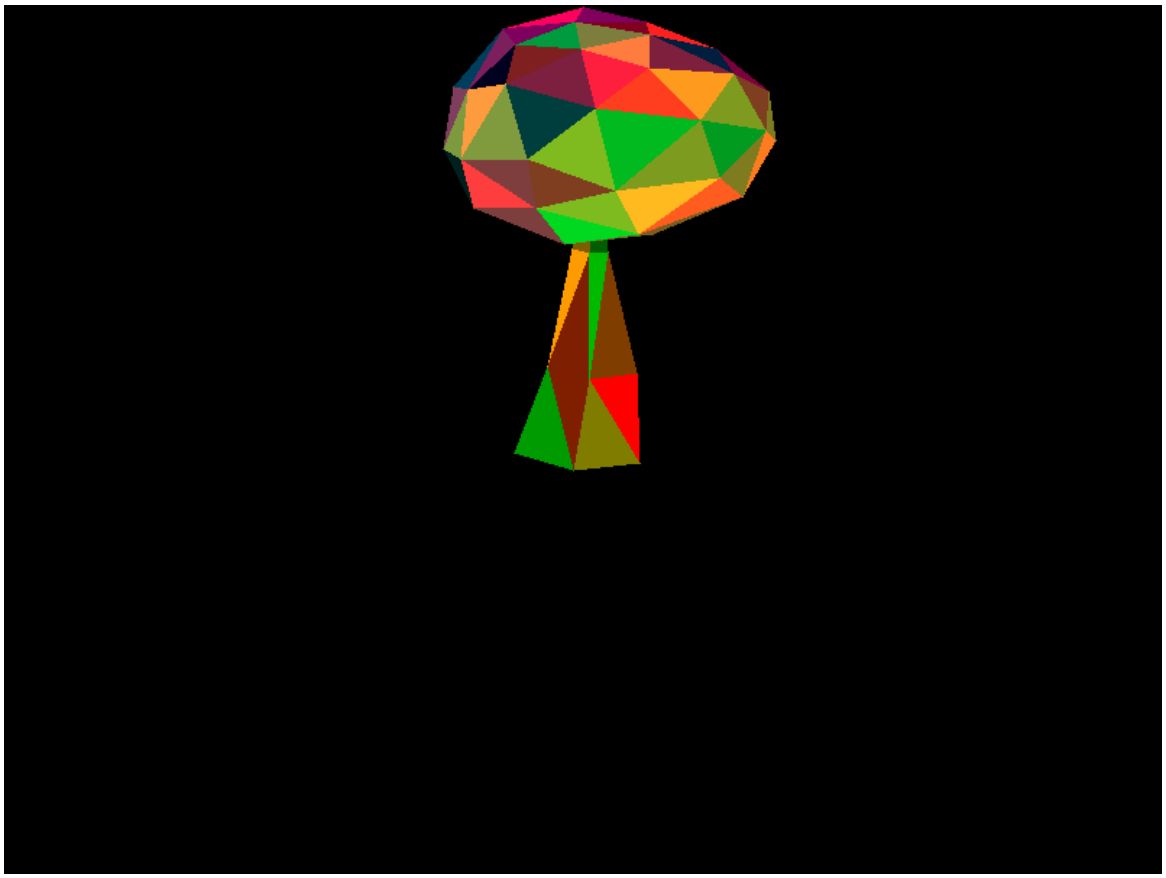


Rysunek 8: Model: /models/FOXU.obj

Rysunek 9: Model: /models/SDUP.obj



Rysunek 10: Model: /models/Low_Poly_Tree_003.obj



Rysunek 11: Model: /models/Low_Poly_Tree_001.obj