STRESZCZENIE

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

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

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

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

ABSTRACT

// TODO

SPIS TREŚCI

WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW

Skróty:

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

1. WSTĘP

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

Rozwój teoretyczny grafiki komputerowej jest zdominowany przez doroczną konferencję SIGGRAPH [2], podczas której prezentacje i dyskusje akademickie są przeplecione z targami branżowymi. Rozwój grafiki komputerowy 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 fotorealisteczne rezultaty takie jak śledzenie promieni (ang. ray tracking) i wymaga od programistów zastosowania technik aproksymacji mniej lub bardziej luźno opertych na prawach fizyki oraz użycia bibliotek graficznych wspierających eerację 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 komputerwej, 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 napisana jako biblioteka programistyczna języka C używającej skryptów Python do automatycznej generacji kodu podczas procesu budowania. Renderowanie grafiki 3D jest obsługiwane przez Vulkan API. Zasoby uzywane 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 renderujacego 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. Nowoczesne silniki graficzne składają się z wielu elementów, z których każdy może być niezależnie rozwijany do dowolnie wysokiego poziomu skomplikowania, dlatego trudno je wszystkie dokładnie i wyczerpująco opisać w ramach jednej pracy.

Zakres pracy obejmuje:

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

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

1.3. Struktura pracy

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

Pierwszy rozdział pracy definiuje cel, zakres i strukturę pracy.

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

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

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

W czwartym rozdziale wyrenderowano przykładową scenę i zbadano wydajność silnika.

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

2. WPROWADZENIE DO DZIEDZINY

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

2.1. Podstawowe pojęcia

// TODO matematyka // TODO podział przestrzeni

2.2. Potok zasobów

// TODO

2.2.1. glTF

// TODO

2.3. Vulkan

Biblioteki graficzne pozwalaja 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 został po raz pierwszy wydany w 2016 i oferuje między innymi:

- ponad 1000+ stron zwięzłej, precyzyjnej i szczgółowej specyfikacji API,
- bezstanowość: nie jest używana globalna maszyna stanów jak w OpenGL, co ułatwia wielowątkowość,
- wszystkie informacje używane podczas renderowania są zaszyte w
- zestaw testów zgodności Vulkan CTS [6],
- wieloplatformowość: w odróżnieniu od DirectX wspieranego tylko przez system Windows i konsole Xbox [1],
- ustandaryzowaną niskopoziomową reprezentację shaderów w postaci kodu bajtowego SPIR-V,
- niskopoziomowe i ręczne zarządzanie pamięcią GPU: odciąża to sterownik,
- warstwy walidacji wykrywające nieprawidłowe użycie API.

// TODO 1000+ linijek trójkąt
// TODO HISTORIA, core vs ext, promowanie

2.3.1. Podstawy API

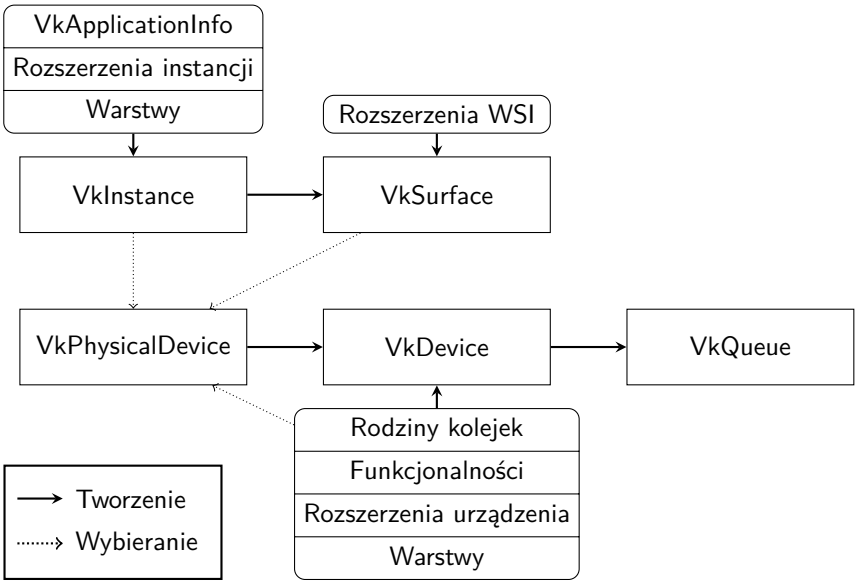
// TODO użycie api, notacje, funcje tworzenia i niszczenia, łańcuch pNext, rozszerzenia instancji i urządzenia

2.3.2. Inicjalizacja podstawowych obiektów

Wszyskie programy używające API Vulkan wymagają wcześniejszego stworzenia następujących obiektów:

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

Poniższy diagram przedstawia kolejność inicjalizacji podstawowych obiektów Vulkan:

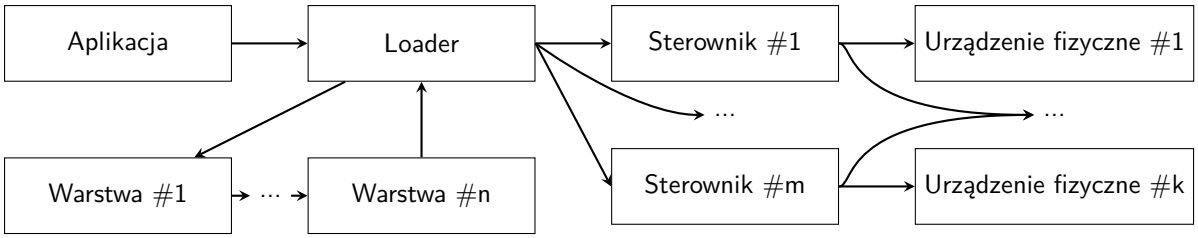


Rysunek 2.1: Kolejność inicjalizacji podstawowych obiektów Vulkan

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 pośredniczącą miedzy 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 [7]:



Rysunek 2.2: Warstwowa architektura biblioteki Vulkan

// HIRO warstwy

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* niewyświetlającym wyniku renderowania na ekranie. W innym wypadku powierzchnia musi być stworzona przed urządzeniem fizycznym, ponieważ jest używana do sprawdzania, czy wybrane urządzenie fizyczne wspiera stworzenie łańcucha wymiany dla powierzchni okna.

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

- *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 pozwalajace na stworzenie VkSurface.

Tworzenie okna jest często obsługiwane przez bibliotekę multimedialną taką jak GLFW [8] czy SDL [9], które posiadają funkcjonalność abstrahującą zawiłości tworzenia powierzchni okna używając WSI.

Urządzenie fizyczne

Po stworzeniu instancji należy wybrać urządzenie fizyczne. Reprezentuje ono pojedyńczą implementację Vulkan - zwykle jedną z kart graficznych obsługiwana przez zainstalowany sterownik graficzny lub renderer programowy taki jak llvmpipe [10] czy [11].

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.

Kazde urządzenie fizyczne jest opisywane ogólnie przy pomocy właściwości i funkcjonalności. Właściwości (*VkPhysicalDeviceProperties*) zawierają wspieraną wersja API, typ, nazwę i producenta GPU oraz jego limity - numeryczne wartości, które muszą być przestrzegane przez program podczas jego użytkowania (przykładowo limit *maxImageDimension2D* definiuje najwyższa obsługiwana wysokość lub szerokość obrazu 2D). Funkcjonalności (*VkPhysicalDeviceFeatures* zawierają długą listę wartości logicznych, które opisują dokładnie możliwości urządzenia (przykładowo *tessellationShader* oznaczaja wsparcie shaderów wyliczania teselacji).

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

Aplikacja musi wybrać z listy kandydatów urządzenie fizyczne, które wspiera wszystkie właściwości i funkcjonalności używane podczas działania aplikacji oraz jest najlepiej najwydajniejsze - powinno uniknąć się sytuacji, w której renderer programowy jest wybierany zamiast GPU.

Urządzenie logiczne

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

Podczas tworzenia urządzenia fizycznego nalezy zdefiniować używane kolejki oraz funkcjonalności i rozszerzenia urządzenia, których wsparcie było sprawdzane podczs wyboru urządzenia fizycznego. Dodatkowo w imię kompatybilności wstecznej 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 urządzeniem logicznego sterownik graficzny automatycznie tworzony żądane kolejki.

Kolejki są używane do wykonywania na urządzeniu fizycznym poleceń zawartych w buforach poleceń wysłanych do kolejki funkcją *vkQueueSubmit()*. Funkcja zwraca kontrolę do aplikacji nieczekając na zakończenie wykonywania bufora poleceń na GPU - wymagana jest synchronizacja GPU z CPU przy pomocy ogrodzeń. Wykonanie buforów poleceń może się odbywać poza kolejnością lub nakładać i wymaga synchronizacji GPU z GPU przy pomocy semafor. Podobnie nie ma silnej gwarancji porządkowania wykonywania poleceń należącego do pojedyńczego bufora i wymaga jawnej synchronizacji używając barier potoku lub zdarzeń.

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

- *GRAPHICS*: kolejka graficzna, wspiera polecenia rysowania *vkCmdDraw*()*,

[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 fańcucha *pNext* i wspierają odpytywanie właściwości i funkcjonalności wprowadzonych przez późniejsze wesje Vulkan oraz rozszerzenia.

- *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()*),
- *SPARSE_BINDING*: kolejka zasobów chronionych, wspiera funkcję *vkQueueBindSparse()* dowiązującą 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 bufory poleceń wysłane do różnych kolejek asynchronicznie zapewniając skalowanie na wielordzeniowych GPU, dlatego warto pomyśleć o użyciu osobnych kolejek transferu, graficznych i obliczeniowych do kopiowania danych i przeplatania poleceń rysowania z obliczeniami GPGPU - pamiętając, że narzut związany z synchronizacją może zniwelować poprawy wydajności.

Uchwyt do 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 w loadera używając funkcji instancji *vkGetInstanceProcAddr()*. Zwrócony wskaźnik nie musi bezpośrednio wskazywać na implementację oferowaną przez sterownik i może być funkcją loadera wykonującą dodatkową logikę dodaną przez załadowane warstwy. Funkcja *vkGetDeviceProcAddr()* pozwala na pominięcie loadera, co gwarantuje szybsze wywołania API, ale zwrócona funkcja może być używana tylko dla urządzenia logicznego użytego do pobrania jej.

2.3.3. Łańcuch wymiany

Łańcuch wymiany to obiekt *VkSwapchainKHR* będący częścia rozszerzenia *VK_KHR_swapchain* WSI i reprezentuje tablica prezentowalnych obrazów należących do powierzchni okna. Jest on używany do prezentacji obrazu, czyli aktualizacji powierzchni okna zawartością wyrenderowanego obrazu. 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 uchwyrów przy użyciu indeksu zwróconego przez funkcję *vkAcquireNextImageKHR()*,
- Wyrenderować scenę do dostępnego prezentowalnego obrazu przy użyciu funkcji vkQueueSubmit(),
- Oddać wyrenderowany obraz łańcuchowi wymiany funkcją *vkQueuePresentKHR()*.

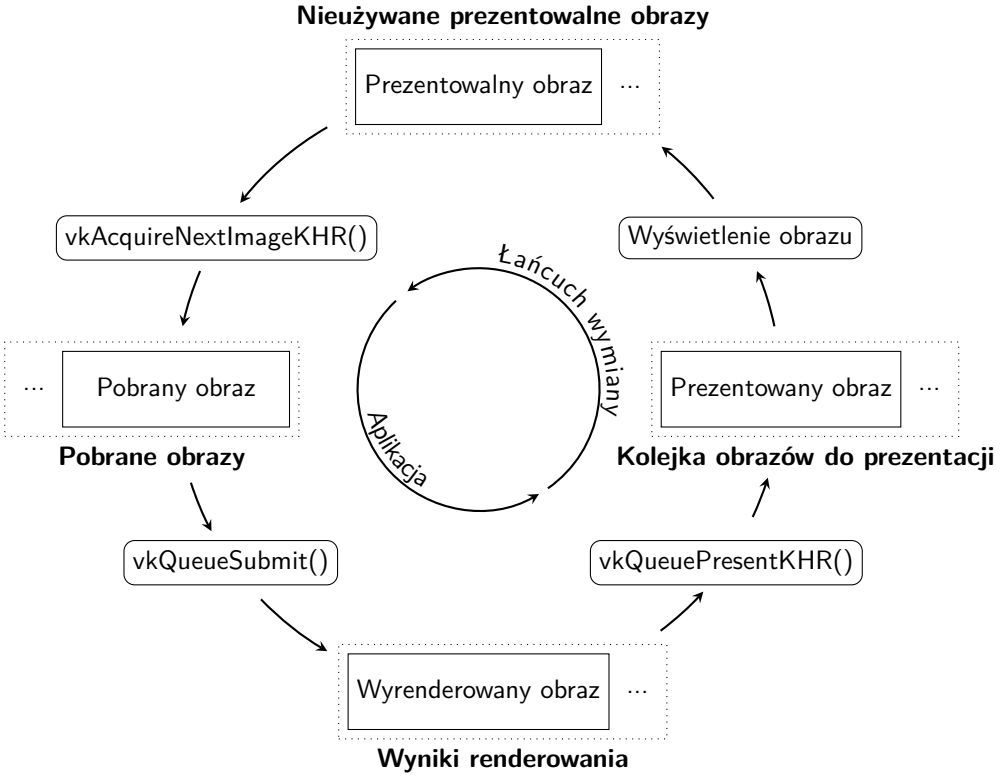
Każdy z tych kroków wymaga synchronizacji z krokiem następnym przy użyciu semaforów. Funkcja *vkAcquireNextImageKHR()* sygnalizuje *semafor dostępności obrazu*, na który czeka funkcja *vkQueu-*

eSubmit() - GPU zaczyna renderowanie dopiero wtedy, gdy prezentowany obraz nie jest używany przez okno. Funkcja *vkQueueSubmit()* sygnalizuje *semafor zakończena renderowania*, na który czeka funkcja *vkQueuePresentKHR()* - okno może zacząć prezentować wynik renderowania dopiero wtedy, gdy GPU zakończył wykonywanie poleceń.

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.

Poniższy diagram ilustruje cykl życia obrazu łańcucha wymiany:



Rysunek 2.3: Cykl życia obrazu łańcucha wymiany

Dokładny mechanizm działania silnika prezentacji zależy od wybranego trybu prezentacji: (*VkPresentModeKHR*):

- *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 pionowa 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 specyfiakcją 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.

Istnieją sytuacja, w których łańcuch wymiany musi być odtworzony (zniszczony i stworzony). Fukcje *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 chciać 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.

2.3.4. Bufory poleceń

// TODO diagram state, bindings

2.3.5. Zasoby

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

2.3.6. Synchronizacja

// TODO

Bariery potoku

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

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

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

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

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

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

- bariery pamięci obrazów: dla zakresu obrazu, dodatkowo pozwala na tranzycje układu obrazu,

- bariery pamięci buforów: dla zakresu bufora,
- globalne bariery pamięci: dla wszystkich istniejących obiektów,

// TODO użycie // TODO listingi?

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 graficzna i kolejką prezentacji w celu zagwarantowania, że prezentowalny obraz łańcucha wymiany jest używany tylko przez jedną kolejkę.

Ogrodzenia

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

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

Zdarzenia

Zdarzenia to obiekty *VkEvent* pozwalające 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 sygnalizowane, 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 optymalniejsze od elastyczniejszych zdarzeń.

2.3.7. Deskryptory i stałe push

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

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

Tworzenie deskryptorów

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

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

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

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

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

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

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

Typ deskryptora zależy od rodzaju opisywanego zasobu, przykładowo:

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

Aktualizacja deskryptorów

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

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

Stałe push

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

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

Zadeklarowanie użycia deskryptorów w układzie potoku

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

Podczas tworzenia należy zadeklarować:

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

Zakres stałej push składa się z:

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

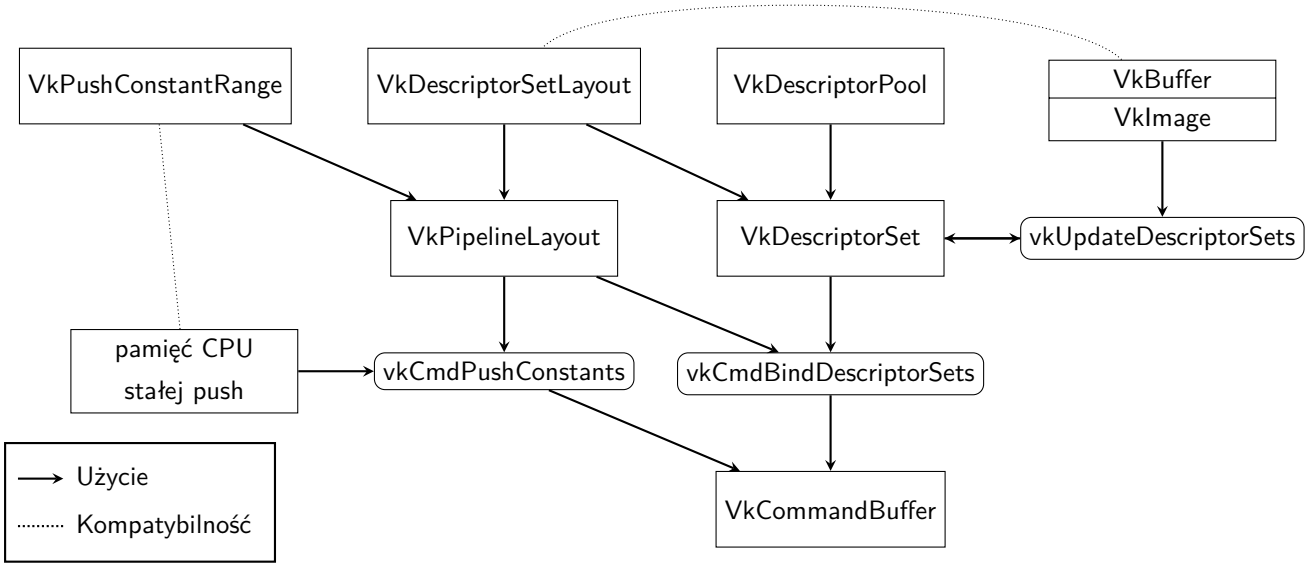
Liczba poszczególnych typów deskryptorów uzwględnionych w potoku renderowania jest ograniczona limitami urządzenia fizycznego. Limit *maxPerStageResources* to maksymalna liczba zasobów, które mogą być dostępne dla pojedyńczego etapu cieniowania. Rodzina limitów *maxDescriptorSet**, gdzie * to typ deskryptora, 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 komeny *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 poniższym diagramie:



Rysunek 2.4: Relacje pomiędzy obiektami Vulkan używanymi do zarządzania deskryptorami

Dostęp do zasobów w shaderach

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

Przykładowo kwalifikator układu dla pojedyńczego deskryptora typu UNIFORM_BUFFER z dowiązania o numerze x ze zbioru o numerze y ma następującą formę:

```
struct bufferStruct {
        vec3 field1;
        mat4 field2;
        ...
};
layout(scalar, set = y, binding = x) uniform bufferBlock {
        bufferStruct buffer;
};
```

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

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

2.4. Rozszerzenie VK_EXT_descriptor_indexing

Rozszerzenie *VK_EXT_descriptor_indexing* wprowadziło szereg dodatkowych funkcjonalności pozwalających na tworzenie dużych zbiorów deskryptorów zawierających wszystkie zasoby używane przez program. Celem tego jest umożliwienie technik renderowania bez dowiązań. Z powodu swojej użyteczności rozszerzenie to zostało promowane w Vulkan 1.2. W kolejnych sekcjach opisano nowe funkcjonalości.

2.4.1. Niejednolite dynamiczne indeksowanie deskryptorów

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

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

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

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

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

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

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

W czasie pisania pracy powyższe funkcjonalności są wspierane przez więcej niż 90% urządzeń [12].

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

```
#extension GL_EXT_nonuniform_qualifier : require
```

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

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

Wymóg jednolitości podczas indeksowania deskryptorów był spowodowany ograniczeniami poprzednich generacji GPU - w modelu renderowania OpenGL dostęp do dowiązanych tekstur odbywał się pośrednio poprzez jednostki teksturujące, które były widoczne przez wszystkie wywołania shaderów i nie pozwalały na zmianę dołączonych tekstur podczas wykonywania polecenia rysowania [13].

2.4.2. Aktualizacja deskryptorów po dowiązaniu

Domyślnie deskryptory nie mogą być aktualizowane po nagraniu ich dowiązania w buforze poleceń, przez co aplikacja musi mieć kompletną wiedzę o wszystkich używanych zasobach w trakcie nagrywania buforów poleceń. Nie dotyczy to jednak deskryptorów używających funkcjonalności aktualizacji po dowiązaniu, co pozwala na elastyczniejsze zarządzanie zasobami poprzez odroczenie aktualizacji zbiorów deskryptorów aż do momentu bezpośrednio przed wykonaniem bufora poleceń.

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

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

W czasie pisania pracy powyższe funkcjonalności są wspierane przez ponad 90% urządzeń wyłączając bufory uniform niewspierane przez ok. 40% platform [12].

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

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

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

2.4.3. Dowiązanie deskryptora o zmiennej wielkości

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

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

Wsparcie dowiązań o zmiennej wielkości wymaga funkcjonalności urządzenia Vulkan 1.2 *descriptorBindingVariableDescriptorCount*, która w czasie pisania pracy jest wspierana przez ponad 90% urządzeń [12].

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

2.4.4. Częściowe dowiązania deskryptorów

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

Dzięki tej funkcjonalności deskryptory muszą być dynamicznie używane: deskryptory nieużywane przez shadery mogą być nieprawidłowe i dodatkowo mogą być nawet aktualizowane gdy zbior deskryptorów jest używany przez GPU - pamiętając, że dostęp przy użyciu nieprawidłowego deskryptora jest wciąż niezdefiniowanym zachowaniem i aplikacja musi zapewnić odpowiednią synchronizację CPU-GPU.

Wsparcie dowiązań o zmiennej wielkości wymaga funkcjonalności urządzenia Vulkan 1.2 *descriptorBindingPartiallyBound*, która w czasie pisania pracy jest wspierana przez ponad 94% urządzeń [12].

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

2.4.5. Tablice deskryptorów czasu wykonania

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

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

Wsparcie tablic deskryptorów czasu wykonania wymaga funkcjonalności urządzenia Vulkan 1.2 *runtimeDescriptorArray*, która w czasie pisania pracy jest wspierana przez ponad 94% urządzeń [12].

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

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

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

```
OpCapability Shader
OpCapability RuntimeDescriptorArray
OpExtension "SPV_EXT_descriptor_indexing"
...
OpName %textures2D "textures2D"
...
OpDecorate %textures2D DescriptorSet 0
OpDecorate %textures2D Binding 2
...
%150 = OpTypeImage %float 2D 0 0 0 1 Unknown
%151 = OpTypeSampledImage %150
%_runtimearr_151 = OpTypeRuntimeArray %151
%_ptr_UniformConstant__runtimearr_151 =OpTypePointer UniformConstant %_runtimearr_1
%textures2D =
        OpVariable %_ptr_UniformConstant__runtimearr_151 UniformConstant
```

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

// TODO

2.5. Przebiegi renderowania i potoki

// TODO

2.6. Rozszerzenie VK_EXT_dynamic_rendering

// TODO

2.7. Renderowanie bez dowiązań

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

W Vulkan renderowanie bez dowiązań jest realizowane poprzez:

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

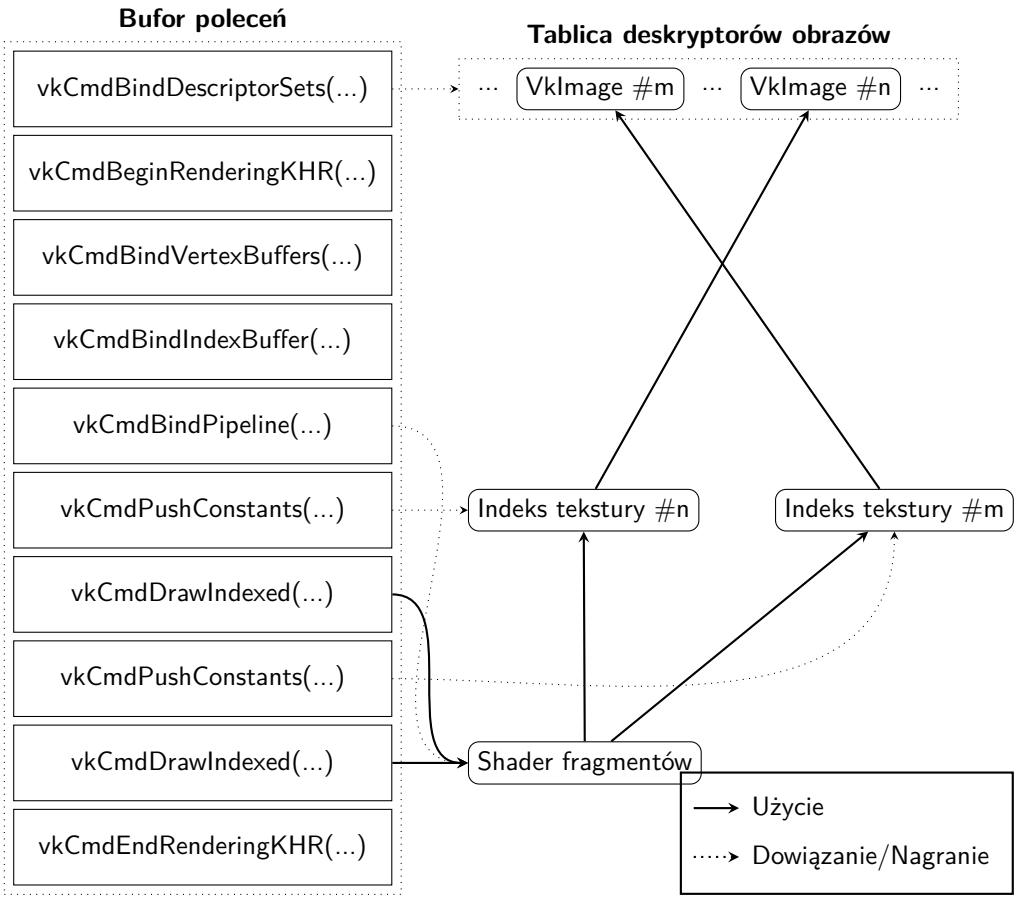
// TODO unified geometry buffer

2.7.1. Tekstury bez dowiązań

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

- dowiązanie całkowicie nowego zbioru deskryptorów: bardzo kosztowaną 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 pojedyńczej instancji: wbudowania zmienna shaderów *gl_InstanceIndex* jest niejednolita tylko dla poleceń rysowania renderujacych wiecej niż jedną instancję.

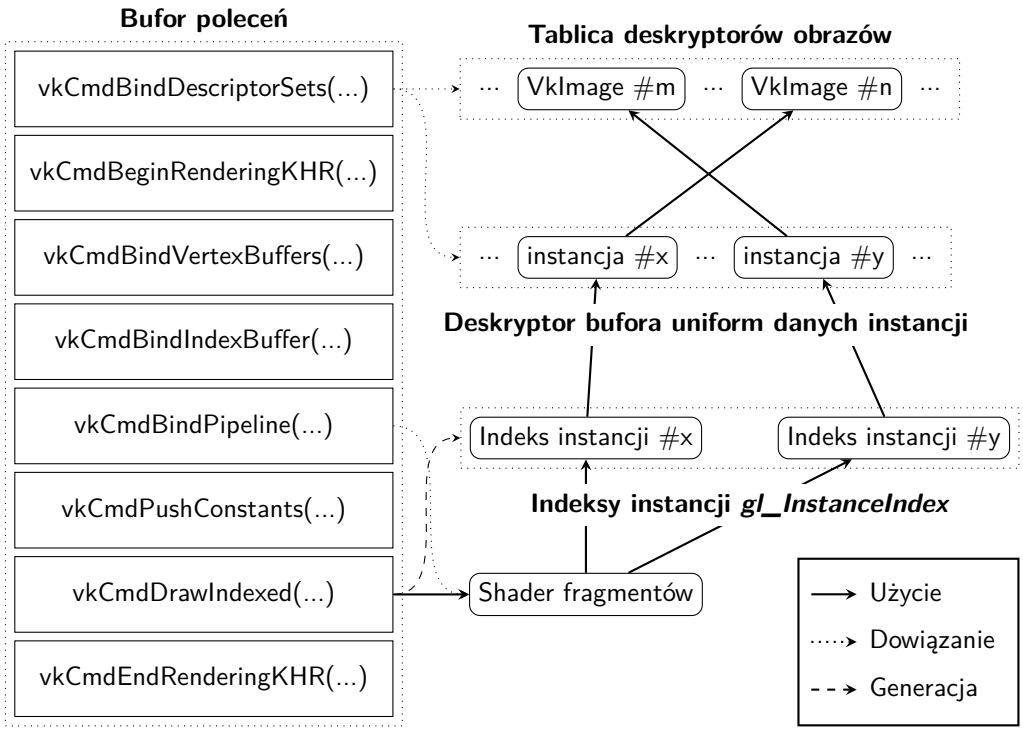
Poniższy diagram przedstawia tradycyjne tekstury z dowiązaniami używające jednolitego indeksu w stałej push:



Rysunek 2.5: Tradycyjne tekstury z dowiązaniami używające jednolitego indeksu w stałej push

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

Poniższy diagram przedstawia tekstury bez dowiązań używając niejednolitych indeksów instancji:



Rysunek 2.6: Tekstury bez dowiązań używając niejednolitych indeksów instancji

2.7.2. Geometria bez dowiązań

// TODO

2.8. Mapowanie tekstur

// TODO

2.9. Oświetlenie

// TODO

2.10. Cieniowanie odroczone

// TODO

2.11. Graf sceny

// TODO

2.12. Graf renderowania

// TODO

3. ARCHITEKTURA I IMPLEMENTACJA

3.1. Użyte narzędzia

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

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

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

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

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

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

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

3.1.1. Proces budowania

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

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

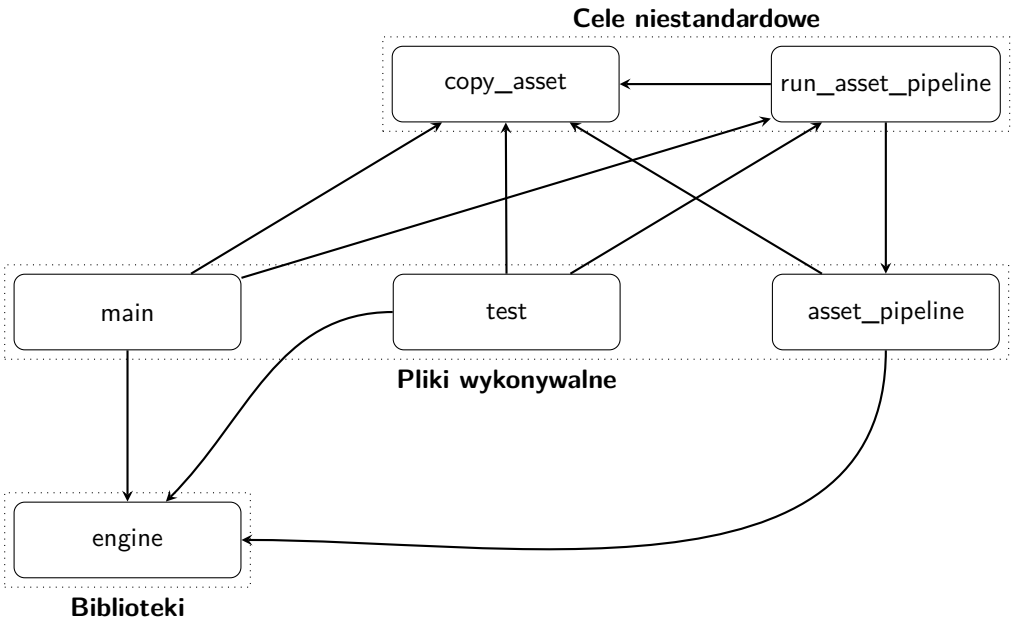
Skrypty w języku Python są obsługiwane pośrednio przez CMake, które wykrywa zainstalowany interpreter języka Python i używa go do stworzenia tzw. środowiska wirtualnego w tymczasowym katalogu venv/ w głównym katalogu projektu. Podczas procesu budowania środowisko wirtualne jest używane do zainstalowania wymaganych zewnętrznych bibliotek w języku Python i wykonywania skryptów generatora kodu. Zaletą użycia środowiska wirtualnego w porównaniu do bezpośredniego wywoływania zainstalowanego interpretera Pythona jest izolacja zarządzania zależnościami od reszty systemu operacyjnego, co pozwala na łatwiejszą powtarzalność podczas debugowania [14].

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

Wyróżniane są trzy rodzaje celów:

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

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



Rysunek 3.1: Proces budowania w formie celów i ich zależności

engine Cel budujący bibliotekę programistyczną zawierającą implementację silnika.

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

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

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

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

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

// TODO więcej o celach

3.1.2. Biblioteki zewnętrzne

Projekt używa następujących zewnętrznych bibliotek programistycznych:

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

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

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

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

3.2. Architektura

// TODO

3.3. Moduły

// TODO

4. BADANIA

5. PODSUMOWANIE

// TODO

WYKAZ LITERATURY

[1] J. F. Hughes i in., *Computer Graphics: Principles and Practice*, 3 wyd. Upper Saddle River, NJ: Addison-Wesley, 2013.

[2] *Konferencja SIGGRAPH*, 2022. adr.: https://www.siggraph.org/.

[3] *Advances in Real-Time Rendering in 3D Graphics and Games*, 2022. adr.: https://advances.realtimerendering.com/.

[4] Społeczność Godot Engine, *Godon Engine*, 2022. adr.: https://godotengine.org.

[5] Epic Games, *Unreal Engine*, wer. 5, 2022. adr.: https://www.unrealengine.com.

[6] *Khronos Vulkan, OpenGL, and OpenGL ES Conformance Tests*, 2022. adr.: https://github.com/KhronosGroup/VK-GL-CTS.

[7] *Architecture of the Vulkan Loader Interfaces*, 2022. adr.: https://github.com/KhronosGroup/Vulkan-Loader/blob/master/docs/LoaderInterfaceArchitecture.md.

[8] *GLFW: An OpenGL library*, 2022. adr.: https://www.glfw.org/.

[9] *Simple DirectMedia Layer (SDL)*, 2022. adr.: https://www.libsdl.org/.

[10] *LLVMpipe - The Mesa 3D Graphics Library*, 2022. adr.: https://docs.mesa3d.org/drivers/llvmpipe.html.

[11] *SwiftShader*, 2022. adr.: https://github.com/google/swiftshader.

[12] *Vulkan Hardware Database - GPUinfo.org*, 2022. adr.: https://vulkan.gpuinfo.org/.

[13] W. Engel, *GPU Pro 4: Advanced Rendering Techniques* (An A K Peters Book t. 4). Taylor & Francis, 2013.

[14] *PEP 405 – Python Virtual Environments*, 2011. adr.: https://peps.python.org/pep-0405/.

SPIS RYSUNKÓW

SPIS TABLIC