



WYDZIAŁ ELEKTRONIKI,
TELEKOMUNIKACJI
I INFORMATYKI

Imię i nazwisko studenta: Szymon Szczyrbak

Nr albumu: 165760

Poziom kształcenia: Studia drugiego stopnia

Forma studiów: stacjonarne

Kierunek studiów: Informatyka

Specjalność: Algorytmy i technologie internetowe

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Silnik renderujący używający technik cieniowania odroczonego

Tytuł pracy w języku angielskim: Rendering engine using deferred shading techniques

Opiekun pracy: dr inż. Piotr Mironowicz

STRESZCZENIE

Cieniowanie odroczone jest techniką grafiki 3D czasu rzeczywistego popularną wśród twórców gier komputerowych pozwalającą na obsługę wielu światów 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, cieniowanie odroczone, renderowanie bez dowiązań, renderowanie pośrednie, Vulkan, glTF.

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

ABSTRACT

Deferred shading is a real-time 3D graphics technique popular among computer game developers. It allows multiple lights on a stage without a significant drop in performance.

In this thesis, a rendering engine was designed and implemented using the C language and the Vulkan graphics library. The elements of the rendering engine as well as low and high-level techniques used in modern 3D games with an emphasis on deferred rendering are described. The engine architecture and implementation are described. A sample scene was rendered and the performance was measured.

Keywords: rendering engine, deferred shading, bindless rendering, indirect rendering, Vulkan, glTF.

Field of science and technology classification: 1.2 Computer and information sciences.

SPIS TREŚCI

Streszczenie	2
Abstract	3
Spis treści	4
Wykaz ważniejszych oznaczeń i skrótów	6
1. Wstęp	7
1.1. Cel pracy	7
1.2. Zakres pracy	8
1.3. Struktura pracy	8
2. Wprowadzenie do dziedziny	9
2.1. Vulkan	9
2.1.1. Podstawy API	10
2.1.2. Szkielet aplikacji graficznej	12
2.1.3. Inicjalizacja podstawowych obiektów	12
2.1.4. Zasoby	16
2.1.5. Łącuch wymiany	20
2.1.6. Bufory polecień	22
2.1.7. Deskryptory i stałe push	23
2.1.8. Rozszerzenie VK_EXT_descriptor_indexing	26
2.1.9. Potoki graficzne i przebiegi renderowania	29
2.1.10. Rozszerzenie VK_KHR_performance_query	33
2.1.11. Synchronizacja	33
2.2. Renderowanie bez dowiązań	35
2.3. Potok zasobów	37
2.3.1. Zasoby wejściowe	37
2.3.2. Zasoby wyjściowe	39
2.3.3. Potok zasobów	40
2.4. Graf sceny	40
2.5. Graf renderowania	41
2.6. Renderowanie oparte na fizyce	42
2.7. Renderowanie odroczone i efekty post-processingu	42
3. Narzędzia, architektura i implementacja	44
3.1. Narzędzia	44
3.1.1. Proces budowania	44
3.1.2. Biblioteki zewnętrzne	45
3.2. Architektura	46
3.3. Implementacja	48
3.3.1. Wygenerowany kod	48
3.3.2. Rdzeń	53
3.3.3. I/O	56
3.3.4. Zasoby	58

3.3.5. Vulkan	62
3.3.6. Scena	74
3.3.7. Renderer	77
4. Wyniki i testy wydajnościowe	85
4.1. Przykładowe sceny	85
4.2. Pomiary wydajnościowe	86
4.3. Analiza pomiarów	88
5. Podsumowanie	89
Wykaz literatury	92
Spis rysunków	93
Spis tablic	94

WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW

Skróty:

- API - Application Programming Interface, interfejs programistyczny aplikacji;
- CPU - Central Processing Unit, procesor;
- CTS - Conformance Test Suite, pakiet testów zgodności;
- FPS - Frames Per Second, klatki na sekundę;
- GPU - Graphics Processing Unit, procesor graficzny;
- IHV - Independent Hardware Vendor, niezależny dostawca sprzętu;
- ISA - Instruction Set Architecture, architektura procesora;
- GLSL - OpenGL Shading Language, język programowania potoku graficznego wprowadzony w API OpenGL;
- HLSL - High-Level Shader Language, język programowania potoku graficznego wprowadzony w API Direct3D 9;
- PBR - Physically Based Rendering, renderowanie oparte na fizyce;
- SDK - Software Development Kit, zestaw narzędzi dla programistów aplikacji;
- WSI - Windowing System Integration, integracja systemu okien.

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 może być śledzony na dorocznej konferencji SIGGRAPH [2], podczas której prezentacje i dyskusje akademickie są przeplecone 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 wyświetlanie 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 tracing) 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. Proces budowania używa skryptów Python do automatycznej generacji kodu, który nie może być wyrażony używając istniejących mechanizmów metaprogramowania języka C. Renderowanie grafiki 3D jest obsługiwane przez Vulkan API [4]. 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 przeglądowy i implementacyjny. Nowoczesne silniki graficzne składają się z wielu elementów, z których każdy może być niezależnie rozwijany do 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 profesjonalnymi silnikami graficznymi rozwijanymi z myślą o zastosowaniu w grach komputerowych (takimi jak otwarty Godot [5] czy komercyjny Unreal Engine [6]). 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 szczegółowo implementacji poszczególnych jego modułów.

W czwartym rozdziale wyrenderowano przykładową scenę używając potoku graficznego używającego cieniowania odroczonego i zbadano wydajność silnika.

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

2. WPROWADZENIE DO DZIEDZINY

Grafika komputerowa czasu rzeczywistego jest szerokim zagadnieniem. W tym rozdziale przybliżono bibliotekę Vulkan oraz techniki renderowania, których zrozumienie jest wymagane przed rozpoczęciem implementacji silnika graficznego.

2.1. Vulkan

Biblioteki graficzne pozwalają aplikacji na użycie ich API do uzyskania dostępu do akceleracji sprzętowej, czyli przeniesienia obliczeń wymaganych przez renderowanie z CPU do specjalnie pod nie zoptymalizowanego GPU.

Biblioteka graficzna mająca na celu równe wsparcie wielu platform rozwijanych przez różnych IHV (*Independent Hardware Vendor*, niezależny dostawca sprzętu) wymaga drobiazgowego ustandaryzowania i dokumentacji. W czasie pisania pracy istnieją trzy popularne standardy: Direct3D od firmy Microsoft oraz OpenGL i Vulkan od konsorcjum non-profit Khronos.

Vulkan w wersji 1.0 został po raz pierwszy wydany w 2016. By zacząć używać Vulkan należy pobrać Vulkan SDK rozwijany przez LunarG [7]. Zawiera on m.in. specyfikację, nagłówki API, biblioteki SPIR-V oraz warstwy, które zostały opisane poniżej.

Specyfikacja Specyfikacja Vulkan [4] to ponad 1000 stron zwięzej i precyzyjnej specyfikacji API przeznaczonej do użytku zarówno przez implementatorów sterowników, jak i programistów aplikacji.

Khronos utrzymuje listę urządzeń, które zaliczyły zestaw testów zgodności Vulkan CTS [8] i spełniają wymagania wieloplatformowości (w odróżnieniu takich API jak DirectX wspieranego tylko przez system Windows i konsole Xbox [1] czy Metal wspierane przez urządzenia firmy Apple).

Nowa funkcjonalność jest dodawana, podobnie jak w OpenGL, używając opcjonalnych rozszerzeń podstawowej specyfikacji. Są one proponowane przez członków Khronos i często są specyficzne dla urządzeń. Przykładowo rozszerzenie *VK_NV_mesh_shader* pozwala na użycie shaderów siatki na GPU firmy Nvidia.

Rozszerzenia są podzielone na trzy kategorie różniące się stopniem adaptacji.

Rozszerzenia mogą być rozwijane tylko przez pojedynczego dostawcę (można to poznać po nazwie zawierającej *NV*, *AMD*, *QCOM*, *VALVE* itp.) i ich wsparcie jest zwykle ograniczone do ich produktów (np. GPU, gier komputerowych lub narzędzi do debugowania).

Rozszerzenia *EXT* zostały częściowo ustandaryzowane i są wspierane przez wielu dostawców.

Dostatecznie popularne rozszerzenie może stać się podstawą rozszerzenia *KHR*, od którego oczekuje się wsparcia przez większość sterowników. Przykładowo na podstawie rozszerzenia *VK_NV_ray_tracing* stworzono rozszerzenie *VK_KHR_ray_tracing_pipeline*.

Rozszerzenia mogą być promowana w nowej wersji Vulkan stając się częścią podstawowej specyfikacji. Przykładowo rozszerzenie *VK_EXT_scalar_block_layout* zostało promowane w Vulkan 1.2.

Biblioteki SPIR-V Vulkan używa niskopoziomowej pośredniej reprezentacji shaderów w postaci kodu bajtowego SPIR-V [9], który jest standardem Khronos będącym przenośnym celem komplikacji shaderów napisanych w wysokopoziomowych językach takich jak GLSL, HLSL czy Cg. Implementatorzy zajmują się tylko translacją ze SPIR-V do kodu maszynowego urządzenia, co znacznie upraszcza sterowniki

niemuszące osadzać całego kompilatora, przyśpiesza proces kompilacji oraz zmniejsza prawdopodobieństwo sytuacji znanej z OpenGL, w której mimo deklarowanej przenośności API sterowniki różnej jakości interpretują ten sam shader na różne sposoby [10]. Vulkan SDK oferuje zestaw gotowych narzędzi i bibliotek pozwalających na komplikację, analizę i deasembację SPIR-V.

Warstwy Vulkan jest niskopoziomowy i skomplikowany w użyciu - słynny tutorial wymaga ponad 1000 linijek kodu C++ do renderowania pojedynczego trójkąta [11]. Sterowniki nie sprawdzają większości błędów i wymagają od programisty chcącego uniknąć niezdefiniowanych zachowań przestrzegania zasad poprawnego użycia API.

Na szczęście Vulkan wspiera tworzenie warstw - małych bibliotek dynamicznych pośredniczących pomiędzy aplikacją i sterownikiem, które przechwytyują wywołania API i pozwala na dodanie do nich dodatkowej logiki. Vulkan SDK oferuje oficjalne warstwy mające uprościć proces debugowania.

Warstwa walidacji *VK_LAYER_KHRONOS_validation* wykrywa oraz raportuje nieprawidłowe i nie-wydajne użycie API.

Warstwy rozszerzeń implementują funkcje rozszerzeń niewspieranych przez sterownik. Przykładowo warstwa *VK_LAYER_KHRONOS_synchronization2* implementuje rozszerzenie *VK_KHR_synchronization2*, które upraszcza użycie API synchronizacji.

Warstwy narzędzi dodają przydatne funkcjonalności takie jak raportowanie wywołań API (*VK_LAYER_LUNARG_api_dump*), robienie zrzutów ekranu (*VK_LAYER_LUNARG_screenshot*) czy symulacja możliwości bardziej ograniczonych GPU (*VK_LAYER_LUNARG_device_simulation*).

Warstwy mają negatywny wpływ na wydajność, który jest zwłaszcza widoczny w przypadku warstwy walidacji. Dlatego też warto używać je w trakcie rozwoju, ale nie w produkcie końcowym.

2.1.1. Podstawy API

Vulkan jest API obiektowym - wszystkie używane koncepty są reprezentowane przez obiekty Vulkan tworzone i niszczone przez aplikację, która ma do nich dostęp poprzez uchwyty. W przeciwieństwie do OpenGL używającego globalnej maszyny stanów, Vulkan jest bezstanowy, używany stan jest całkowicie zaszyty w obiektach i wszystkie funkcje API operują tylko na stanie obiektów przekazanych do nich w postaci parametrów i mogą być wywoływane współbieżnie z wielu wątków. Wyjątkiem są parametry zdefiniowane jako *zewnętrznie synchronizowane*, dla których aplikacja musi zagwarantować, że tylko jeden wątek używa obiektu takiego parametru w danym momencie. Przykładowo większość funkcji *vkCmd** wymaga zewnętrznej synchronizacji obiektu bufora poleceń *VkCommandBuffer*, co oznacza, że ich wywoływanie powinno się odbywać najlepiej w ramach jednego wątku.

API Vulkan posiada regularną i przewidywalną strukturę nazewnictwa oraz użycia obiektów, którego elementy zostały opisane poniżej.

Konwencje nazewnictwa

Nagłówek *vulkan.h* dołącza funkcje, struktury, typy wyliczeniowe i stałe preprocesora języka C mające następujące przedrostki:

- *vk*: funkcje (np. *vkBeginCommandBuffer*),
- *Vk*: struktury i typy wyliczeniowe (np. *VkPipeline* i *VkPipelineBindPoint*),
- *VK_*: wyliczenia i stałe (np. *VK_PIPELINE_BIND_POINT_GRAPHICS* i *VK_NULL_HANDLE*).

W imię zwięzości dalej w pracy części nazwy będą pomijane jeśli można je wywnioskować z kontekstu. Przykładowo *VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT* może być zapi-

sywany jako etap potoku *COLOR_ATTACHMENT_OUTPUT*.

Funkcje *vkCmd**() są nazywane poleceniami. Mogą być one używane tylko pomiędzy wywołaniami funkcji *vkBeginCommandBuffer()* i *vkEndCommandBuffer()*. Ich wywołanie nagrywa polecenie, czyli dodaje instrukcję przeznaczone do wykonania na GPU do bufora polecień. Przykładowo polecenie *vkCmdCopyBufferToImage* nagrywa operację kopiowania bufora do obrazu.

Funkcje *vkEnumerate**() i *vkGet**() służą do odpytywania właściwości obiektów. Przykładowo funkcja *vkGetPhysicalDeviceQueueFamilyProperties()* rapportuje właściwości rodzin kolejek udostępnianych przez dane urządzenie fizyczne.

Nowe elementy nagłówka wprowadzane przez rozszerzenia kończą się przyrostkami. Przykładowo rozszerzenie *VK_NV_device_diagnostic_checkpoints* wprowadza funkcję *vkCmdSetCheckpointNV* używającą struktury *VkCheckpointDataNV*.

Model obiektów

Wszystkie obiekty Vulkan mogą być albo tworzone funkcją *vkCreate**(), albo alokowane z puli wcześniejszej utworzonego obiektu funkcją *vkAllocate**().

Tworzenie obiektu wymaga przygotowania struktury informacji tworzenia *Vk*CreateInfo* przedstawionego na listingu 2.1.

```
Vk*CreateInfo createInfo = {
    .sType = VK_STRUCTURE_TYPE_*_CREATE_INFO;
    .flags = ...,
    ...
    .pNext = NULL,
};

Vk* object;
assert(vkCreate(..., &createInfo, ..., &object) == VK_SUCCESS);
```

Listing 2.1: Tworzenie obiektu Vulkan

Alokacja obiektu wymaga przygotowania struktury informacji alokacji *Vk*AllocateInfo* przedstawionego na listingu 2.2.

```
Vk*Pool pool;

Vk*AllocateInfo allocInfo = {
    .sType = VK_STRUCTURE_TYPE_*_ALLOCATE_INFO;
    .*Pool = pool,
    ...
    .pNext = NULL,
};

Vk* object;
assert(vkAllocate(..., &allocInfo, ..., &object) == VK_SUCCESS);
```

Listing 2.2: Alokacja obiektu Vulkan

Stworzone obiekty są niszczone funkcją *vkDestroy**(). Zaalokowane obiekty są zwalniane funkcją *vkFree**() lub poprzez zniszczenie puli obiektów.

Struktury Vulkan często wspierają łańcuch *pNext* - wskaźnik *void** na kolejną strukturę w liście jednokierunkowej lub *NULL*. Jest on używany przez rozszerzenia dodające nowe struktury i może być iterowany w wygodny sposób używając struktur *VkBaseInStructure* i *VkBaseOutStructure* - typ struktury może być wywnioskowany używając jej pierwszego pola *sType*.

2.1.2. Szkielet aplikacji graficznej

Vulkan, z racji swojej niskopoziomowości, nie narzuca jednej konkretnej struktury aplikacji - większość problemów można rozwiązać wieloma technikami, z których każda ma swoje zalety i wady. Programista powinien zaprojektować program w taki sposób, żeby rozwiązywał on problem, nie był zawiły i jego profilowanie ujawniało maksymalne użycie GPU.

W strukturze wszystkich aplikacji graficznych można wyróżnić następujące ogólne kroki:

- Stworzenie bądź wybór podstawowych obiektów: instancja (*VkInstance*), urządzenie fizyczne (*VkPhysicalDevice*), urządzenie logiczne (*VkDevice*) oraz kolejka graficzna (*VkQueue*),
- Przygotowanie obiektów służących do prezentacji wyników renderowania: powierzchnia okna (*VkSurface*), łańcuch wymiany (*VkSwapchain*), prezentowalne obrazy (*VkImage*) i ich widoki (*VkImageView*) oraz kolejka prezentacji (*VkQueue*),
- Transfer zasobów z CPU do GPU opisanych w zbiorach deskryptorów (*VkDescriptorSet*),
- Stworzenie obiektów opisujących proces renderowania: potoki (*VkPipeline*) z shaderami (*VkShaderModule*) oraz przebiegi renderowania (*VkRenderPass*),
- Wykonanie procesu renderowania w pętli głównej programu: pobranie nieużywanego prezentowalnego obrazu, nagranie i wykonanie buforów poleceń (*VkCommandBuffer*) realizujących pożądane techniki renderowania oraz prezentacja wyrenderowanego obrazu.

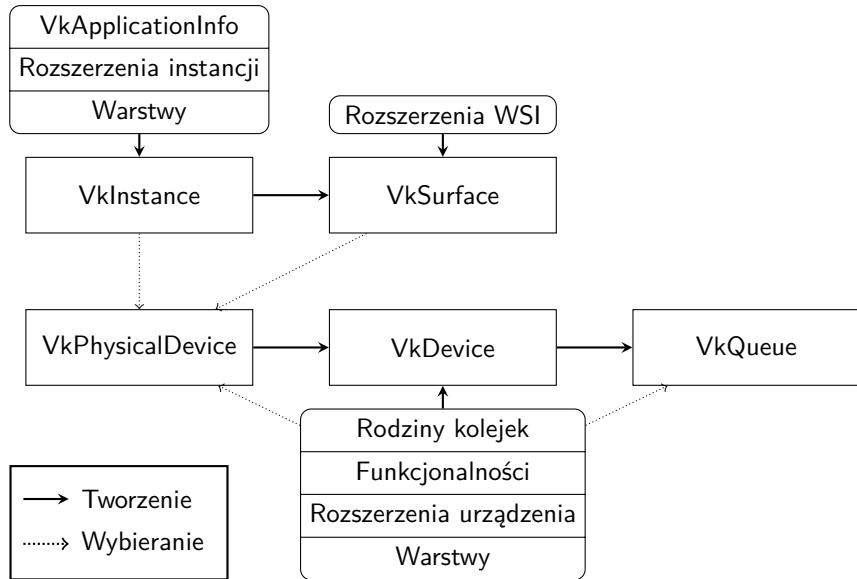
W kolejnych sekcjach pracy przybliżono wspomniane powyżej obiekty Vulkan i zaprezentowano metody ich użycia.

2.1.3. Inicjalizacja podstawowych obiektów

Wszystkie programy używające API Vulkan wymagają wcześniejszego stworzenia obiektów w następującej kolejności:

- instancji (*VkInstance*),
- powierzchni okna (*VkSurface*),
- urządzenia fizycznego (*VkPhysicalDevice*),
- urządzenia logicznego (*VkDevice*),
- wskaźników funkcji rozszerzeń,
- kolejek (*VkQueue*),

Poniższy diagram przedstawia kolejność inicjalizacji podstawowych obiektów Vulkan:

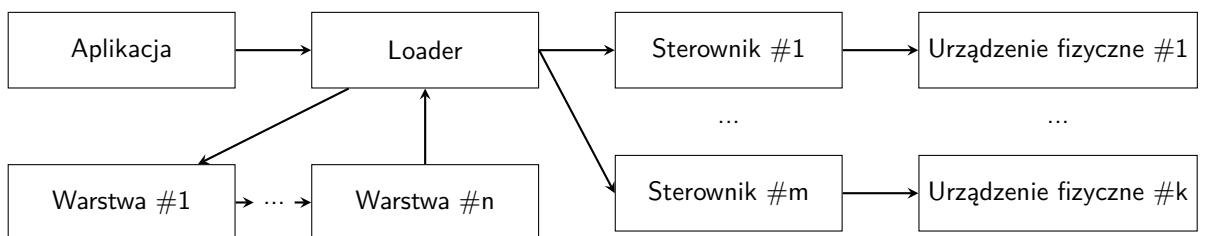


Rysunek 2.1: Kolejność inicjalizacji podstawowych obiektów Vulkan (opracowanie własne)

Instancja

Pierwszym krokiem każdego programu chcącego używać Vulkan jest stworzenie instancji, która pozwala programowi na komunikację z loaderem Vulkan.

Loader Vulkan to zewnętrzna warstwa biblioteki Vulkan pośrednicząca między aplikacją i urządzeniami fizycznymi. Jest on odpowiedzialny za wykrywanie sterowników wspierających Vulkan i przekazywanie do nich wywołań API po wcześniejszym przefiltrowaniu ich przez załadowane warstwy. Poniższy diagram przedstawia warstwową architekturę biblioteki Vulkan [12]:



Rysunek 2.2: Warstwowa architektura biblioteki Vulkan (opracowanie własne na podstawie [12])

Instancja musi zostać stworzona przed użyciem jakichkolwiek innych funkcji API Vulkan. Jest ona używana przez funkcje instancji, które są używane do:

- stworzenia powierzchni okna,
- stworzenia komunikatora debugowania,
- uzyskania wskaźników funkcji rozszerzeń,
- pobrania listy urządzeń fizycznych.

Podczas tworzenia instancji należy zdefiniować podstawowe informacje o aplikacji (`VkApplicationInfo` zawierające nazwę i wersję aplikacji, używanego silnika i API Vulkan) oraz listę używanych rozszerzeń instancji i warstw.

Powierzchnia

Po stworzeniu instancji Vulkan program chcący prezentować wyniki renderowania musi stworzyć powierzchnię okna.

Ten krok może być pominięty dla programów używających Vulkan w trybie *headless* niewyswietlającym wyniku renderowania na ekranie. W innym wypadku powierzchnia musi być stworzona przed urządzeniem fizycznym, ponieważ jest używana do sprawdzania, czy wybrane urządzenie fizyczne będzie wspierało stworzenie łańcucha wymiany dla powierzchni okna.

Stworzenie powierzchni okna odbywa się przy użyciu WSI (Windowing System Integration, integracja systemu okien), który jest zbiorem rozszerzeń udostępnianych przez środowisko uruchomieniowe programu pozwalających na integrację API Vulkan z systemem okien w celu wyświetlenia wyników renderowania. Użycie WSI wymaga trzech rozszerzeń instancji:

- `VK_KHR_surface`: udostępnia obiekt `VkSurface` bez funkcji tworzenia,
- `VK_KHR_swapchain`: udostępnia obiekt `VkSwapchain`,
- `VK_KHR_*_surface`, gdzie * to nazwa systemu okien (przykładowo `VK_KHR_win32_surface` dla Windows): udostępnia specyficzne funkcje instancji pozwalające na stworzenie `VkSurface`.

Tworzenie okna jest często obsługiwane przez bibliotekę multimedialną taką jak GLFW [13] czy SDL [14], które posiadają funkcjonalność abstrahującą proces tworzenia powierzchni okna - każda wspierana platforma wymaga osobnej implementacji używającej odpowiedniego rozszerzenia WSI w parze z API systemu okien.

Urządzenie fizyczne

Po stworzeniu instancji należy wybrać urządzenie fizyczne. Reprezentuje ono pojedynczy element systemu operacyjnego wspierający Vulkan - zwykle jedną z kart graficznych obsługiwanych przez zainstalowany sterownik graficzny lub renderer programowy taki jak llvmpipe [15] czy SwiftShader [16].

Rozróżniane są następujące typy urządzeń fizycznych (`VkPhysicalDeviceType`):

- `DISCRETE_GPU`: dedykowana karta graficzna,
- `INTEGRATED_GPU`: zintegrowane GPU,
- `VIRTUAL_GPU`: wirtualne GPU oferowane przez środowisko wirtualizacji,
- `CPU`: renderer programowy.

Każde urządzenie fizyczne jest opisywane ogólnie przy pomocy właściwości i funkcjonalności. Właściwości (`VkPhysicalDeviceProperties`) zawierają wspieraną wersję API, typ, nazwę i producenta GPU oraz jego limity - numeryczne wartości, które muszą być przestrzegane przez program podczas jego użytkowania. Przykładowo limit `maxImageDimension2D` definiuje najwyższą obsługiwana wysokość lub szerokość obrazu 2D. Z kolei funkcjonalności `VkPhysicalDeviceFeatures` zawierają długą listę wartości logicznych, które opisują dokładnie możliwości urządzenia. Przykładowo wartość `tessellationShader` oznacza wsparcie shaderów wyliczania teselacji.

Funkcja instancji `vkEnumeratePhysicalDevices()` zwraca listę dostępnych urządzeń fizycznych. Funkcje `vkGetPhysicalDeviceProperties2()` i `vkGetPhysicalDeviceFeatures2()`¹ pozwalają na określenie kolejno właściwości i funkcjonalności urządzenia fizycznego. Funkcja `vkEnumerateDeviceExtensionProperties()` zwraca listę wspieranych rozszerzeń urządzenia.

¹ Te funkcje są częścią rozszerzenia instancji `VK_KHR_get_physical_device_properties2`, które zostało promowane w Vulkan 1.1. W przeciwieństwie do wcześniejszych funkcji `vkGetPhysicalDeviceProperties()` i `vkGetPhysicalDeviceFeatures()` używają one tzw. łańcucha `pNext` i wspierają odpytywanie właściwości i funkcjonalności wprowadzonych przez późniejsze wersje Vulkan oraz rozszerzenia.

Aplikacja musi wybrać z listy kandydatów urządzenie fizyczne, które wspiera wszystkie właściwości i funkcjonalności używane podczas działania aplikacji. Należy też wziąć pod uwagę wymagania wydajnościowe - gra komputerowa powinna wybrać GPU zamiast renderera programowego.

Urządzenie logiczne

Po wybraniu urządzenia fizycznego należy użyć go do stworzenia urządzenia logicznego. Reprezentuje ono sterownik graficzny urządzenia fizycznego i jest używane przez większość funkcji i poleceń Vulkan.

Podczas tworzenia urządzenia fizycznego należy zdefiniować używane kolejki oraz funkcjonalności i rozszerzenia urządzenia, których wsparcie było sprawdzane podczas wyboru urządzenia fizycznego. Dodatkowo w imię kompatybilności wstępnej powinno się ponownie podać listę używanych warstw. Jest to spowodowane przestarzałym i zlikwidowanym podziałem na warstwy instancji i urządzenia - obecnie wszystkie warstwy są traktowane jako oba rodzaje.

Kolejki

Podczas tworzenia urządzenia logicznego sterownik graficzny automatycznie tworzy żądane kolejki.

Kolejki są używane do wykonywania na urządzeniu fizycznym polecień zawartych w buforach polecień wysłanych do kolejki funkcją `vkQueueSubmit()`. Funkcja zwraca kontrolę do aplikacji, nie czekając na zakończenie wykonywania bufora polecień na GPU - wymagana jest synchronizacja GPU z CPU przy pomocy tzw. ogrodzeń (ang. fence). Wykonanie buforów polecień może się odbywać poza kolejnością lub nakładać i wymaga synchronizacji przy pomocy semafor (ang. semaphore). Podobnie nie ma silnej gwarancji porządkowania wykonywania polecień należącego do pojedynczego bufora i wymaga jawniej synchronizacji używając barier potoku lub zdarzeń.

Każda kolejka należy do pewnej rodziny kolejek (`VkQueueFlagBits`) sygnalizując tym wsparcie pewnego rodzaju polecień:

- **GRAPHICS**: kolejka graficzna, wspiera polecenia rysowania `vkCmdDraw*()`,
- **COMPUTE**: kolejka obliczeniowa, wspiera polecenia GPGPU (General-Purpose Computing on GPU, obliczenia ogólnego przeznaczenia na GPU) `vkCmdDispatch*()` oraz polecenia śledzenia promieni (np. `vkCmdTraceRays*()`),
- **TRANSFER**: kolejka transferowa, wspiera polecenia transferu (np. `vkCmdCopyBuffer*()`, `vkCmdFillBuffer()`),
- **PARSE_BINDING**: kolejka zasobów chronionych, wspiera funkcję `vkQueueBindSparse()` dowieżającą do zasobu indywidualne strony pamięci,
- **PROTECTED**: kolejka pamięci chronionej, promowana w Vulkan 1.1, pozwala na ochronę pamięci zasobów.

Jedna kolejka może należeć do kilku rodzin. Przykładowo kolejki graficzne i obliczeniowe zawsze wspierają operacje transferu. Sterowniki mogą wykonywać bufore polecień wysłane do różnych kolejek asynchronicznie zapewniając skalowanie na wielordzeniowych GPU, dlatego warto pomyśleć o użyciu osobnych kolejek transferu, graficznych i obliczeniowych do kopowania danych i przeplatania polecień rysowania z obliczeniami GPGPU - pamiętając, że narzucający związek z synchronizacją może zniewelować poprawy wydajności.

Uchwyt kolejki urządzenia logicznego jest uzyskiwany używając funkcji `vkGetDeviceQueue()`.

Wskaźniki funkcji rozszerzeń

Użycie funkcji niebędących częścią używanej wersji API Vulkan i oferowanej przez wspierane rozszerzenia instancji i urządzenia wymaga pobrania wskaźników funkcji loadera używając funkcji instancji `vkGetInstanceProcAddress()`. Zwrócony wskaźnik nie musi bezpośrednio wskazywać na implementację oferowaną przez sterownik i może być funkcją loadera wykonującą dodatkową logikę dodaną przez załadowane warstwy. Funkcja `vkGetDeviceProcAddress()` pozwala na pominięcie loadera, co gwarantuje szybsze wywołania API, ale zwrócona funkcja może być używana tylko dla urządzenia logicznego użytego do pobrania jej.

Rozszerzenie VK_EXT_debug_utils

Podczas inicjalizacji Vulkan warto rozważyć użycie rozszerzenia `VK_EXT_debug_utils`, które pozwala na stworzenie komunikatora debugowania (ang. debug messenger) przechwytyującego wszystkie komunikaty wygenerowane przez loader, warstwy i sterownik Vulkan. Przechwycone komunikaty debugowania wraz z priorytetami są przekazywane do wywołania zwrotnego zdefiniowanego przez programistę, które może przykładowo logować wiadomość. Użycie debugera wspierającego warunkowe punkty przerwania (ang. conditional breakpoint) dla wiadomości o priorytecie `error` lub `fatal` pozwala na zatrzymanie działania programu tuż po zgłoszeniu błędu przez warstwy walidacji, co upraszcza proces debugowania.

Rozszerzenie pozwala też na dodawanie nazw do obiektów Vulkan funkcją `vkSetDebugUtilsObjectNameEXT()` oraz etykiet do regionów buforów poleceń funkcjami `vkCmdInsertDebugUtilsLabelEXT()`, `vkCmdBeginDebugUtilsLabelEXT()` i `vkCmdEndDebugUtilsLabelEXT()`.

Nazwy i etykiety są używane w wiadomościach debugujących i pokazywane przez zewnętrzne narzędzia takie jak RenderDoc [17], co znacznie upraszcza proces debugowania.

2.1.4. Zasoby

Vulkan wyróżnia dwa rodzaje zasobów: bufore (`VkBuffer`) i obrazy (`VkImage`).

Zasoby są widokami na pamięć GPU reprezentowaną przez alokację pamięci `VkDeviceMemory`. Fragment alokacji pamięci musi zostać dowiązany do nowo stworzonego zasobu przy użyciu funkcji `vkBindImageMemory()` lub `vkBindBufferMemory()`.

W kolejnych sekcjach przybliżono właściwości buforów, obrazów, widoków oraz próbników.

Bufory

Bufor to ciągły blok pamięci który może być odczytany i zapisywany przez GPU. Jest on najprostszym zasobem oferowanym przez Vulkan.

Stworzenie bufora wymaga ustalenia jego wielkości w bajtach oraz flag użycia bufora `VkBufferUsageFlags`.

Flagi użycia bufora określają jego dozwolone użycie. Przykładowo:

- `TRANSFER_SRC`: może być źródłem polecenia transferu;
- `TRANSFER_DST`: może być miejscem docelowym polecenia transferu;
- `UNIFORM_TEXEL_BUFFER`: może być użyty do stworzenia widoku bufora magazynowego używanego przez deskryptory typu `UNIFORM_TEXEL_BUFFER`;
- `STORAGE_TEXEL_BUFFER`: może być użyty do stworzenia widoku bufora magazynowego używanego przez deskryptory typu `UNIFORM_TEXEL_BUFFER`;

- *UNIFORM_BUFFER*: może być buforem uniform (pozwalającym tylko na odczyt) opisanym de-skryptorem typu *UNIFORM_BUFFER*;
- *STORAGE_BUFFER*: może być buforem magazynowym (pozwalającym na odczyt i zapis) opisanym deskryptorem typu *STORAGE_BUFFER*;
- *INDEX_BUFFER*: może być dowiązany jako bufor indeksów poleceniem *vkCmdBindIndexBuffer()*;
- *VERTEX_BUFFER*: może być dowiązany jako bufor wierzchołków poleceniem *vkCmdBindVertexBuffer()*;
- *INDIRECT_BUFFER*: może być używany jako bufor parametrów rysowania pośredniego w poleceniach *vkCmdDrawIndirect()* i *vkCmdDrawIndexedIndirect()*.

Widoki bufora *VkBufferView* pozwala shaderom na traktowanie ich zawartości jako obrazy. Jednak z powodu ich prostoty dostęp do buforów w praktyce jest odbywa się bezpośrednio używając deskryptorów typu *UNIFORM_BUFFER* lub *STORAGE_BUFFER*.

Obrazy

Obraz, podobnie jak bufor, reprezentuje ciągły blok pamięci, ale jego wewnętrzna struktura jest o wiele bardziej skomplikowana i wymaga wcześniejszego ustalenia następujących informacji tworzenia:

- typ (*VkImageType*),
- rozmiar (*VkExtent*),
- liczba warstw,
- flagi tworzenia (*VkImageCreateFlags*),
- liczba poziomów mipmap,
- format (*VkFormat*),
- liczba próbek na tekSEL (*VkSampleCountFlagBits*),
- kafelkowanie (*VkImageTiling*),
- flagi użycia (*VkImageUsageFlags*),
- początkowy układ (*VkUsageFlags*).

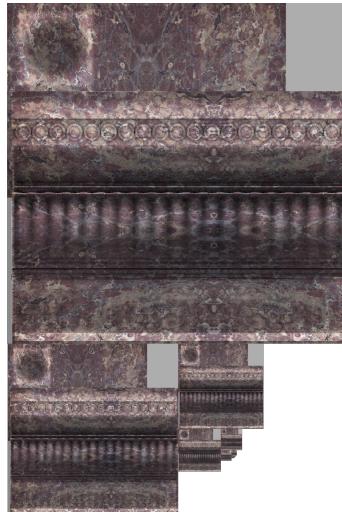
Typ obrazu określa jego liczbę wymiarów (1D, 2D lub 3D).

Rozmiar opisuje liczbę pikseli obrazu (tekSELi) wzdłuż każdego wymiaru (szerokość, wysokość i głębokość).

Obraz Vulkan może być traktowany jako tablica identycznych obrazów, której pojedynczy element jest zwany warstwą. Warstwy nie są liczone jako wymiar obrazu - obraz 2D z warstwami nie jest obrazem 3D.

Flagi tworzenia obrazu pozwalają na definicję dodatkowych funkcjonalności. Przykładowo flaga *CUBE_COMPATIBLE* pozwala na traktowanie obrazu z 6 kwadratowymi warstwami jako obrazu sześcienneego. Takie obrazy są często używane w technice renderowania sklepienia nieba używając sześcianu oteksturowanego przy pomocy sześciennnej tekstury skybox przedstawiającej niebo.

Mipmapa to kopia obrazu z każdym wymiarem zmniejszonym dwukrotnie. W obrazie z mipmapami pierwotny obraz jest traktowany jako mipmapa poziomu zerowego, z której generowane są mipmapy kolejnych poziomów aż do osiągnięcia wymiaru 1x1x1. Przykładowo obraz 2.3 przedstawia obraz 2D 1024x1024 z modelu Sponza [18] i jego 10 automatycznie wygenerowanych mipmap. Obraz posiadający mipmapy zajmuje więcej pamięci, ale pozwala na użycie filtrowania mipmapowego, które jest popularną



Rysunek 2.3: Obraz 2D 1024x1024 z sceny Sponza [18] i jego 10 mipmap (opracowanie własne)

techniką zwiększającą prędkość i jakość renderowania poprzez wprowadzenie nowych metod filtrowania podczas próbkowania obrazu.

Format (*VkFormat*) definiuje rozmiar, strukturę i sposób kodowania podczas zapisu i dekodowania podczas próbkowania pojedynczego piksela obrazu. Przestrzeń kolorów (*VkColorSpaceKHR*) definiuje metodę interpretacji pikseli obrazu przez silnik prezentacji podczas prezentacji obrazu. Przestrzeń kolorów liniowa (niesprecyzowana) jest używana w obliczeniach shaderów. Przestrzeń kolorów SRGB jest przeznaczona do wyświetlania na monitorach i jest powszechnie używana w teksturach. Przykładowe często używane formaty i ich użycie:

- *R8G8B8A8_UNORM*: 4 komponenty koloru B,G,R,A, z których każdy zajmuje 8 bitów i jest interpretowany jako znormalizowana wartość bez znaku (8-bitowa liczba zmiennoprzecinkowa pomiędzy 0 i 1), używany przez tekstury pozaekranowe,
- *B8G8R8A8_SRGB*: podobny do poprzedniego, ale podczas próbkowania GPU przeprowadza automatyczną konwersję z SRGB do przestrzeni liniowej (odwrotna konwersja podczas zapisu), używany przez prezentowalne obrazy łańcucha wymiany,
- *D32_SFLOAT_S8_UINT*: 32-bity komponentu głębi (liczba zmiennoprzecinkową ze znakiem), 8-bitów komponentu szablonu (liczba całkowita bez znaku), używany przez bufore głębi.

Liczba próbek na teksel większa niż 1 jest wymagana podczas implementacji technik antialiasingu takich jak multisampling pozwalających na wygładzanie zrasteryzowanych prymitywów [1].

Kafelkowanie obrazu (*VkImageTiling*) definiuje ułożenie tekselów w pamięci GPU i może być liniowe lub optymalne. W kafelkowaniu liniowym teksele są uszeregowane w pamięci wierszami (ang. row-major order) podobnie jak tablicach dwuwymiarowych języka C. Vulkan wspiera też kafelkowanie optymalne, w którym sterownik samodzielnie wybiera kafelkowanie obrazu na podstawie jego zawartości, co ma na celu zwiększenie prędkości dostępu poprzez poprawę lokalności przestrzennej w pamięci podręcznej GPU.

Flagi użycia obrazu określają jego zamierzone użycie. Przykładowo:

- *TRANSFER_SRC*: może być źródłem polecenia transferu;
- *TRANSFER_DST*: może być miejscem docelowym polecenia transferu;
- *SAMPLED*: może być użyty do stworzenia widoku próbkowanego obrazu używanego przez deskryptory typu *SAMPLED*;

- *STORAGE*: może być użyty do stworzenia widoku obrazu magazynowego używanego przez deskryptory typu *STORAGE*;
- *COLOR_ATTACHMENT*: może być dodaniem koloru przebiegu renderowania;
- *DEPTH_STENCIL_ATTACHMENT*: może być dodaniem głębi/szablonu przebiegu renderowania.

Obrazy istnieją w pamięci GPU w układach zdefiniowanych przez sterownik. Każdy układ ogranicza zbiór możliwych operacji na obrazie w zamian za optymalizację dozwolonych operacji.

Układ obrazu nie jest tym samym co jego kafelkowanie - rodzaj używanego kafelkowania nie może być zmieniany po utworzeniu obrazu, a układ obrazu jest zwykle często zmieniany podczas działania programu aby pozwolić GPU na optymalizację sposobu użycia obrazu.

Przykładowe układy obrazu i dozwolone użycie:

- *UNDEFINED*: zawartość pamięci obrazu jest niesdefiniowana, początkowy układ nowego obrazu;
- *GENERAL*: wszystkie rodzaje dostępu, nieoptymalny
- *COLOR_ATTACHMENT_OPTIMAL*: dodanie koloru przebiegu renderowania,
- *DEPTH_STENCIL_ATTACHMENT_OPTIMAL*: dodanie głębi/szablonu przebiegu renderowania pozwalające na odczyt (testy głębi i szablonu) i zapis (wyjście głębi),
- *DEPTH_STENCIL_READ_ONLY_OPTIMAL*: dodanie głębi/szablonu przebiegu renderowania pozwalające tylko na odczytu,
- *SHADER_READ_ONLY_OPTIMAL*: odczyt w shaderach jako próbkowany obraz,
- *TRANSFER_SRC_OPTIMAL*: obraz źródłowy w poleceniu transferu,
- *TRANSFER_DST_OPTIMAL*: obraz docelowy w poleceniu transferu,
- *PRESENT_SRC_KHR*: prezentowanie obrazu łańcuchem wymiany.

Dozwolone układy obrazu mogą zależeć od wcześniej sprecyzowanych flag użycia. Przykładowo układ *SHADER_READ_ONLY_OPTIMAL* wymaga flagi użycia *SAMPLED*.

Obrazy, w przeciwieństwie do buforów, nie są używane bezpośrednio i zawsze wymagają utworzenia widoku *VkImageView* opisującego używane części obrazu - warstwy i poziomy mipmap - oraz sposób interpretacji tekscela - możliwe aliasowanie formatu obrazu lub remapowanie jego komponentów.

Przykładami użycia widoków obrazów są aktualizacja deskryptorów i specyfikacja dowiązań przebiegu renderowania.

Próbniki

Teksele obrazu (lub rzadziej bufora) zwykle nie są bezpośrednio odczytywane w shaderach. Zamiast tego próbkowane obrazy są używane jako tekstury, które mogą być tylko odczytywane pośrednio używając próbników GPU.

Próbnik GPU przyjmuje od shadera koordynaty tekstury i używa ich do obliczenia i zwrócenia próbki będącej wynikiem automatycznego procesu filtrowania tekstury mieszającej wartość adresowanego współrzędnymi tekstury tekscela z jego sąsiadami w warstwie lub poziomach mipmap. Przykładowo większość GPU wspiera następujące metody filtracji:

- najbliższego sąsiada: próbka jest równa tekscelowi najbliższemu współrzędny tekscury;
- dwuliniowe: próbka jest średnią czterech najbliższych tekscelów;
- trójliniowe: rozszerzenie filtrowania dwuliniowego o liniową interpolację dwóch najbliższych poziomów mipmap;

- anizotropowe: rozszerzenie filtrowanie trójliniowe uwzględniające większą liczbę poziomów mip-map, implementacja zależna od sterownika.

Stan próbnika GPU używanego przez shader jest zaszyty w obiekcie próbnika `VkSampler` i jego następujące parametry tworzenia mogą być użyte do sterowania filtrowaniem:

- `magFilter`: filtr powiększania `VkFilter`;
- `minFilter`: filtr pomniejszania `VkFilter`;
- `mipmapMode`: tryb mipmapowania `VkSamplerMipmapMode`;
- `addressModeU`, `addressModeV`, `addressModeW`: tryb adresowania współrzędnych tekstur poza przedziałem $[0, 1]$ `VkSamplerAddressMode`;
- `anisotropyEnable`: włączenie filtrowania anizotropowego, wymaga funkcjonalności urządzenia Vulkan 1.0 `samplerAnisotropy`;
- `maxAnisotropy`: maksymalna wartość anizotropii, musi być mniejsza od limitu `maxSamplerAnisotropy`.

2.1.5. Łańcuch wymiany

Łańcuch wymiany jest reprezentowany przez obiekt `VkSwapchainKHR` będący częścią rozszerzenia WSI `VK_KHR_swapchain`. Można o nim myśleć jako o tablicy prezentowalnych obrazów udostępnianych aplikacji.

Prezentowalny obraz to obraz należący do powierzchni okna, który może być używany do prezentacji obrazu, czyli aktualizacji powierzchni okna zawartością procesu renderowania.

Dodatkowo łańcuch wymiany może być używany do synchronizacji pionowej (vertical synchronization, V-sync), czyli synchronizacji prezentacji obrazów z częstotliwością odświeżania ekranu, której brak powoduje rozrywanie obrazu - korupcję polegającą na jednoczesnym wyświetlaniu zawartości kilku klatek w tym samym czasie.

Program nie może bezpośrednio prezentować obrazu. Zamiast tego musi on:

- Pobrać tablicę uchwytów prezentowalnych obrazów funkcją `vkGetSwapchainImagesKHR()`,
- Wybrać dostępny prezentowalny obraz z tablicy uchwytów przy użyciu indeksu zwróconego przez funkcję `vkAcquireNextImageKHR()`,
- Wyrenderować scenę do dostępnego prezentowalnego obrazu przy użyciu funkcji `vkQueueSubmit()`,
- Oddać wyrenderowany obraz łańcuchowi wymiany funkcją `vkQueuePresentKHR()`.

Każdy z tych kroków wymaga synchronizacji z krokiem następnym przy użyciu semaforów. Funkcja `vkAcquireNextImageKHR()` sygnalizuje *semafor dostępności obrazu*, na który czeka funkcja `vkQueueSubmit()`. GPU zaczyna renderowanie dopiero wtedy, gdy prezentowany obraz nie jest używany przez okno. Funkcja `vkQueueSubmit()` sygnalizuje *semafor zakończenia renderowania*, na który czeka funkcja `vkQueuePresentKHR()` - okno może zacząć prezentować wynik renderowania dopiero wtedy, gdy GPU zakończył wykonywanie poleceń.

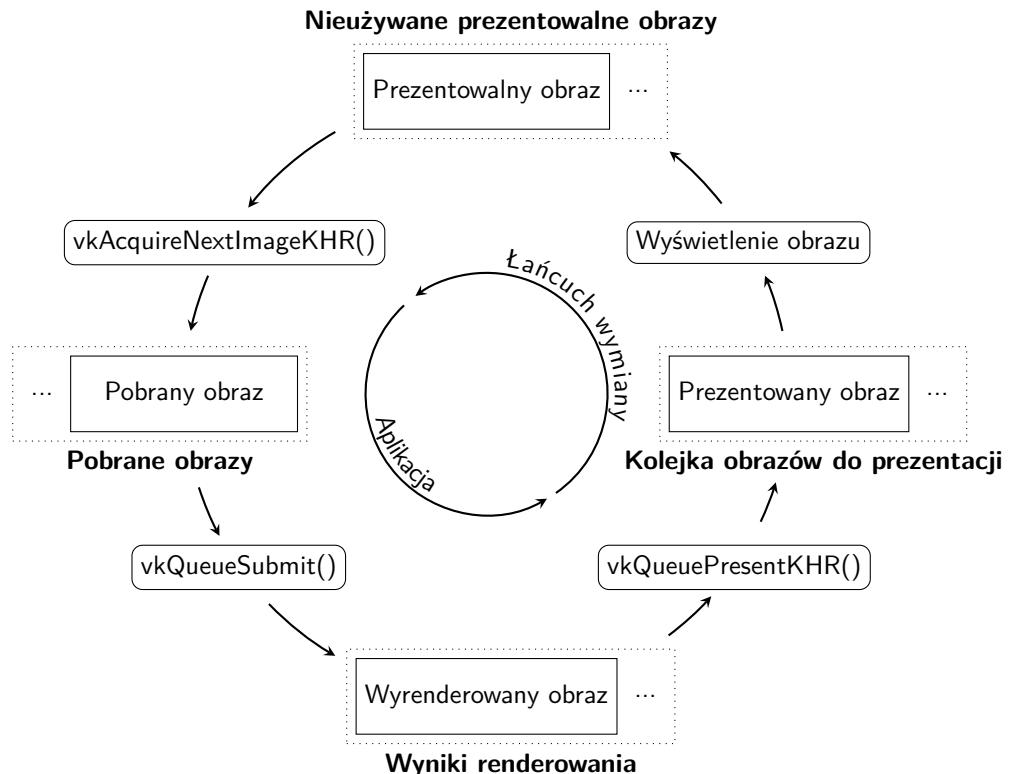
Prezentacja funkcją `vkQueuePresentKHR()` wymaga użycia uchwytu kolejki prezentacji, która jest dowolną kolejką wspierającą prezentację. Nie istnieje osobna rodzina kolejek prezentacji, zwykłe kolejki graficzne deklarują wsparcie, które może zostać potwierdzone funkcją `vkGetPhysicalDeviceSurfaceSupportKHR()`.

Okno wyświetla tylko jeden prezentowalny obraz na raz, ale istnieje możliwość umieszczania kilku obrazów w kolejce do prezentacji. Aktualnie prezentowany obraz jest często nazywany *buforem przednim*

(front buffer), a reszta obrazów w kolejce od prezentowania jest zwana *buforami tylnymi* (back buffers).

Część sterownika graficznego zwana silnikiem prezentacji wybiera z kolejki obraz służący jako bufor przedni i używa go do prezentacji. Po zakończeniu prezentacji obraz zostaje oznaczony jako nieużywany i może być ponownie pobrany przez program.

Diagram 2.4 ilustruje cykl życia obrazu łańcucha wymiany.



Rysunek 2.4: Cykl życia obrazu łańcucha wymiany (opracowanie własne)

Informacje tworzenia łańcucha wymiany muszą być wspierane przez powierzchnię okna. Wymagane jest ustalenie trybu prezentacji (funkcja `vkGetPhysicalDeviceSurfacePresentModesKHR()`), liczba prezentowalnych obrazów i ich rozmiar (funkcja `vkGetPhysicalDeviceSurfaceFormatsKHR()`) oraz ich formatu i przestrzeń kolorów (funkcja `vkGetPhysicalDeviceSurfaceCapabilitiesKHR()`).

Tryb prezentacji (`VkPresentModeKHR`) definiuje dokładny mechanizm działania silnika prezentacji:

- **IMMEDIATE**: wyrenderowane obrazy są natychmiastowo prezentowane. Brak synchronizacji pionowej może powodować rozrywanie obrazu.
- **FIFO**: łańcuch wymiany zachowuje się jak kolejka FIFO. Przed odświeżeniem ekranu obraz z przodu kolejki jest usuwany i prezentowany. Wyrenderowane obrazy są dodawane na koniec kolejki. Jeśli kolejka jest pełna, to program jest blokowany na funkcji `vkQueuePresentKHR()` i musi czekać na zwolnienie miejsca w kolejce. Ten tryb zapewnia synchronizację pionową i jego dostępność jest gwarantowana przez specyfikację Vulkan. Jednak w sytuacji, w której GPU renderuje obrazy szybciej, niż ekran je prezentuje, program jest blokowany.
- **FIFO_RELAXED**: podobny do **FIFO**, ale prezentacja obrazu może pomijać synchronizację pionową wywołując rozrywanie gdy kolejka jest pełna, ale zmniejszając czas blokowania aplikacji,
- **MAILBOX**: podobny do **FIFO**, ale zamiast blokowania programu gdy kolejka jest pełna, starsze obrazy w kolejce są zastępowane przez nowsze. Ten tryb zapewnia synchronizację pionową i

zgodnie ze specyfikacją Vulkan gwarantuje, że program nie jest blokowany podczas prezentacji oraz może zawsze pobrać nieużywany obraz z łańcucha wymiany, ale tylko pod warunkiem, że liczba prezentowalnych obrazów jest większa od minimalnej liczby wymaganej przez powierzchnię okna.

Po stworzeniu łańcucha obrazów aplikacja może pobrać tablicę uchwytów jego prezentowalnych obrazów funkcją `vkGetSwapchainImagesKHR()` - należy oczywiście pamiętać, że pomimo posiadania uchwytu obrazu może być on używany dopiero, gdy jego indeks jest zwrocony przez funkcję `vkAcquireNextImageKHR()`. Następnie należy utworzyć widoki prezentowalnych obrazów, które będą później używane do renderowania do nich.

Istnieją sytuacje, w których łańcuch wymiany musi być odtworzony (zniszczony i stworzony). Funkcje `vkAcquireNextImageKHR()` i `vkQueuePresentKHR()` mogą zwrócić rezultat `ERROR_OUT_OF_DATE_KHR` oznaczający, że powierzchnia okna zmieniła się w taki sposób, że nie jest już kompatybilna z łańcuchem wymiany. Aplikacja może chcieć odtworzyć łańcuch wymiany także dla rezultatu `SUBOPTIMAL_KHR` oznaczającego, że rozmiar obrazów łańcucha wymiany przestał dokładnie pokrywać się z powierzchnią okna i prezentacja musi przeprowadzać dodatkowe operacje skalowania. Najczęstszym sprawcą obu tych sytuacji jest zmiana rozmiaru okna.

2.1.6. Bufory poleceń

Wszystkie operacje do wykonania na GPU są grupowane w obiektach Vulkan zwanych buforami poleceń `VkCommandBuffer`.

Bufory poleceń są alokowane z wcześniej stworzonej puli poleceń `VkCommandPool`.

Bufor poleceń jest zawsze w jednym z następujących stanów:

- początkowy (ang. initial),
- nagrywania (ang. recording),
- wykonywalny (ang. executable),
- oczekujący: (ang. pending),
- nieprawidłowy (ang. invalid).

Nowo zaallokowany bufor poleceń jest całkowicie pusty - nie zawiera poleceń i znajduje się w stanie początkowym. Bufory poleceń mogą być ponownie sprowadzone do stanu początkowego poprzez zresetowanie funkcją `vkResetCommandBuffer()`. Podobnie zresetowanie całej puli poleceń funkcją `vkResetCommandPool()` resetuje wszystkie zaallokowane z niej bufory poleceń.

Bufor poleceń przechodzi w stan nagrywania poprzez wywołanie funkcji `vkBeginCommandBuffer()`. W tym stanie polecenia `vkCmd*`() mogą być używane do nagrywania (dodawania) operacji do wykonania na GPU. Efekty nagrywanych poleceń mogą być nieformalnie podzielone na:

- zmianę wewnętrznego stanu GPU:
 - zmiana dowiązanego obiektu `vkCmdBind*`(),
 - dynamiczne ustawienie stanu `vkCmdSet*`(),
- wykonanie operacji GPU wspieranych przez rodzinę kolejki.

Bufor poleceń przechodzi w stan wykonywalny po wywołaniu funkcji `vkEndCommandBuffer()` kończącej nagrywanie.

Bufor poleceń w stanie wykonywalnym może zostać wysłany funkcją `VkQueueSubmit()` do kolejki w celu asynchronicznego wykonania na GPU. Bufor poleceń przechodzi ze stanu wykonywalnego do stanu oczekującego i po zakończeniu wykonywania na GPU wraca do stanu wykonywalnego.

Modyfikacja bufora polecień w stanie oczekującym, zniszczenie używanych zasobów bądź rozpoczęcie nagrywania z flagą użycia *ONE_TIME_SUBMIT* powoduje przejście do stanu nieprawidłowego. Bufor polecień w tym stanie może być tylko zresetowany bądź zwolniony.

2.1.7. 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 [19], ale może być intuicyjnie rozumiana jako struktura zawierająca wskaźnik do 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 tablicę poprzez zbiory deskryptorów (*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 zaallokowanych 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 próbkiowanego obrazu, może być próbkiowany w shaderze zmienną typu *sampler2D*;

- *STORAGE_IMAGE*: widok obrazu magazynowego, może być odczytywany w shaderze zmienną typu *image2D*;
- *COMBINED_IMAGE_SAMPLER*: próbkowany obraz, pojedynczy deskryptor jest skojarzony zarówno z próbnikiem, jak 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 polecen. 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 polecen *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 małej ilości często zmienianych pomiędzy poleceniami danych. 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 dowiązania 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.

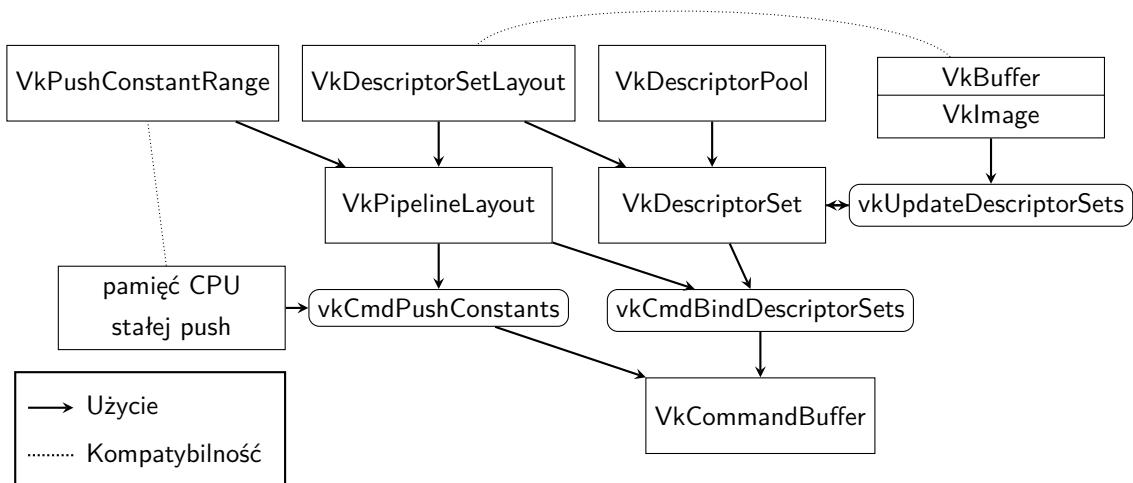
Liczba poszczególnych typów deskryptorów uwzględnionych w potoku renderowania jest ograniczona limitami urządzenia fizycznego. Limit *maxPerStageResources* to maksymalna liczba zasobów, które mogą być dostępne dla pojedynczego etapu cieniowania. Rodzina limitów *maxDescriptorSet**, gdzie * to typ deskryptora, 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 dowiązania ich do bufora poleceń przy użyciu polecenia `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 używającymi deskryptorów i stałych push zostały przedstawione na diagramie 2.5. Widać na nim, że układ zbioru deskryptorów używany do tworzenia zbioru deskryptorów musi być kompatybilny z zasobami używanymi aktualizacji. Relacja ta nie jest bezpośrednio wyrażona parametrami tworzenia - deskryptory mogą być aktualizowane dowolnymi zasobami, o ile ich typy są kompatybilne.



Rysunek 2.5: Relacje pomiędzy obiektami Vulkan używanymi do zarządzania deskryptorami (opracowanie własne)

Dostęp do zasobów w shaderach

Po dowiązaniu 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 dowiązaniem o numerze *x* ze zbioru o numerze *y* ma formę przedstawioną na listingu 2.3.

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

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

Listing 2.3: Kwalifikator układu dla bufora uniform

Analogicznie kwalifikator układu dla tablicy deskryptorów typu COMBINED_IMAGE_SAMPLER o rozmiarze r z dowiązaniem o numerze x ze zbioru o numerze y próbkiwanych obrazów 2D ma formę przedstawioną na listingu 2.4.

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

Listing 2.4: Kwalifikator układu dla tablicy tekstur

Układ pamięci scalar dla buforów i stałych push

Układ pamięci definiuje wyrównania pól w strukturze znajdującej się w opisywanym deskryptorem pamięci. Używany układ musi być zadeklarowany w kwalifikatorze układu dla buforów i stałych push.

OpenGL wspiera dwa standardy układów: *std140* i *std430*. Vulkan odziedziczył te układy i dodatkowo wprowadził *scalar*. Każdy kolejny standard pozwala na ciasniejsze upakowanie pól, co skutkuje mniejszym zużyciem pamięci GPU.

Układ pamięci scalar wymaga wsparcia funkcjonalności urządzenia Vulkan 1.2 *scalarBlockLayout*, które w czasie pisania pracy jest wspierane przez więcej niż 98% urządzeń [20]. Dodatkowo kod GLSL shadera musi posiadać dyrektywę preprocesora przedstawioną na listingu 2.5.

```
#extension GL_EXT_scalar_block_layout : require
```

Listing 2.5: Dyrektywa preprocesora dla układu pamięci skalar

W przeciwieństwie do swoich poprzedników, układ scalar ma bardzo proste wymagania dotyczące wyrównań pól:

- dla typu skalarnego jest równe jego wielkości,
- dla wektora bądź macierzy jest równe wyrównaniu jego komponentu,
- dla tablicy jest równe wyrównaniu jego elementu,
- dla struktury jest równe największemu wyrównaniu jej pól.

Powysze zasady skutkują maksymalnym upakowaniem pól dla buforów używających tylko standardowych 32-bitowych skalarów mających to samo wyrównanie, co obejmuje większość shaderów nieużywających rozszerzeń takich jak *GL_EXT_shader_16bit_storage* wprowadzające 16-bitowe skalary.

2.1.8. 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 dowiązań. Z powodu swojej użyteczności rozszerzenie to zostało promowane w Vulkan 1.2. W kolejnych sekcjach opisano nowe funkcjonalności.

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 komilacji. 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 wywołaniach shadera spowodowanych przez pojedyncze polecenie rysowania. Użycie różnych indeksów jest błędem

i może skutkować niezdefiniowanym zachowaniem. Przykładem tego rodzaju indeksowania jest użycie indeksu tekstuury w stałej push lub buforze uniform. Wymaga ona wsparcia następujących funkcjonalności urządzenia Vulkan 1.0:

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

Nie jednolite 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 tekstuury 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 [20].

Nie jednolite dynamiczne indeksowanie wymaga zadeklarowania rozszerzenia SPIR-V *SPV_EXT_descriptor_indexing*. Może być to uczynione z poziomu kodu GLSL przez dodanie dyrektywy preprocesora przedstawionej na listingu 2.6.

```
#extension GL_EXT_nonuniform_qualifier : require
```

Listing 2.6: Dyrektywa preprocesora dla niejednolitych indeksów

Dodatkowo każde użycie niejednolitego indeksu powinno być oznaczone funkcją *nonuniformEXT* przedstawioną na listingu 2.7.

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

Listing 2.7: Próbkowanej używając niejednolitego indeksu

Wymóg jednolitości podczas indeksowania deskryptorów był spowodowany ograniczeniami poprzednich generacji GPU - w modelu renderowania OpenGL dostęp do dowiązanych tekstur odbywał się pośrednio poprzez jednostki teksturowe, które były widoczne przez wszystkie wywołania shaderów i nie pozwalały na zmianę dołączonych tekstur podczas wykonywania polecenia rysowania [21].

Aktualizacja deskryptorów po dowiązaniu

Domyślnie deskryptory nie mogą być aktualizowane po nagraniu ich dowiązania w buforze polecień, przez co aplikacja musi mieć pełną wiedzę o wszystkich używanych zasobach w trakcie nagrywania buforów polecień. 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 polecień.

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

- *descriptorBindingUniformBufferUpdateAfterBind*: bufore uniform,
- *descriptorBindingSampledImageUpdateAfterBind*: próbkowane obrazy,
- *descriptorBindingStorageBufferUpdateAfterBind*: bufore magazynowe,

- *descriptorBindingStorageImageUpdateAfterBind*: obrazy magazynowe.

W czasie pisania pracy powyższe funkcjonalności są wspierane przez ponad 90% urządzeń wyłączając bufory uniform niewspierane przez ok. 40% platform [20].

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.

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 na obsługę sytuacji, w której dokładna liczba deskryptorów wymaganych do opisania zasobów nie jest znana podczas tworzenia układu zbioru deskryptorów.

Wsparcie dowiązań o zmiennej wielkości wymaga funkcjonalności urządzenia Vulkan 1.2 *descriptor-BindingVariableDescriptorCount*, która w czasie pisania pracy jest wspierana przez ponad 90% urządzeń [20].

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

Częściowo dowiązane deskryptory

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 zbiór 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 *descriptor-BindingPartiallyBound*, która w czasie pisania pracy jest wspierana przez ponad 94% urządzeń [20].

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

Nieograniczone tablice deskryptorów

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

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

Wsparcie nieograniczonych tablic deskryptorów wymaga funkcjonalności urządzenia Vulkan 1.2 *runtimeDescriptorArray*, która w czasie pisania pracy jest wspierana przez ponad 94% urządzeń [20].

Użycie tej funkcjonalności odbywa się poprzez pominięcie rozmiaru tablicy w kwalifikatorze układu w kodzie GLSL, co zostało przedstawione na listingu 2.8.

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

Listing 2.8: Kwalifikator układu dla nieograniczonej tablicy tekstur

Dostęp do zmiennej w GLSL się nie zmienia, ale wygenerowany kod SPIR-V używa rozszerzenia *SPV_EXT_descriptor_indexing* i typu *OpTypeRuntimeArray* zamiast *OpTypeArray*, co widać na listingu 2.9.

```
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_151
%textures2D =
    OpVariable %_ptr_UniformConstant__runtimearr_151 UniformConstant
```

Listing 2.9: Kod SPIR-V wygenerowany dla nieograniczonej tablicy deskryptorów

Tablica jest nieograniczona tylko z punktu widzenia kodu shadera - dowiązanie deskryptora wciąż posiada pewną wielkość zdefiniowaną podczas tworzenia zbioru deskryptorów. Dlatego też indeksowanie poza długością tablicy jest niezdefiniowanym zachowaniem.

2.1.9. Potoki graficzne i przebiegi renderowania

Renderowanie w Vulkan wymaga wysłania do kolejki graficznej bufora poleceń wypełnionego poleceniami rysowania.

Vulkan 1.2 oferuje następujące sześć wariantów polecenia rysowania:

- *vkCmdDraw()*: Polecenie rysowania pozwalające na wyrenderowanie dowiązanego bufora wierzchołków. Przyjmuje następujące parametry:

- *vertexCount*: liczba wierzchołków do narysowania,
- *instanceCount*: liczba instancji do narysowania,
- *firstVertex*: indeks pierwszego wierzchołka,
- *firstInstance*: indeks pierwszej instancji.

Specyfikacja *instanceCount* większej niż 1 pozwala na wyrenderowanie kilku kopii tej samej siatki w tym samym czasie. Shader wierzchołków może uzyskać dostęp do indeksu obecnie renderowanej instancji używając zmiennej *gl_InstanceIndex*, która może zostać użyta do indeksowania deskryptorów z danymi specyficznymi dla instancji takich jak macierz modelu. Ta technika jest użyteczna podczas renderowania powtarzających się elementów sceny takich jak trawy czy drzewa.

- *vkCmdDrawIndexed()*: Rozszerzenie *vkCmdDraw()* używające dowiązanego bufora indeksów do indeksowania dowiązanego bufora wierzchołków. Użycie bufora indeksów pozwala na zmniejszenie duplikacji w buforze wierzchołków.
- *vkCmdDrawIndirect()*: Polecenie rysowanie pośredniego. Zachowuje się podobnie do *vkCmdDraw()*, ale jego parametry nie są zaszyte w poleceniu i są odczytywane przez GPU pośrednio z bufora parametrów rysowania pośredniego zawierającego tablicę struktur *VkDrawIndirectCommand*, którego rozmiar jest przekazywany parametrem *drawCount*. Użycie *drawCount* większego niż 1 wymaga funkcjonalności urządzenia *multiDrawIndirect* i skutkuje poleceniem wielokrotnego rysowania pośredniego. Takie polecenie potencjalnie odpowiada kilku następujących bezpośrednio po sobie poleceniom *vkCmdDraw()* używających niezmienionego stanu GPU, co pozwala na renderowanie skomplikowanych scen przy użyciu jednego polecenia rysowania. Dodatkowo znajdujący się w pamięci GPU bufor parametrów może być wypełniany używając operacji GPGPU (np. polecienniem *vkCmdDispatch()*), co pozwala na przyśpieszenie renderowania poprzez ograniczenie transferów z CPU do GPU.
- *vkCmdDrawIndexedIndirect()*: Wariant łączący polecenia *vkCmdDrawIndirect()* i *vkCmdDrawIndexed()*.
- *vkCmdDrawIndirectCount()*: Polecenie rysowania pośredniego z pośrednią liczbą poleceń. Zachowuje się podobnie do *vkCmdDrawIndirect()*, ale parametr *drawCount* jest zastępowany odczytem z drugiego bufora parametrów zawierającym liczbę poleceń rysowania. To polecenie pozwala na elastyczniejszą generację poleceń rysowania na GPU - ich liczba nie musi być znana z góry.
- *vkCmdDrawIndexedIndirectCount()*: Wariant łączący polecenia *vkCmdDrawIndirectCount()* i *vkCmdDrawIndexed()*.

Wydajność renderowania może być zwiększała poprzez zmniejszenie całkowitej liczby poleceń rysowania, które muszą być wykonane przez GPU. Ten proces jest często nazywany optymalizacją polecień rysowania (ang. draw call optimization), w którym nieoptymalizowane polecenia rysowania są konwertowane na grupy (ang. batch) możliwe do wyrenderowania jednym zoptymalizowanym poleceniem rysowania. W Vulkan ten proces jest realizowany poprzez użycie funkcjonalności polecień rysowania takich jak instancjonowanie czy wielokrotne rysowanie pośrednie.

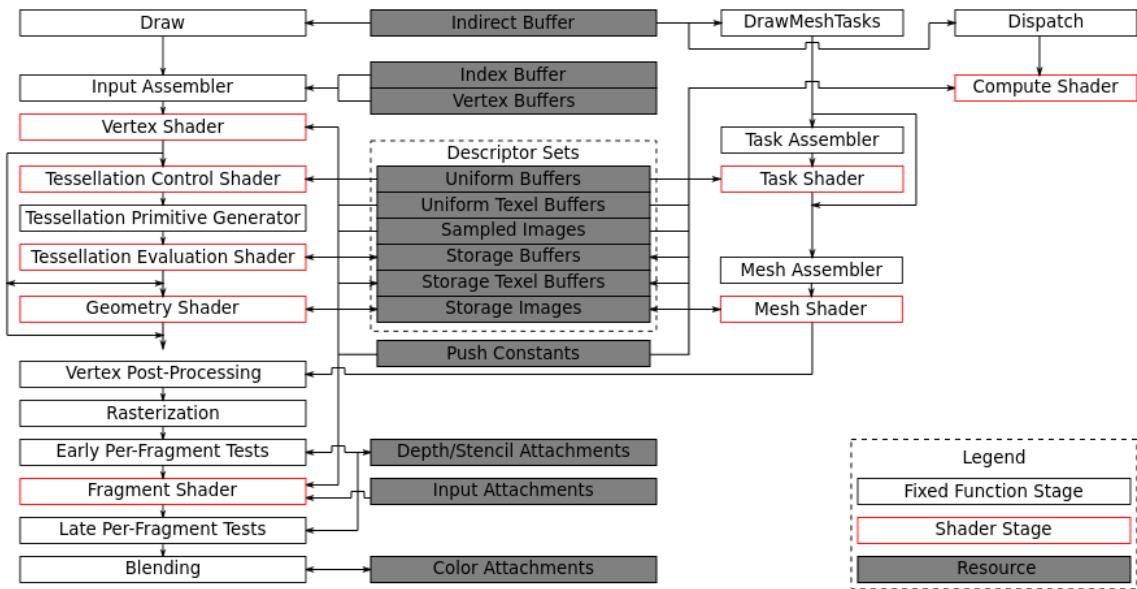
Pojedyncze polecenie rysowania nie zawiera wszystkich informacji potrzebnych do wyrenderowania dowiązanych zasobów i wymaga wcześniejszego stworzenia dwóch obiektów: potoku graficznego i przebiegu renderowania.

Potok graficzny

Termin „potok” może się odnosić do abstrakcyjnego modelu potoku opisującego proces przetwarzania polecień na GPU oraz od obiektu *VkPipeline* zawierającego jego wstępnie skonfigurowany stan.

Vulkan 1.2 wspiera trzy rodzaje potoków pozwalające na przetwarzanie odpowiedniego rodzaju polecień: potok graficzny, potok obliczeniowy oraz potok śledzenia promieni.

Każdy potok składa się z serii etapów przetwarzających dane dowiązanych zasobów oraz wyniki poprzedniego etapu. Każdy etap potoku jest albo etapem cieniowania, albo etapy funkcji stałych. Diagram 2.6 przedstawia wszystkie możliwe etapy potoku i wraz z używanymi przez nie zasobami.



Rysunek 2.6: Schemat blokowy potoku ze specyfikacji Vulkan [4]

Każdy etap cieniowania wykonuje pewien shader - program specyfikujący operacje wykonywane dla każdego przetwarzanego przez elementu (np. wierzchołka, punktu sterującego, fragmentu lub grupy roboczej). Kod źródłowy GLSL reprezentuje shader w formie tekstuowej przy użyciu języka programowania potoku graficznego składniowo zbliżonego do języka C. Kod bajtowy SPIR-V reprezentuje shader w formie ustandaryzowanej binarnej reprezentacji pośredniej. Jest on zwykle uzyskiwany poprzez komplikację kodu źródłowego w języku wyższego poziomu takiego jak GLSL. Moduł shaderów to obiekt *vkShaderModule* uzyskiwany poprzez komplikację kodu bajtowego SPIR-V do kodu binarnego w ISA GPU używając funkcję *vkCreateShaderModule()*. Po komplikacji moduł shadera jest w formie, która może być używana bezpośrednio przez etapy cieniowania.

Kod Vulkan może odnosić się do niektórych etapów potoku graficznego używając flag *VkPipelineStageFlags*. Przykładowo:

- *DRAW_INDIRECT*: odczyt buforów parametrów rysowania pośredniego,
- *VERTEX_INPUT*: wejście wierzchołków (odczyt dowiązanych buforów wierzchołków i indeksów),
- *VERTEX_SHADER*: shader wierzchołków,
- *TESSELLATION_CONTROL_SHADER*: shader sterowania teselacją,
- *TESSELLATION_EVALUATION_SHADER*: shader wyliczania teselacją,
- *GEOMETRY_SHADER*: shader geometrii,

- *EARLY_FRAGMENT_TESTS*: wczesne testy fragmentów (testy głębi i szablonu przed cieniowaniem fragmentów),
- *FRAGMENT_SHADER*: shader fragmentów,
- *LATE_FRAGMENT_TESTS*: późne testy fragmentów (testy głębi i szablonu po cieniowaniu fragmentów),
- *COLOR_ATTACHMENT_OUTPUT*: zapis do dołączeń kolorów końcowego wyniku renderowania.

Dodatkowe prymitywy synchronizacji często używają następujących etapów należących do potoku graficznego:

- *TOP_OF_PIPE*: pseudo-etap przed rozpoczęciem wykonywania polecenia,
- *BOTTOM_OF_PIPE*: pseudo-etap po zakończeniu wykonywania polecenia,
- *TRANSFER*: wykonywanie poleceń transferu.

Polecenia rysowania używają potoku graficznego, który jest tworzony funkcją *vkCreateGraphicsPipelines()*. Jej parametry tworzenia *VkGraphicsPipelineCreateInfo* to ogromna struktura specyfikująca konfigurację wszystkich etapów potoku graficznego używanych przez polecenia rysowania.

Parametry tworzenia potoku dla etapów cieniowania to lista struktur *VkPipelineShaderStageCreateInfo* zawierających moduły shaderów oraz układ potoku *VkPipelineLayout*. Parametry te pozwalają na pełną specyfikację wszystkich shaderów, deskryptorów i stałych push używanych przez polecenia rysowania.

Działanie etapów funkcji stałych jest sterowane parametrami tworzenia zawierającymi wskaźniki do struktur *VkPipelineCreateInfo*. Przykłady:

- stan wejścia wierzchołków *VkPipelineVertexInputStateCreateInfo* opisujący sposób odczytu wierzchołków z dowiązanych buforów oraz jego wewnętrzną strukturę widzianą przez shadery w postaci atrybutów wierzchołka;
- stan składania prymitywów wejściowych *VkPipelineInputAssemblyCreateInfo* sterujący łączeniem odczytanych wierzchołków w prymitywy używając topologii *VkPrimitiveTopology* i trybu restartu prymitywów;
- stan rasteryzacji *VkPipelineRasterizationStateCreateInfo* pozwalający na specyfikację takich parametrów jak kierunek rysowania (ang. winding order) trójkąta czy grubość linii;
- stan głębokości/szablonu *VkPipelineDepthStencilCreateInfo* definiujący testy głębokości i szablonu;
- stan mieszania kolorów *VkPipelineColorBlendStateCreateInfo* określający sposób mieszania kolorów będących wynikiem cieniowania fragmentów przed zapisem do dołączeń koloru.

Ostatecznie parametry tworzenia muszą też zawierać informacje dotyczące rodzaju przebiegu renderowania, z którymi potok graficzny jest kompatybilny. Zostanie to opisane w następnej sekcji.

Utworzony potok graficzny jest dowiązywany poleceniem *vkCmdBindPipeline()* i wpływa na wykonywanie wszystkich kolejnych nagranych poleceń rysowania.

Sam potok graficzny nie wystarcza do rozpoczęcia procesu renderowania - wymagane jest ustalenie jego wyjścia używając przebiegu renderowania.

Przebieg renderowania

Polecenia rysowania muszą być nagrywane w ramach pojedynczego przebiegu renderowania definiującego zbiór widoków obrazów nazywanych dołączaniami (ang. attachments) służących jako cele

renderowania wszystkich poleceń wykonywanych w ramach przebiegu.

Tradycyjny sposób definicji przebiegów renderowania w Vulkan wymaga stworzenia obiektów przebiegu renderowania *VkRenderPass* i bufora ramki *VkFramebuffer* kompatybilnych wzajemnie ze sobą oraz z potokiem *VkPipeline*. Pozwala to na szczegółowy opis procesu renderowania pozwalający sterownikom na maksymalną optymalizację, ale jest skomplikowane i wymaga tworzenia osobnych przebiegów renderowania i buforów ramki dla każdej zmiany obrazów służących jako dołączenia.

Definicja przebiegów renderowania została uproszczona rozszerzeniem *VK_EXT_dynamic_rendering*. Wprowadził one dynamiczne przebiegi renderowania niewymagające tworzenia dodatkowych obiektów Vulkan. Zamiast tego podczas tworzenia potoku graficznego do jego łańcucha *pNext* dodawana jest struktura *VkPipelineRenderingCreateInfoKHR* definiująca liczbę i formaty używanych dołączeń koloru i głębi/szablonu. Widoki obrazów służących jako dołączenia są specyfikowane dopiero podczas rozpoczęcia przebiegu renderowania polecienniem *vkCmdBeginRenderingKHR()*.

Technika renderowania sceny przy użyciu kilku przebiegów renderowania jest nazywana renderowaniem wieloprzebiegowym (ang. multi-pass rendering). Podczas implementacji jej używając Vulkan należy pamiętać o zapewnieniu odpowiedniej synchronizacji pomiędzy nimi. Przykładowo aplikacja musi nagrać odpowiednią barierę pamięci pomiędzy dwoma przebiegami, jeśli drugi przebieg próbuje teksturować będącą dołączeniem pierwszego przebiegu.

2.1.10. Rozszerzenie *VK_KHR_performance_query*

Rozszerzenie *VK_EXT_performance_query* dodaje mechanizm umożliwiający aplikacjom i narzędziom do profilowania sprawdzanie wartości tzw. liczników wydajności (ang. performance counter), które są udostępniane przez GPU i pozwalają na badanie obciążenia GPU.

Narzędzie RenderDoc [17] pozwala na przechwycenie liczników wydajności dla każdego polecenia, co znacznie upraszcza proces profilowania.

2.1.11. Synchronizacja

Sterowniki Vulkan, w przeciwieństwie do OpenGL, nie wykonują automatycznej synchronizacji pomiędzy operacjami używającymi zasobów wymagając od programisty używania poniżej opisanych prymitywów synchronizacji.

Bariery potoku

Vulkan nie zapewnia silnych gwarancji dotyczących kolejności wykonywania operacji na GPU. Przykładowo polecenia w buforze poleceń rozpoczynają wykonywanie w kolejności nagrywania, ale nie muszą być kończone w tej samej kolejności, co może powodować wyścigi podczas dostępu do współdzielonych zasobów.

Bariera potoku to prymityw synchronizacji definiowany polecienniem *vkCmdPipelineBarrier()* pozwalający na zdefiniowanie zależności wykonania oraz zależności pamięci pomiędzy poleceniami przed i po barierce.

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, które różnią się typem pamięci opisywanym przez zależność pamięci:

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

Semafory

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

Ogródzenia

Vulkan pozwala na bardzo prostą synchronizację GPU z CPU poprzez funkcje *vkQueueWaitIdle()* i *vkDeviceWaitIdle()* blokujące wykonywanie programu aż do zakończenia wszystkich operacji wykonywanych na kolejce bądź całym urządzeniu.

Ogródzenia (ang. fence) to obiekty *VkFence* pozwalające na bardziej drobnoziarnistą synchronizację poleceń wykonywanych w kolejce na GPU z CPU. Ogródzenie 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.

Zdarzenia

Zdarzenia (ang. event) to obiekty *VkEvent* pozwalające na ogólną synchronizację wykonywanych poleceń z innymi poleceniami bądź CPU. Funkcja *vkCmdSetEvents()* sygnalizuje zdarzenie po rozpoczęciu wykonywania źródłowych etapów potoku zależności wykonania. Wraz z funkcją *vkCmdWaitEvents()* pozwala na specyfikację pełnej zależności pamięci. Aplikacja może manualnie sygnalizować, resetować i czekać na zdarzenia na funkcjami *vkSetEvents()*, *vkResetEvent()* i *vkGetEventStatus()*, co pozwala na blokowanie GPU przez CPU i jest jedyną funkcjonalnością niemożliwą do zaimplementowania przez poprzednio opisane prymitywy synchronizacji, które powinny być wydajniejsze od elastyczniejszych zdań.

2.2. Renderowanie bez dowiązań

Renderowania bez dowiązań to grupa technik mająca na celu maksymalizację czasu GPU spędzanego na rzeczywistym renderowaniu, a nie na synchronizacji i zmianach stanu renderowania [22].

W Vulkan renderowanie bez dowiązań jest realizowane poprzez:

- eliminację zmiany stanu pomiędzy poleceniami rysowania,
- optymalizację poleceń rysowania,
- umożliwienia GPU bezpośredniego dostępu do tekstur i buforów poprzez niejednolite dynamiczne indeksowanie deskryptorów.

Eliminacja zmiany stanu pomiędzy poleceniami rysowania sprowadza się do dowiązania wszystkich zasobów na początku bufora polecień. Rozszerzenie *VK_EXT_descriptor_indexing* zostało zaprojektowane z myślą o tworzeniu jednego dużego zbioru deskryptorów dowiązywanego poleceniem *vkCmdBindDescriptorSets()* i używanego przez wszystkie przebiegi renderowania. Bufory wierzchołków i indeksów mogą być skonsolidowane na CPU w pojedyncze duże bufore dowiązane poleceniami *vkCmdBindVertexBuffers()* i *vkCmdBindIndexBuffers()*. Polecenia rysowania mogą używać parametrów poleceń rysowania takich jak *firstIndex* i *vertexOffset* do uzyskania dostępu do różnych części buforów geometrii bez zmiany stanu GPU.

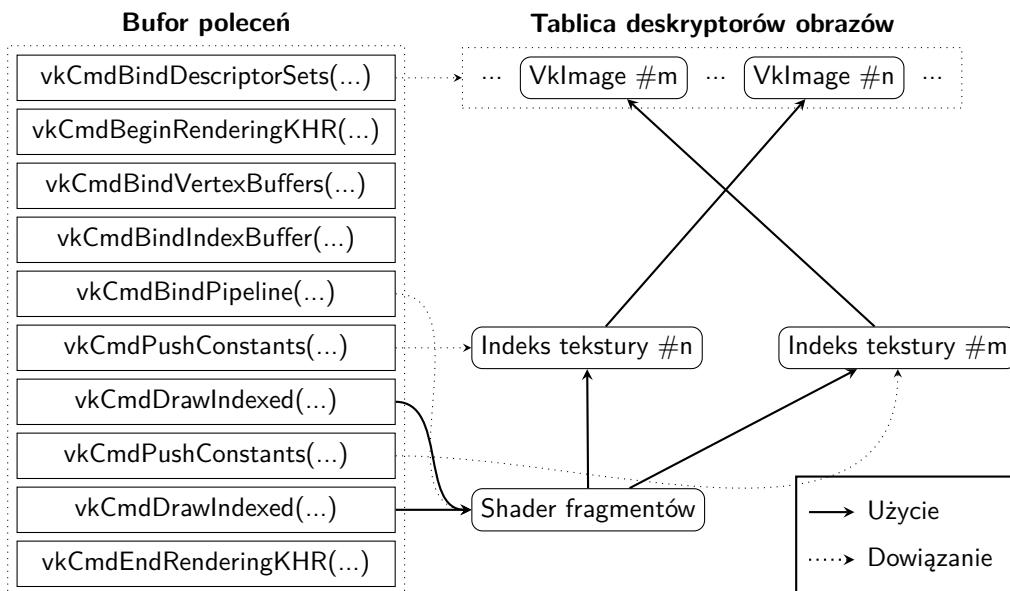
Optymalizacja poleceń rysowania sprowadza się do nagrania polecenia wielokrotnego rysowania pośredniego *vkCmdDrawIndexedIndirect()*. Używany przez niego bufor parametrów rysowania pośredniego może być zminimalizowany poprzez grupowanie polecień - dwa polecenia renderujące ten sam prymityw mogą być zastąpione jednym poleceniem rysującym dwie instancje.

Niestety użycie pojedynczego zoptymalizowanego polecenia rysowania renderującego scenę zawierającą obiekty różniące się teksturami jest utrudniane przez wymóg jednolitości indeksowania deskryptorów.

Tradycyjnie podczas renderowania obiektów każda zmiana używanej tekstury wymaga nagrania nowego polecenia rysowania, które jest informowane o zmianie używanych tekstur poprzez:

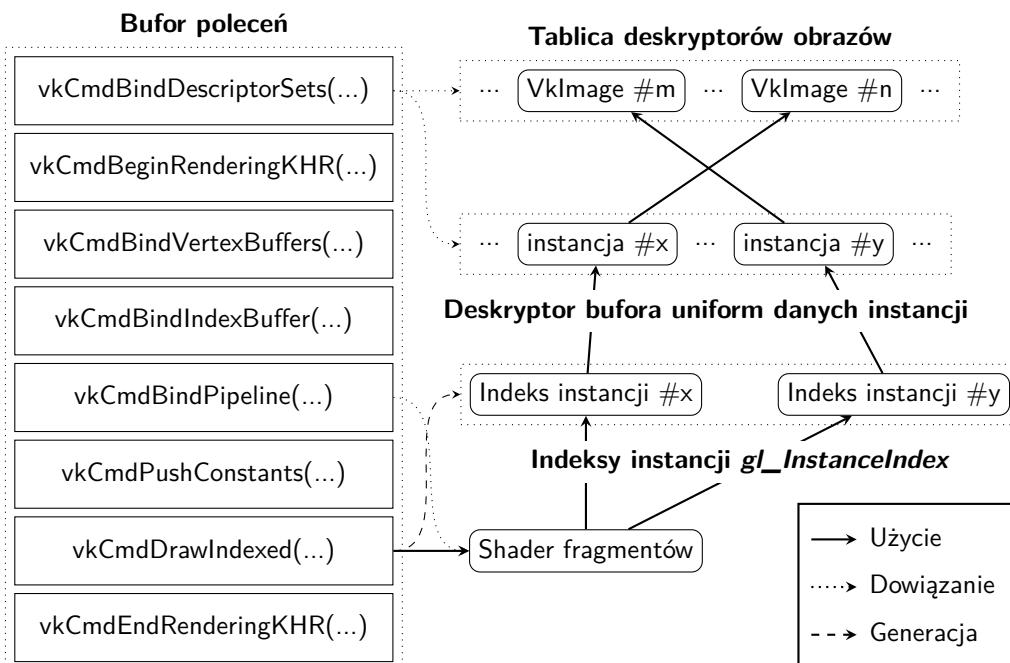
- dowiązanie całkowicie nowego zbioru deskryptorów: bardzo kosztowną operacją, ale wymaga wsparcia tylko statycznego indeksowania,
- ponowne dowiązanie deskryptora dynamicznego bufora uniform ze zmienionym dynamicznym offsetem,
- zmianę jednolitego indeksu używanego do indeksowania deskryptorów poprzez: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

Diagram 2.7 przedstawia tradycyjne tekstury z dowiązaniemami używającymi jednolitego indeksu w stałej push.



Rysunek 2.7: Tradycyjne tekstury z dowiązaniami używające jednolitego indeksu w stałej push (opracowanie własne)

Optymalizacja poleceń rysowania wymaga innego podejścia nieopartego na jednolitym indeksowaniu. Niejednolite dynamiczne indeksowanie pozwala na dowiązanie wszystkich używanych zasobów na początku bufora poleceń i wyemitowanie pojedynczego polecenia rysowania, którego shadery używają niejednolitego indeksu instancji do pobrania indeksów używanych tekstur z bufora uniform opisującego obiekty na scenie. Diagram 2.8 przedstawia tekstury bez dowiązań używając niejednolitych indeksów instancji.



Rysunek 2.8: Tekstury bez dowiązań używając niejednolitych indeksów instancji (opracowanie własne)

2.3. Potok zasobów

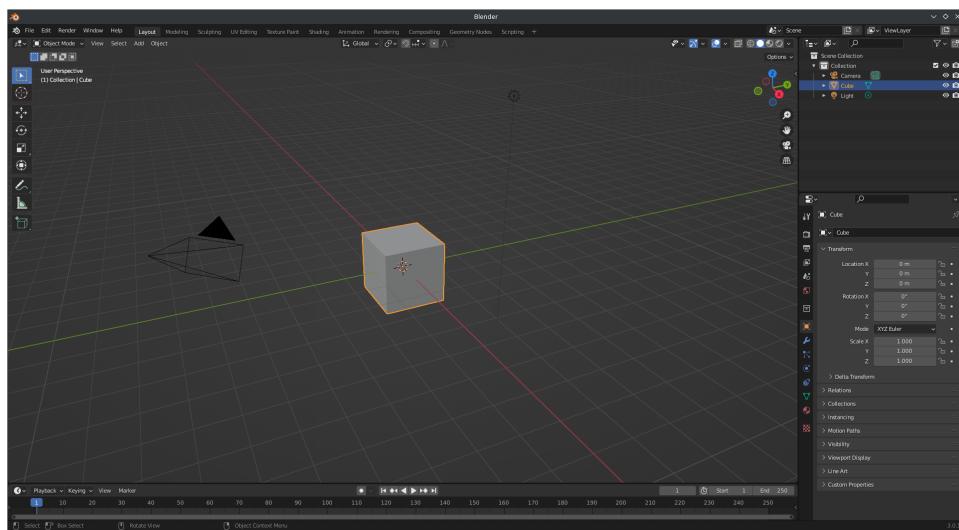
W kontekście silnika zasób (ang. asset) można zdefiniować jako ogólnie jego elementów, które nie są częścią jego kodu źródłowego i mogą być niezależnie od niego dodawane, usuwane i modyfikowane. Ich przepływ pracy jest przedstawiony na poniższym diagramie:



Rysunek 2.9: Przepływ pracy dla zasobów (opracowanie własne)

2.3.1. Zasoby wejściowe

Zasoby wejściowe mają formę plików w formacie przystosowanym do edycji przy użyciu zewnętrznego oprogramowania. Oczywiście przykładami zasobów wejściowych są zasoby 3D obejmujące modele wraz z siatkami, teksturami, materiałami, transformacjami, animacjami, szkieletami i innymi elementami używanymi do renderowania, ale do zasobów wejściowych można też wliczyć tak różnorodne elementy jak oprawa dźwiękowa, efekty cząsteczkowe, pliki konfiguracyjne, shadery czy skrypty.



Rysunek 2.10: Interfejs programu Blender [23] używanego do modelowania 3D (opracowanie własne)

Formaty opisu sceny glTF

Liczba możliwych elementów składającymi się na zasób 3D skłoniła branżę do opracowania ustanowionych formatów opisu sceny, wśród których szczególną popularność zdobyły:

- *FBX* (Filmbox): rozwijany od 2006 przez Autodesk zamknięty format szeroko używany w branży gier z powodu łatwego eksportu oferowanego przez programy Autodesk 3ds Max i Maya. Jego zamknięta natura wymaga importowania albo przy pomocy oficjalnego SDK wymagającego akceptacji restrykcyjnego EULA, albo samodzielnej implementacji wymaganych części formatu inżynierią wsteczną.
- *USD* (Universal Scene Description): otwarty format rozwijany od 2016 przez Pixar pozwala na łatwą i kompletną wymianę informacji pomiędzy różnymi najnowocześniejszymi programami grafiki 3D używanymi przez Pixar do produkcji animacji. Jego skompresowany i okrojony zamknięty wariant *USDZ* jest używany przez Apple.
- *glTF* (Graphics Language Transmission Format): rozwijany od 2015 przez Khronos otwarty format przystosowany do przechowywania ostatecznej wersji sceny w sposób prosty do sparsowania i wrenderowania bądź dalszego przetworzenia. Wersja 2.0 wydana w 2021 zerwała kompatybilność wsteczną z wersją 1.0 i całkowicie ją zastąpiła.

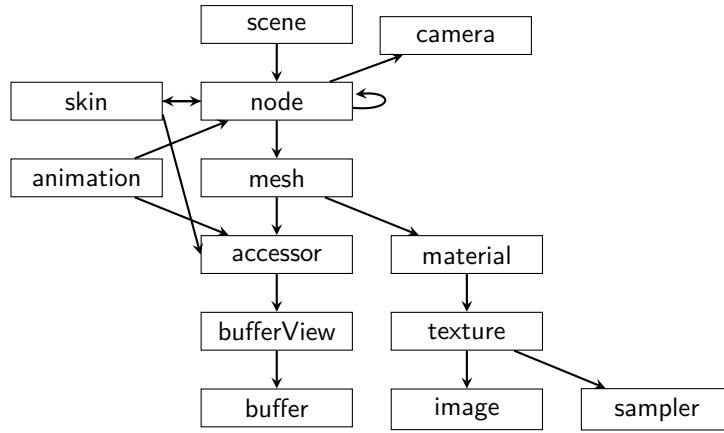
Wśród wymienionych formatów jedynie glTF jest w pełni otwarty i łatwo importowany, dlatego jest on dobrym wyborem jako format zasobów 3D używanych przez silnik graficzny. Na przykład Godot [5] wspiera glTF jako główny i rekommendowany format opisy sceny.

Zgodnie ze specyfikacją glTF [24] zasób jest reprezentowany przez:

- plik tekstowy **.gltf* w formacie JSON zawierający pełny opis sceny i jej poszczególnych elementów,
- pliki binarne **.bin* zawierający dane buforów zawierających geometrię bądź animacje,
- pliki obrazów **.png* i **.jpg* opisujące tekstury,

Pliki binarne i obrazy mogą być osadzone bezpośrednio w obiekcie JSON używając kodowania Base64. Możliwe jest też użycie binarnej wersji formatu glTF pozwalający na przechowywanie wszystkich danych w jednym binarnym pliku *.glb*.

Zasób glTF składa się z obiektu JSON zawierającego metadane oraz osobne tablice dla każdego typu elementu zasobu. Elementy mogą odnosić się do innych elementów używając ich indeksów w odpowiednich tablicach. Relacje pomiędzy różnymi typami elementów zostały pokazana na poniższym diagramie:



Rysunek 2.11: Relacje pomiędzy różnymi typami elementów w formacie glTF (opracowanie własne na podstawie [24])

Listing 2.3.1 zawiera przykładowy plik *triangle.gltf* opisujący scenę zawierającą pojedynczy trójkąt z geometrią składającą się z wierzchołków pozycji zawartych w pliku *triangle.bin*.

```

{
  "asset" : {
    "version" : "2.0"
  },
  "scene" : 0,

  "scenes" : [
  {
    "nodes" : [ 0 ]
  } ],
  "nodes" : [
  {
    "mesh" : 0
  } ],
  "meshes" : [
  {
    "primitives" : [ {
      "attributes" : {
        "POSITION" : 0
      }
    } ]
  } ],
  "buffers" : [
  {
    "uri" : "triangle.bin",
    "byteLength" : 36
  } ],
  "bufferViews" : [
  {
    "buffer" : 0,
    "byteOffset" : 0,
    "byteLength" : 36,
    "target" : 34962
  } ],
  "accessors" : [
  {
    "bufferView" : 0,
    "byteOffset" : 0,
    "componentType" : 5126,
    "count" : 3,
    "type" : "VEC3",
    "max" : [ 1.0, 1.0, 0.0 ],
    "min" : [ 0.0, 0.0, 0.0 ]
  } ]
}

```

2.3.2. Zasoby wyjściowe

Zasoby wyjściowe mają formę plików w formacie przystosowanym do manipulacji przez aplikację. Baza zasobów (ang. asset database) to pojedynczy zasób wyjściowy mający na celu zgromadzenie informacji o wszystkich zasobach używanych przez silnik. Przykładami bazy zasobów są pliki z rozszerzeniami

.uasset i .umap używane przez silnik Unreal Engine [6] do przechowywania zasobów w zoptymalizowanym formacie binarnym czy pliki .streamdb i .resources, które według twórców narzędzia do ekstrakcji zasobów z gier UnArch [25] są używane w grze Doom Eternal i zastąpiły pliki .pak wcześniejszych gier od id Software.

2.3.3. Potok zasobów

Potok zasobów (ang. asset pipeline) to część silnika odpowiadająca za konwersję zasobów wejściowych na zasoby wyjściowe. Konwersja ta jest wymagana, ponieważ istnieją różne wymagania dotyczące formatów: zasoby wejściowe są zoptymalizowane pod kątem oszczędności miejsca na dysku i interoperacyjności między oprogramowaniem zewnętrznym, kiedy zasoby wyjściowe zwykle wymagają mniejszego zakresu możliwych funkcjonalności i powinny być w formacie dostosowanym do maksymalnie szybkiego wczytywania przez aplikację - szybkość ich zapisu jest mniej ważna ponieważ musi się odbyć tylko jeden raz na komputerach twórców oprogramowania uruchamiających potok zasobów.

2.4. Graf sceny

Wyrenderowanie sceny opisanej wysokopoziomowym formatem używanym przez program do grafiki 3D wymaga konwersji jej do listy poleceń rysowania biblioteki graficznej.

Wyemitowanie pojedynczego polecenia rysowania wymaga ustalenia stanu renderowania używanego przez potok graficzny. W przypadku Vulkan API sprowadza się to do dowiązania zasobów bądź przekazania informacji deskryptorami i stałymi push dla następujących elementów renderowanego obiektu:

- geometria: `vkCmdBindVertexBuffer()`, `vkCmdBindIndexBuffer()`;
- materiał: `vkCmdBindPipeline()`, `vkCmdBindDescriptorSets()`;
- globalne przekształcenie (macierz modelu): `vkCmdPushConstants()`, `vkCmdBindDescriptorSets()`.

Proces konwersji sceny na polecenia rysowania zależy od jej formatu. Nieskomplikowane sceny mogą być opisane przy pomocy listy renderowanych obiektów. Bardziej złożone sceny są zwykle reprezentowane przy użyciu hierarchicznego modelowania. Jest to technika reprezentowania złożonych scen polegająca na podzieleniu ich modelów na części i śledzeniu hierarchicznych relacji rodzic-dziecko pomiędzy nimi przy użyciu struktury zwanej grafem sceny. Ten sposób reprezentacji sceny bardzo popularny wśród aplikacji 3D i jest używany przez formaty FBX i glTF.

Każdy węzeł grafu sceny składa się z lokalnej transformacji przestrzeni świata, opcjonalnej referencji do renderowanej geometrii oraz listy węzłów potomnych. Jeden z węzłów jest oznaczony jako korzeń. Emitowanie poleceń rysowania sprowadza się do rekurencyjnego przemierzania grafu sceny zaczynając od korzenia [22] i zostało przedstawione na listingu 2.10:

RenderujGrafSceny(węzeł):

```
jeżeli węzeł.rodzic istnieje:  
    węzeł.globalnaTransformacja = węzeł.rodzic.globalnaTransformacja * węzeł.  
    ↪ lokalnaTransformacja  
w przeciwnym wypadku:  
    węzeł.globalnaTransformacja = węzeł.lokalnaTransformacja  
EmitujPolecenieRysowania(węzeł.geometria, węzeł.materiał, węzeł.  
    ↪ globalnaTransformacja)  
dla każdego dziecka w węzeł.dzieci:
```

RenderujGrafSceny(dziecko)

Listing 2.10: Emitowanie poleceń rysowania na podstawie grafu sceny

Liczba poleceń rysowania wyemitowanych algorytmem 2.10 może być zoptymalizowana. Przykładowo w scenie z samochodem jego cztery koła są renderowane przy pomocy czterech osobnych poleceń rysowania różniących się tylko pozycją, które mogą być narysowane używając jednego polecenia rysującego cztery instancje geometrii.

2.5. Graf renderowania

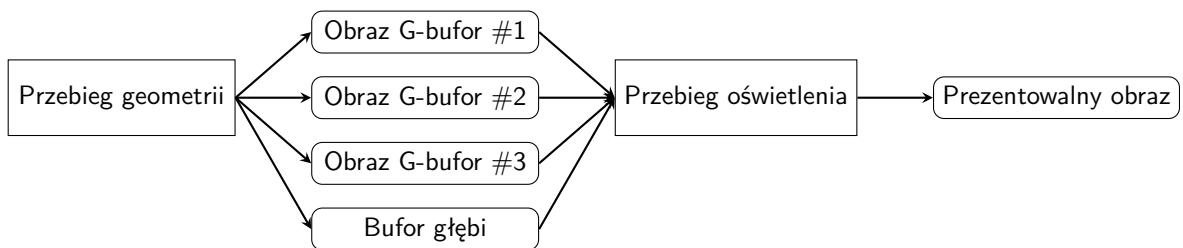
W przeciwieństwie do poprzednich API takich jak OpenGL, Vulkan nie zmienia automatycznie układu obrazu, co zmusza programistę do samodzielnego śledzenia stanu układu obrazów i wykonywania przejść na poprawny układ przed użyciem ich przez polecenia poprzez użycie odpowiedniej bariery pamięci obrazu *VkImageMemoryBarrier*.

Dodatkowo renderowanie wieloprzebiegowe wymaga zapewnienia zależności pamięci pomiędzy przebiegami renderowania odczytującymi i zapisującymi te same zasoby poprzez ręczne nagrywanie odpowiednich barier pamięci.

Problem ten został rozwiązany w silniku Frostbite poprzez wprowadzenie struktury zwanej grafem renderowania [26] będącej wysokopoziomową reprezentacją procesu renderowania pojedynczej klatki.

Graf renderowania jest skierowanym grafem acyklicznym, którego węzły to przemienne przebiegi renderowania i zasoby. Krawędzie reprezentują zależności odczytu i zapisu zasobów przez przebiegi. W przypadku tradycyjnego potoku graficznego zasoby to obrazy, które są próbkiowane i używane jako dołączenia.

Rysunek 2.12 pokazuje przykładowy graf renderowania ilustrujący renderowanie odroczone. Zawiera on dwa przebiegi geometrii i oświetlenia, które odczytują i zapisują obrazy G-bufora, bufor głębi oraz prezentowalny obraz.



Rysunek 2.12: Przykładowy graf renderowania ilustrujący renderowanie odroczone (opracowanie własne)

Użycie grafu renderowania można podzielić na trzy etapy:

- konstrukcja: dodanie wszystkich przebiegów renderowania, używanych zasobów oraz zależności między nimi;
- komplikacja: analiza skonstruowanego grafu mająca na celu obliczenie zmian stanu zasobów pomiędzy przebiegami;
- wykonanie: użycie skompilowanego grafu do nagrania polecen rysowania.

W Vulkan graf renderowania może być używany do śledzenia użycia obrazów przez przebiegi renderowania oraz nagrania do bufora polecen przebiegów renderowania nagrywania wraz z barierami pamięci obrazu gwarantującymi odpowiednie przejście układu obrazów i zależności pamięci.

2.6. Renderowanie oparte na fizyce

Renderowanie oparte na fizyce (ang. Physically Based Rendering, PBR) to metodologia renderowania mająca na celu zwiększenie realizmu renderowanych scen poprzez użycie materiałów i modeli oświetlenia odwzorowujących rzeczywistość.

Materiał to kompletny opis właściwości powierzchni renderowanej geometrii będący zbiorem parametrów i tekstur używanych do przez shadery podczas renderowania prymitywów.

Renderowana scena może definiować różne rodzaje światła. Najprostszym typem światła jest światło otoczenia padające równomiernie na wszystkie powierzchnie prymitywów. Światło bezpośrednie pada na obszar powierzchni prymitywu bezpośrednio ze źródła światła. Światło pośrednie jest światłem odbitym od innych powierzchni na scenie i z powodu kosztowności jego obliczania jest albo pomijane, albo przybliżane używając okluzji otoczenia (ang. ambient occlusion).

Materiał i informacje o światłach na scenie są używane przez zaimplementowany w shaderach model oświetlenia to obliczenia ostatecznego koloru powierzchni prymitywu. Model oświetlenie PBR, w przeciwieństwie do tradycyjnych modelów takich jak Blinn-Phong [1], stara się przestrzegać praw fizyki.

Nie istnieje jedna standardowa implementacja PBR, ale specyfikacja glTF [24] definiuje prosty model materiałów PBR. Materiał w formacie glTF jest zdefiniowany przy użyciu modelu metaliczności-chropowatości posiadającego następujące właściwości:

- kolor podstawowy (ang. base color): informacja o kolorze obiektu;
- metaliczności (ang. metalness): określa metaliczność materiału, tj. czy materiał jest metalem czy dielektrykiem;
- chropowatość (ang. roughness): określa chropowatość materiału, tj. czy materiał jest błyszczący czy matowy.

Kolor podstawowy jest obliczany poprzez przemnożenie dwóch parametrów materiału: współczynnika vec4 oraz tekstury RGBA. Podobnie używane są współczynniki chropowatości i metaliczności będące wartościami w przedziale $[0,1]$ mnożonymi przez tekstury metaliczności-chropowatości - wartości metaliczności są próbkiowane z komponentu B, a wartości metaliczności z komponentu G.

Khronos udostępnia referencyjną implementację PBR używaną przez glTF przeznaczoną do celów edukacyjnych [27]. Jest ona łatwa do zrozumienia i może być łatwo rozszerzony o dodatkowe funkcjonalności takie jak mapowanie nierówności.

2.7. Renderowanie odroczone i efekty post-processingu

Scena może być wyrenderowana w ramach jednego przebiegu renderowania używającego potoku graficznego zaczynającego od pobrania wierzchołków geometrii i kończącego na zapisaniu ostatecznego koloru do dołączenia koloru będącego prezentowalnym obrazem. Ta technika renderowania jest nazywana renderowaniem wprzód (ang. forward rendering) i jest tradycyjnym oraz intuicyjnym sposobem myślenia o renderowaniu.

Niestety renderowanie wprzód posiada wadę widoczną podczas renderowania skomplikowanych scen zawierających obiekty z geometrią zasłaniającą inne obiekty, dla których następuje spadek wydajności spowodowany przerysowaniem (ang. overdraw). Przerysowanie to sytuacja, w której ostateczny kolor jest wielokrotnie nadpisywany przez shader fragmentów wykonywane dla prymitywów zasłaniających ten sam teksel dołączenia koloru. Przerysowanie jest problematyczne zwłaszcza dla przebiegów renderowania używających kosztownych shaderów fragmentów, np. obliczających modele oświetlenia dla dużej liczby

świateł na scenie.

Problem przerysowania może zostać rozwiązany przy użyciu techniki renderowania odroczonego (ang. deferred rendering). Renderowanie odroczone rozbija przebieg renderowania wprzód na dwa następujące przebiegi renderowania:

- przebieg geometrii: wypełnia poszczególne komponenty G-bufora,
- przebieg oświetlenia: generuje właściwy obraz na podstawie komponentów G-bufora.

Rysunek 2.12 zawiera graf renderowania ilustrujący cieniowanie odroczone.

G-bufor to zbiór tekstur pozaekranowych zmiennoprzecinkowych zawierających mniej kosztowne obliczeniowo informacje o powierzchni wyrenderowanej geometrii. Przykładowo G-bufor może zawierać kolor podstawowy powierzchni, normalną oraz położenie w przestrzeni świata.

Ważne jest, żeby shader fragmentów używany podczas renderowania sceny do G-bufora był mały i wydajny, dzięki czemu wystąpienie przerysowania nie spowoduje znaczącego spadku wydajności - przebieg geometrii nie wykonuje kosztownych obliczeń oświetlenia.

Po zakończeniu przebiegu geometrii G-bufor może zostać użyty do wyrenderowania ostatecznego obrazu. Potok graficzny przebiegu oświetlania zaczyna się od polecenia rysowania pełnoekranowego trójkąta, dzięki czemu emitowane jest dokładnie jedno wywołanie shadera fragmentów dla każdego tekscela dowiązania koloru - kosztowny shader fragmentów obliczający oświetlenie jest obliczany tylko jeden raz i jest niezależny od stopnia skomplikowania geometrii sceny.

Opisana wyżej technika renderowania pełnoekranowanego trójkąta wywołującego shaderów fragmentów dla każdego tekscela dowiązania koloru jest podstawą efektów post-processingu. Są one przebiegami renderowania występującymi po zakończeniu rysowania właściwej sceny mających na celu poprawienie wyglądu ostatecznie prezentowanego obrazu.

W grafice komputerowej znane jest wiele efektów post-processingu, poniżej wymieniono popularniejsze:

- SSAO (ang. screen space ambient occlusion): przybliżona okluzja otoczenia imitująca światło pośrednie,
- rozmycie ruchu (ang. motion blur): rozmazanie obrazu spowodowane nagłym ruchem kamery,
- rozkwit (ang. bloom): poświata dookoła jasnych powierzchni,
- mgła:
- skybox: renderowania sklepienia nieba używając oteksturowanego sześcianu.

3. NARZĘDZIA, ARCHITEKTURA I IMPLEMENTACJA

3.1. Narzędzia

Silnik została napisany jako biblioteka w języku C w standardzie C11 [28]. Budowanie biblioteki ze źródeł wymaga generacji dodatkowego kodu przy pomocy skryptów w języku Python w wersji 3.9.7 [29].

Silnik został w całości opracowany na przy użyciu środowiska programistycznego CLion w wersji 2021.2.3 [30].

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 [31].

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 [32], 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 komplikacji (pliki Makefile w systemie Linux, projekty Microsoft Visual C++ w systemie Windows). Silnik był napisany i przetestowany wyłącznie na systemie Linux, ale wieloplatformowa natura używanych narzędzi i bibliotek powinna pozwolić na dokonanie komplikacji skróśnej. Użyto prekompilowanych nagłówków do przyśpieszenia komplikacji 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 wywoływanego zainstalowanego interpretera Pythona jest izolacja zarządzania zależnościami od reszty systemu operacyjnego, co pozwala na łatwiejszą powtarzalność podczas debugowania [33].

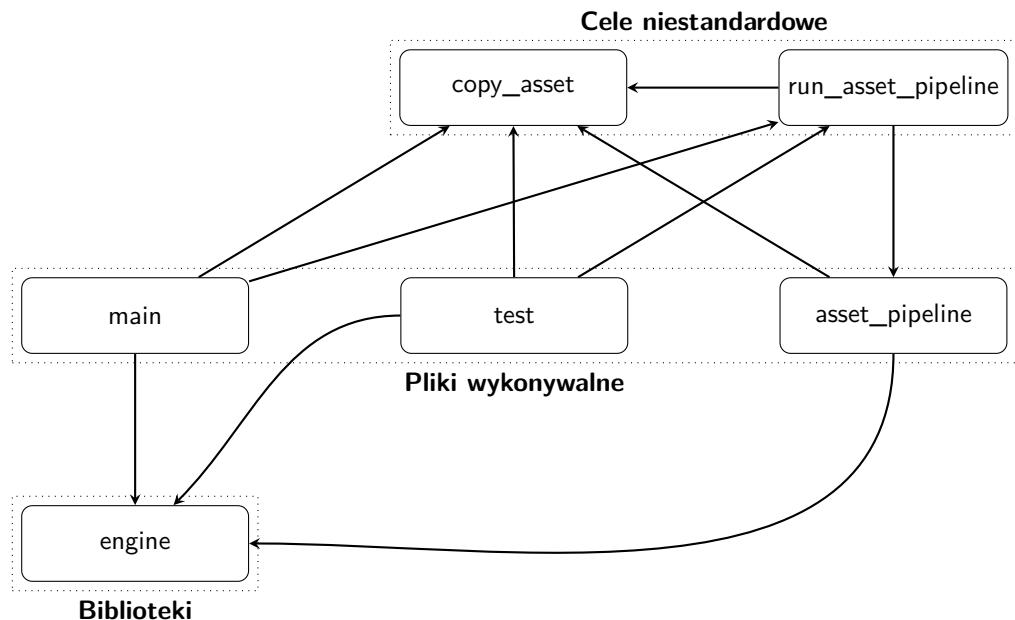
CMake organizuje proces budowania jako graf, w którym wierzchołki to cele połączonych ze sobą zależnościami. Budowa celu wymaga wcześniejszego zbudowania wszystkich innych celów od których zależy budowany cel.

Wyróżniane są trzy rodzaje celów:

- plik wykonywalny;
- biblioteka: statyczna lub dynamiczna;

- cel niestandardowy: używany do uruchamiania zewnętrznych programów podczas procesu komplikacji, np. generatorów kodu.

Diagram 3.1 przedstawia proces budowania projektu w formie celów i ich zależności.



Rysunek 3.1: Proces budowania w formie celów i ich zależności (opracowanie własne)

engine Cel budujący bibliotekę programistyczną zawierającą implementację silnika.

main Cel budujący plik wykonywalny demonstруjący 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 kopiący podkatalogu głównego `assets` zawierającego nieprzetworzone zasoby wejściowe do katalogu budowania.

run_asset_pipeline Niestandardowy cel realizujący potok zasobów poprzez uruchomienie skryptu Python wielokrotne uruchamiającego narzędzie **asset_pipeline** na zasobach wejściowych.

3.1.2. Biblioteki zewnętrzne

Projekt używa następujących zewnętrznych bibliotek programistycznych:

- **Vulkan SDK 1.3.211.0 [7]:**
 - pliki nagłówkowe dla Vulkan,
 - *shaderc*: komplikacja shaderów z kodu źródłowego GLSL do kodu bajtowego SPIR,
 - *SPIRV-Reflect*: mechanizm refleksji dla kodu bajtowego SPIR-V,

- *glfw 3.4* [13]: wieloplatformowa obsługa tworzenia okien, obsługa wejścia klawiatury i myszy,
- *sqlite 3.35.5* [34]: relacyjna baza danych SQL,
- *uthash 2.3.0* [35]: proste struktury danych (tablica dynamiczna, lista dwukierunkowa, tablica mieszająca),
- *xxHash 0.8.1* [36]: niekryptograficzny algorytm mieszający,
- *cgltf 1.11* [37]: wczytywanie plików w formacie glTF,
- *cglm 0.8.5* [38]: biblioteka matematyczna,
- *stb_image 2.27* [39]: wczytywanie obrazów,
- *stb_truetype 1.26* [39]: rasteryzacja tekstu czcionek,
- biblioteka standardowa języka C [28],
- API systemu operacyjnego: pliki nagłówkowe POSIX [40] albo WinAPI [41],
- biblioteka standardowa języka Python [29],
- *libclang 12.0.0* [42]: analizowanie kodu C w skryptach Python.

Dodatkowo biblioteka zbudowana w konfiguracji *Debug* statycznie linkuje biblioteki *ASan* (Address-Sanitizer) 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

Silnik jest zaprojektowany w duchu architektury modułowej - funkcjonalność biblioteki jest rozdzielona na bloki zwane modułami, które mogą być rozwijane niezależnie od pozostałych modułów.

Obecne silnik składa się z następujących modułów:

- Wygenerowany kod: mechanizmy metaprogramowania;
- Rdzeń: funkcje pomocnicze;
- I/O: wczytywanie bazy zasobów i konfiguracji;
- Zasoby: serializacja i deserializacja danych sceny z bazy zasobów;
- Vulkan: obsługa API Vulkan;
- Scena: operacje na grafie sceny;
- Renderer: graf renderowania i główna pętla programu.

Dodatkowo budowany jest plik wykonywalny demonstrujący użycie silnika do wyrenderowanie przykładowej sceny używając grafu renderowania odroczonego.

Silnik był rozwijany metodą *bottom-up* - jego pierwsza iteracja była pojedynczym plikiem źródłowym wyświetlającym trójkąt [11], który w procesie dekompozycji i refaktoryzacji organicznie rozrosł się do

7 modułów znajdujących się w osobnych podkatalogach zawierających łącznie 9 skryptów Python .py, 65 nagłówków *.h i 63 plików źródłowych *.c.

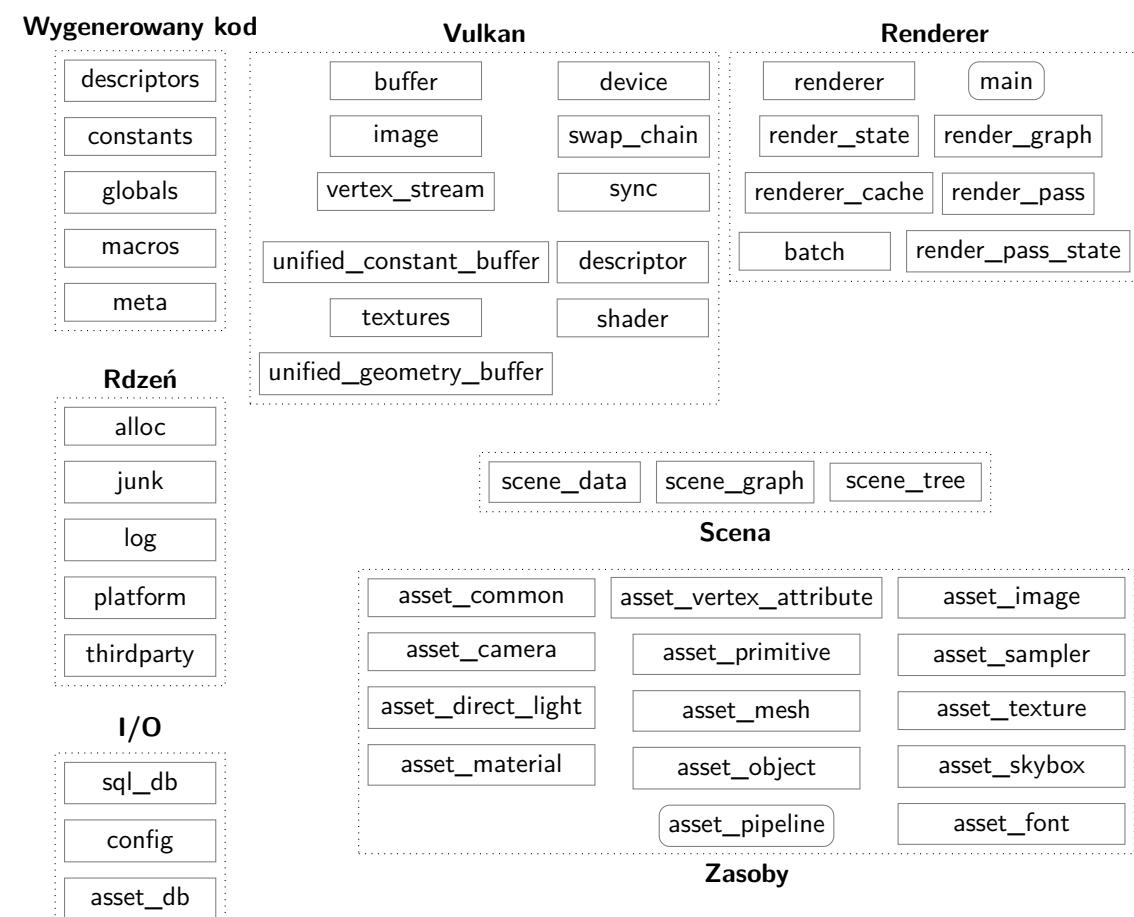
Moduł jest dalej podzielony na jednostki, które zostały na potrzeby projektu zdefiniowane jako para składająca się z pliku nagłówkowego z odpowiadającym plikiem źródłowym o tej samej nazwie.

Pliki nagłówkowe zawierają deklaracje funkcji, struktur oraz typów wyliczeniowych widocznych dla użytkownika końcowego i powinny być dołączone do programu przy użyciu dyrektywy `#include` preprocesora. Pliki źródłowe zawierają definicje deklaracji pliku nagłówkowego i powinny być dołączone do programu używając argumentów kompilatora (jeśli dodawane są niezbudowane pliki źródłowe) bądź linkera (jeśli dodawane są zbudowane pliki biblioteczne), co jest automatycznie wykonywane przez CMake.

Struktury są zorganizowane w sposób obiektowy. Język C nie posiada wbudowanej koncepcji klasy, ale w projekcie przyjęto założenie, że dla klasy `struct` jej stan jest reprezentowany przez strukturę `struct`, która może posiadać metodę `func()`, jeśli istnieje funkcja `struct_func()` przyjmująca wskaźnik do `struct` jako pierwszy argument. Dla obiektów globalnych nie jest istnieje osobna struktura przekazywana do jej metod - stan obiektu jest zaszyty w zmiennych globalnych jednostki translacji pliku źródłowego.

Obiekty mogą oferować metody `create()` i `destroy()`, które alokują lub dealokująinstancję obiektu oraz tworzą bądź niszczą jej wewnętrzny stan. Analogiczne metody `init()` i `deinit()` tworzą i niszczą instancję, której pamięć została wcześniej zaalokowaną (np. na stosie lub w tablicy). Opcjonalna metoda `debug_print()` loguje informacje o wewnętrzny stanie instancji użyteczne podczas debugowania.

Diagram 3.2 przedstawia moduły silnika i ich najważniejsze jednostki.



Rysunek 3.2: Relacje pomiędzy modułem silnika i ich najważniejszymi klasami (opracowanie własne)

3.3. Implementacja

Ta sekcja opisuje szczegóły implementacyjne poszczególnych modułów silnika.

3.3.1. Wygenerowany kod

Silnik używa kodu w języku C wygenerowanego przez automatyczny generator kodu będący skryptem Python uruchamianym przez CMake na początku procesu budowania przed rozpoczęciem komplikacji właściwego kodu źródłowego biblioteki.

Język C nie posiada mechanizmów pozwalających na metaprogramowanie z wyjątkiem makr preprocesora, które mogą zaspokoić część potrzeb programisty chcącego przykładowo dodać nowy rodzaj pętli [43], ale nie pozwalają na bardziej skomplikowaną analizę i przekształcanie kodu, które muszą być wykonywane przez zewnętrzne narzędzia.

Działanie skryptu jest sterowane konfiguracją generatora, który jest plikiem w formacie INI (zgodnym z biblioteką *configparser* [44]) znajdującym się w katalogu ze skryptem. Format INI nie posiada standardowej specyfikacji, ale tradycyjnie jest on plikiem tekstowym podzielonym na sekcje zawierające pary klucz-wartość.

Skrypt parsuje pliki nagłówkowe języka C znajdujący się w katalogu /src z pominięciem katalogu /src/codegen, do którego skrypt zapisuje wygenerowane pliki nagłówkowe i źródłowe, które są kolejno dołączane w innych modułach silnika i dodawane jako argumenty kompilatora. Razem wszystkie wygenerowane pliki tworzą jednostki modułu wygenerowanego kodu.

Jednostka constants

Zawiera wygenerowane stałe: wartości zdefiniowane w sekcji *CONSTANTS* konfiguracji generatora używane przez resztę modułów, które zostały uznane za zbyt niepraktyczne aby pozwolić na ich modyfikację przy użyciu konfiguracji globalnej. Poniżej wymieniono stałe, ich wartości oraz interpretacje:

- *FRAMES_IN_FLIGHT*: 2, liczba klatek w locie (ang. in flight frames), czyli jednocześnie renderowanych przez GPU, domyślna wartość pozwala na podwójne buforowanie;
- *MAX_OFFSCREEN_TEXTURE_COUNT*: 16, maksymalna liczba tekstur pozaekranowych;
- *MAX_RENDER_TARGET_COUNT*: 8, maksymalną liczbę tekstur pozaekranowych, które mogą być używane jako cele renderowania podczas jednego przebiegu;
- *MAX_FRAMEBUFFER_ATTACHMENT_COUNT*: $\text{MAX_RENDER_TARGET_COUNT} + 1 + 1$, maksymalna liczba dołączeń używana przez potok graficzny - wystarcza na dołączenia celów renderowania, prezentowalnego obrazu i bufor głębi;
- *MAX_INDIRECT_DRAW_COMMAND_COUNT*: 1024, maksymalna liczba poleceń rysowania które mogą być wykonana przez jedno polecenie rysowania pośredniego;
- *MAX_MATERIAL_COUNT*: 128, maksymalna liczba materiałów;
- *MAX_DIRECTIONAL_LIGHT_COUNT*: 1, maksymalna liczba światel kierunkowych na scenie;
- *MAX_POINT_LIGHT_COUNT*: 128, maksymalna liczbę światel punktowych na scenie;
- *MAX_TEXT_CHARACTER_COUNT*: 256, maksymalną liczbę znaków w renderowanym ciągu znaków;
- *MIN_DELTA_TIME*: (1.0/30.0), minimalny czas pomiędzy wywołaniami funkcji zwrotnej *update* w pętli głównej, domyślnie $\frac{1}{30}$ sekundy (30 FPS);

- *WORLD_UP*: 0,1,0; wektor interpretowany jako "w góre" w przestrzeni świata.

Wygenerowane stałe mogą być używane przez shadery - ich definicje są umieszczane na początku kodu GLSL shadera przed jego komplikacją - dlatego są one udostępniane w formie X-makro *CODEGEN_CONSTANTS*.

X-makro to przydatna technika preprocesora pozwalająca na pisanie kodu, który jest automatycznie aktualizowany po zmianie danych opisywanych przez X-makro [45]. Przykładowo funkcja przedstawiona na listingu 3.1 wymaga manualnej aktualizacji po zmianie używanego typu wyliczeniowego.

```
typedef enum key {
    key_space,
    key_enter,
    key_count,
} key;

int key_to_glfw_key(key value) {
    switch (value) {
        case key_space: return GLFW_KEY_SPACE;
        case key_enter: return GLFW_KEY_ENTER;
        default: return GLFW_KEY_UNKNOWN;
    }
}
```

Listing 3.1: Przykładowy kod przed zastosowaniem X-makro

Listing 3.2 przedstawia użycie X-makro do przepisania kodu z listingu 3.1:

```
#define END_OF_KEYS
#define KEYS(X, ...) \
    X(space, GLFW_KEY_SPACE) \
    X(enter, GLFW_KEY_ENTER) \
    END_OF_KEYS

typedef enum key {
#define x(_name, ...) key_##_name,
    KEYS(x, )
#undef x
    key_count,
} key;

int key_to_glfw_key(key value) {
    switch (value) {
#define x(_name, _value, ...) case key_##_name: return _value;
    KEYS(x, )
#undef x
        default: return GLFW_KEY_UNKNOWN;
    }
}
```

```
}
```

Listing 3.2: Przykładowy kod po zastosowaniu X-makro

Jednostka `globals`

Obiekt globalny `globals` reprezentujący wygenerowane zmienne. Ich wartości, w przeciwieństwie do stałych, mogą być ustalone dopiero w czasie wykonywania. Obiekt jest używany do specyfikacji struktury różnych ścieżek katalogów i plików używanych przez silnik.

Listing 3.3 przedstawią sekcję konfiguracji generacji opisującą ścieżki dla katalogu zasobów, konfiguracji globalnej, bazy zasobów, katalogu shaderów i ich współdzielonego kodu GLSL oraz pliku logowania.

```
[GLOBALS]
assetsDirname = assets
assetDatabaseFilepath = ${assetsDirname}/data.db
assetConfigFilepath = ${assetsDirname}/config.ini
assetsShaderDirpath = ${assetsDirname}/shaders
assetsShaderCommonFilepath = ${assetsShaderDirpath}/common.gls
logFileName = log.txt
```

Listing 3.3: Konfiguracja generacji zmiennych

Listing 3.4 przedstawia wygenerowaną metodę `init()`.

```
void globals_create() {
    globals.assetsDirname =
        get_executable_dir_file_path("", "assets");
    globals.assetDatabaseFilepath =
        get_executable_dir_file_path("", "assets/data.db");
    globals.assetConfigFilepath =
        get_executable_dir_file_path("", "assets/config.ini");
    globals.assetsShaderDirpath =
        get_executable_dir_file_path("", "assets/shaders");
    globals.assetsShaderCommonFilepath =
        get_executable_dir_file_path("", "assets/shaders/common.gls");
    globals.logFileName =
        get_executable_dir_file_path("", "log.txt");
}
```

Listing 3.4: Wynik generacji zmiennych

Jednostka `macros`

Zbiór X-makr używanych przez moduł I/O obsługujący następujące zasoby wejściowe.

Makra opisują wewnętrzną strukturę plików INI konfiguracji globalnej i konfiguracji zasobów: używane sekcje i ich dopuszczalne pary klucz-wartość z domyślnymi wartościami (liczby całkowite bądź ciągi znaków). Listing 3.5 przedstawia przykładowy fragment konfiguracji generatora opisujący konfigurację globalną i pozwalający na sparsowanie pliku INI przedstawionego na listingu 3.6.

```
[GLOBAL.CONFIG]
graphics.WindowWidth = 640
controls.Enabled = 1
settings.StartScene = "sponza"
```

Listing 3.5: Konfiguracja generacji konfiguracji globalnej

```
[settings]
StartScene = MetalRoughSpheresNoTextures

[graphics]
WindowWidth = 1024

[controls]
Enabled = 1
```

Listing 3.6: Przykładowa konfiguracja globalna

Podobnie opisywana jest struktura bazy zasobów: typy podstawowe i ich odpowiedniki w języku C oraz tabele i ich kolumny. Ilustruje to fragment konfiguracji generatora przedstawiony na listingu 3.7.

```
[ASSET.DB]
types = "BYTE:uint8_t, INT:uint32_t, FLOAT:float, TEXT:UT_string *, KEY:hash_t"
image = "key KEY, width INT, height INT, depth INT, channels INT, type INT, data
        ↪ BYTE_ARRAY"
sampler = "key KEY, magFilter INT, minFilter INT, addressWrapU INT, addressWrapV
          ↪ INT"
texture = "key KEY, image KEY, sampler KEY"
```

Listing 3.7: Fragment konfiguracji generatora opisujący strukturę bazy zasobów

Struktura zasobów wejściowych zostanie dokładniej opisana w dalszym podrozdziale o module I/O.

Jednostka meta

Funkcje pomocnicze wygenerowane na podstawie nagłówków silnika i Vulkan SDK.

Dla każdego napotkanego typu wyliczeniowego *EnumName* jest generowana jedna z funkcji, których prototypy zostały przedstawione na listingu 3.8.

```
const char *EnumName_debug_str(int value);
void EnumName_debug_print(int flags, int indent);
```

Listing 3.8: Wygenerowane funkcje dla typów wyliczeniowych

Funkcje pozwalające na konwersję liczby całkowitej będącej wartością zmiennej wyliczeniowej na ciąg znaków i są używane przez metody *debug_print()* do logowania wartości w formie przyjazniejszej dla użytkownika.

Funkcja **_debug_str()* jest generowana tylko wtedy, jeśli literał wyliczeniowe nie są flagami, tj. nie są kolejnymi potęgami liczby 2.

Jednostka descriptors

Jednostka zawierająca struktury i funkcje upraszczające pracę z deskryptorami.

Nagłówek *descriptor* modułu Vulkan zawiera definicje struktur języka C opisujących wewnętrzną strukturę pamięci buforów i stałych push znajdujących się na GPU. W zależności od nazwy dzielą się one na 3 rodziny:

- **_push_constant_struct*: stała push o nazwie ***,
- **_uniform_buffer_struct*: bufor uniform o nazwie ***,
- **_helper_struct*: struktura pomocnicza o nazwie *** używana w powyższych.

Listing 3.9 przedstawia przykłady powyższych struktur.

```
// stała push 'draw'
typedef struct draw_push_constant_struct {
    uint currentFrameInFlight;
} draw_push_constant_struct;

// struktura pomocnicza 'offscreen_texture'
typedef struct offscreen_texture_helper_struct {
    uint textureId; ///< array=MAX_OFFSCREEN_TEXTURE_COUNT
} offscreen_texture_helper_struct;

// stała push 'global'
typedef struct global_uniform_buffer_struct {
    mat4 viewMat;
    mat4 projMat;
    ...
    offscreen_texture_helper_struct offscreenTextures;
} global_uniform_buffer_struct;
```

Listing 3.9: Przykładowe struktury w nagłówku *descriptor* opisujące wewnętrzną strukturę deskryptorów

Układ pamięci struktur zdefiniowanych w języku C nie jest koniecznie kompatybilny z układem pamięci wymaganym przez GPU. Dlatego dla każdej sparsowanej struktury **_struct* jest generowana analogiczna struktura **_element*, w których użyto specyfikatorów *alignas* i atrybutów *packed* udostępnianych przez C11 i rozszerzenia GCC w celu wyrównania pól struktury w zgodzie ze standardem układu pamięci scalar. Generowana jest też funkcja *glsl_add_**() dodająca do ciągu znaków z kodem GLSL definicje struktury i kwalifikator układu. Listing 3.10 pokazuje przykładowe wejście i wyjście generacji dla bufora uniform *instances*.

```
// descriptor.h:
typedef struct instances_uniform_buffer_struct {
    mat4 modelMat;
    uint materialId;
} instances_uniform_buffer_struct;

// descriptors.h
typedef struct PACKED_STRUCT instances_uniform_buffer_element {
    alignas(4) mat4 modelMat ;
```

```

    alignas(4) uint materialId ;
} instances_uniform_buffer_element;
void glsl_add_instances_uniform_buffer(
    UT_string *s, uint32_t set, uint32_t binding, uint32_t count);

// descriptors.c
void glsl_add_instances_uniform_buffer(
    UT_string *s, uint32_t set, uint32_t binding, uint32_t count) {
    utstring_printf(s, "struct instancesStruct {\n");
    utstring_printf(s, "    mat4 modelMat ;\n");
    utstring_printf(s, "    uint materialId ;\n");
    utstring_printf(s, "};\n");
    utstring_printf(s, "layout(scalar, set = %u, binding = %u) "
                    "uniform instancesBlock {\n", set, binding);
    utstring_printf(s, "    instancesStruct instances");
    if (count > 1) {utstring_printf(s, "[%u]", count);}
    utstring_printf(s, ";\n};\n");
}

```

Listing 3.10: Przykładowe wejście i wyjście generacji dla bufora uniform

Generacja jest kończona X-makrami wyliczającymi nazwy wszystkich sparsowanych rodzin struktur.

Dzięki automatycznej generacji kodu modyfikacja sposobu organizacji pamięci GPU buforów sprawdza się do modyfikacji struktur w nagłówku *descriptors*, co pozwala na szybkie testowanie nowych parametrów i metod dostępu do nich podczas pisania shaderów. Mechanizm ten został zainspirowany implementacją jednolitych buforów w grze *Tom Clancy's Rainbow Six Siege* [46].

Wygenerowane struktury, funkcje i X-makra są używane podczas kopирования danych z CPU do pamięci GPU oraz generacji shaderów, co zostanie dokładniej opisane w dalszym podrozdziale o module Vulkan.

3.3.2. Rdzeń

Rdzeń to moduł zawierający funkcje pomocnicze i obiekty globalne zapewniające podstawowe funkcjonalności używane przez resztę modułów.

Jednostka thirdparty

Jednostka odpowiedzialna za udostępnienia bibliotek zewnętrznych reszcie kodu.

Nagłówek dołącza nagłówki bibliotek zewnętrznych i z powodów wydajnościowych podczas procesu budowania jest traktowany jako nagłówek prekompilowany (ang. precompiled header, PCH).

Plik źródłowy obsługuje część bibliotek zewnętrznych składających się jedynie z nagłówków (ang. header-only library). W przeciwieństwie do tradycyjnych bibliotek języka C w których kod jest podzielony na pliki nagłówkowe i źródłowe, w tym przypadku dostęp do definicji tradycyjnie znajdujących się z plikach źródłowych jest uzyskiwany poprzez ponowne dołączenie nagłówka przy użyciu dyrektywy `#include` po wcześniejszym zdefiniowaniu odpowiedniego symbolu preprocesora. Listing 3.11 przedstawia wymagany sposób ponownego dołączenia nagłówka biblioteki *cgltf*.

```
#define CGLTF_IMPLEMENTATION
```

```
#include "cgltf.h"
```

Listing 3.11: Przykład dołączenia implementacji biblioteki *cgltf*

Jednostka alloc

Funkcje pomocnicze wspomagające zarządzanie pamięcią CPU, co obejmuje alokację, dealokację, kopiowanie, duplikowanie i porównywanie bloków pamięci CPU.

Funkcje te są potrzebne, ponieważ działanie odpowiednich funkcji oferowane przez bibliotekę standardową języka C, chociaż oferują żądaną funkcjonalność, opiera się na mechanizmie niezdefiniowanych zachowań (ang. undefined behaviour) dla niektórych argumentów (wskaźnik NULL, rozmiar 0) i zachowań OOM (ang. out-of-memory, brak pamięci).

Funkcje pomocnicze są wrapperami z dodatkowymi instrukcjami warunkowymi sprawdzającymi, czy wywołanie funkcji nie skutkuje niezdefiniowanym zachowaniem.

Jednostka definiuje też makra ułatwiające zarządzanie pamięcią struktur danych biblioteki *uthash*.

Jednostka log

Obiekt globalny *log* reprezentujący system logowania komunikatów wygenerowanych podczas działania kodu mający na celu w uproszczenie procesu debugowania.

Komunikat jest ciągiem znaków z przypisanym poziomem logowania określającym jego ważność z domyślnie wspieranymi wartościami *debug*, *info*, *warn*, *error* i *fatal*. Komunikaty *debug* są logowane tylko w konfiguracji *Debug*.

Komunikaty są zapisywane do standardowego wyjścia (*stdout* albo *stderr*) oraz do pliku tekstowego na dysku, którego nazwa została zdefiniowana w wygenerowanych zmiennych (domyślnie *log.txt*).

Logowanie komunikatu odbywa się poprzez grupę funkcji *log_**(*poziom*), gdzie *** to poziom logowania, zachowujące się tak samo jak funkcja *printf()* z biblioteki standardowej języka C - pierwszy argument to ciąg znaków z znakami formatującymi, reszta argumentów to formatowane wartości.

Listing 3.12 demonstruje logowanie i powinien zapisać do pliku *log.txt* w katalogu z plikiem wykonywalnym komunikaty podobne do przedstawionych na listingu 3.13:

```
log_create();
log_debug("komunikat #%d", 1);
log_debug("komunikat #%d", 2);
log_fatal("%s #%d", "komunikat", 3);
log_destroy();
```

Listing 3.12: Demonstracja logowania

```
[DEBUG] (/home/user/repo/src/main.c:45) main:
komunikat #1
komunikat #2
[FATAL] (/home/user/repo/src/main.c:47) main:
komunikat #3
```

Listing 3.13: Wynik logowania

Jednostka junk

Proste funkcje i makra które mogą być potencjalnie używane we wszystkich modułach biblioteki, ale nie zostały uznane za wystarczająca skomplikowane, aby uzasadnić wydzielenia do osobnej jednostki.

Jednostka definiuje stałe preprocessora *PLATFORM_** przedstawione na listingu 3.14 używane do rozpoznania systemu operacyjnego, na którym budowany jest silnik (Linux, MacOS, Windows).

```
#if defined(_linux) || defined(__linux__) || defined(linux)
#define PLATFORM_LINUX
#elif defined(__APPLE__)
#define PLATFORM_APPLE
#elif defined(_WIN32) || defined(__WIN32__)
|| defined(WIN32) || defined(_WIN64)
#define PLATFORM_WINDOWS
#endif
```

Listing 3.14: Stałe preprocessora używane do rozpoznania systemu operacyjnego

Funkcja *strstrip()* usuwa początkowe i końcowe białe znaki z ciągu znaków.

Funkcja *count_bits()* zlicza bity w liczbie całkowitej używając metody Briana Kernighana [47]. Przykładem użycia jest określenie liczby flag ustawionych w wyliczeniu.

Makra *HASH_** ukrywają detale użycie funkcji skrótu biblioteki *xxHash* i zostały przedstawione na listingu 3.15.

```
hash_t hash;
HASH_START(hashState)
HASH_UPDATE(hashState, &num, sizeof(num))
HASH_UPDATE(hashState, str, strlen(str))
HASH_UPDATE(hashState, &object->field, sizeof(object->field))
HASH_DIGEST(hashState, hash)
HASH_END(hashState)
log_debug("Hash value is %zu", hash);
```

Listing 3.15: Przykład użycia funkcji skrótu

Makro *UNREACHABLE* pozwala na optymalizację kodu poprzez oznaczenie punktów programu, które nigdy nie są napotykane przez przepływ sterowania. Jego definicja zależy od konfiguracji: w *Debug* sprowadza się do asercji *assert(0)*, a w *Release* do funkcji wbudowanej kompilatora GCC *__builtin_unreachable()*. Przykładowo określenie nieosiągalności przypadku domyślnej instrukcji *switch* bądź bloku *else* informuje o kompletności sprawdzanych warunków, co zostało przedstawione na listingu 3.16.

```
if (type == directional) {
    ...
} else if (type == point) {
    ...
} else {
    UNREACHABLE;
}
```

Listing 3.16: Przykład użycia makra UNREACHABLE

Jednostka definiuje też makra używające formy metaprogramowania w celu dodania nowych struktur kontrolnych [43] upraszczających iterowanie po strukturach danych biblioteki *uthash*, której użycie zostało przedstawione na listingu 3.17.

```
utarray_FOREACH_ELEM_DEREF (tree_node *, node, tree->nodes) {  
    tree_set_dirty(tree, node);  
}
```

Listing 3.17: Przykład iteracji używając makra `utarray_FOREACH_ELEM_DEREF`

Jednostka platform

Główna część rdzenia implementująca obiekt globalny *platform* odpowiedzialny za tworzenie i niszczenie globalnego stanu używanego przez system logowania i funkcje wieloplatformowe, z których najważniejsze zostały opisane poniżej.

Funkcja *panic()* pozwala na zamknięcie programu z kodem wyjścia oznaczającym nieudane wykonanie po wystąpieniu fatalnego błędu. Jest ona używana przez makro *verify()*, które podobnie do makra *assert()* pozwala na testowanie warunku logicznego i przerwanie działania programu gdy przyjmuje on wartość fałsz, ale w przeciwnieństwie do niego działa też w konfiguracji *Release*.

Funkcje *get_executable_dir_path()* i *get_path dirname()* pozwalają na odkrycie ścieżki z katalogiem zawierającym plik wykonywalny, co jest potrzebne do pełnego określenia struktury plików opisanych przez wygenerowane stałe. Na systemie Linux używana jest funkcja *readlink()* do odczytania pliku */proc/self/exe* oraz funkcja *dirname()*. Na systemie Windows używana jest funkcja *GetModuleFileName()* oraz funkcja *PathRemoveFileSpec()*.

Funkcje *write_text_file()* i *read_text_file()* pozwalają na odczyt i zapis plików tekstowych i są używane do obsługi konfiguracji i kodu źródłowego shaderów.

3.3.3. I/O

Silnik wczytuje ze ścieżek zaszytych w zmiennych globalnych następujące zasoby wyjściowe:

- konfiguracja globalna (plik tekstowy INI),
- kod GLSL shadera (plik tekstowy GLSL),
- baza zasobów (baza danych).

Wszystkie zasoby wejściowe które muszą być łatwo edytowalne przez użytkownika są plikami tekstowymi i są bezpośrednio kopowane przez potok zasobów do katalogu budowania stojąc się zasobami wyjściowymi. Reszta zasobów wejściowych staje się częścią bazy zasobów będącej plikiem bazy danych SQLite.

SQLite [34] to biblioteka języka C implementująca silnik relacyjnej bazy danych SQL. Jest ona bardzo popularnym wyborem jako format pliku używany do utrwalania stanu aplikacji na dysku z wielu powodów, do których zalicza się prosta użycia, wysoka wydajność, bogata wewnętrzna struktura oferowana przez relacyjną bazę danych i formę łatwego do dystrybucji pojedynczego samodzielnego pliku na dysku [48].

Moduł I/O zawiera obiekty używane do wczytywania i zapisywania powyższych zasobów wraz z walidacją ich formatu i wewnętrznej struktury - interpretacją danych zajmują się dalsze moduły.

Jednostka config

Obiekt `data_config` reprezentujący pojedynczy plik INI zawierający jeden z dwóch rodzajów konfiguracji: konfigurację globalną lub konfigurację zasobów.

Konfiguracja globalna jest ładowana na samym początku inicjalizacji silnika i pozwala użytkownikowi na sterowanie jego działania poprzez zmianę następujących zmiennych:

- sekcja `graphics`:
 - `WindowWidth`, `WindowHeight`, `WindowTitle`: szerokość, wysokość i tytuł okna,
 - `EnabledInstancing`: włączenie instancjonowania,
 - `MaxPrimitiveElementCount`: maksymalna liczba prymitywów renderowania,
 - `Font`: czcionka,
- sekcja `controls`:
 - `Enabled`: obsługa danych wejściowych myszy i klawiatury,
- sekcja `settings`:
 - `StartScene`: nazwa sceny ładowanej z bazy zasobów.

Konfiguracja zasobów jest używana wyłącznie przez potok zasobów i zawiera dodatkowe informacje o przetwarzanym zasobie wejściowym takie jak:

- sekcja `skybox`:
 - `Name`: nazwa używanej tekstury skybox.
- Konfiguracja jest zbiorem par klucz-wartość. Wartości mogą być liczbą całkowitą lub ciągiem znaków i dla brakujących kluczy mają wartość domyślną. Odczyt i zapis odbywa się przy pomocy metod `load()` i `save()`.

Jednostka sql_db

Obiekt `sql_db` reprezentujący połączenie z plikiem bazy danych SQLite.

SQLite posiada dynamiczny i słaby system typowania posiadający 5 typów prostych: NULL, INTEGER (liczba całkowita maksymalnie 64-bitowa), REAL (64-bitowa liczba zmiennoprzecinkowa), TEXT (ciąg znaków UTF-8/16) i BLOB: (blok pamięci). Obiekt `db` rozszerza ten ubogi system typów nowymi typami złożonymi bezpośrednio odpowiadającymi typom języka C zdefiniowanych w konfiguracji generatora: BYTE (`uint8_t`), INT (`uint32_t`), FLOAT (`float`), VEC2(`vec2`), VEC3(`vec3`), VEC4(`vec4`), MAT4(`mat4`), TEXT(`UT_string *`), i KEY(`hash_t`). Dodatkowo każdy typ posiada wersję tablicową `*_ARRAY` (`BYTE_ARRAY`, `INT_ARRAY` itd.).

Obiekt `sql_db` pozwala na przeprowadzanie standardowych operacji wyboru (ang. `select`) i umieszczania (ang. `insert`) rekordów do wybranej tabeli. Baza danych SQLite wciąż wewnętrznie używa typów prostych, ale wygenerowane przy użyciu X-makro metody `select_*`() i `insert_*`() automatycznie przeprowadzają serializację i deserializację typów złożonych.

Jednostka asset_db

Obiekt `asset_db` reprezentuje bazę zasobów. Używa on wewnętrznie obiektu `sql_db` i podobnie jak w nim użyto X-makro do dodania metod wyboru i umieszczania wartości dla specyficznej tabeli, kolumny i klucza.

Listing 3.18 demonstruje wybór i umieszczenie kod wartości FLOAT z tabeli `directLight`, kolumny `intensity` i klucza `key`.

```

float value =
    asset_db_select_directLight_intensity_float(assetDb, key).value;
asset_db_insert_directLight_intensity_float(assetDb, key,
    data_float_temp(value));

```

Listing 3.18: Deserializacja i serializacja wartości zmiennoprzecinkowej

Obiekt ten jest intensywnie używany przez moduł zasobów do implementacji serializacji i deserializacji.

3.3.4. Zasoby

Moduł zawierający obiekty zasobów, które pozwalają na serializację i deserializację indywidualnych zasobów z bazy zasobów do formy używalnej przez resztę kodu silnika.

Wszystkie obiekty zasobów mają nazwy w formie `asset_*` i wspólnie posiadają następujące pola i metody:

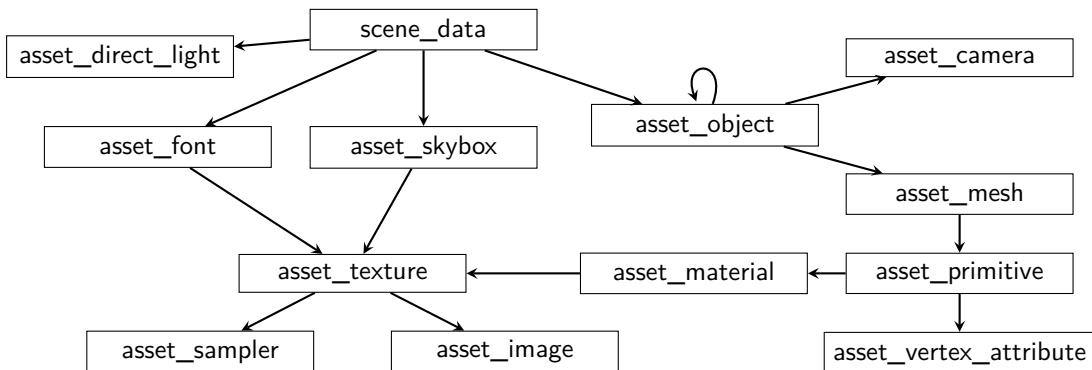
- klucz zasobu: jednoznacznie identyfikuje obiekt jako unikalny zasób i jest otrzymywany poprzez użycie funkcji skrótu na jego polach.
- wskaźnik do obiektu `scene_data` z modułu sceny: obiekty zasobów są zarządzane przez wskazywany obiekt zawierający dane sceny. Może być on używany podczas deserializacji.
- wskaźniki `prev` i `next`: pozwalają na użycie obiektu w liście dwukierunkowej biblioteki `uthash`.
- `calculate_key()`: oblicza klucz zasobu, musi być wywoływany po każdej modyfikacji obiektu.
- `serialize()`: serializacja, czyli zapis pól do bazy zasobów. Pola będące typami złożonymi wspieranymi przez obiekt `asset_db` są serializowane przy użyciu jego odpowiedniej metody `insert_*`() przyjmującej klucz zasobu. Dla pól będących obiektami zasobów wywoływana jest ich metoda `serialize()`.
- `deserialize()`: deserializacja, czyli odczyt pól z bazy zasobów. Metoda przyjmuje klucz zasobu żądanego zasobu i w sposób analogiczny do serializacji wypełnia pola używając metod `select_*`() obiektu `asset_db` i metod `deserialize()`.

Powyższy interfejs jest zdefiniowany przez makra w jednostce `common` i jest używany w reszcie jednostek modułu do implementacji obiektów zasobów.

Wspierane są następujące rodzaje obiektów zasobów zawierające dane opisujące:

- `asset_object`: węzeł sceny,
- `asset_mesh`: siatkę,
- `asset_primitive`: prymityw,
- `asset_vertex_attribute`: atrybut wierzchołka,
- `asset_camera`: kamera,
- `asset_direct_light`: światło bezpośrednie,
- `asset_skybox`: skybox,
- `asset_font`: czcionka,
- `asset_material`: materiał,
- `asset_texture`: tekstura,
- `asset_image`: obraz,
- `asset_sampler`: próbnik.

Relacje pomiędzy nimi są inspirowane formatem glTF i zostały pokazane na diagramie 3.3.



Rysunek 3.3: Relacje pomiędzy obiektami zasobów w silniku (opracowanie własne)

Zasób węzła **asset_object**

Zasób węzła **asset_object** jest kontenerem zawierającym referencje do obiektów zasobów.

Węzeł zawiera macierz 4x4 z lokalną transformacją przestrzeni oraz wskaźniki do zasobów (albo wartość NULL):

- siatki,
- kamery,
- potomnych węzłów.

Ostateczna pozycja na scenie dla powyższych zasobów nie jest opisana bezpośrednio i musi zostać obliczana używając modelowania hierarchicznego przy pomocy grafu sceny - każdy zasób węzła jest używany do stworzenia odpowiedniego węzła grafu sceny.

Zasób siatki **asset_mesh**

Zasób siatki **asset_mesh** reprezentuje geometrię na scenie i składa się z listy prymitywów. Podział na siatkę i prymitywy ma na celu zmniejszenie redundancji. Przykładowo siatka modelu auta może zawierać 4 identyczne koła renderowanych 4 razy tym samym prymitywem. Siatka jest tożsama z geometrią pojedynczego modelu przygotowanego w programie do modelowania 3D.

Zasób siatki **asset_primitive**

Zasób prymitywu **asset_primitive** reprezentuje część siatki obiektu. Jeden prymityw zawiera wszystkie dane wymagane do wygenerowania jednego polecenia rysowania i składa się z następujących elementów:

- rodzaj topologii (*VkPrimitiveTopology*),
- atrybuty wierzchołka,
- indeksy wierzchołków,
- materiał.

Prymityw zawiera po jednym zasobie atrybutu wierzchołka dla każdego wspieranego typu atrybutów (*vertex_attribute_type*):

- *position*: pozycje,
- *normal*: normalne,

- *color*: kolory,
- *texcoord*: koordynaty tekstury,
- *tangent*: styczne.

Dodatkowo jeden zasób atrybutów jest używany do przechowywania indeksów wierzchołków.

Zasoby zawierają jedynie dane potrzebne do konstrukcji wierzchołków siatki - ostatecznie używany format wierzchołka (w tym rozdzielenie lub separacja atrybutów), tylko przez strumień wierzchołków *vertex_stream* w module renderera.

Zasób siatki *asset_vertex_attribute*

Zasób atrybutu wierzchołka *asset_vertex_attribute* reprezentuje dane pojedynczego atrybutu przechowywane w postaci tablicy komponentów, których typ to *uint32_t*, *vec2*, *vec3* lub *vec4*.

Zasób siatki *asset_camera*

Zasób kamery *asset_camera* zawiera parametry, których część zależy od rodzaju używanego rzutu:

- rzutowanie perspektywiczne:
 - *fovY*: pionowy kąt widzenia,
 - *aspectRatio*: proporcje okna (stosunek szerokości do wysokości),
- rzutowanie ortogonalne:
 - *magX*: poziome powiększenie widoku,
 - *magY*: pionowe powiększenie widoku.

Dodatkowo pola *nearZ* oraz *farZ* definiują są odległości bliskiej i dalekiej płaszczyzny przycinania wzdłuż osi +Z.

Powysze parametry są używane do uzyskania macierzy rzutowania. Ostateczna pozycja i rotacja kamery (i tym samym macierz widoku) musi być wyliczana na podstawie wynikowej transformacji przestrzeni węzła z kamerą.

Zasób siatki *asset_direct_light*

Zasób światła bezpośredniego *asset_direct_light* reprezentuje jedno światło na scenie. Jego struktura jest inspirowana rozszerzeniem *KHR_lights_punctual* formatu glTF. Dostępne są dwa rodzaje światel:

- kierunkowe (ang. directional light),
- punktowe (ang. point light).

Wszystkie rodzaje światel posiadają parametry:

- intensywność: jasność światła (float),
- kolor: wartość RGB w liniowej przestrzeni kolorów (vec3).

Światło kierunkowe definiuje dodatkowy parametr kierunku będący wektorem w przestrzeni świata (vec3). Światło punktowe definiuje:

- pozycja: punkt w przestrzeni świata (vec3),
- zakres: promień sfery zdefiniowanej w pozycji światła. poza którą przez tłumienie intensywność osiąga zero (float).

Zasoby skybox *asset_skybox*

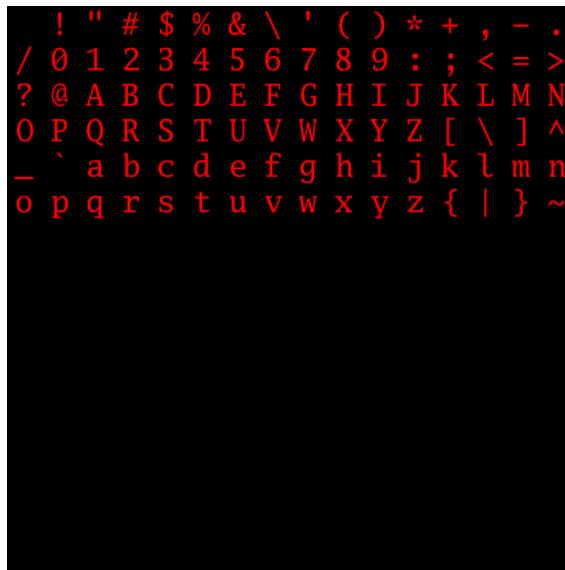
Zasób skybox *asset_skybox* opisuje składa się z zasobu tekstuury oraz nazwy używanej przez konfigurację globalną.

Zasób czcionki *asset_font*

Zasób czcionki *asset_font* jest opisuje czcionkę bitmapową używaną do renderowania tekstu. Składa się z:

- nazwa: używana przez konfigurację globalną,
- zasób tekstuury,
- alfabet: ciąg znaków ASCII,
- rozmiar znaku: rozmiar jednego glifu w pikselach (uint32_t).

Rysunek 3.4 przedstawia przykładową teksturę dla czcionki Go-Mono [49].



Rysunek 3.4: Przykładowa tekstura dla czcionki Go-Mono [49] (opracowanie własne)

Zasób materiału *asset_material*

Zasób materiału *asset_material* reprezentuje parametry używanego podczas renderowania powierzchni prymitywów przy pomocy następujących parameterów:

- *baseColorFactor*: współczynnik koloru podstawowego (vec4);
- *metallicFactor*: współczynnik metaliczności zakres, [0, 1];
- *roughnessFactor*: współczynnik chropowatości [0, 1];
- *metallicRoughnessTexture*: tekstura metaliczności-chropowatości, opcjonalna;
- *normalMapTexture*: mapa normalnych, opcjonalna.

Zasób tekstuury *asset_texture*

Zasób tekstuury *asset_texture* reprezentuje próbkowany obraz i składa się z zasobu obrazu oraz zasobu próbnika.

Zasób obrazu *asset_image*

Zasób obrazu *asset_image* zawiera dane obrazu w postaci nieskompresowanej bitmapy. Bitmapa to prostokątna tablica pikseli opisywana przez następujące parametry:

- szerokość i wysokość (`uint32_t`);
- liczba ścian: domyślnie 1 ściana, 6 ścian dla tekstur sześciennych (`uint32_t`);
- liczba kanałów: specyfikuje liczbę komponentów i tym samym rozmiar piksela, jeden kanał jest reprezentowany przez jeden bajt (`uint32_t`).

Zasób próbnika *asset_sampler*

Zasób próbnika *asset_sampler* reprezentuje parametry używane do stworzenia próbnika obrazu:

- *magFilter*: filtr powiększania (`VkFilter`),
- *minFilter*: filtr pomniejszania (`VkFilter`),
- *addressModeU*: tryb adresowania współrzędnych tekstur poza przedziałem [0, 1] dla osi X (`VkSamplerAddressMode`),
- *addressModeV*: tryb adresowania współrzędnych tekstur poza przedziałem [0, 1] dla osi Y (`VkSamplerAddressMode`).

Potok zasobów *asset_pipeline*

Potok zasobów składa się z dwóch części:

- narzędziwa wiersza poleceń *asset_pipeline*,
- skryptu Python *asset_pipeline*.

Skrypt skanuje podkatalog z zasobami wejściowymi i wywołuje narzędzie z argumentami będącymi ścieżką zasoby wejściowego rodzajem konwertowanego zasobu wyjściowego.

Przykładowo potok zasobów może wywołać narzędzie 5 razy w sposób pokazany na listingu 3.19 tworząc pustą konfigurację globalną oraz pustą bazę zasobów wypełnioną zasobem skybox, zasobem czcionki Go-Mono oraz zasobami składającymi się na scenę Sponza opisanej plikiem glTF.

```
asset_pipeline empty_config
asset_pipeline empty_assets
asset_pipeline cubemap "skybox1" "/home/user/repo/cmake-build-debug/assets/
    ↪ cubemap/skybox1" png
asset_pipeline font "Go-Mono" "/home/sszczyrb/repo/cmake-build-debug/assets/font/
    ↪ Go-Mono.ttf"
asset_pipeline gltf "sponza" "/home/user/repo/cmake-build-debug/assets/gltf/
    ↪ sponza"
```

Listing 3.19: Komendy wywoływane przez przykładowy potok zasobów

3.3.5. Vulkan

Vulkan to moduł zapewniający obiekty abstrahujące API Vulkan przeznaczone do użycia przez moduł renderera.

Moduł implementuje też mechanizm jednolitych buforów inspirowany grą *Tom Clancy's Rainbow Six Siege* [46]. Używa on automatycznie wygenerowanego kodu z jednostki descriptors i pozwala na

wymaganą przez techniki renderowania bez dowiązań konsolidację zasobów buforów i obrazów w pojedyncze zasoby opisane jednym zbiorem deskryptorów.

Urządzenie device

Obiekt *device* reprezentuje urządzenie. Jest on podstawowym obiektem przygotowującym podstawowe funkcjonalności używane przez resztę obiektów modułu.

Obiekt oferuje następujące funkcjonalności:

- obsługa okna,
- inicjalizacja Vulkan,
- wykonywanie poleceń one-shot.

Okno jest tworzone przy użyciu biblioteki GLFW i jest reprezentowane obiektem *GLFWwindow*. Listing 3.20 ilustruje stworzenie okna i użycie go do zarejestrowania funkcji wywołań zwrotnych używane do wykrywania zmiany rozmiaru okna oraz przechwytywania danych wejściowych myszy i klawiatury. Kursor myszy jest ukryty i jego fizyczna pozycja w oknie jest centrowana co klatkę - program ma dostęp do wirtualnej pozycji, co pozwala na nieograniczony ruch myszy. Wirtualna pozycja kursowa jest używana do implementacji sterowania kamery.

```
// Inicjalizacja biblioteki GLFW.  
assert(glfwInit() == GLFW_TRUE);  
assert(glfwVulkanSupported() == GLFW_TRUE);  
  
// Stworzenie okna.  
glfwDefaultWindowHints();  
glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);  
GLFWwindow *window = glfwCreateWindow(640, 480, "Window caption", NULL, NULL);  
  
// Rejestracja funkcji wywołania zwrotnego.  
glfwSetWindowUserPointer(window, callbackData);  
glfwSetFramebufferSizeCallback(window, framebuffer_resize_callback);  
glfwSetKeyCallback(window, key_callback);  
glfwSetCursorPosCallback(window, mouse_callback);  
  
// Wirtualny kurSOR myszy.  
glfwSetInputMode(vkd->window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

Listing 3.20: Użycie biblioteki GLFW do stworzenia okna i rejestracji funkcji wywołań zwrotnich

Obiekt inicjalizuje podstawowe obiekty Vulkan: instancja, powierzchnia okna, urządzenia fizyczne, urządzenie logiczne, kolejki oraz wskaźniki funkcji rozszerzeń.

Instancja deklaruje użycie wersji Vulkan 1.2 oraz następujących rozszerzeń instancji:

- *VK_KHR_get_physical_device_properties2*;
- rozszerzenia zwrócone przez funkcję *glfwGetRequiredInstanceExtensions()* używane do stworzenia powierzchni okna (*VK_KHR_surface* i dodatkowe rozszerzenie zależące od systemu okien);
- w trybie debugowania *VK_EXT_debug_utils*.

Dodatkowo w trybie debugowania używana jest warstwa walidacji *VK_LAYER_KHRONOS_validation* oraz komunikator debugowania dla instancji.

Powierzchnia okna jest tworzona przy użyciu funkcji `glfwCreateWindowSurface()`.

Lista urządzeń fizycznych jest przefiltrowana do listy kandydatów przy użyciu następujących kryteriów wsparcia:

- wersja Vulkan: Vulkan 1.2;
- dostępne rodziny kolejek: graficzne i prezentacji;
- rozszerzenia urządzenia:
 - `VK_KHR_swapchain`;
 - `VK_KHR_dynamic_rendering`;
 - `VK_KHR_shader_non_semantic_info`: w trybie debugowania, używane przez `debugPrintf`;
- wcześniej utworzonej powierzchni okna;
- funkcjonalności urządzenia fizycznego:
 - Vulkan 1.0 Core:
 - * `samplerAnisotropy`: filtrowanie anizotropowego;
 - * `shaderUniformBufferArrayDynamicIndexing`: jednolite dynamiczne indeksowanie tablic buforów uniform;
 - * `shaderSampledImageDynamicIndexing`: jednolite dynamiczne indeksowanie tablic próbkowanych obrazów;
 - * `multiDrawIndirect`: polecenia wielokrotnego rysowania pośredniego;
 - * `drawIndirectFirstInstance`: polecenia rysowania pośredniego z offsetem indeksu instancji;
 - Vulkan 1.2 Core:
 - * `descriptorIndexing`: indeksowanie deskryptorów;
 - * `shaderSampledImageNonUniformIndexing`: niejednolite dynamiczne indeksowanie tablic próbkowanych obrazów;
 - * `descriptorBindingVariableDescriptorCount`: dowiązania deskryptora o zmiennej wielkości;
 - * `descriptorBindingPartiallyBound`: częściowe dowiązanie deskryptorów;
 - * `runtimeDescriptorArray`: nieograniczone tablice deskryptorów;
 - * `scalarBlockLayout`: układ pamięci scalar;
 - `VK_KHR_dynamic_rendering`:
 - * `dynamicRendering`: dynamiczne przebiegi renderowania.

Każde urządzenie fizyczne listy kandydatów ma nadawany ranking poprzez oceniane jego typu od najlepszego do najgorszego:

- dyskretne GPU,
- zintegrowane GPU,
- wirtualne GPU,
- CPU.

Urządzenie fizyczne z najwyższym rankingiem jest ostatecznie wybierane z listy kandydatów.

Urządzenie logiczne jest tworzone wspierając funkcjonalności sprawdzanie podczas wyboru urządzenia fizycznego oraz albo po jednej kolejce graficznej i prezentacji, albo jedna kolejka „uniwersalna” wspierająca obie rodziny poleceń.

Po stworzeniu urządzenia logicznego funkcja `vkGetDeviceProcAddr()` jest używana do pobrania następujących funkcji rozszerzeń:

- *VK_KHR_dynamic_rendering*
 - vkCmdBeginRenderingKHR,
 - vkCmdEndRenderingKHR,
- *VK_EXT_debug_utils*
 - vkCmdBeginDebugUtilsLabelEXT,
 - vkCmdEndDebugUtilsLabelEXT,
 - vkCmdInsertDebugUtilsLabelEXT,
 - vkSetDebugUtilsObjectNameEXT.

Polecenia one-shot są przeznaczone do jednokrotnego transferu dużych ilości danych z bazy zasobów do pamięci DEVICE_LOCAL przez rozpoczęciem pętli głównej renderowania.

Bufor poleceń one-shot jest alokowany podczas tworzenia obiektu *device* z osobnej puli komend one-shot. Jest on przeznaczona do użycia z kolejką graficzną i stworzona z flagą TRANSIENT, która wskazuje sterownikowi graficznemu, że zaalokowane bufory komend będą krótkotrwałe i zresetowane bądź zwolnione w stosunkowo krótkim czasie, co teoretycznie pozwala sterownikowi na optymalizację metody alokacji pamięci.

Nagrywanie poleceń one-shot rozpoczyna się wywołaniem metody *begin_one_shot_commands()* rozpoczynającej nagrywanie bufora poleceń funkcją *vkBeginCommandBuffer()* z flagą użycia ONE_TIME_SUBMIT wskazującą, że będzie on wykonany tylko jeden raz.

Nagrywanie jak kończone wywołaniem metody *end_one_shot_commands()*, która wykonuje następujące czynności:

- kończy nagrywanie bufora komend one-shot funkcją *vkEndCommandBuffer()*,
- wysyła bufor poleceń do kolejki graficznej funkcją *vkQueueSubmit()*,
- czeka na CPU aż kolejka zakończy wykonywanie poleceń na GPU funkcją *vkQueueWaitIdle()*,
- resetuje bufor komend one-shot poprzez reset całej puli komend one-shot funkcją *vkResetCommandPool()*.

Synchronizacja między krokiem 2. i 4. zapobiega próbie zresetowania bufora poleceń wciąż używanego przez GPU. Resetowanie puli poleceń automatycznie resetuje zaalokowane z niego bufory poleceń i jest uznawane za szybsze od manualnego resetowania buforów poleceń przez warstwy walidacji, której ostrzeżenie przedstawia listing 3.21

Validation Performance Warning:

```
[ UNASSIGNED-BestPractices-vkCreateCommandPool-command-buffer-reset ]
Object 0: handle = 0x626000015100, type = VK_OBJECT_TYPE_DEVICE;
| MessageID = 0x8728e724
| vkCreateCommandPool(): VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER is set.
Consider resetting entire pool instead.
```

Listing 3.21: Ostrzeżenie wydajnościowe wyemitowane przez warstwy walidacji

Bufor poleceń one-shot może być wypełniony dowolnymi poleceniami graficznymi i transferu, ale obiekt oferuje metody pomocnicze wykonujące podstawowe operacje używane podczas transferu danych.

Metody *one_shot_copy_buffer_to_buffer()* i *one_shot_copy_buffer_to_image()* kopią dane w obrębie GPU z bufora do bufora lub obrazu używając polecień transferu *vkCmdCopyBuffer()* i *vkCmdCopyBufferToImage()*. Metoda *one_shot_copy_buffer_to_image()* wymaga układu docelowego obrazu *TRANSFER_DST_OPTIMAL*.

Metoda `one_shot_generate_mipmaps()` generuje poziomy mipmap dla tekstur 2D. Baza zasobów nie przechowuje poziomów mipmap, dlatego metoda jest wywoływania po transferze danych zasobu obrazu do pierwszego poziomu mipmapy obrazu w celu automatycznej generacji reszty poziomów. Metoda zakłada układ obrazu `TRANSFER_DST_OPTIMAL` i pozostawia go w układzie `SHADER_READ_ONLY_OPTIMAL`.

Metoda `one_shot_transition_image_layout()` zmienia układ całego obrazu, tj. jego wszystkich warstw i poziomów mipmap. Używa ona do tego metody `transition_image_layout_command()`.

Metoda `transition_image_layout_command()` przeprowadza dla części obrazu przejście ze starego układu na nowy układ poprzez nagranie bariery pamięci obrazu `VkImageMemoryBarrier`. Użycie bariery potoku wymaga zdefiniowania całej zależności pamięci, dlatego każde przejście układu obrazów wymaga zdefiniowania jak dokładnie obraz będzie używany po przejściu i zakodowania tej wiedzy używając źródłowych i docelowych etapów potoku oraz zakresów dostępów. Z tego powodu implementacja metody musi być świadoma późniejszego użycie obrazu po przejściu, co zostało podsumowane w tabeli 3.1.

przejście układu obrazu użycie obrazu po przejściu	zależność pamięci (zakresy dostępu, etapy potoku)
UNDEFINED -> <code>TRANSFER_DST_OPTIMAL</code> inicjalizacja poleceniem transferu	0 -> <code>TRANSFER_WRITE</code> , <code>TOP_OF_PIPE</code> -> <code>TRANSFER</code>
<code>TRANSFER_DST_OPTIMAL</code> -> <code>TRANSFER_SRC_OPTIMAL</code> źródło polecenia transferu po inicjalizacji (generacje mmap)	<code>TRANSFER_WRITE</code> -> <code>TRANSFER_READ</code> , <code>TRANSFER</code> -> <code>TRANSFER</code>
<code>TRANSFER_SRC_OPTIMAL</code> -> <code>SHADER_READ_ONLY_OPTIMAL</code> odczyt przez shader fragmentów (po generacji mmap)	<code>TRANSFER_READ</code> -> <code>SHADER_READ</code> , <code>TRANSFER</code> -> <code>FRAGMENT_SHADER</code>
<code>TRANSFER_DST_OPTIMAL</code> -> <code>SHADER_READ_ONLY_OPTIMAL</code> odczyt przez shader fragmentów (brak generacji mmap)	<code>TRANSFER_WRITE</code> -> <code>SHADER_READ</code> , <code>TRANSFER</code> -> <code>FRAGMENT_SHADER</code>
UNDEFINED -> <code>COLOR_ATTACHMENT_OPTIMAL</code> dołączenie koloru (pierwszy raz z klatce)	0 -> <code>COLOR_ATTACHMENT_WRITE</code> , <code>TOP_OF_PIPE</code> -> <code>COLOR_ATTACHMENT_OUTPUT</code>
UNDEFINED -> <code>DEPTH_STENCIL_ATTACHMENT_OPTIMAL</code> dołączenie gębi/szablonu (odczyt i zapis, pierwszy raz z klatce)	0 -> <code>DEPTH_STENCIL_ATTACHMENT_READ</code> <code>DEPTH_STENCIL_ATTACHMENT_WRITE</code> , <code>TOP_OF_PIPE</code> -> <code>EARLY_FRAGMENT_TESTS</code> <code>LATE_FRAGMENT_TESTS</code>

DEPTH_STENCIL_ATTACHMENT_OPTIMAL -> DEPTH_STENCIL_READ_ONLY_OPTIMAL dołczanie głębi/szablonu (tylko do odczytu, wcześniejszy odczyt i zapis)	DEPTH_STENCIL_ATTACHMENT_WRITE -> DEPTH_STENCIL_ATTACHMENT_READ SHADER_READ, EARLY_FRAGMENT_TESTS LATE_FRAGMENT_TESTS -> EARLY_FRAGMENT_TESTS FRAGMENT_SHADER
DEPTH_STENCIL_READ_ONLY_OPTIMAL -> DEPTH_STENCIL_ATTACHMENT_OPTIMAL dołczanie głębi/szablonu (odczyt i zapis, wcześniejszy tylko do odczytu)	DEPTH_STENCIL_ATTACHMENT_READ SHADER_READ -> DEPTH_STENCIL_ATTACHMENT_WRITE, EARLY_FRAGMENT_TESTS FRAGMENT_SHADER -> EARLY_FRAGMENT_TESTS FRAGMENT_SHADER
COLOR_ATTACHMENT_OPTIMAL -> SHADER_READ_ONLY_OPTIMAL dołczanie koloru (tylko do odczytu, wcześniejszy odczyt i zapis)	COLOR_ATTACHMENT_WRITE -> SHADER_READ, COLOR_ATTACHMENT_OUTPUT -> FRAGMENT_SHADER
SHADER_READ_ONLY_OPTIMAL -> COLOR_ATTACHMENT_OPTIMAL dołczanie koloru (odczyt i zapis, wcześniejszy tylko do odczytu)	SHADER_READ -> COLOR_ATTACHMENT_WRITE, FRAGMENT_SHADER -> COLOR_ATTACHMENT_OUTPUT
COLOR_ATTACHMENT_OPTIMAL -> COLOR_ATTACHMENT_OPTIMAL dołczanie koloru (odczyt i zapis w poprzednim i obecnym przebiegu renderowania)	COLOR_ATTACHMENT_WRITE -> COLOR_ATTACHMENT_READ COLOR_ATTACHMENT_WRITE, COLOR_ATTACHMENT_OUTPUT -> COLOR_ATTACHMENT_OUTPUT
DEPTH_STENCIL_ATTACHMENT_OPTIMAL -> DEPTH_STENCIL_ATTACHMENT_OPTIMAL dołczanie głębi/szablonu (odczyt i zapis w poprzednim i obecnym przebiegu renderowania)	DEPTH_STENCIL_ATTACHMENT_WRITE -> DEPTH_STENCIL_ATTACHMENT_READ DEPTH_STENCIL_ATTACHMENT_WRITE, LATE_FRAGMENT_TESTS -> EARLY_FRAGMENT_TESTS LATE_FRAGMENT_TESTS
COLOR_ATTACHMENT_OPTIMAL -> PRESENT_SRC_KHR prezentacja obrazu	COLOR_ATTACHMENT_WRITE -> MEMORY_READ, COLOR_ATTACHMENT_OUTPUT -> BOTTOM_OF_PIPE

Tablica 3.1: Logika przejść układu i zależności pamięci obrazu (opracowanie własne)

Metoda jest używana w poleceniach one-shot oraz przez graf renderowania do zapewnienia przejść układu i zależności pamięci obrazów pomiędzy przebiegami renderowania.

Łańcuch wymiany swap_chain

Obiekt *swap_chain* reprezentuje łańcuch wymiany i zawiera następujące elementy:

- łańcuch wymiany *VkSwapchain*,
- lista uchwytów prezentowalnych obrazów *VkImage*,
- lista widoków prezentowalnych obrazów *VkImageView*.

Preferowany tryb prezentacji to *MAILBOX*. Jeśli jest niedostępny, to wybierany jest zawsze wspierany tryb prezentacji *FIFO*.

Preferowany format i przestrzeń kolorów to *B8G8R8A8_SRGB* i *SRGB*. Jeśli nie są dostępne, to wybierane są pierwszy dostępny. Formaty z rodziny *BGRA* i przestrzeń kolorów *SRGB* powierzchni okna są zawsze wspierane na systemie Windows i w czasie pisania pracy są one wspierane w ponad 96% systemów Linux [20]. Obliczenia modelu oświetlenia będące wyjściem shaderów fragmentów odbywają się w liniowej przestrzeni kolorów. Użycie formatu *B8G8R8A8_SRGB* zamiast *B8G8R8A8_UNORM* pozwala na ominięcie ręcznej konwersji z przestrzeni liniowej do *SRGB* podczas renderowania bezpośrednio do prezentowalnej tekstury - konwersja zostanie automatycznie przeprowadzona przez sterownik.

Lista uchwytów prezentowalnych obrazów jest wypełniana w pętli używając funkcji *vkGetSwapchainImagesKHR()*. Następnie tworzone są widoki obrazów pozwalające przebiegom renderowania *render_pass* używać prezentowalne obrazy jako dodatki koloru.

Bufor buffer

Obiekt *buffer* reprezentuje pojedynczy bufor Vulkan. Rozróżniane są następujące typy bufora (*buffer_type*):

- *geometry_index*: bufor indeksów geometrii,
- *geometry_vertex*: bufor wierzchołków geometrii,
- *uniform*: bufor uniform,
- *indirect_draw*: bufor poleceń rysowania pośredniego.

Bufory wierzchołków, indeksów i poleceń rysowania pośredniego są źródłem danych odczytywanych przez stałe funkcji potoku graficznego. Bufory uniform są źródłem danych odczytywanych przez shadery.

Bufor zawiera dwa obiekty Vulkan: bufor *VkBuffer* oraz dowiązaną do niego alokację pamięci *VkDeviceMemory*. Typ bufora przekłada się na flagi używane podczas ich tworzenia, co zostało podsumowane w tabeli 3.2.

Typ bufora	flagi użycia bufora <i>VkBufferUsageFlags</i>	flagi właściwości pamięci <i>VkMemoryPropertyFlags</i>
<i>geometry_index</i>	TRANSFER_DST VERTEX_BUFFER	DEVICE_LOCAL
<i>geometry_vertex</i>	TRANSFER_DST INDEX_BUFFER	DEVICE_LOCAL
<i>uniform</i>	UNIFORM_BUFFER	HOST_VISIBLE HOST_COHERENT
<i>indirect_draw</i>	TRANSFER_DST INDIRECT_BUFFER	HOST_VISIBLE HOST_COHERENT

Tablica 3.2: Zależność flag obiektów Vulkan od typu bufora (opracowanie własne)

Obiekt utrzymuje listę elementów bufora *buffer_element*. Zawiera ona bloki pamięci CPU, które są kopiowane do GPU metodą *send_to_device()*. Sposób transferu pamięci z CPU do GPU zależy od użytej flagi właściwości pamięci. Dla pamięci *HOST_VISIBLE* używane jest mapowanie pamięci - funkcja *vkMapMemory()* zwraca wskaźnik CPU do regionu pamięci GPU, do którego bezpośrednio kopiwana jest pamięć używając funkcji *memcpy()*. Dla reszty rodzajów pamięci niemogących być bezpośrednio zapisywanych przez CPU (w tym pamięć *DEVICE_LOCAL*) tworzony jest bufor tymczasowy *VkBuffer*, do którego pamięć CPU jest kopiwana używając polecenia one-shot *one_shot_copy_buffer_to_buffer()*.

Obraz image

Obiekt *image* reprezentuje pojedynczy obraz Vulkan. Wspierane są następujące typy obrazu (*image_type*):

- *material_base_color*: tekstura koloru podstawowy materiału,
- *material_parameters*: tekstura parametrów (metaliczności-chropowatości) materiału,
- *material_normal_map*: mapa normalnych materiału,
- *cubemap*: tekstura sześcienna,
- *font_bitmap*: tekstura czcionki bitmapowej,
- *offscreen_f16*: pozaekranowa tekstura o formacie *R16_SFLOAT*,
- *offscreen_depth_buffer*: pozaekranowy bufor głębi,
- *offscreen_r8*: pozaekranowa tekstura o formacie *R8_UNORM*.

Wszystkie rodzaje tekstur mogą być próbkiowane w shaderach poprzez deskryptory. Tekstury pozaekranowe mogą być też używane jako cele renderowania, tj. dołączenia koloru i głębi/szablonu potoku graficznego.

Obraz utrzymuje trzy obiekty Vulkan: obraz *VkImage*, dowiązaną do niego alokację pamięci *VkDeviceMemory* oraz widok na niego *VkImageView*. Ich informacje tworzenia zależą od wejścia metody *init()*: wysokości, szerokości, liczby kanałów oraz typu obrazu.

Typ obrazu przekłada się głównie na flagi użycia obrazu *VkImageUsageFlags*. Tekstury pozaekranowe używają flag *SAMPLED* i *COLOR_ATTACHMENT*. Reszta obrazów używa flag *SAMPLED*, *TRANSFER_SRC* i *TRANSFER_DST*.

Obrazy używają wyłącznie kafelkowania optymalnego, które jest wydajniejsze od kafelkowania liniowego i znacznie szerzej wspierane [20].

Obrazy posiadają jedną warstwę i jeden poziom mipmap z wyjątkiem tekstur sześciennych (sześć warstw) oraz tekstur koloru podstawowy materiału (maksymalna możliwa liczba poziomów mipmap).

Format obrazu zależy od typu obrazu i liczby kanałów i jest ustalany funkcją *find_image_format()*. Generuje ona listę formatów, które spełniają wymagania specyfikowane przez typ obrazu i posiadają liczbę komponentów równą lub większą liczbie kanałów używanej przez obraz. Ostateczny format to ten z najmniejszą liczbą komponentów wciąż wspierany przez sterownik, co pozwala na zaoszczędzenie pamięci GPU.

Pamięć obrazu jest zawsze w pamięcią *DEVICE_LOCAL*, ponieważ transport z GPU do CPU nigdy nie zachodzi i ten typ pamięci powinien skutkować szybszym dostępem po stronie GPU.

Typ stworzonego widoku *VkImageViewType* to *2D* pozwalający shaderom GLSL na próbkiwanie przy użyciu zmiennych shaderów o typie *sampler2D*. Wyjątkiem są tekstury sześcienne, które wymagają widoku *CUBE* pozwalającego na próbkiwanie zmiennymi *samplerCube*.

Kopiowanie danych obrazu z CPU do GPU to ważna operacja pozwalająca na wstępne wypełnienie obrazów danymi załadowanymi z bazy zasobów. Nie jest ona wymagana dla obrazów niebędących teksturami pozaekranowymi. Jest ona wykonywana metodą `send_to_device()`, która podobnie do obiektów `buffer` używa poleceń one-shot do wypełnienia zmapowanego bufora tymczasowego danymi obrazu i skopiowania go poleceniem one-shot `one_shot_copy_buffer_to_image()` do pamięci GPU. Dodatkowo bufor poleceń one-shot jest używany do wygenerowania wymaganych poziomów mipmap.

Jednolity bufor stałych `unified_constant_buffer`

Obiekt `unified_constant_buffer` reprezentuje jednolity bufor stałych - jeden duży bufor uniform zawierający wszystkie informacje używane przez shadery.

Obiekt zawiera struktury `*_uniform_buffer_data` z wygenerowanego nagłówka `descriptors` pozwalające aplikacji na aktualizację wewnętrznej struktury bufora uniform z poziomu języka C z poszanowaniem zasad układu pamięci scalar.

Dodatkowo obiekt utrzymuje dwie kopie danych bufora i pozwala na aktualizację tylko tej kopii używanej przez obecną klatkę. Podwaja to rozmiar zaalokowanej pamięci GPU, ale pozwala na renderowanie klatek w locie.

Metoda `update()` używa wejściowego wskaźnika funkcji do aktualizacji struktur `*_uniform_buffer_data`, które są następnie kopiowane do pamięci GPU bufora uniform metodą `send_to_device()`.

Obecnie jednolity bufor stałych ma następującą strukturę:

- dane instancji `instances`: Tablica przeznaczona do indeksowania zmiennej shadera `gl_InstanceIndex` pozwalająca na otrzymanie danych renderowanego prymitywu:
 - `mat4 modelMat`: macierz modelu,
 - `uint materialId`: identyfikator materiału.

Rozmiar tablicy jest sterowany konfiguracją globalną (`MaxPrimitiveElementCount`).

- dane globalne `global`: Dane współdzielone przez wszystkie polecenia rysowania:
 - macierze widoku i rzutowania,
 - tablica danych materiałów,
 - tablice danych światel bezpośrednich,
 - dane skybox: identyfikator tekstury skybox,
 - czcionka i tekst debugowania,
 - dane używane do renderowania tekstu debugowania,
 - identyfikatory tekstur pozaekranowych.

Konsolidacja buforów uniform jest techniką renderowanie bez dowiązań. Większość przebiegów renderowania używa tylko części danych jednolitego bufora stałych, ale jest on zawsze w całości dowiązany przez deskryptory `descriptors` na początku klatki i tym samym dostępny w całym potoku graficznym.

Jednolity bufer geometrii `unified_geometry_buffer`

Obiekt `unified_geometry_buffer` reprezentuje jednolity bufor geometrii - bufor wierzchołków i bufor indeksów zawierający całą geometrię używaną przez polecenia rysowania.

Metoda `record_bind_command` dowiązuje do bufora polecień buforów wierzchołków i indeksów poleceniami `vkCmdBindVertexBuffers()` i `vkCmdBindIndexBuffers()`.

Metody `update()` i `send_to_device()` są używane do aktualizacji i transferu do GPU zawartości buforów używając obiektu `vertex_stream`.

Strumień wierzchołków vertex_stream

Obiekt *vertex_stream* reprezentuje strumień wierzchołków zawierający całą geometrię sceny.

Obiekt składa się z dwóch tablic: wierzchołków (tablica *vertex*) i ich indeksy (tablica *uint32_t*).

Metoda *add_geometry()* pozwala na dodanie do strumienia wierzchołka tablic zawierających poszczególne atrybuty i indeksy. Metoda jest używana przez pamięć podręczną renderowania *render_cache* do zebrania geometrii wszystkich renderowanych prymitywów w pojedynczy blok pamięci CPU przekazywany jednolitemu buforowi geometrii.

Silnik używa przekładanych atrybutów (ang. interleaved attributes) - wszystkie atrybuty wierzchołka są przechowywane w ramach jednej struktury *vertex* przedstawionej na listingu 3.22.

```
typedef struct vertex {
    vec3 position;
    vec3 normal;
    vec3 color;
    vec2 texCoord;
    vec4 tangent;
} vertex;
```

Listing 3.22: Struktura wierzchołka *vertex*

Metody *get_vertex_buffer_binding_count()*, *get_vertex_buffer_binding_description()* i *get_vertex_buffer_attribute_descriptions()* zwracają struktury Vulkan używane podczas tworzenia potoku graficznego kompatybilnego ze strumieniem wierzchołków.

Tekstury textures

Obiekt *textures* zarządza wszystkimi teksturami, które są tylko do odczytu i nie są pozaekranowe, i używającymi ich materiałami.

Tekstura składa się z następujących elementów:

- zasób tekstuury,
- obraz,
- próbnik,
- identyfikator tekstuury.

Tekstury są przechowywane w tablicy mieszającej, której kluczem jest jej zasób, co pozwala na uniknięcie duplikacji pamięci.

Metoda *add_texture()* tworzy nową teksturę na podstawie wejściowego zasobu tekstuury. Obraz i próbnik są tworzone na podstawie danych zawartych w zasobie tekstuury. Identyfikator tekstuury to 32-bitowa liczba całkowita. Pierwszy identyfikator to zero, każdy następny jest uzyskiwany poprzez inkrementację. Maksymalna liczba stworzonych tekstuur jest równa maksymalnej liczbie deskryptorów próbkowanych obrazów.

Metoda *send_to_device()* wysyła obrazy tekstuur do GPU.

Dostęp do tekstuur w shaderach odbywa się przy użyciu tablicy deskryptorów próbkowanych obrazów znajdującej się w ostatnim dowiązaniu deskryptorów *descriptors*. Identyfikator tekstuury jest indeksem w tej tablicy używanym przez shadery i podczas aktualizacji deskryptorów.

Funkcja *glsl_add_textures()* dodaje kwalifikatory układu do kodu GLSL shadera pozwalające na dostęp do tablicy tekstuur z dowiązania o numerze *x*, których forma została przedstawiona na listingu 3.23.

```

layout(set = 0, binding = x) uniform sampler2D textures2D[];
layout(set = 0, binding = x) uniform samplerCube texturesCube[];

```

Listing 3.23: Kwalifikatory układu dla tekstur *textures*

Można zauważyć, że definiowane są dwie zmienne shadera posiadające identyczne numery zbioru i dołączania deskryptorów, ale różne typy zmiennych shadera. Ta sytuacja jest dozwolona przez specyfikację Vulkan z zastrzeżeniem, że shader może używać jedynie tych zmiennych shadera, których typ odpowiada rodzajowi dołączanego deskryptora. Przykładowo, jeśli indeks *i* w tablicy deskryptorów tekstur opisuje teksturę 2D, to dostęp do niej w shaderze musi się odbywać przy użyciu wyrażenia *textures2D[i]* - użycie wyrażenia *texturesCube[i]* jest niezdefiniowanym zachowaniem. Technika ta eliminuje potrzebę tworzenia i zarządzania osobnymi dołączaniami deskryptorów dla różnych rodzajów tekstur i pozwala na unifikację próbkowania. Dostęp do tekstury wymaga jedynie wiedzy o jego identyfikatorze *i* i rodzaju, co zostało przedstawione na listingu 3.24.

```

vec4 tex2DSample = texture(textures2D[i], vec2(0));
vec4 texCubeSample = texture(texturesCube[], vec3(0));

```

Listing 3.24: Przykład próbkowania tekstur

Materiał składa się z następujących elementów:

- zasób materiału,
- tekstury materiału:
 - tekstura koloru podstawowego,
 - tekstura metaliczności-chropowatości,
 - mapa normalnych,
- identyfikator materiału.

Metoda *add_material()* tworzy materiał w sposób analogiczny do metody *add_texture()*.

Dostęp do materiałów w shaderach odbywa się poprzez użycie identyfikatora materiału do indeksowania tablicy *materials* w danych globalnych ujednoliconego bufora stałych. Listing 3.25 przedstawia przykładowy sposób dostępu do materiału podczas renderowania przykładowej sceny.

```

uint globalIdx = getGlobalIdx();
uint instanceId = getInstanceId();
uint materialId = getMaterialId(instanceId);
vec4 baseColorFactor = global[globalIdx].materials[materialId].
    ↪ baseColorFactor;
uint baseColorTextureId = global[globalIdx].materials[materialId].
    ↪ baseColorTextureId;
vec4 baseColorSample = texture(textures2D[baseColorTextureId], inTexCoord
    ↪ );

```

Listing 3.25: Przykład dostępu do materiału podczas renderowania przykładowej sceny

Deskryptory *descriptors*

Obiekt *descriptors* reprezentuje deskryptory udostępniające przebiegom renderowania tekstury i jednolity bufor stałych. Obiekt zarządza też stałymi push.

Obiekt zawiera wskaźniki do obiektów *textures* i *unified_constant_buffer* używanych do tworzenia i aktualizacji obiektów Vulkan pozwalających na użycie deskryptorów:

- pula deskryptorów *VkDescriptorPool*,
- układ zbioru deskryptorów *VkDescriptorSetLayout*,
- zbiór deskryptorów *VkDescriptorSet*,
- układ potoku *VkPipelineLayout*.

Zgodnie z duchem renderowania bez dowiązań tworzony jest jeden globalny zbiór deskryptorów mający trzy dowiązania. Dwa pierwsze dowiązania zawierają pojedyncze deskryptory buforów uniform opisujące fragmenty jednolitego bufora stałych zawierające dane instancji i dane globalne. Ostatnie dowiązanie zawiera nieograniczoną tablicę tekstur.

Metoda *send_to_device()* aktualizuje każde dowiązanie zbioru deskryptorów wywołaniami funkcji *vkUpdateDescriptorSets()*.

Metoda *record_bind_commands()* dowiązuje zbiór deskryptorów polecienniem *vkCmdBindDescriptorSets()* oraz nagrywa wejściowe stałe push polecienniem *vkCmdPushConstants()*.

Shader

Obiekt *shader* reprezentuje pojedynczy shader i jest odpowiedzialny za komplikację ich do formy używalnej przez Vulkan.

Obiekt składa się z następujących elementów:

- typ shadera,
- kod źródłowy GLSL,
- kod bajtowy SPIR-V,
- moduł shadera *vkShaderModule*,
- obiekt *shader_reflect*.

Typ shadera zależy od tego, dla którego etapu potoku graficznego jest on przeznaczony. Wspierane są dwa typy: wierzchołków i fragmentów.

Kod źródłowy GLSL musi być znany podczas tworzenia - jest on uzyskiwany poprzez użycie obiektu *shader_generator* z modułu renderowania.

Kod bajtowy SPIR-V jest uzyskiwany poprzez komplikację kodu źródłowego GLSL biblioteką *shaderc*, której użycie ilustruje listing 3.26.

```
shaderc_compiler_t compiler = shaderc_compiler_initialize();

shaderc_compile_options_t options = shaderc_compile_options_initialize();
shaderc_compile_options_set_target_env(options, shaderc_target_env_vulkan, 0);

const char *glslCode = ...;
size_t glslLen = strlen(glslCode);
shaderc_shader_kind shaderType = ...;
const char *inputFileName = "shader";
const char *entryPointName = "main";
shaderc_compilation_result_t result = shaderc_compile_into_spv(
compiler, glslCode, glslLen,
shaderType, inputFileName, entryPointName, NULL);
shaderc_compile_options_release(options);
```

```

if (shaderc_result_get_num_errors(result)) {
    const char *errorMsg = shaderc_result_get_error_message(result);
    panic("compilation error: %s\n", errorMsg);
}

size_t spvSize = shaderc_result_get_length(result);
uint32_t *spvCode = (uint32_t *)malloc(spvSize);
core_memcpy(spvCode, (uint32_t *)shaderc_result_get_bytes(result), spvSize);

shaderc_result_release(result);
shaderc_compiler_release(compiler)

```

Listing 3.26: Kompilacja kodu źródłowego GLSL biblioteką *shaderc*

Moduł shadera jest uzyskiwany poprzez komplikację kodu bajtowego SPIR-V funkcją `vkCreateShaderModule()`, który jest też używany do uzyskania obiektu *shader_reflect*.

Obiekt *shader_reflect* reprezentuje mechanizm refleksji shadera pozwalający na badanie jego struktury. Operuje on na kodzie bajtowym SPIR-V i jest on używany podczas testów oraz do logowania informacji debugujących.

Synchronizacja sync

Obiekt *sync* zawiera obiekty Vulkan które są używane do renderowania klatek w locie, ale nie mogą być pomiędzy nimi współdzielone i muszą być przetrzymywane w tablicach zawierających *FRAMES_IN_FLIGHT* kopii. Są to:

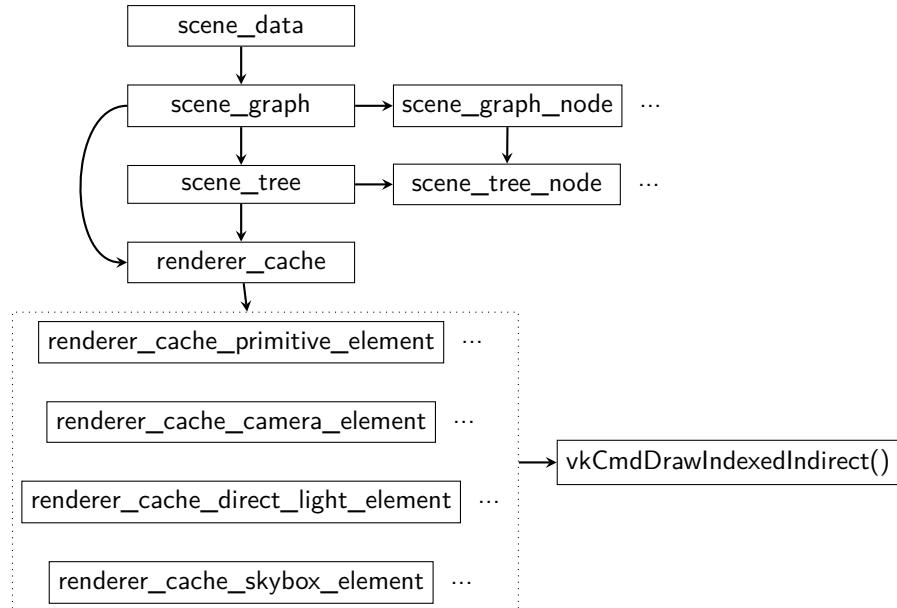
- semafory *imageAvailableSemaphores* sygnalizujące, że prezentowalny obraz zwrócony przez funkcję `vkAcquireNextImageKHR()` jest gotowy do renderowania;
- semafory *renderFinishedSemaphores* sygnalizujące, że wykonywany bufor zakończył renderowanie do prezentowalnego obrazu, który może zostać zaprezentowany funkcją `vkQueuePresentKHR()`;
- ogrodzenia *inFlightFences* sygnalizowane po zakończeniu renderowania klatki i używane do ograniczenia ich liczby do *FRAMES_IN_FLIGHT* jednocześnie renderowanych klatek;
- pule poleceń i bufory poleceń, które są równolegle nagrywane i wysyłane do kolejki w pętli głównej programu.

Pole *currentFrameInFlight* jest indeksem używanym do śledzenia która kopia powinna być używana w obecnej klatce. Metoda *advance_to_next_frame()* wywoływana na początku klatki inkrementuje *currentFrameInFlight* i dzieli modulo *FRAMES_IN_FLIGHT*.

3.3.6. Scena

Moduł sceny jest odpowiedzialny za konwersję sceny z danych sceny *scene_data* do pamięci podręcznej renderera *renderer_cache* - wysokopoziomowa forma używanej przez bazę zasobów jest zamieniana na niskopoziomową formę łatwiej używalną przez renderer do emitowania poleceń rysowania pośredniego.

Obiekty biorące udział w procesie konwersji sceny są przedstawione na diagramie 3.5.



Rysunek 3.5: Obiekty biorące udział w procesie konwersji sceny (opracowanie własne)

Dane sceny `scene_data`

Obiekt `scene_data` reprezentuje dane sceny wczytane z bazy zasobów. Jest one używany przez moduł sceny do konstrukcji grafu sceny `scene_graph`.

Obiekt utrzymuje listy dwukierunkowe zawierającą wszystkie utworzone obiekty zasobów, w tym ich domyślne warianty. Przykładowo domyślny obraz `asset_image` to obraz 2D o rozmiarze 1x1 mający 4 8-bitowe komponenty o wartości 255.

Wśród wszystkich obiektów zasobów składających się na dane sceny dodatkowo wyróżnia się:

- węzły główne: używane jako punkty początkowe podczas tworzenia grafu sceny,
- używany skybox: może być zmieniony w konfiguracji zasobów,
- aktywna czcionka: sterowana konfiguracją globalną,
- domyślna kamera: używana w przypadku braku węzła z przypisaną kamerą.

Metody `serialize()` i `deserialize()` podobnie jak analogiczne metody obiektów zasobów pozwalają na zapis i odczyt danych sceny do bazy zasobów.

Metoda `create_with_gltf_file()` jest wywoływana wyłącznie przez potok zasobów. Jej wejściem jest nazwa tworzonej sceny i ścieżka do katalogu zawierającego zasób 3D w formacie `gltf` wraz z konfiguracją zasobów. Oba zasoby wejściowe są parsowane przy użyciu biblioteki `cgltf` i obiektu `config`. Wynik parsowania jest używany do stworzenia i wypełnienia danych sceny. Ta metoda wraz z metodą `serialize()` stanowi główną częścią potoku zasobów - obiekt stworzony na podstawie zasobów wejściowych jest serializowany do zasobu wyjściowego (bazy zasobów).

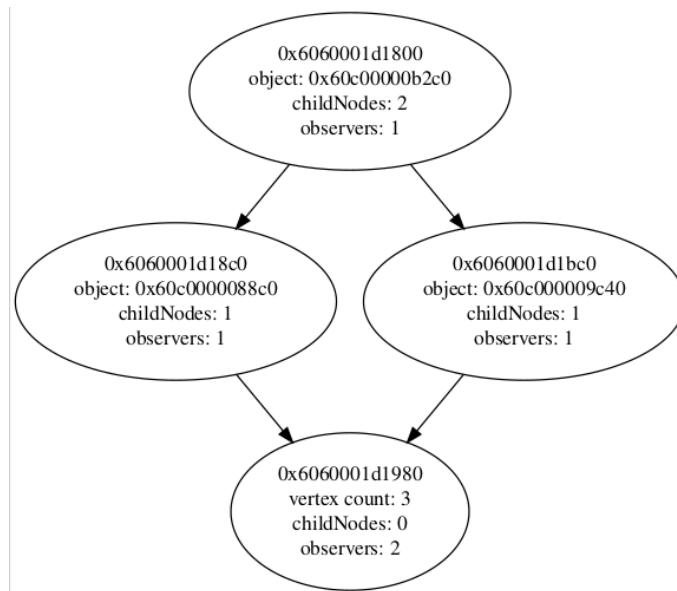
Metoda `create_with_asset_db()` jest wywoływana w czasie wykonywania i wczytuje dane sceny o żądanej nazwie z bazy zasobów.

Graf sceny `scene_graph` i drzewo sceny `scene_tree`

Obiekty `scene_graph` i `scene_tree` reprezentują kolejno graf i drzewo sceny. Są one tworzone na podstawie danych sceny używając metodę `create_with_scene_data()`.

Implementacja została zainspirowana techniką modelowania hierarchicznego opracowaną w firmie Nvidia [50]. Graf sceny jest konwertowany na formę pośrednią zwaną drzewem sceny w sposób zapewniający unikalną ścieżkę z korzenia drzewa do jego liści. Uprasza to klasyczny proces konwersji sceny algorytmem 2.10 - podczas przechodzenia drzewa sceny każdy węzeł może być napotkany tylko raz. Właściwość ta pozwala na wprowadzenie do węzła stanu nagromadzonego (struktura `scene_tree_node_accumulated`) zawierającego aktualny wynik działania algorytmu. Konwersja sceny sprowadza się wtedy do całkowitej propagacji stanu nagromadzonego od korzenia do liści, w których jest on całkowicie wypełniony i tym samym zawiera wszystkie informacje wymagane do wyemitowania polecenia rysowania.

Diagramy 3.6 i 3.7 przedstawiają wygenerowane metodami `debug_print()` graf i drzewo sceny składającej się z dwóch trójkątów używających tej samej siatki, ale różniących się pozycją. Graf sceny składa

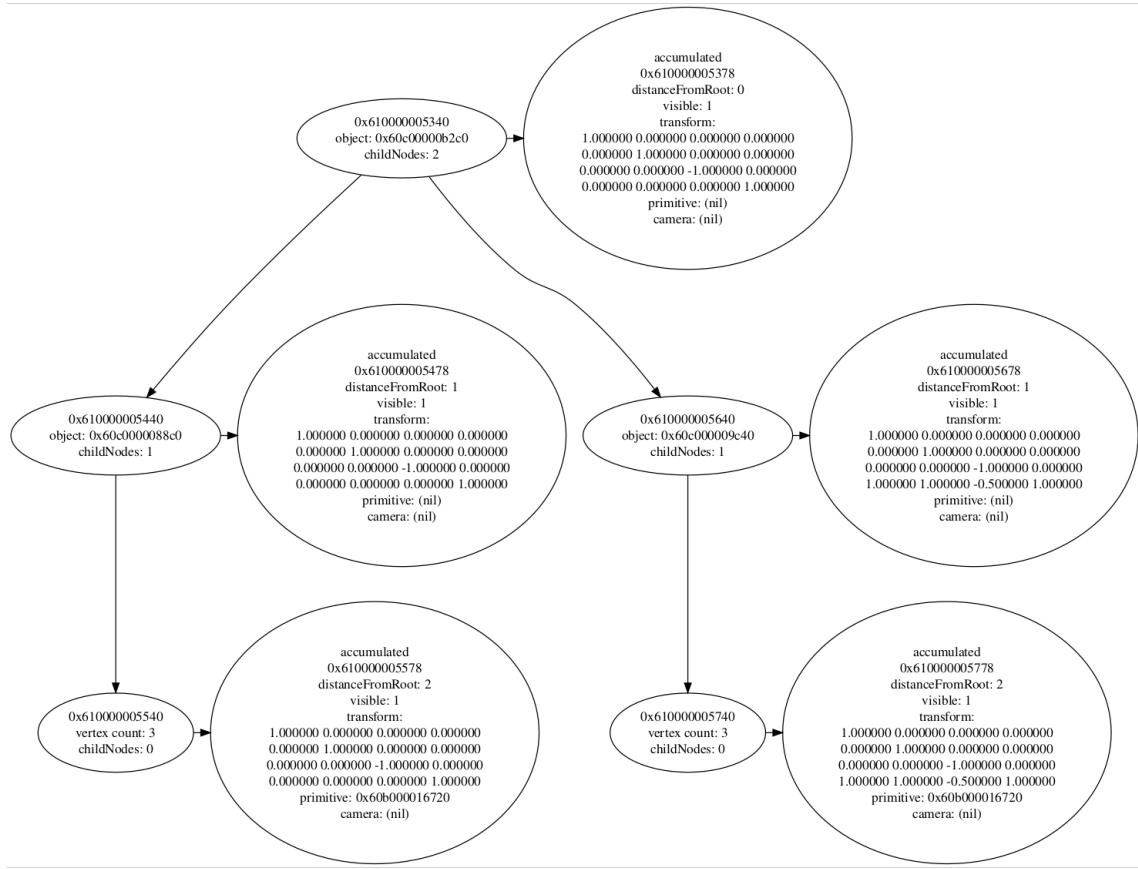


Rysunek 3.6: Przykładowy graf sceny zawierający dwa trójkąty (opracowanie własne)

się z czterech węzłów `scene_graph_node`: korzenia oraz dwóch węzłów utworzonych na podstawie zasobów obiektów wskazujących na ten sam węzeł z zasobem prymitywu. Węzły grafu sceny obserwują utworzone na ich podstawie podczas procesu konwersji węzły drzewa sceny, co pozwala na aktualizację i propagację ich stanu nagromadzonego po zmianach w węźle grafu sceny bez konieczności całkowitego odtworzenia drzewa sceny. Drzewo sceny składa się z pięciu węzłów `scene_tree_node` posiadających stan nagromadzony:

- korzeń: stan posiada lokalnym przekształceniem będącym macierz obrotu zmieniającą prawo-skrętną przestrzeń modelu glTF na lewoskrętną przestrzeń używaną przez silnik;
- dwa węzły utworzone na podstawie węzłów z zasobami obiektów: stan posiada lokalne przekształcenie zmieniające pozycję trójkątów;
- dwa liście utworzone na podstawie węzłów z zasobami prymitywu: stan posiada wskaźnik do zasobu prymitywu.

Po zakończeniu propagacji stanu nagromadzonego liście posiadają informacje o prymitywie i jego globalnym przekształceniu. Liście drzewa sceny są dodawane do pamięci podrzcznej renderera.



Rysunek 3.7: Przykładowe drzewo sceny zawierające dwa trójkąty (opracowanie własne)

3.3.7. Renderer

Główny moduł silnika odpowiedzialny za właściwy proces renderowania. Moduł renderera jest używany do zaimplementowania aplikacji renderującej przykładową scenę.

Renderer

Obiekt *renderer* to główny obiekt modułu odpowiedzialny za integrację wszystkich obiektów silnika w celu rozpoczęcia pętli głównej programu i wyrenderowanie sceny. Obiekt tworzy i używa następujące obiekty:

- konfiguracja globalna *data_config*,
- baza zasobów *asset_db*,
- dane sceny *scene_data*,
- graf sceny *scene_graph*,
- pamięć podrzczna renderowania *renderer_cache*,
- urządzenie *device*,
- łańcuch wymiany *swap_chain*,
- stan renderera *render_state*,
- graf renderowania *render_graph*.

Po stworzeniu obiektu aplikacja musi skonstruować i skompilować początkowo pusty graf renderowania. Następnie może ona rozpoczęć pętlę główną programu metodą `run_main_loop()`, która renderuje pojedynczą klatkę w locie używając następujących kroków:

- zaktualizuj stan renderera wejściową funkcją zwrotną `updateFunc`;
- czekaj aż ogrodzenie `inFlightFences` obecnej klatki w locie zasygnalizuje zakończenie wykonywania poprzednio nagranego bufora poleceń i gotowość do ponownego nagrywania;
- pobierz prezentowalny obraz funkcją `vkAcquireNextImageKHR()`;
- odtwórz łańcuch wymiany jeśli jest nieaktualny i pomiń resztę iteracji pętli;
- zaktualizuj i skopiuj do GPU stan renderera i graf renderowania używając ich metod `update()` i `send_to_device()`;
- zresetuj bufor poleceń i rozpocznij nagrywanie;
- dowiąż stan renderera (jednolite bufory, zbiór deskryptorów i stałe push);
- nagraj przebiegi renderowania używając grafu renderowania;
- zakończ nagrywanie;
- wyślij bufor poleceń do kolejki graficznej funkcją `vkQueueSubmit()` zapewniając synchronizację poprzez oczekiwanie na semafor `imageAvailableSemaphores` oraz sygnalizację semafor `renderFinishedSemaphores` i zresetowanego ogrodzenia `inFlightFences`;
- prezentuj wynik renderowania funkcją `vkQueuePresentKHR()` zapewniając synchronizację poprzez oczekiwanie na semafor `renderFinishedSemaphores`
- odtwórz łańcuch wymiany jeśli jest nieaktualny;
- przejdź do kolejnej klatki w locie (metoda `advance_to_next_frame()` obiektu `sync`).

W dalszych sekcjach przybliżono obiekty renderera i używającą go przykładową aplikację.

Pamięć podręczna renderera `renderer_cache`

Pamięć podręczna renderera `renderer_cache` jest niskopoziomową reprezentacją sceny pośredniczącą pomiędzy modułami sceny i renderera. Obiekt jest podstawowym źródłem informacji o scenie używanym przez resztę obiektów modułu.

Inicjalizację pamięci podręcznej renderera można podzielić na dwie fazy - graf sceny wypełniania obiekt informacjami o scenie, który jest następnie używany do wypełnienia obiektów używanych przez renderer:

- stan renderera,
- stany przebiegów renderowania,
- grup poleceniami rysowania.

Obiekt jest złożony z list elementów `renderer_cache_*_element`:

- `renderer_cache_primitive_element`: element prymitywu;
- `renderer_cache_camera_element`: element kamery;
- `renderer_direct_light_element`: element światła bezpośrednie;
- `renderer_skybox_element`: element skybox;
- `renderer_font_element`: element czcionki.

Elementy są dodawane przez graf sceny używając metod `add_*_element()`.

Obiekt jest używany do aktualizacji stanu renderera:

- Metoda *update_geometry()* dodaje geometrię prymitywów do jednolitego bufora geometrii.
- Metoda *update_textures()* dodaje tekstury używane przez elementy do teksturow *textures*.

Metoda *add_new_primitive_elements_to_batches()* dodaje polecenia rysowania prymitywów do grup poleceniami rysowania.

Aktualizacja stanu renderera i grup poleceń rysowania przy użyciu powyższych metod jest odzwierciedlana aktualizacją stanu elementu. Przykładowo wywołanie metody *update_geometry()* aktualizuje strukturę *vertex_stream_element* elementów prymitywów offsetami buforów strumienia wierzchołków. Podobnie metoda *update_textures()* dodaje tekstury elementu skybox, których identyfikatory są zapamiętywane w elementach.

Zachowanie to pozwala stanom przebiegów renderowania na aktualizację jednolitego bufora stałych informacjami o scenie. Przykładowo element skybox jest używany do aktualizacji zmiennej *skyboxCubemapTextureId* używanej przez shadery renderujące skybox.

Grupy poleceń rysowania batches

Grupa poleceń rysowania *batches* reprezentuje jedno optymalizowane polecenia rysowania utworzone na podstawie elementów prymitywów dodanych metodą *add_primitive_element()*.

Metoda *update()* sortuje i konsoliduje dodane elementy prymitywów tworząc tablicę pojedynczych grup poleceń rysowania *batch*. Pojedyncza grupa poleceń *batch* zawiera wszystkie informacje wymagane do nagrania pojedynczego polecenia rysowania *vkCmdDrawIndexed()* rysującego *n* kopii mających tą samą geometrię, ale różniących się materiałem i macierzą modelu.

Metoda *record_draw_command()* nagrywa do wejściowego bufora poleceń pojedyncze polecenie wielokrotnego rysowania pośredniego *vkCmdDrawIndexedIndirect*, której bufora parametrów jest wypełniany wcześniej stworzoną tablicą obiektów *batch*. Bufor parametrów jest przekazywany do metody strukturą *batches_buffer*, która jest zarządzana przez stan przebiegów renderowania.

Stan renderera *renderer_state* i stan przebiegów renderowania *render_pass_state*

Stan renderera *renderer_state* zawiera wszystkie obiekty renderera, które są niezależne od stanu przebiegów renderowania. Są to:

- strumień wierzchołków *vertex_stream*,
- jednolity bufor geometrii *unified_geometry_buffer* (dane globalne współdzielone przez klatki w locie),
- tekstury *textures*,
- jednolity bufor stałych *unified_constant_buffer*,
- deskryptory *descriptors*,
- synchronizacja *sync*.

Podział ten jest spowodowany obsługą odtwarzania łańcucha wymiany - większość stanu przebiegów renderowania, w przeciwieństwie do stanu renderera, zależy od wielkości łańcucha wymiany i dlatego musi być niszczony i ponownie tworzony przez renderer wraz z łańcuchem wymiany.

Metody *update()* i *send_to_device()* wywołuje analogiczne metody obiektów renderera.

Stan przebiegów renderowania *render_pass_state* jest obiektem analogicznym do stanu renderera i odpowiada za aktualizację jednolitego bufora stałych (dane instancji oraz dane globalne zależne od obecnej klatki w locie) oraz zarządzanie buforami parametrów polecień rysowania pośredniego.

Graf renderowania render_graph

Graf renderowania *render_graph* opisuje proces renderowania klatki w formie przebiegów renderowania i używanych przez nie obrazów. Obiekt jest też odpowiedzialny za zarządzanie wszystkimi przebiegami renderowania *render_pass* i współdzielonym przez nie stanem przebiegów renderowania *render_pass_state*.

Metoda *add_image_resource()* jest używana przez aplikację do dodania wszystkich obrazów, które będą używane przez przebiegi renderowania. Wyjątkiem są prezentowalne obrazy, które są dodawane automatycznie.

Dodane obrazy są przechowywane w liście struktur *render_graph_resource* posiadającej:

- nazwę zasobu;
- typ obrazu;
- identyfikator tekstury pozaekranowej: obsługa klatek w locie wymaga wewnętrznej duplikacji używanych tekstury pozaekranowych, dlatego indeks obrazu należącego do obecnej klatki jest przekazywany w jednolitym buforze stałych do shaderów chcących je próbować;
- harmonogram użycia *image_render_pass_usage_timeline*

Harmonogram użycia obrazu *image_render_pass_usage_timeline* opisuje sposób użycia (odczyt, zapis, odczyt i zapis), format i wartość czyszczenia obrazu na wskroś przebiegów renderowania. Jest on wypełniony podczas komplikacji i używany do nagrania wymaganych przejść układów obrazów pomiędzy przebiegami renderowania.

Listing 3.27 przedstawia kod przykładowej aplikacji dodający obrazy (G-bufor, bufor głębi i bufor okluzji otoczenia) do grafu renderowania.

```
render_graph_add_image_resource(renderer->renderGraph, "depthBuffer",
    image_type_offscreen_depth_buffer);
render_graph_add_image_resource(renderer->renderGraph, "gBuffer0",
    ↪ image_type_offscreen_f16);
render_graph_add_image_resource(renderer->renderGraph, "gBuffer1",
    ↪ image_type_offscreen_f16);
render_graph_add_image_resource(renderer->renderGraph, "gBuffer2",
    ↪ image_type_offscreen_f16);
render_graph_add_image_resource(renderer->renderGraph, "aoBuffer",
    ↪ image_type_offscreen_r8);
```

Listing 3.27: Dodawanie obrazów do grafu renderowania przykładowej aplikacji

Metoda *add_render_pass()* jest używana przez aplikację do dodawania następujących po sobie przebiegów renderowania opisywanych strukturą *render_pass_desc*.

Dodane przebiegi renderowania są przechowywane jako węzły grafu renderowania *render_graph_render_pass_element* posiadające:

- przebieg renderowania *render_pass*;
- wskaźniki do zasobów *render_graph_resource*, które będą używane przez przebieg renderowania:
 - prezentowalny obraz: dołączenie koloru);
 - tekstury pozaekranowe: dołączenia koloru lub próbkowany przez shader);
 - bufor głębi: dołączenie głębi/szablonu używane do testów głębi z możliwością aktualizacji lub próbkowany przez shader;

- stan komplikacji: struktura *rendering_info* używana nagrania przejść układów powyższych obrazów i rozpoczęcia przebiegu renderowania poleciem *vkCmdBeginRenderingKHR()*.

Wszystkie powyższe obiekty są tworzone na podstawie opisu przebiegu renderowania *render_pass_desc*.

Listing 3.28 przedstawia kod przykładowej aplikacji dodający przebieg geometrii, który:

- używa shadera wierzchołków *deferred_geometry_vertex.glsl*;
- używa shadera fragmentów *deferred_geometry_fragment.glsl*;
- nie renderuje do prezentowalnego obrazu;
- używa trzech obrazów G-bufora, które są czyszczone wartością 0,0,0,1 jeśli są używane po raz pierwszy oraz używane jako dowiązania kolorów;
- używa bufora głębi jako dołączenia głębi/szablonu, czyszczonego wartością 0,0, używanego do testów głębi z operatorem porównania "większy lub równy" i włączonym zapisem;
- brak mieszania kolorów;
- nagrywa zoptymalizowane polecenie rysowania wszystkich elementów prymitywów podręcznej pamięci renderera.

```
void render_pass_record_primitive_geometry_draws(render_pass *renderPass,
    ↪ render_pass_frame_state *frameState, VkCommandBuffer commandBuffer) {
    batches_record_draw_command(renderPass->renderPassState->sharedState.
    ↪ rendererCacheBatches,
    commandBuffer, &frameState->rendererCacheBatchesData);
}

render_graph_add_render_pass(renderer->renderGraph, (render_pass_desc){
    .vertexShader = "deferred_geometry_vertex.glsl",
    .fragmentShader = "deferred_geometry_fragment.glsl",
    .useOnscreenColorAttachment = false,
    .offscreenColorAttachmentCount = 3,
    .offscreenColorAttachments =
    {
        {.name = "gBuffer0", .clearValue = {{0.0f, 0.0f, 0.0f, 1.0f}}},
        {.name = "gBuffer1", .clearValue = {{0.0f, 0.0f, 0.0f, 1.0f}}},
        {.name = "gBuffer2", .clearValue = {{0.0f, 0.0f, 0.0f, 1.0f}}},
    },
    .offscreenDepthAttachment =
    {
        .name = "depthBuffer",
        .depthWriteEnable = true,
        .depthTestEnable = true,
        .depthTestOp = VK_COMPARE_OP_GREATER_OR_EQUAL,
        .clearValue = {0.0f, 0.0f},
    },
    .colorBlendingType = color_blending_type_none,
}
```

```
.recordFunc = render_pass_record_primitive_geometry_draws});
```

Listing 3.28: Dodawanie przebiegu geometrii do grafu renderowania przykładowej aplikacji

Metoda *compile()* kompliluje graf renderowania. Dla każdego węzła metoda oblicza harmonogramy użycia dla używanych przez niego obrazów i następnie używa ich do wypełnienia stanu kompilacji celami renderowania z wymaganymi przez nie przejściami układów obrazów.

Metoda *record_commands()* iteruje po węzłach używając ich stanu kompilacji do nagrania przebiegów renderowania.

Metody *update()* i *send_to_device()* aktualizuje i kopiuje do GPU stan przebiegów renderowania.

Przebieg renderowania render_pass

Przebieg renderowania *render_pass* jest tworzony na podstawie opisu przebiegu renderowania *render_pass_desc* i składa się z:

- potoku graficznego *VkPipeline*,
- programu shaderów *render_pass_shader_program*;

Potok graficzny jest tworzony na podstawie opisu *render_pass_desc* oraz stanu renderowania (stan wejścia wierzchołków zależny od *vertex_stream*). Moduły shaderów używane przez potok są tworzone przez program shaderów wywołujący generator shaderów *render_pass_shader_generator*.

Metoda *record()* nagrywa dynamiczny przebieg renderowania z dowiązanym potokiem graficznym i poleceniami rysowania zdefiniowanymi polem *recordFunc* opisu *render_pass_desc*.

Generator shaderów render_pass_shader_generator

Generator shaderów *render_pass_shader_generator* generuje kod źródłowy GLSL pojedynczego shadera.

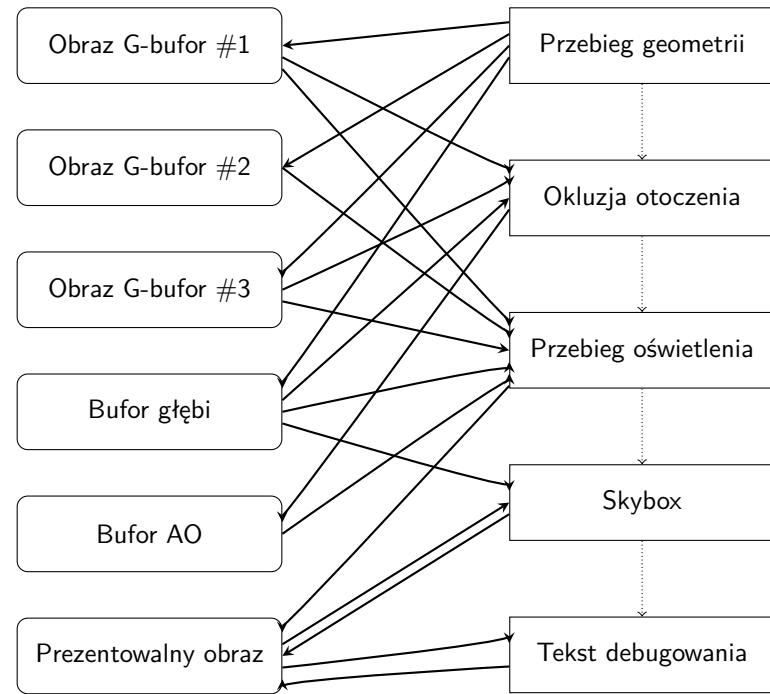
Generacja kodu źródłowego sprowadza się do konkatenacji:

- dyrektyw preprocesora deklarujących używane rozszerzenia,
- stałych preprocesora opisujących:
 - używane atrybuty wierzchołka (*IN_POSITION*, *IN_NORMAL* itp.),
 - typ shadera (*SHADER_VERTEX*, *SHADER_FRAGMENT*),
 - wygenerowane stałe (np. *MAX_DIRECTIONAL_LIGHT_COUNT*),
 - nazwy używanych tekstur pozaekranowych (np. *OFFSCREEN_TEXTURE_gBuffer1*),
- kwalifikatorów układów dla stałych push, buforów uniform i tablicy tekstur,
- zmiennych wejściowych i wyjściowych,
- współdzielonego kodu GLSL wczytanego z pliku tekstowego *common.gsl*,
- funkcji *main()*:
 - zmienne z indeksami tekstur pozaekranowych,
 - kod GLSL wczytany z pliku tekstowego zdefiniowanego w opisie *render_pass_desc*.

Przykładowa scena main

Plik wykonywalny *main* demonstruje działanie silnika poprzez wyrenderowanie sceny zdefiniowanej w konfiguracji globalnej używając grafu renderowania.

Diagram 3.8 przedstawia używany graf renderowania opisujący cieniowane odroczone zintegrowane z okluzję otoczenia [51] i dwoma dodatkowymi efektami post-processingu: skybox i tekst debugujący.



Rysunek 3.8: Graf renderowania przykładowej sceny (opracowanie własne)

Listing 3.29 przedstawia kod funkcji *main()* przykładowej aplikacji uruchamiający renderer.

```
void render_pass_record_draws(render_pass *renderPass, render_pass_frame_state *
    ↪ frameState, VkCommandBuffer commandBuffer) {
    batches_record_draw_command(renderPass->renderPassState->sharedState.
    ↪ rendererCacheBatches, commandBuffer, &frameState->rendererCacheBatchesData
    ↪ );
}

void update_func(renderer *renderer, double fps, double dt) {
    device *vkd = renderer->vkd;
    render_state *renderState = renderer->renderState;
    if (renderState->config->global.controlsEnabled == 1) { ... }
}

int main(int argc, char *argv[]) {
    platform_create(argc, argv);
    renderer *renderer = renderer_create();

    render_graph_add_image_resource(renderer->renderGraph, "gBuffer0",
    ↪ image_type_offscreen_f16);
    ...
    render_graph_add_render_pass(
        renderer->renderGraph,
```

```
(render_pass_desc){  
    ...  
    .recordFunc = render_pass_record_primitive_geometry_draws  
}  
);  
...  
  
renderer_run_main_loop(renderer, update_func);  
  
renderer_destroy(renderer);  
platform_destroy();  
return 0;  
}
```

Listing 3.29: Uruchomienie renderera w funkcji *main()* przykładowej aplikacji

4. WYNIKI I TESTY WYDAJNOŚCIOWE

Ten rozdział zawiera zrzuty ekranu pokazujące wyrenderowane przykładowe sceny oraz testy wydajnościowe mierzące wydajność implementacji cieniowania odroczonego.

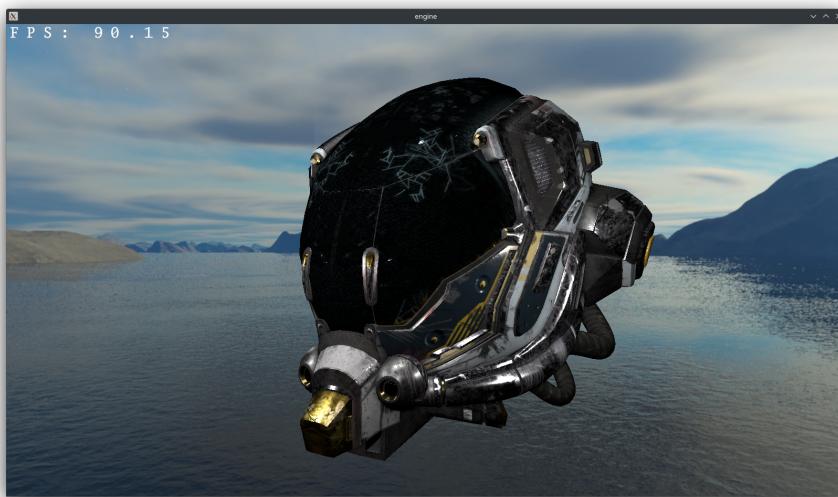
4.1. Przykładowe sceny

Używane sceny zostały przygotowane na podstawie zebranych przez Khronos w repozytorium [18] następujących przykładowych modelów glTF przedstawiających:

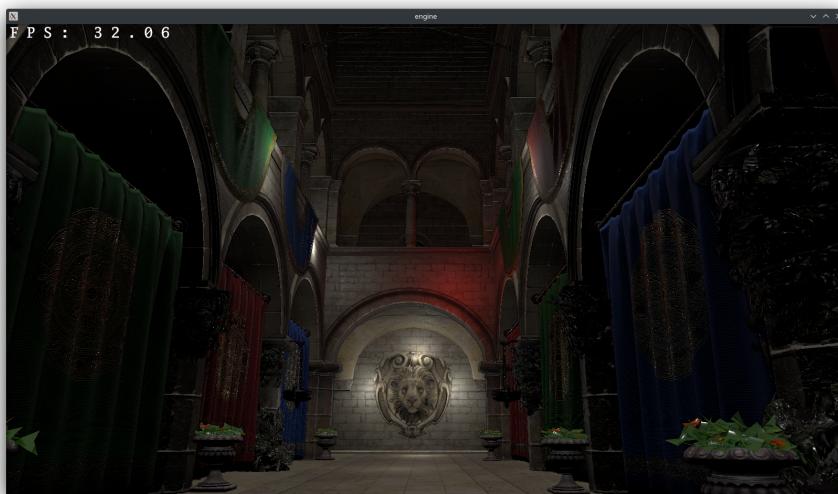
- *MetalRoughSpheres*: różne wartości metalu i chropowatości materiałów używając sfer, patrz rys. 4.1;
- *DamagedHelmet*: uszkodzony w walce hełm sci-fi, patrz rys. 4.2;
- *Sponza*: wnętrze budynku inspirowane pałacem Sponza, często używane do testowania oświetlenia, patrz rys. 4.3.



Rysunek 4.1: Wynik renderowania sceny MetalRoughSpheres (opracowanie własne)



Rysunek 4.2: Wynik renderowania sceny DamagedHelmet (opracowanie własne)



Rysunek 4.3: Wynik renderowania sceny Sponza (opracowanie własne)

4.2. Pomiary wydajnościowe

Zbadano wydajność implementacji cieniowania odroczonego poprzez renderowania sceny Sponza w 10 eksperymentach spowodowanych zmianą następujących zmiennych:

- rozdzielczości okna: 640x480, 1600x900;
- liczby światel punktowych: 1, 30, 75, 10, 100.

Zmierzono liczbę klatek na sekundę podczas 60 sekund działania programu używając narzędzia *MangoHud* [52]. W przeciwieństwie do prostego licznika FPS silnika w lewym górnym rogu okna i warstwy *VK_LAYER_MESA_overlay*, *MangoHud* pozwala na pomiar percentyli FPS, które ilustrują problemy przycinania (ang. *stuttering*) lepiej od minimalnej, średniej i maksymalnej wartości FPS [53]. Przykładowo percentyl 97 równy 45 FPS oznacza, że 97% klatek wykonało się wolniej (mniejsza wartość FPS), a 3% wykonało się szybciej (większa wartość FPS). Tabela 4.1 przedstawia wyniki pomiarów FPS.

Rozdzielcość okna	Liczba światel	percentyl 0.1 FPS	percentyl 1 FPS	percentyl 97 FPS	średnia arytmetyczna FPS
640x480	1	27.5	45.4	162.4	118.2
640x480	10	27.9	34.7	131.4	102.8
640x480	30	26.8	32.2	91.8	83.8
640x480	75	20.4	22.7	53.9	46.1
640x480	100	18.1	20.7	44.4	44.1
1600x900	1	19.1	21.7	49.8	45.3
1600x900	10	16.4	17.7	36.9	33.3
1600x900	30	7.7	12.2	23.8	27.3
1600x900	75	6.4	9.3	13.0	15.5
1600x900	100	5.0	6.1	10.5	14.7

Tablica 4.1: Wyniki pomiarów FPS dla sceny Sponza (opracowanie własne)

Zmierzono udział poszczególnych etapów potoku graficznego podczas wykonywania poleceń rysowania poprzez przechwycenie 10 klatek aplikacji narzędziem RenderDoc i zbadanie liczników wydajności. Po porównaniu wartości liczników wydajności z przechwyconych klatek okazało się, że największe zmierzone zmiany, które nie mogą być wyjaśnione błędem przypadkowym spowodowanym nieprzewidywalnością działania potoku graficznego GPU, zachodzą tylko dla liczników wydajności najdłużej wykonującego się polecenia rysowania w przebiegu oświetlenia G-bufora: W pomiarach zostały uwzględnione poniższe liczniki:

- *GPU Time Elapsed (μ)*: czas jaki upłynął na GPU podczas wykonywania polecenia;
- *L3 Shader Throughput (bytes)*: całkowita liczba bajtów pamięci GPU przesyłanych pomiędzy shaderami a pamięcią podręczną L3;
- *Shader Memory Accesses*: całkowita liczba operacji dostępu do pamięci buforów w shaderach;
- *Sampler Texels*: całkowita liczba tekself widocznych na wejściu wszystkich próbników (z dokładnością 2x2),
- *Samplers Busy (%)*: procent czasu spędzony na próbkowaniu obrazów.

Wartości wszystkich zmierzonych liczników mają stałą wartość z wyjątkiem *GPU Time Elapsed (μ)* i *Samplers Busy (%)* obarczonych błędem przypadkowym. Wyniki pomiarów wydajności prezentuje tabela 4.2.

Rozdzielczość okna	Liczba świateł	GPU Time Elapsed (μ)	L3 Shader Throughput (bytes)	Shader Memory Accesses	Sampler Texels	Samplers Busy (%)
640x480	1	1859.916	19685376	307584	1228800	97.3006
640x480	10	7457.666	68837824	1075591	1228800	88.82873
640x480	30	7299.75	179433088	2803642	1228800	50.89936
640x480	75	37235.50	428882496	6701289	1228800	26.02086
640x480	100	23358.666	566511168	8851737	1228800	20.69141
1600x900	1	3819.166	92281664	1441901	5760000	99.45081
1600x900	10	14263.333	322695168	5042112	5760000	89.91253
1600x900	30	34444.083	841100544	13142196	5760000	57.64096
1600x900	75	82389.333	2010387968	31412312	5760000	26.82165
1600x900	100	109205.75	2655508160	41492315	5760000	21.54303

Tablica 4.2: Wyniki pomiarów liczników wydajności dla sceny Sponza (opracowanie własne)

4.3. Analiza pomiarów

Działanie renderera spowalnia wraz z zwiększającą się liczbą świateł oraz rozdzielczością okna.

Spowolnienie spowodowane zmianą rozdzielczości może być wyjaśnione zmianą wielkości G-bufora i tym samym zwiększeniem liczby wykonywanych shaderów fragmentów.

Zwiększenie liczby świateł zwiększa liczbę obiegów pętli w przebiegu oświetlania pobierających informacje o każdym świetle z bufora uniform, co tłumaczy rosnącą liczbę dostępów do pamięci i malejący udział procentowy czasu spędzonego na próbkowaniu G-bufora.

Wartość FPS zachowuje się na wskroś wszystkich eksperymentów w podobny sposób: jest stabilna z wyjątkiem pojawiających się sporadycznie co kilkanaście lub kilkadziesiąt sekund zacięć mających formę pojedynczych dłużej wykonujących się klatek. Podobne zacięcia zostały zaobserwowane dla innych aplikacji Vulkan uruchamianych na maszynie testowej (aplikacja vkcube z Vulkan SDK i przykłady od Sascha Willems [54]).

5. PODSUMOWANIE

Efektem niniejszej pracy jest silnik graficzny pozwalający na renderowanie używając cieniowania odroczonego i innych technik graficznych.

Zaimplementowano potok zasobów pozwalający na konwersję scen w formacie *g/TF* do bazy SQLite przechowującej wszystkie zasoby używane przez silnik.

Zastosowano bibliotekę graficzną Vulkan do realizacji technik renderowania bez dowiązań.

Zaimplementowano graf renderowania pozwalający na wysokopoziomowy opis użycia potoku graficznego w formie przebiegów renderowania operujących na teksturach pozaekranowych.

Zademonstrowano działanie silnika poprzez zdefiniowanie grafu renderowania odroczonego i użycie go do wyrenderowania przykładowych scen. Przeprowadzono testy wydajnościowe.

Projekt w obecnym kształcie nie został wykorzystany w grze komputerowej i powinien być rozpatrywany w kategoriach prototypu. Rozwój funkcjonalności silnika nie jest zakończony i w czasie realizacji projektu pojawiło się wiele pomysłów na ulepszenia.

Silnik był napisany i przetestowany wyłącznie na systemie Linux, ale wieloplatformowa natura używanych narzędzi i bibliotek powinna pozwolić na dokonanie komplikacji skróśnej.

Renderer odroczony w obecnym kształcie obsługuje tylko nieprzeźroczyste obiekty. Rysowanie obiektów przeźroczystych mogłoby być rozwiązane dodatkowym przebiegiem renderowania wprzód dla posortowanych obiektów z mieszaniem alfa. Bardziej dokładna, lecz wolniejsza, przeźroczystość dla skomplikowanej geometrii może być osiągnięta używając technik przeźroczystości niezależnej od kolejności (ang. *order-independent transparency*, OIT).

Wszystkie zasoby są jednokrotnie kopowane z GPU do CPU przed rozpoczęciem pętli głównej renderowania. Bardziej efektywnym pamięciowo podejściem byłoby przesyłanie strumieniowe zasobów polegające na ładowaniu ich tylko gdy są rzeczywiście używane do renderowania.

Z przesyaniem strumieniowym dobrze współpracują techniki usuwania niewidocznych powierzchni. Najprostszą do implementacji jest odrzucanie na CPU obiektów poza bryłą widzenia pozwalające na niewysyłanie nierenderowanej geometrii do GPU. Znacznie bardziej skomplikowane, ale zgodne z duchem renderowania bez dowiązań, jest usuwanie niewidocznych powierzchni na GPU. Może być ono zrealizowane w Vulkan używając shaderów obliczeń modyfikujących bufore pośrednie używane przez polecenia rysowania.

Użycie shaderów obliczeń pozwoliłoby też na zastosowanie techniki programowanego pobierania wierzchołków (ang. *programmable vertex pulling*) [22], w której etap asemblera wejścia potoku graficznego jest całkowicie pomijany i zastępowany odczytywaniem i dekodowaniem danych geometrii z zasobów shadera. Podejście to pozwoliłoby na eliminację ujednoliconego bufora geometrii oraz zastąpienie sztywnego definiowania formatów wierzchołków w potoku graficznym elastyczniejszym dekodującym kodem w shaderach.

Model oświetlenia może być rozszerzony o powierzchnie emitujące światło poprzez dodanie tekstur emisyjnych. Podobnie tekstury skybox mogą być użyte do mapowania środowiskowego symulującego odbicia światła dla połyskliwych powierzchni.

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/> (dostęp 20 paź. 2022).
- [3] *Advances in Real-Time Rendering in 3D Graphics and Games*, 2022. adr.: <https://advances.realtimerendering.com/> (dostęp 20 paź. 2022).
- [4] T. K. V. W. Group, *Vulkan 1.2.225 - A Specification (with KHR extensions)*, 2022. adr.: <https://registry.khronos.org/vulkan/specs/1.2-khr-extensions/html/> (dostęp 20 paź. 2022).
- [5] Społeczność Godot Engine, *Godot Engine*, 2022. adr.: <https://godotengine.org> (dostęp 20 paź. 2022).
- [6] Epic Games, *Unreal Engine*, ver. 5, 2022. adr.: <https://www.unrealengine.com> (dostęp 20 paź. 2022).
- [7] LunarG, *Vulkan SDK*, 2022. adr.: <https://www.lunarg.com/vulkan-sdk/> (dostęp 20 paź. 2022).
- [8] Khronos Vulkan, OpenGL, and OpenGL ES Conformance Tests, 2022. adr.: <https://github.com/KhronosGroup/VK-GL-CTS> (dostęp 20 paź. 2022).
- [9] K. Group, *SPIR-V Specification*, 2022. adr.: <https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html> (dostęp 20 paź. 2022).
- [10] J. Barczak, *OpenGL Is Broken*, 2022. adr.: <http://www.joshbarczak.com/blog/?p=154> (dostęp 20 paź. 2022).
- [11] A. Overvoorde, *Vulkan SDK*, 2022. adr.: <https://vulkan-tutorial.com/> (dostęp 20 paź. 2022).
- [12] *Architecture of the Vulkan Loader Interfaces*, 2022. adr.: <https://github.com/KhronosGroup/Vulkan-Loader/blob/master/docs/LoaderInterfaceArchitecture.md> (dostęp 20 paź. 2022).
- [13] *GLFW: An OpenGL library*, 2022. adr.: <https://www.glfw.org/> (dostęp 20 paź. 2022).
- [14] *Simple DirectMedia Layer (SDL)*, 2022. adr.: <https://www.libsdl.org/> (dostęp 20 paź. 2022).
- [15] *LLVMpipe - The Mesa 3D Graphics Library*, 2022. adr.: <https://docs.mesa3d.org/drivers/llvmpipe.html> (dostęp 20 paź. 2022).
- [16] *SwiftShader*, 2022. adr.: <https://github.com/google/swiftshader> (dostęp 20 paź. 2022).
- [17] Baldur Karlsson, *Debugger graficzny RenderDoc*, 2022. adr.: <https://renderdoc.org/> (dostęp 20 paź. 2022).
- [18] K. Group, *gltf Sample Models*, 2022. adr.: <https://github.com/KhronosGroup/gltf-Sample-Models> (dostęp 20 paź. 2022).
- [19] J. Ekstrand, *Descriptors are hard*, 2022. adr.: <https://www.jlekstrand.net/jason/blog/2022/08/descriptors-are-hard/> (dostęp 20 paź. 2022).
- [20] *Vulkan Hardware Database - GPUinfo.org*, 2022. adr.: <https://vulkan.gpuinfo.org/> (dostęp 20 paź. 2022).
- [21] W. Engel, *GPU Pro 4: Advanced Rendering Techniques* (An A K Peters Book t. 4). Taylor & Francis, 2013.
- [22] S. Kosarevsky i V. Latypov, *3D Graphics Rendering Cookbook: A comprehensive guide to exploring rendering algorithms in modern OpenGL and Vulkan*. Packt Publishing, 2021. adr.: <https://books.google.pl/books?id=Nys7EAAAQBAJ>.

- [23] Blender Foundation, *Modeler 3D Blender*, 2022. adr.: <https://www.blender.org/> (dostęp 20 paź. 2022).
- [24] K. Group, *gltf 2.0 Specification*, 2022. adr.: <https://registry.khronos.org/gltf/specs/2.0/gltf-2.0.html> (dostęp 20 paź. 2022).
- [25] ArwgLacyProgramming, *Narzędzie do ekstrakcji archiw gier UnArch*, 2022. adr.: www.alprogramming.com (dostęp 20 paź. 2022).
- [26] Y. O'Donnell, *FrameGraph: Extensible Rendering Architecture in Frostbite*, 2022. adr.: <https://kayru.org/publications/> (dostęp 20 paź. 2022).
- [27] K. Group, *Physically-Based Rendering in glTF 2.0 using WebGL*, 2022. adr.: <https://github.com/KhronosGroup/gltf-Sample-Viewer/tree/gltf-WebGL-PBR> (dostęp 20 paź. 2022).
- [28] C11 Reference, 2022. adr.: <https://en.cppreference.com/w/c/11/> (dostęp 20 paź. 2022).
- [29] The Python Standard Library, 2022. adr.: <https://docs.python.org/3/library/index.html#library-index> (dostęp 20 paź. 2022).
- [30] JetBrains, *CLion: A Cross-Platform IDE for C and C++ by JetBrains*, 2022. adr.: <https://www.jetbrains.com/clion> (dostęp 20 paź. 2022).
- [31] Microsoft, *GitHub*. adr.: <https://github.com/> (dostęp 20 paź. 2022).
- [32] Kitware, *CMake*, 2022. adr.: <https://cmake.org> (dostęp 20 paź. 2022).
- [33] PEP 405 – Python Virtual Environments, 2011. adr.: <https://peps.python.org/pep-0405/> (dostęp 20 paź. 2022).
- [34] D. Richard Hipp, *Baza danych SQLite*, 2022. adr.: <https://www.sqlite.org> (dostęp 20 paź. 2022).
- [35] uthash: C macros for hash tables and more, 2022. adr.: <https://github.com/troydhanson/uthash> (dostęp 20 paź. 2022).
- [36] xxHash - Extremely fast hash algorithm, 2022. adr.: <https://github.com/Cyan4973/xxHash> (dostęp 20 paź. 2022).
- [37] cgltf: Single-file glTF 2.0 loader and writer written in C99, 2022. adr.: <https://github.com/jkuhlmann/cgltf> (dostęp 20 paź. 2022).
- [38] cglm: Highly Optimized Graphics Math (glm) for C, 2022. adr.: <https://github.com/recp/cglm> (dostęp 20 paź. 2022).
- [39] stb: single-file public domain libraries for C/C++, 2022. adr.: <https://github.com/nothings/stb> (dostęp 20 paź. 2022).
- [40] POSIX (The GNU C Library), 2022. adr.: https://www.gnu.org/software/libc/manual/html_node/POSIX.html (dostęp 20 paź. 2022).
- [41] Windows API index, 2022. adr.: <https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list> (dostęp 20 paź. 2022).
- [42] libclang: Clang Python Bindings, 2022. adr.: <https://pypi.org/project/libclang/> (dostęp 20 paź. 2022).
- [43] S. Tatham, Metaprogramming custom control structures in C, 2022. adr.: <https://www.chiark.greenend.org.uk/~sgtatham/mp/> (dostęp 20 paź. 2022).
- [44] configparser — Configuration file parser, 2022. adr.: <https://docs.python.org/3/library/configparser.html> (dostęp 20 paź. 2022).
- [45] W. Bright, The X Macro, 2010. adr.: <https://digitalmars.com/articles/b51.html> (dostęp 20 paź. 2022).

- [46] J. E. E. Mansouri, *Rendering 'Rainbow Six / Siege'*, 2018. adr.: <https://www.gdcvault.com/play/1022990/Rendering-Rainbow-Six-Siege> (dostęp 20 paź. 2022).
- [47] S. E. Anderson, *Bit Twiddling Hacks*, 2022. adr.: <http://graphics.stanford.edu/~seander/bithacks.html> (dostęp 20 paź. 2022).
- [48] *SQLite As An Application File Format*, 2022. adr.: <https://www.sqlite.org/appfileformat.html> (dostęp 20 paź. 2022).
- [49] R. P. Nigel Tao Chuck Bigelow, *Go fonts*, 2022. adr.: <https://go.dev/blog/go-fonts> (dostęp 20 paź. 2022).
- [50] C. K. Markus Tavenrath, *Advanced Scenegraph Rendering Pipeline*, 2013. adr.: <https://on-demand.gputechconf.com/gtc/2013/presentations/S3032-Advanced-Scenegraph-Rendering-Pipeline.pdf> (dostęp 20 paź. 2022).
- [51] D. Holden, *Pure Depth SSAO*, 2011. adr.: <https://theorangeduck.com/page/pure-depth-ssao> (dostęp 20 paź. 2022).
- [52] FlightlessMango, *MangoHud: A Vulkan and OpenGL overlay for monitoring FPS, temperatures, CPU-GPU load and more*. 2022. adr.: <https://github.com/flightlessmango/MangoHud> (dostęp 20 paź. 2022).
- [53] L. N. Iain Cantlay, *Analysing Stutter – Mining More from Percentiles*, 2014. adr.: <https://developer.nvidia.com/content/analysing-stutter-%E2%80%93-mining-more-percentiles-0> (dostęp 20 paź. 2022).
- [54] S. Willems, *Vulkan C++ examples and demos*, 2022. adr.: <https://github.com/SaschaWillems/Vulkan> (dostęp 20 paź. 2022).

SPIS RYSUNKÓW

2.1	Kolejność inicjalizacji podstawowych obiektów Vulkan (opracowanie własne)	13
2.2	Warstwowa architektura biblioteki Vulkan (opracowanie własne na podstawie [12])	13
2.3	Obraz 2D 1024x1024 z sceny Sponza [18] i jego 10 mipmap (opracowanie własne)	18
2.4	Cykl życia obrazu łańcucha wymiany (opracowanie własne)	21
2.5	Relacje pomiędzy obiektyami Vulkan używanymi do zarządzania deskryptorami (opracowanie własne)	25
2.6	Schemat blokowy potoku ze specyfikacji Vulkan [4]	31
2.7	Tradycyjne tekstury z dowiązaniami używające jednolitego indeksu w stałej push (opracowanie własne)	36
2.8	Tekstury bez dowiązań używając niejednolitych indeksów instancji (opracowanie własne)	36
2.9	Przepływ pracy dla zasobów (opracowanie własne)	37
2.10	Interfejs programu Blender [23] używanego do modelowania 3D (opracowanie własne) .	37
2.11	Relacje pomiędzy różnymi typami elementów w formacie glTF (opracowanie własne na podstawie [24])	39
2.12	Przykładowy graf renderowania ilustrujący renderowanie odroczone (opracowanie własne)	41
3.1	Proces budowania w formie celów i ich zależności (opracowanie własne)	45
3.2	Relacje pomiędzy modułami silnika i ich najważniejszymi klasami (opracowanie własne)	47
3.3	Relacje pomiędzy obiektymi zasobów w silniku (opracowanie własne)	59
3.4	Przykładowa tekstura dla czcionki Go-Mono [49] (opracowanie własne)	61
3.5	Obiekty biorące udział w procesie konwersji sceny (opracowanie własne)	75
3.6	Przykładowy graf sceny zawierającej dwa trójkąty (opracowanie własne)	76
3.7	Przykładowe drzewo sceny zawierającej dwa trójkąty (opracowanie własne)	77
3.8	Graf renderowania przykładowej sceny (opracowanie własne)	83
4.1	Wynik renderowania sceny MetalRoughSpheres (opracowanie własne)	85
4.2	Wynik renderowania sceny DamagedHelmet (opracowanie własne)	86
4.3	Wynik renderowania sceny Sponza (opracowanie własne)	86

SPIS TABLIC

3.1 Logika przejść układu i zależności pamięci obrazu (opracowanie własne)	67
3.2 Zależność flag obiektów Vulkan od typu bufora (opracowanie własne)	68
4.1 Wyniki pomiarów FPS dla sceny Sponza (opracowanie własne)	87
4.2 Wyniki pomiarów liczników wydajności dla sceny Sponza (opracowanie własne)	88