

STRESZCZENIE

Cieniowanie odroczone jest techniką grafiki 3D czasu rzeczywistego popularną wśród twórców gier komputerowych pozwalającą na obsługę wielu światła na scenie bez znaczącego spadku wydajności.

W niniejszej pracy zaprojektowano i zaimplementowano silnik renderujący używając języka C i biblioteki graficznej Vulkan. Opisano elementy silnika renderującego oraz nisko i wysokopoziomowe techniki graficzne używane w nowoczesnych grach 3D z naciskiem na renderowanie odroczone. Opisano architekturę silnika i szczegóły implementacji. Wyrenderowano przykładową scenę i zbadano wydajność użytych technik graficznych.

Słowa kluczowe: silnik renderujący, renderowanie odroczone, renderowanie bez dowiązań, renderowanie pośrednie, Vulkan, GLFW.

Dziedzina nauki i techniki według OECD: 1.2 Nauki o komputerach i informatyka.

ABSTRACT

// TODO

SPIS TREŚCI

Streszczenie	1
Abstract	2
Spis treści	3
Wykaz ważniejszych oznaczeń i skrótów	4
1. Wstęp	5
1.1. Cel pracy	5
1.2. Zakres pracy	6
1.3. Struktura pracy	6
2. Wprowadzenie do dziedziny	7
2.1. Podstawowe pojęcia	7
2.2. Potok graficzny	7
2.3. Potok zasobów	7
2.4. Vulkan	7
2.4.1. Synchronizacja	7
2.4.1.1 Bariery potoku	7
2.4.1.2 Semafore	8
2.4.1.3 Ogrodzenia	8
2.4.2. Deskryptory	8
2.5. Mapowanie tekstur	8
2.6. Oświetlenie	8
2.7. Graf sceny	8
2.8. Graf renderowania	8
3. Architektura i implementacja	9
3.1. Użyte narzędzia	9
3.1.1. Proces budowania	9
3.1.2. Biblioteki zewnętrzne	10
3.2. Architektura	11
3.3. Moduły	11
4. Badania	12
5. Podsumowanie	13
Wykaz literatury	14
Spis rysunków	15
Spis tablic	16

WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW

Skróty:

- API - Application Programming Interface, interfejs programistyczny aplikacji;
- CPU - Central Processing Unit, procesor;
- GPU - Graphics Processing Unit, procesor graficzny.

1. WSTĘP

Podręcznik [1] definiuje grafikę komputerową jako dziedzinę interdyscyplinarną zajmującą się komunikacją wizualną za pomocą wyświetlacza komputera i jego urządzeń wejścia-wyjścia.

Rozwój teoretyczny grafiki komputerowej jest zdominowany przez doroczną konferencję SIGGRAPH [2], podczas której prezentacje i dyskusje akademickie są przeplecione z targami branżowymi. Rozwój grafiki komputerowej jest w znacznej mierze napędzany wymaganiami stawianymi przez przemysł rozrywkowy. Przykładem jest dobrze znana seria kursów przeznaczona dla twórców gier komputerowych obejmująca najnowsze prace i postępy w technikach renderowania czasu rzeczywistego używanych w silnikach graficznych rozwijanych przez producentów gier komputerowych [3].

Renderowanie to proces konwersji pewnych prymitywów na obraz przeznaczony do wyświetlania na ekranie. Wyświetlany obraz jest nazywany klatką (ang. frame).

Renderowanie czasu rzeczywistego nakłada ograniczenie czasowe dotyczące liczby klatek na sekundę (ang. frames per second, FPS), która musi być na tyle wysoka, by dawać iluzję ciągłości ruchu. Przyjmuje się, że ograniczenie to jest spełniane poprzez zapewnienie wyświetlania minimum 30 klatek na sekundę (renderowanie trwa krócej niż $\frac{1}{30}$ sekundy). Eliminuje to kosztowne obliczeniowo techniki renderowania dające fotorealistyczne rezultaty takie jak śledzenie promieni (ang. ray tracking) i wymaga od programistów zastosowania technik aproksymacji mniej lub bardziej luźno opartych na prawach fizyki oraz użycia bibliotek graficznych wspierających akcelerację sprzętową takich jak Vulkan lub OpenGL.

Silnik renderujący, zwany też silnikiem graficznym to element aplikacji odpowiadający za renderowanie czasu rzeczywistego. Zapewnia on wysokopoziomową warstwę abstrakcji pozwalającą użytkownikowi na operowanie używając takich konceptów jak sceny, obiekty, materiały lub światła oraz ukrywając niskopoziomowe detale użytych bibliotek i technik graficznych.

Zaprojektowanie i zaimplementowanie silnika graficznego jest złożonym procesem wymagającym znajomości szerokiego wachlarza technik graficznych z całego możliwego spektrum poziomów abstrakcji, dlatego też coraz więcej twórców gier komputerowych decyduje się na licencjonowanie i użycie gotowego silnika graficznego zamiast powolnej i mozolnej pracy nad własnymi rozwiązaniami.

Jeśli jednak celem inżyniera jest poszerzenie osobistego zrozumienia grafiki komputerowej, to warto podjąć próbę stworzenia własnego silnika graficznego.

1.1. Cel pracy

Celem pracy jest zaprojektowanie i zaimplementowanie silnika graficznego, którego potok graficzny używa techniki cieniowania odroczonego.

Silnik został napisany jako biblioteka programistyczna języka C używającej skryptów Python do automatycznej generacji kodu podczas procesu budowania. Renderowanie grafiki 3D jest obsługiwane przez Vulkan API. Zasoby używane podczas renderowania są wczytywane z bazy zasobów, która jest wyjściem potoku zasobów działającego podczas procesu budowania. Działanie biblioteki jest sterowane plikiem konfiguracyjnym dostarczonym przez użytkownika i jest demonstrowane przy użyciu pliku wykonywalnego renderującego przykładową scenę używając cieniowania odroczonego.

Celem autora było zapoznanie się z teorią stojącą za elementami składającymi się na silnik graficzny i praktyczne zademonstrowanie zdobytej wiedzy.

1.2. Zakres pracy

Niniejsza praca ma charakter przeglądu. Nowoczesne silniki graficzne składają się z wielu elementów, z których każdy może być niezależnie rozwijany do dowolnie wysokiego poziomu skomplikowania, dlatego trudno je wszystkie dokładnie i wyczerpująco opisać w ramach jednej pracy.

Zakres pracy obejmuje:

- opis algorytmów i technik graficznych używanych w nowoczesnych silnikach graficznych, ze szczególnym naciskiem na Vulkan API i renderowanie odroczone,
- omówienie architektury i implementacji projektu,
- demonstrację użycia silnika graficznego do wyrenderowania przykładowej sceny,
- analizę wydajności silnika graficznego.

Stworzony silnik nie może konkurować z silnikami graficznymi profesjonalnie rozwijanymi przez duże drużyny z myślą o zastosowaniu w grach komputerowych (takimi jak otwarty Godot [4] czy komercyjny Unreal Engine [5]). Jest on jednak przystosowany do względnie łatwej, szybkiej i elastycznej modyfikacji potoku graficznego oraz wspiera mechanizmy zgłaszania informacji debugowania oferowane przez Vulkan API, co pozwala na szybki cykl prototypowania i debugowania podczas zapoznawania się z technikami graficznymi.

1.3. Struktura pracy

Praca została podzielona na pięć rozdziałów, z których każdy jest rozwinięciem rozdziału poprzedniego.

Pierwszy rozdział pracy definiuje cel, zakres i strukturę pracy.

Drugi rozdział zawiera wprowadzenie do wybranych części dziedziny grafiki komputerowej użytych podczas późniejszej implementacji silnika renderującego.

W trzecim rozdziale opisano architekturę silnika i szczegóły implementacji poszczególnych jego modułów.

W trzecim rozdziale opisano specyficzny potok graficzny używający cieniowania odroczonego do realizacji modelu oświetlenia opartego o renderowanie bazujące na fizyce.

W czwartym rozdziale wyrenderowano przykładową scenę i zbadano wydajność silnika.

Ostatni rozdział zawiera podsumowanie oraz opis przewidywanych kierunków przyszłego rozwoju silnika.

2. WPROWADZENIE DO DZIEDZINY

W tej sekcji przybliżono serię pojęć oraz technik związanych z grafiką komputerową, których zrozumienie jest wymagane przed rozpoczęciem implementacji silnika graficznego.

2.1. Podstawowe pojęcia

// TODO

2.2. Potok graficzny

// TODO

2.3. Potok zasobów

// TODO

2.4. Vulkan

// TODO

2.4.1. Synchronizacja

// TODO

2.4.1.1 Bariery potoku

Bariera potoku to prymityw synchronizacji definiowany poleceniem `vkCmdPipelineBarrier()` pozwalający na zdefiniowanie zależności wykonania oraz zależności pamięci pomiędzy poleceniami przed i po barierze.

Zależność wykonania to gwarancja, że praca pewnych *źródłowych etapów potoku* (określonych używając `VkPipelineStageFlags`) dla wcześniejszego zestawu poleceń została zakończona przed rozpoczęciem wykonywania pewnych *docelowych etapów potoku* dla późniejszego etapu poleceń.

Przykładowo zależność wykonania pomiędzy etapami `COLOR_ATTACHMENT_OUTPUT` i `FRAGMENT_SHADER` gwarantuje, że zapis do dołączeń kolorów został skończony przed rozpoczęciem wykonywania shadera fragmentów.

Zależność pamięci to zależność wykonania z dodatkową gwarancją, że rezultat zapisów wyspecyfikowanych przez pewien *źródłowy zakres dostępów* (określony używając `VkAccessFlags`) wygenerowanych przez wcześniejszy zestaw poleceń jest udostępniony późniejszemu zestawowi poleceń dla pewnego *docelowego zakresu dostępów*.

Przykładowo zależność pamięci pomiędzy etapami `COLOR_ATTACHMENT_OUTPUT` i `FRAGMENT_SHADER` z zakresami dostępów `COLOR_ATTACHMENT_WRITE` i `SHADER_READ` gwarantuje, że zapis do dołączeń kolorów zostanie skończony i będzie mógł być odczytany przez shader fragmentów.

Istnieją trzy rodzaje barier w zależności od rodzaju pamięci zarządzanego przez zależności pamięci:

- bariery pamięci obrazów: dla zakresu obrazu, dodatkowo pozwala na tranzykcje układu obrazu,
- bariery pamięci buforów: dla zakresu bufora,
- globalne bariery pamięci: dla wszystkich istniejących obiektów,

// TODO

2.4.1.2 Semafor

Semafor to obiekty `VkSemaphore` pozwalające na synchronizację wykonywania buforów poleceń w tej samej lub pomiędzy kolejkami. GPU może sygnalizować semafor po zakończeniu wykonywania poleceń oraz może czekać na sygnalizację semafora przed rozpoczęciem wykonywania następnego bufora poleceń.

Przykładowo semafor jest używany do synchronizacji pomiędzy kolejką graficzną i kolejką prezentacji w celu zagwarantowania, że prezentowany obraz łańcucha wymiany jest używany tylko przez jedną kolejkę.

2.4.1.3 Ogrodzenia

Ogrodzenia to obiekty `VkFence` pozwalające na synchronizację poleceń wykonywanych w kolejce na GPU z CPU. Ogrodzenie może być sygnalizowane przez GPU po zakończeniu wykonywania funkcji używających GPU, CPU może zresetować ogrodzenie funkcją `vkResetFences()` lub czekać na jego sygnalizację funkcją `vkWaitForFences()` chwilowo blokując wykonywanie programu.

Przykładowo ogrodzenia są używane do zagwarantowania, że program nie używa funkcji `vkQueueSubmit()` do wysyłania buforów poleceń szybciej, niż GPU je wykonuje.

2.4.2. Deskryptory

// TODO

2.5. Mapowanie tekstur

// TODO

2.6. Oświetlenie

// TODO

2.7. Graf sceny

// TODO

2.8. Graf renderowania

// TODO

3. ARCHITEKTURA I IMPLEMENTACJA

3.1. Użyte narzędzia

Silnik została napisany jako biblioteka w języku C w standardzie C11. Budowanie biblioteki ze źródeł wymaga generacji dodatkowego kodu przy pomocy skryptów w języku Python w wersji 3.9.7.

Silnik został w całości opracowany na przy użyciu środowiska programistycznego CLion w wersji 2021.2.3.

Proces testowania i debugowania odbywał się na maszynie o następującej konfiguracji:

- OS: Kubuntu 22.04.1 LTS x86-64,
- CPU: 11th Gen Intel Core i5-11400 (2.60GHz),
- GPU: Intel UHD Graphics 730 (Rocket Lake GT1).

Podczas pracy stosowano rozproszony system kontroli wersji git. Repozytorium jest utrzymywane na serwisie GitHub.

Pliki `.clang-tidy` i `.clang-format` znajdujące się w strukturze plików projektu pozwalają na automatyczne formatowanie kodu źródłowego zgodnie ze uprzednio zdefiniowanym standardem kodowania.

Proces budowania projektu jest zautomatyzowany przy użyciu narzędzia CMake, które w przypadku języków C i C++ jest praktycznie standardem podczas rozwoju wieloplatformowych projektów.

3.1.1. Proces budowania

Proces budowania silnika jest zdefiniowany w pliku `*CMakeLists.txt*` znajdującym się w katalogu głównym projektu.

Kompilacja kodu źródłowego w języku C jest obsługiwana bezpośrednio przez CMake, które generuje standardowe pliki kompilacji (pliki Makefile w systemie Unix, projekty Microsoft Visual C++ w systemie Windows). Użyto prekompilowanych nagłówków do przyśpieszenia kompilacji bibliotek zewnętrznych.

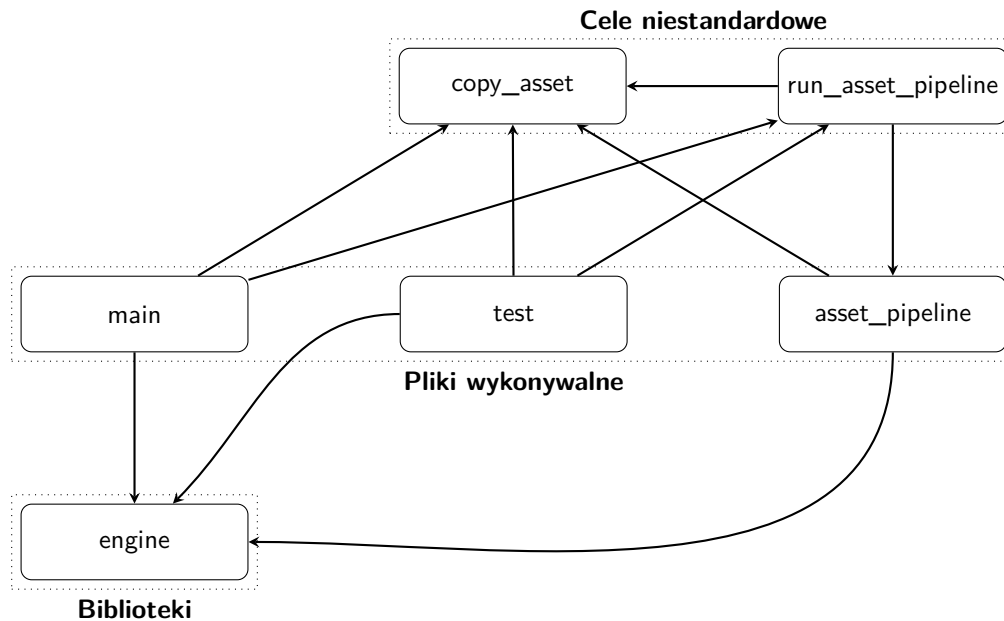
Skrypty w języku Python są obsługiwane pośrednio przez CMake, które wykrywa zainstalowany interpreter języka Python i używa go do stworzenia tzw. środowiska wirtualnego w tymczasowym katalogu `venv/` w głównym katalogu projektu. Podczas procesu budowania środowisko wirtualne jest używane do zainstalowania wymaganych zewnętrznych bibliotek w języku Python i wykonywania skryptów generatora kodu. Zaletą użycia środowiska wirtualnego w porównaniu do bezpośredniego wywoływania zainstalowanego interpretera Pythona jest izolacja zarządzania zależnościami od reszty systemu operacyjnego, co pozwala na łatwiejszą powtarzalność podczas debugowania [7].

CMake organizuje proces budowania jako graf, w którym wierzchołki to cele połączonych ze sobą zależnościami. Budowa celu wymaga wcześniejszego zbudowania wszystkich innych celów od których zależy budowany cel.

Wyróżniane są trzy rodzaje celów:

- plik wykonywalny
- biblioteka: statyczna lub dynamiczna
- cel niestandardowy: używany do uruchamiania zewnętrznych programów podczas procesu kompilacji, np. generatorów kodu

Poniższy diagram przedstawia proces budowania projektu w formie celów i ich zależności:



Rysunek 3.1: Proces budowania w formie celów i ich zależności

engine Cel budujący bibliotekę programistyczną zawierającą implementację silnika.

main Cel budujący plik wykonywalny demonstrujący użycie silnika poprzez wyrenderowanie przykładowej sceny.

test Cel budujący plik wykonywalny z testami jednostkowymi napisanymi i używanymi podczas implementowania projektu.

asset_pipeline Cel budujący plik wykonywalny służący jako narzędzie wiersza poleceń wykonujące operacje potoku zasobów.

copy_assets Niestandardowy cel kopiujący podkatalogu głównego `assets` zawierającego nieprzetworzone zasoby wejściowe do katalogu budowania.

run_asset_pipeline Niestandardowy cel realizujący potoku zasobów poprzez uruchomienie skryptu Python wielokrotnie uruchamiającego narzędzie `asset_pipeline` na zasobach wejściowych.

// TODO więcej o celach

3.1.2. Biblioteki zewnętrzne

Projekt używa następujących zewnętrznych bibliotek programistycznych:

- *Vulkan SDK 1.3.211.0*:
 - pliki nagłówkowe dla Vulkan,
 - *shaderc*: kompilacja shaderów z kodu źródłowego GLSL do kodu bajtowego SPIR,
 - *SPIRV-Reflect*: mechanizm refleksji dla kodu bajtowego SPIR-V,
- *glfw 3.4*: międzyplatformowa obsługa tworzenia okien, obsługa wejścia klawiatury i myszy,
- *sqlite 3.35.5*: relacyjna baza danych SQL,

- *uthash 2.3.0*: proste struktury danych (tablica dynamiczna, lista dwukierunkowa, tablica mieszająca),
- *xxHash 0.8.1*: niekryptograficzny algorytm mieszający,
- *glTF 1.11*: wczytywanie plików w formacie glTF,
- *glm 0.8.5*: biblioteka matematyczna,
- biblioteka standardowa C,
- API systemu operacyjnego: pliki nagłówkowe POSIX albo WinAPI,
- biblioteka standardowa Python,
- *libclang 12.0.0*: analizowanie kodu C w skryptach Python.

Dodatkowo biblioteka zbudowana w konfiguracji *Debug* statycznie linkuje biblioteki *ASan* (AddressSanitizer) i *UBSan* (UndefinedBehaviorSanitizer) wykrywające szeroką klasę błędów dotyczących niewłaściwego użycia pamięci i niezdefiniowanych zachowań. Błędy te w języku C są nieoczywiste i trudne do wykrycia przez programistę. Podczas rozwoju projektu ASan wielokrotnie pozwolił na wykrycie i naprawienie następujących rodzajów błędów:

- wycieki pamięci,
- dereferencje zwisających wskaźników,
- dereferencja wskaźników NULL,
- dereferencja źle wyrównanych struktur,
- odczyt i zapis poza granicami tablicy.

3.2. *Architektura*

// TODO

3.3. *Moduły*

// TODO

4. BADANIA

5. PODSUMOWANIE

// TODO

WYKAZ LITERATURY

- [1] J. F. Hughes i in., *Computer Graphics: Principles and Practice*, 3 wyd. Upper Saddle River, NJ: Addison-Wesley, 2013.
- [2] *Konferencja SIGGRAPH*, 2022. adr.: <https://www.siggraph.org/>.
- [3] *Advances in Real-Time Rendering in 3D Graphics and Games*, 2022. adr.: <https://advances.realtimerendering.com/>.
- [4] Społeczność Godot Engine, *Godon Engine*, 2022. adr.: <https://godotengine.org>.
- [5] Epic Games, *Unreal Engine*, wer. 5, 2022. adr.: <https://www.unrealengine.com>.
- [6] Piotr Piwowarczyk, "Microsoft DirectX 12 Feature Level 12_2 - nowa wersja funkcji API <https://www.purepc.pl/microsoft-directx-12-feature-level-12-2-nowa-wersja-funkcji-api>", (urldate 2020-11-02).
- [7] *PEP 405 – Python Virtual Environments*, 2011. adr.: <https://peps.python.org/pep-0405/>.

SPIS RYSUNKÓW

3.1	Proces budowania w formie celów i ich zależności	10
-----	------------------------------------------------------------	----

SPIS TABLIC