

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	5
1. Wstęp	6
1.1. Cel pracy	6
1.2. Zakres pracy	7
1.3. Struktura pracy	7
2. Wprowadzenie do dziedziny	8
2.1. Podstawowe pojęcia	8
2.2. Potok zasobów	8
2.2.1. glTF	8
2.3. Vulkan	8
2.3.1. Podstawy API	8
2.3.2. Inicjalizacja	8
2.3.3. Bufory poleceń	8
2.3.4. Zasoby	8
2.3.5. Łańcuch wymiany	8
2.3.6. Synchronizacja	8
2.3.7. Deskryptory i stałe push	10
2.4. Rozszerzenie VK_EXT_descriptor_indexing	12
2.4.1. Niejednolite dynamiczne indeksowanie deskryptorów	13
2.4.2. Aktualizacja deskryptorów po dowiązaniu	13
2.4.3. Dowiązanie deskryptora o zmiennej wielkości	14
2.4.4. Częściowe dowiązania deskryptorów	14
2.4.5. Tablice deskryptorów czasu wykonania	15
2.5. Przebiegi renderowania i potoki	15
2.6. Rozszerzenie VK_EXT_dynamic_rendering	16
2.7. Renderowanie bez dowiązań	16
2.7.1. Tekstury bez dowiązań	16
2.7.2. Geometria bez dowiązań	18
2.8. Mapowanie tekstur	18
2.9. Oświetlenie	18
2.10. Cieniowanie odroczone	18
2.11. Graf sceny	18
2.12. Graf renderowania	18
3. Architektura i implementacja	19
3.1. Użyte narzędzia	19
3.1.1. Proces budowania	19
3.1.2. Biblioteki zewnętrzne	20

3.2. Architektura.....	21
3.3. Moduły	21
4. Badania	22
5. Podsumowanie.....	23
Wykaz literatury.....	24
Spis rysunków	25
Spis tablic.....	26

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 ukrywają 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 przegląadowy. 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 zasobów*

// TODO

2.2.1. *glTF*

// TODO

2.3. *Vulkan*

// TODO HISTORIA, core vs ext, promowanie

2.3.1. *Podstawy API*

// TODO użycie api, notacje, łańcuch pNext

2.3.2. *Inicjalizacja*

// TODO surface, urządzenie fizyczne i logiczne

2.3.3. *Bufory poleceń*

// TODO surface, urządzenie fizyczne i logiczne

2.3.4. *Zasoby*

// TODO pamięć // TODO bufory uniform i bufory magazynowe, obrazy, próbniki, obrazy magazynowe

2.3.5. *Łańcuch wymiany*

// TODO surface

2.3.6. *Synchronizacja*

// TODO

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 dostępu* (określony używając `VkAccessFlags`) wygenerowanych przez wcześniejszy zestaw poleceń jest udostępniony późniejszemu zestawowi poleceń dla pewnego *docelowego zakresu dostępu*.

Przykładowo zależność pamięci pomiędzy etapami `COLOR_ATTACHMENT_OUTPUT` i `FRAGMENT_SHADER` z zakresami dostępu `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 tranzycje układu obrazu,
- bariery pamięci buforów: dla zakresu bufora,
- globalne bariery pamięci: dla wszystkich istniejących obiektów,

// TODO użycie // TODO listingi?

Semaforey

Semaforey 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 semaforey są używane do synchronizacji pomiędzy kolejką graficzną i kolejką prezentacji w celu zagwarantowania, że prezentowalny obraz łańcucha wymiany jest używany tylko przez jedną kolejkę.

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.

2.3.7. Deskryptory i stałe push

Vulkan nie pozwala na bezpośredni dostęp do zasobów z poziomu shadera i wymaga użycia deskryptorów.

Deskryptor to blok pamięci z opisem pojedynczego zasobu używanego przez GPU. Dokładna wewnętrzna struktura deskryptora jest w formacie specyficznym dla GPU, ale może być intuicyjnie rozumiana jako struktura zawierająca wskaźnik to adresu pamięci GPU z danymi zasobu oraz dodatkowe metadane opisujące rodzaj zasobu oraz w jaki sposób zasób będzie używany przez shader.

Tworzenie deskryptorów

Vulkan nie pozwala na tworzenie i używanie pojedynczych deskryptorów i wymaga grupowania ich w tablice poprzez zbiory deskryptorów (obiekt *VkDescriptorSet*).

Stworzenie zbiorów deskryptorów wymaga wcześniejszego stworzenia dwóch obiektów: puli deskryptorów (*VkDescriptorPool*) oraz układu zbioru deskryptorów (*VkDescriptorSetLayout*).

Pula deskryptorów to źródło, z którego alokowane są deskryptory w postaci zbiorów deskryptorów. Podczas tworzenia należy zadeklarować:

- maksymalną liczbę zaalokowanych zbiorów deskryptorów,
- maksymalną liczbę rodzajów deskryptorów.

Układ zbioru deskryptorów reprezentuje wewnętrzną strukturę zbioru deskryptorów - programista języka C może o nim myśleć jako o deklaracji struktury używanej później do definiowania zmiennych (zbiorów deskryptorów). Układ składa się z listy dowiązań deskryptorów (*VkDescriptorSetLayoutBinding*).

Jedno dowiązanie deskryptora reprezentuje fragment zbioru deskryptorów zajmowany przez deskryptory tego samego typu. Każde dowiązanie deskryptora jest opisane poprzez:

- *numer dowiązania*: używany do odnoszenia się w shaderze do dowiązania i uzyskania dostępu do zasobu,
- *typ deskryptora*,
- *liczba deskryptorów*,
- *zbiór etapów cieniowania*: określa które shadery w potoku graficznym mają dostęp do zasobów.

Typ deskryptora zależy od rodzaju opisywanego zasobu, przykładowo:

- *UNIFORM_BUFFER*: bufor uniform,
- *UNIFORM_BUFFER_DYNAMIC*: dynamiczny bufor uniform, dodatkowy dynamiczny offset jest specyfikowany podczas dowiązywania zbioru deskryptorów,
- *STORAGE_BUFFER*: bufor magazynowy,
- *STORAGE_BUFFER_DYNAMIC*: dynamiczny bufor magazynowy,
- *SAMPLER*: próbnik,
- *SAMPLED_IMAGE*: widok próbkowalnego obrazu,
- *STORAGE_IMAGE*: widok obrazu magazynowego,
- *COMBINED_IMAGE_SAMPLER*: próbkowany obraz, pojedynczy deskryptor jest skojarzony zarówno z próbnikiej, jaki i z widokiem obrazu,
- *UNIFORM_TEXEL_BUFFER*: widok bufora uniform,
- *STORAGE_TEXEL_BUFFER*: widok bufora magazynowego.

Aktualizacja deskryptorów

Po stworzeniu zbioru deskryptorów zawartość jego deskryptorów jest niezdefiniowana i musi być zaktualizowana funkcją *vkUpdateDescriptorSets()*. Jej wejściem jest *tablica struktur VkWriteDescriptorSet*, której każdy pojedynczy element opisuje który wycinek tablicy wybranego dowiązania w zbiorze deskryptorów powinien być zaktualizowany informacjami o zasobach.

Aktualizacja zbioru deskryptorów odbywa się na CPU natychmiastowo po wywołaniu *vkUpdateDescriptorSets()* i jest możliwa tylko zanim zbiór deskryptorów zostanie użyty przez jakiekolwiek polecenie w nagrywanym bądź wykonywanym buforze poleceń. Jednym z wyjątków jest aktualizacja zbiorów deskryptorów zaalokowanych z puli deskryptorów wspierającej funkcjonalność uaktualnienia deskryptorów po dowiązaniu.

Stałe push

Stałe push to sposób przekazywania danych do shaderów będący szybszą i łatwiejszą alternatywą dla deskryptorów. Nie wymagają one tworzenia i aktualizacji zasobów opartych na pamięci GPU - pamięć CPU stałej push jest bezpośrednio kopiowana i przechowywana w nagrywanym buforze poleceń komendą *vkCmdPushConstants()*.

Niestety ta metoda ma poważne ograniczenie - minimalny rozmiar pamięci udostępniany shaderowi gwarantowany przez specyfikację Vulkan to tylko 128 bajtów, co odpowiada dwóm macierzom 4x4. Z tego powodu stałe push powinny być używane do przekazywania danych, które zmieniają się na tyle często, że narzut wydajnościowy synchronizacji modyfikowanych buforów uniform. Przykładem mogą być macierze transformacji albo indeksy tekstur używane przez polecenia rysowania.

Zadeklarowanie użycia deskryptorów w układzie potoku

Układ potoku (*VkPipelineLayout*) zawiera informacje o sposobie organizacji wszystkich zbiorów deskryptorów i stałych push, które mogą być używane w potoku (*VkPipeline*). Jest on używany do dowiązywania zbiorów deskryptorów i nagrywania stałych push.

Podczas tworzenia należy zadeklarować:

- listę układów zbiorów deskryptorów,
- listę zakresów stałych push (*VkPushConstantRange*).

Zakres stałej push składa się z:

- zbioru etapów cieniowania mających dostęp do stałej push,
- offset i rozmiar pamięci, który może być używany przez powyższe etapy cieniowania.

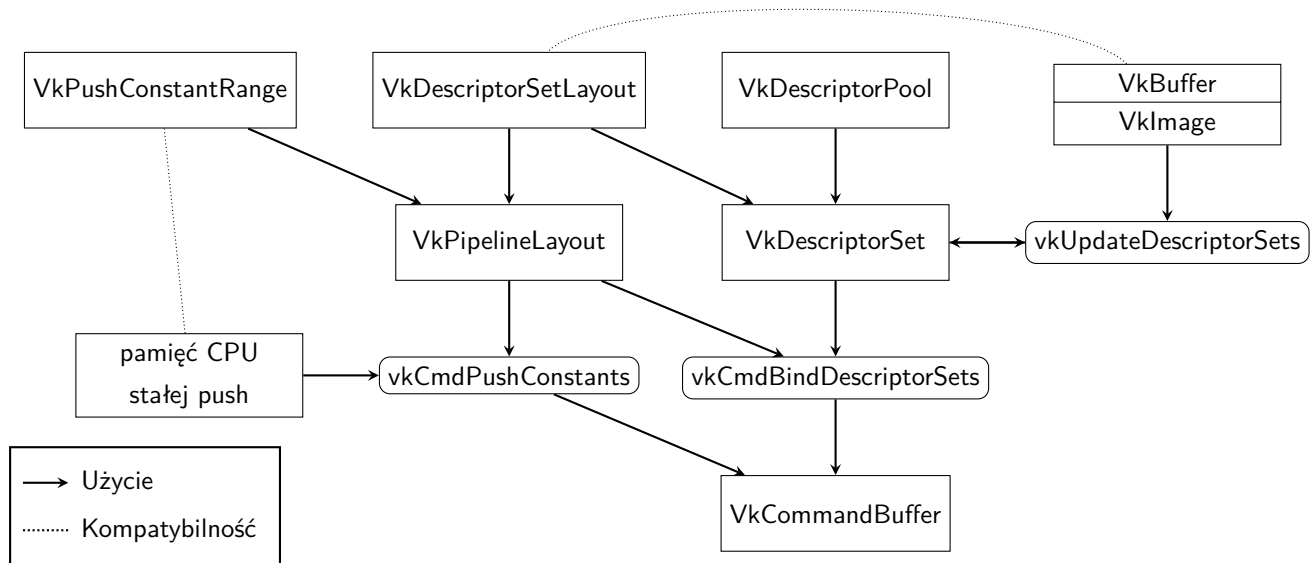
Liczba poszczególnych typów deskryptorów uwzględnionych w potoku renderowania jest ograniczona limitami urządzenia fizycznego. Limit *maxPerStageResources* to maksymalna liczba zasobów, które mogą być dostępne dla pojedynczego etapu cieniowania. Rodzina limitów *maxDescriptorSet**, gdzie * to typ deskryptora, kontroluje maksymalną liczbę deskryptorów danego typu w układzie potoku.

Dowiązanie deskryptorów do bufora poleceń

Przed użyciem zasobów opisanych zbiorem deskryptorów przez polecenia rysowania wymagane jest dowiązanie ich do bufora poleceń przy użyciu komendy *vkCmdBindDescriptorSets()*. Jednym z jej wejść jest *numer zbioru*, który wraz z numerami dowiązań służy do identyfikacji zasobu w shaderach.

Diagram użycia deskryptorów

Relacje pomiędzy obiektami, funkcjami i komendami używanymi deskryptorów i stałych push zostały przedstawione na poniższym diagramie:



Rysunek 2.1: Relacje pomiędzy obiektami Vulkan używanymi do zarządzania deskryptorami

Dostęp do zasobów w shaderach

Po dowiezaniu zbiorów deskryptorów i stałych push do bufora poleceń dostęp do zasobów z poziomu kodu GLSL shadera odbywa się poprzez zmienną posiadającą odpowiednie kwalifikatory układu.

Przykładowo kwalifikator układu dla pojedynczego deskryptora typu `UNIFORM_BUFFER` z dowiezania o numerze x ze zbioru o numerze y ma następującą formę:

```
struct bufferStruct {
    vec3 field1;
    mat4 field2;
    ...
};

layout(scalar, set = y, binding = x) uniform bufferBlock {
    bufferStruct buffer;
};
```

Analogicznie kwalifikator układu dla tablicy deskryptorów typu `COMBINED_IMAGE_SAMPLER` o rozmiarze r z dowiezania o numerze x ze zbioru o numerze y próbkowanych obrazów 2D ma następującą formę:

```
layout(set = y, binding = x) uniform sampler2D texture[r];
```

2.4. Rozszerzenie `VK_EXT_descriptor_indexing`

Rozszerzenie `VK_EXT_descriptor_indexing` wprowadziło szereg dodatkowych funkcjonalności pozwalających na tworzenie dużych zbiorów deskryptorów zawierających wszystkie zasoby używane przez

program. Celem tego jest umożliwienie technik renderowania bez dowiezań. Z powodu swojej użyteczności rozszerzenie to zostało promowane w Vulkan 1.2. W kolejnych sekcjach opisano nowe funkcjonalności.

2.4.1. Niejednolite dynamiczne indeksowanie deskryptorów

Deskryptory są traktowane przez shadery jako tablice, do których dostęp odbywa się używając indeksu.

Statyczne indeksowanie pozwala na dostęp do zasobu przy użyciu indeksu będącego stałą czasu kompilacji. Jest to najstarszy i zawsze wspierany sposób indeksowania.

Dynamiczne indeksowanie pozwala na dostęp do zasobu przy użyciu wartości czasu wykonywania.

Jednolite dynamiczne indeksowanie wymaga, żeby indeks był taki sam we wszystkich wywołań shadera spowodowanych przez pojedyncze polecenie rysowania - użycie różnych indeksów jest błędem i może skutkować korupcjami. Przykładem tego rodzaju indeksowania jest użycie indeksu tekstury w stałej push lub buforze uniform. Wymaga ona wsparcia następujących funkcjonalności urządzenia Vulkan 1.0:

- *shaderUniformBufferArrayDynamicIndexing*: tablice buforów uniform,
- *shaderSampledImageArrayDynamicIndexing*: tablice próbkowalnych obrazów,
- *shaderStorageBufferArrayDynamicIndexing*: tablice buforów magazynowych,
- *shaderStorageImageArrayDynamicIndexing*: tablice obrazów magazynowych.

Niejednolite dynamiczne indeksowanie pozwala na swobodny dostęp do zasobów znajdujących się w pamięci GPU przy użyciu dowolnych indeksów. Przykładem może być indeksowanie tablicy tekstur używając indeksu instancji polecenia rysowania albo próbki tekstury pozaekranowej. Wymaga ona wsparcia analogicznych funkcjonalności urządzenia Vulkan 1.2:

- *shaderUniformBufferArrayNonUniformIndexing*,
- *shaderSampledImageArrayNonUniformIndexing*,
- *shaderStorageBufferArrayNonUniformIndexing*,
- *shaderStorageImageArrayNonUniformIndexing*.

W czasie pisania pracy powyższe funkcjonalności są wspierane przez więcej niż 90% urządzeń [6].

Niejednolite dynamiczne indeksowanie wymaga zadeklarowania rozszerzenia SPIR-V *SPV_EXT_descriptor_indexing*. Może być to uczynione z poziomu kodu GLSL przez dodanie następującej dyrektywy preprocesora:

```
#extension GL_EXT_nonuniform_qualifier : require
```

Dodatkowo każde użycie niejednolitego indeksu powinno być oznaczone funkcją *nonuniformEXT*:

```
vec4 color = texture(textures2D[nonuniformEXT(index)], uv);
```

Wymóg jednolitości podczas indeksowania deskryptorów był spowodowany ograniczeniami poprzednich generacji GPU - w modelu renderowania OpenGL dostęp do dowieżanych tekstur odbywał się pośrednio poprzez jednostki teksturujące, które były widoczne przez wszystkie wywołania shaderów i nie pozwalały na zmianę dołączonych tekstur podczas wykonywania polecenia rysowania [7].

2.4.2. Aktualizacja deskryptorów po dowieżaniu

Domyślnie deskryptory nie mogą być aktualizowane po nagraniu ich dowieżania w buforze poleceń, przez co aplikacja musi mieć kompletną wiedzę o wszystkich używanych zasobach w trakcie nagrywania

buforów poleceń. Nie dotyczy to jednak deskryptorów używających funkcjonalności aktualizacji po dowiązaniu, co pozwala na elastyczniejsze zarządzanie zasobami poprzez odroczenie aktualizacji zbiorów deskryptorów aż do momentu bezpośrednio przed wykonaniem bufora poleceń.

Wsparcie aktualizacji po dowiązaniu dla wybranego typu deskryptora wymaga odpowiedniej funkcjonalności urządzenia Vulkan 1.2:

- *descriptorBindingUniformBufferUpdateAfterBind*: bufory uniform,
- *descriptorBindingSampledImageUpdateAfterBind*: próbkowalne obrazów,
- *descriptorBindingStorageBufferUpdateAfterBind*: bufory magazynowe,
- *descriptorBindingStorageImageUpdateAfterBind*: obrazy magazynowe.

W czasie pisania pracy powyższe funkcjonalności są wspierane przez ponad 90% urządzeń wyłączając bufory uniform niewspierane przez ok. 40% platform [6].

Użycie tej funkcjonalności wprowadza nowe limity *maxPerStageUpdateAfterBindResources* i *maxDescriptorSetUpdateAfterBind** zastępujące stare limity *maxPerStageResources* i *maxDescriptorSet**. Rozszerzenie gwarantuje, że nowe limity są takie same lub znacznie większe od starych limitów. Przykładowo na maszynie testowej limity *maxPerStageDescriptorSampledImages* i *maxDescriptorSetUpdateAfterBindSampledImages* to kolejno 65535 i 1048576.

Użycie tej funkcjonalności odbywa się poprzez stworzenia puli deskryptorów z flagą *UPDATE_AFTER_BIND*. Dowiązania zdefiniowane podczas tworzenia układu zbioru deskryptorów muszą posiadać flagę *UPDATE_AFTER_BIND_POOL*.

Aplikacja musi zapewnić odpowiednią synchronizację - aktualizowane deskryptory nie mogą być używane przez potok graficzny w momencie aktualizacji.

2.4.3. Dowiązanie deskryptora o zmiennej wielkości

Domyślnie wielkość dowiązania deskryptora jest stałą wartością określoną podczas stworzenia układu zbioru deskryptora. Ograniczenie to nie dotyczy dowiązań deskryptora o zmiennej wielkości.

Dzięki tej funkcjonalności wielkość zbioru deskryptorów jest niezależna od układu zbioru deskryptorów i jest specyfikowana dopiero podczas tworzenia zbioru deskryptorów, co pozwala to obsługi sytuacji, w której dokładna liczba deskryptorów wymaganych do opisanie zasobów nie jest znana podczas tworzenia układu zbioru deskryptoru.

Wsparcie dowiązań o zmiennej wielkości wymaga funkcjonalności urządzenia Vulkan 1.2 *descriptorBindingVariableDescriptorCount*, która w czasie pisania pracy jest wspierana przez ponad 90% urządzeń [6].

Użycie tej funkcjonalności odbywa się poprzez stworzenia układu zbioru deskryptorów, którego ostatnie dowiązanie posiada flagę *VARIABLE_DESCRIPTOR_COUNT* i zamiast liczby deskryptorów podawana jest jej górna granica. Rzeczywista liczba jest ustalana podczas tworzenia zbioru deskryptorów przy użyciu struktury *VkDescriptorSetVariableDescriptorCountAllocateInfo* w łańcuchu *pNext*.

2.4.4. Częściowe dowiązania deskryptorów

Domyślnie wszystkie deskryptory w dołączonym zbiorze deskryptorów nie mogą być w stanie nieprawidłowym i muszą koniecznie być zaktualizowane przez dowiązaniem - jest to nazywane wymogiem statycznego użycia deskryptorów. Ograniczenie to nie dotyczy częściowo dowiązanych deskryptorów.

Dzięki tej funkcjonalności deskryptory muszą być dynamicznie używane: deskryptory nieużywane przez shadery mogą być nieprawidłowe i dodatkowo mogą być nawet aktualizowane gdy zbior deskryptorów jest używany przez GPU - pamiętając, że dostęp przy użyciu nieprawidłowego deskryptora jest

wciąż niezdefiniowanym zachowaniem i aplikacja musi zapewnić odpowiednią synchronizację CPU-GPU.

Wsparcie dowiązań o zmiennej wielkości wymaga funkcjonalności urządzenia Vulkan 1.2 *descriptorBindingPartiallyBound*, która w czasie pisania pracy jest wspierana przez ponad 94% urządzeń [6].

Użycie tej funkcjonalności odbywa się poprzez stworzenia układu zbioru deskryptorów, którego dowiązanie posiada flagę *PARTIALLY_BOUND_BIT*.

2.4.5. Tablice deskryptorów czasu wykonania

Domyślnie rozmiar tablicy deskryptorów musi być znany podczas kompilacji shaderów. Przykładowo w języku GLSL jest on podawany w kwalifikatorze układu. Wymaganie to nie dotyczy tablic deskryptorów czasu wykonania.

Ta funkcjonalność pozwala na deklarację w shaderach tablic deskryptorów których rozmiar nie jest znany podczas kompilacji, co pozwala na kompilację shaderów bez wiedzy o dokładnej liczbie deskryptorów w dowiązaniach.

Wsparcie tablic deskryptorów czasu wykonania wymaga funkcjonalności urządzenia Vulkan 1.2 *runtimeDescriptorArray*, która w czasie pisania pracy jest wspierana przez ponad 94% urządzeń [6].

Użycie tej funkcjonalności odbywa się poprzez pominięcie rozmiaru tablicy w kwalifikatorze układu w kodzie GLSL:

```
layout (set = y, binding = x) uniform sampler2D texture [];
```

Dostęp do zmiennej w GLSL się nie zmienia, ale wygenerowany kod SPIR-V używa rozszerzenia *SPV_EXT_descriptor_indexing* i typu *OpTypeRuntimeArray* zamiast *OpTypeArray*:

OpCapability Shader

OpCapability RuntimeDescriptorArray

OpExtension "SPV_EXT_descriptor_indexing"

...

OpName %textures2D "textures2D"

...

OpDecorate %textures2D **DescriptorSet** 0

OpDecorate %textures2D **Binding** 2

...

%150 = **OpTypeImage** %float 2D 0 0 0 1 Unknown

%151 = **OpTypeSampledImage** %150

_%runtimearr_151 = **OpTypeRuntimeArray** %151

_%ptr_UniformConstant__runtimearr_151 = **OpTypePointer UniformConstant** %_runtimearr_1

%textures2D =

OpVariable %_ptr_UniformConstant__runtimearr_151 **UniformConstant**

Indeksowanie poza długością tablicy czasu wykonania jest niezdefiniowanym zachowaniem i może skutkować korupcją.

```
// TODO
```

2.5. Przebiegi renderowania i potoki

```
// TODO
```

2.6. Rozszerzenie `VK_EXT_dynamic_rendering`

// TODO

2.7. Renderowanie bez dowiezań

Renderowania bez dowiezań to grupa technik mająca na celu minimalizację liczby poleceń rysowania w celu maksymalizacji czasu GPU, który jest spędzany na rzeczywistym renderowaniu, a nie na zmianach stanu pomiędzy poleceniami [COOKBOOK].

W Vulkan renderowanie bez dowiezań jest realizowane poprzez:

- maksymalne zmniejszenie liczby alokowanych deskryptorów,
- eliminację kosztownego dowieżywania zasobów między poleceniami rysowania,
- łączenie // HIRO multidraw
- reifikacja sceny // HIRO
- umożliwienia GPU bezpośredniego dostępu do buforów i tekstur poprzez niejednolite dynamiczne indeksowanie deskryptorów.

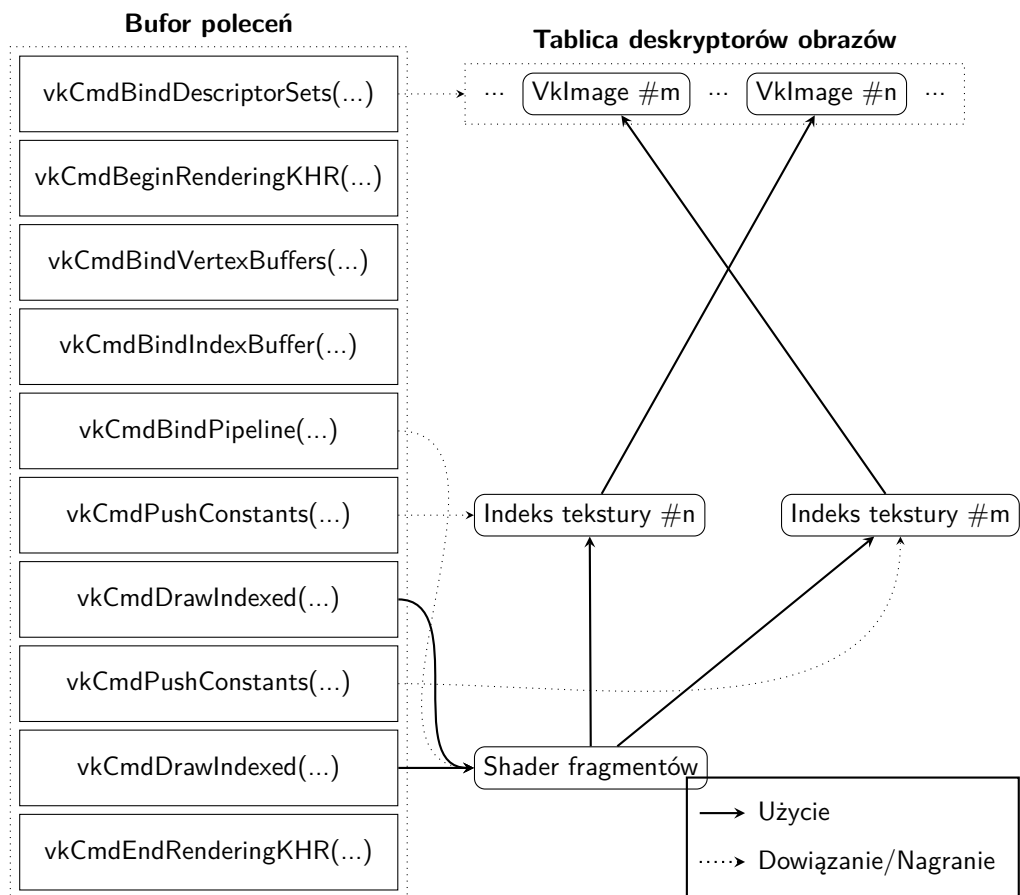
// TODO unified geometry buffer

2.7.1. Tekstury bez dowiezań

Wymóg jednolitości podczas dynamicznego indeksowania deskryptorów wywołuje problemy z renderowaniem scen wypełnionych obiektami używającymi różnych tekstur. Podczas renderowania obiektów każda zmiana używanej tekstury wymaga nagrania nowego polecenia rysowania po zmianie używanych deskryptorów poprzez:

- dowieżanie całkowicie nowego zbioru deskryptorów: bardzo kosztowaną operacją, ale wymaga wsparcia tylko statycznego indeksowania,
- ponowne dowieżanie deskryptora dynamicznego bufora uniform ze zmienionym dynamicznym offsetem,
- zmianę jednolitego indeksu używanego do indeksowania deskryptorów poprzez:
 - nagranie nowej stałej push,
 - zmianę bazowego indeksu instancji w poleceniu rysowania pojedynczej instancji: wbudowania zmienna shaderów `gl_InstanceIndex` jest niejednolita tylko dla poleceń rysowania renderujących więcej niż jedną instancję.

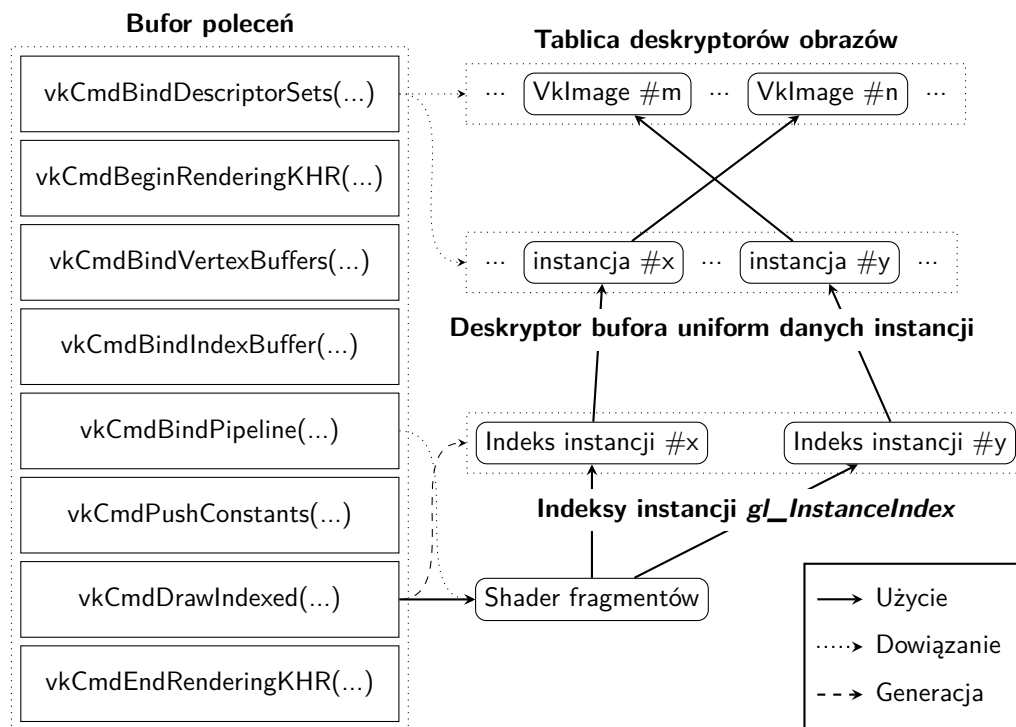
Poniższy diagram przedstawia tradycyjne tekstury z dowieżaniami używające jednolitego indeksu w stałej push:



Rysunek 2.2: Tradycyjne tekstury z dowiązaniami używające jednolitego indeksu w stałej push

Niejednolite dynamiczne indeksowanie pozwala na dowiązanie wszystkich używanych zasobów na początku bufora poleceń i wyemitowanie pojedynczego polecenia rysowania, którego shadery używają niejednolitego indeksu do pobrania indeksów używanych tekstur z bufora uniform opisującego obiekty na scenie.

Poniższy diagram przedstawia tekstury bez dowiązań używając niejednolitych indeksów instancji:



Rysunek 2.3: Tekstury bez dowiązań używając niejednorodnych indeksów instancji

2.7.2. Geometria bez dowiązań

// TODO

2.8. Mapowanie tekstur

// TODO

2.9. Oświetlenie

// TODO

2.10. Cieniowanie odroczone

// TODO

2.11. Graf sceny

// TODO

2.12. 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.

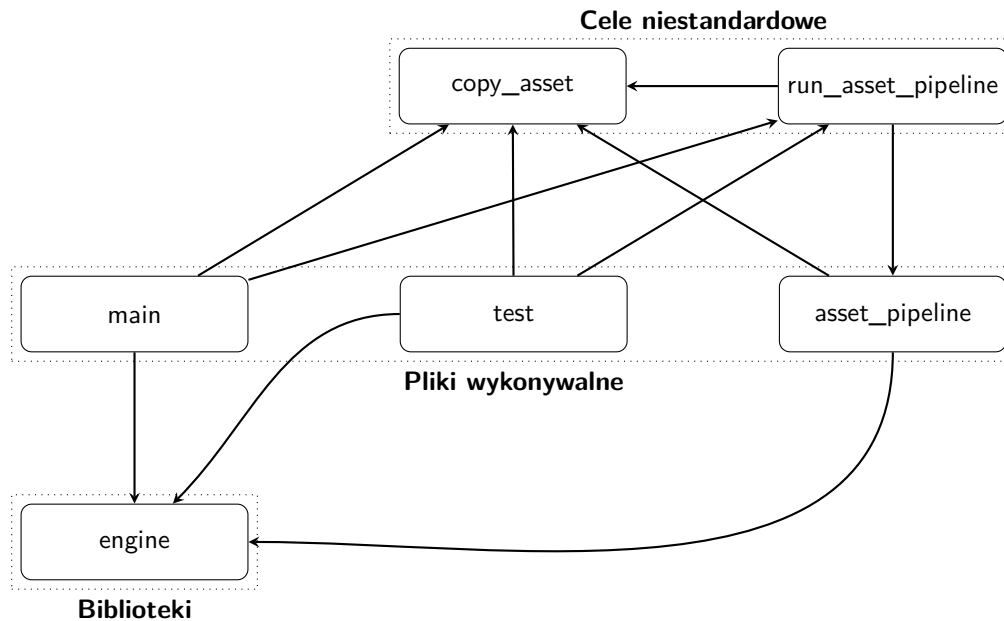
Skrypty w języku Python są obsługiwane pośrednio przez CMake, które wykrywa zainstalowany interpreter języka Python i używa go do stworzenia tzw. środowiska wirtualnego w tymczasowym katalogu `venv/` w głównym katalogu projektu. Podczas procesu budowania środowisko wirtualne jest używane do zainstalowania 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 [8].

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] *Vulkan Hardware Database - GPUinfo.org*, 2022. adr.: <https://vulkan.gpuinfo.org/>.
- [7] W. Engel, *GPU Pro 4: Advanced Rendering Techniques* (An A K Peters Book t. 4). Taylor & Francis, 2013.
- [8] *PEP 405 – Python Virtual Environments*, 2011. adr.: <https://peps.python.org/pep-0405/>.

SPIS RYSUNKÓW

2.1	Relacje pomiędzy obiektami Vulkan używanymi do zarządzania deskryptorami	12
2.2	Tradycyjne tekstury z dowiązaniem używające jednolitego indeksu w stałej push	17
2.3	Tekstury bez dowiązań używając niejednorodnych indeksów instancji	18
3.1	Proces budowania w formie celów i ich zależności	20

SPIS TABLIC