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.
- ISA - Instruction Set Architecture, architektura procesora,
- SDK - Software Development Kit, zestaw narzędzi dla programistów aplikacji.

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. 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 w wersji 1.0 został po raz pierwszy wydany w 2016. By zacząć używać Vulkan należy pobrać Vulkan SDK rozwijany przez LunarG [6]. Zawiera on m.in. specyfikację, nagłówki API, biblioteki (de)kompilujące kod SPIR-V oraz warstwy walidacji.

Specyfikacja Specyfikacja Vulkan [7] to ponad 1000 stron zwięzłej, precyzyjnej i szczgółowej 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 przeez urządzenia firmy Apple).

Obiektowość i bezstanowość 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 całkowiecie zaszyty w obiektach i wszystkie funkcje API operują tylko 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 parametu w danym momencie - przykładowo większość funkcji nagrywania poleceń *vkCmd** wymagają zewnętrznej synchronizacji obiektu bufora poleceń *VkCommandBuffer*, co oznacza, że nagrywanie pojedyńczego bufora poleceń powinno się najlepiej odbywać w ramach jednego wątku.

SPIR-V Vulkan używa niskopoziomowej pośredniej reprezentacji shaderów w postaci kodu bajtowego SPIR-V [9], który jest standardem Khronos będącym przenośnym celem kompilacji shaderów napisanych w wysokopoziomowych językach takich jak GLSL, HLSL czy Cg. Implementatorzy zajmują się tylko

translację 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].

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

Na szczęście Vulkan wspiera tworzenie warstw - małych bibliotek dynamicznych pośredniczących pomiędzy apikacją i sterownikiem, które przechwytują 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* wykrywaja i raportuje nieprawidłowe i niewydajne użycie API.

Warstwy rozszerzeń implementują funkcje rozszerzeń niewspieranych przez sterownik - przykładowo warstwa *VK_LAYER_KHRONOS_synchronization2* implementuje rozszerzenie *VK_KHR_synchronization2*.

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.

Rozszerzenia Nowa funkcjonalność jest dodawana podobnie jak w OpenGL uzywając 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 *EXT* 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 postawowej specyfikacji - przykładowo rozszerzenie *VK_EXT_scalar_block_layout* zostało promowane w Vulkan 1.2.

2.2.1. Podstawy API

API Vulkan posiadaja regularną i przewidywalną strukturę nazewnictwa oraz użycia obieków.

Konwencje nazewnictwa

Konwencje nazewnictwa API Vulkan posiadają regularną i przewidywalną strukturę. 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 zwięzłości dalej w pracy części nazwy będą pomijane jeśli można je wywnioskować z kontekstu - przykładowo *VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT* może być zapisywany jako etap potoku *COLOR_ATTACHMENT_OUTPUT*.

Funkcje *vkCmd*()* są nazywane poleceniami i nagrywają polecenie przeznaczone do wykonania na GPU do bufora poleceń - przykładowo polecenie *vkCmdCopyBufferToImage* nagrywa polecenie kopiowania bufora do obrazu.

Funkcje *vkEnumerate*()* i *vkGet*()* służą do odpytywania właściwości obiektów - przykładowo funkcja *vkGetPhysicalDeviceQueueFamilyProperties()* raportuje 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 wcześniej utworzonego obiektu funkcją *vkAllocate*()*.

Tworzenie obiektu wymaga przygotowania struktury informacji tworzenia *Vk*CreateInfo*:

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

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

Alokacja obiektu wymaga przygotowania struktury informacji alokacji *Vk*AllocateInfo*:

```
Vk*Pool pool;

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

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

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 stuktury może być wywnioskowany używając jej pierwszego pola *sType*.

2.2.2. 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.

Wszystkie aplikacjie graficzne mogą być jednak podsumowane następującymi ogólnymi krokami:

- 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 wyrenderowaniego obrazu.

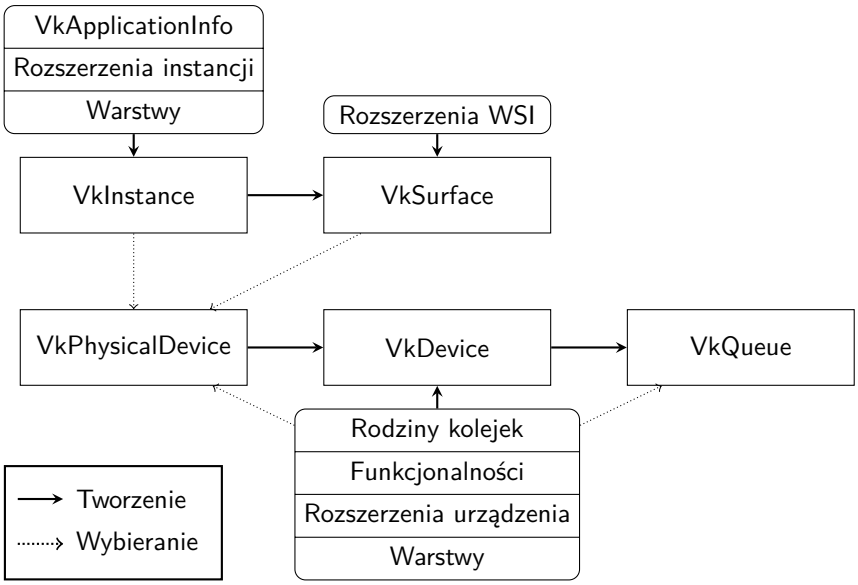
W kolejnych sekcjach pracy przybliżono wspomniane powyżej obiekty Vulkan i zaprezentowano metody ich użycia.

2.2.3. Inicjalizacja podstawowych obiektów

Wszyskie 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:

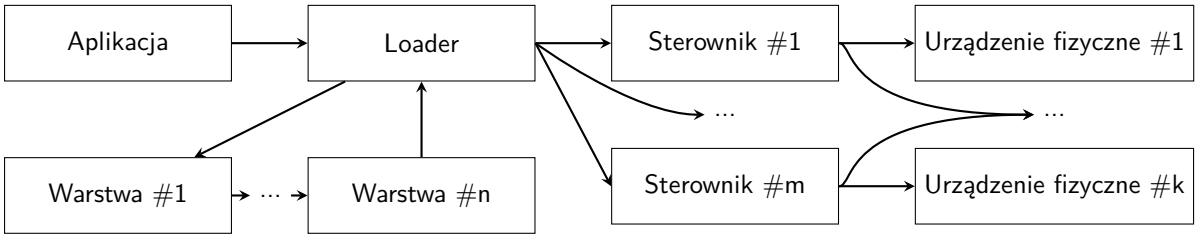


Rysunek 2.1: Kolejność inicjalizacji podstawowych obiektów Vulkan

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 [12]:



Rysunek 2.2: Warstwowa architektura biblioteki Vulkan

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 będzie wspierało 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 [13] czy SDL [14], 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 [15] czy [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.

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.

[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.

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*()*,
- *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 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.

Rozszerzenie VK_EXT_debug_utils

Podczas inicjalizacji Vulkan warto rozważyć użycie rozszerzenia *VK_EXT_debug_utils*, które pozwala na stworzenie komunikatora debugowania (ang. debug messanger) przechwytującego wszystkie komunikaty wygenerowane przez loader, warstwy i sterownik Vulkan. Przechwycone komunikaty debugowania wraz z priorytetami są przekazywne do wywołania zwrotnego zdefiniowanego przez programistę, które może przykładowo logować wiadomość. Użycie debuggera 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 debugownia.

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.

2.2.4. Zasoby

Vulkan wyróżnia dwa rodzaje zasobów: bufory (*VkBuffer*) i obrazy (*VkImage*).

Bufory

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

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 (*VkUsageFlags*),

- tryb udostępniania (*VkSharingMode*)
- początkowy układ (*VkUsageFlags*).

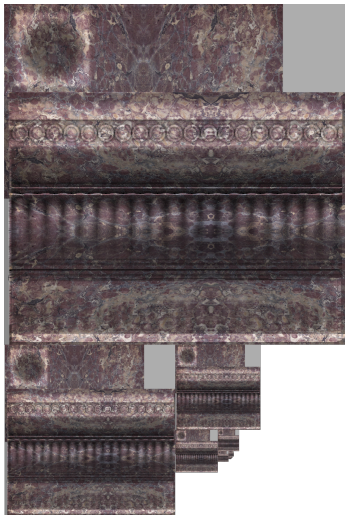
// HIRO opisz powyższe

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 pojedeńczy element jest zwany warstwą. Warstwy nie są liczone jako wymiar obrazu - obraz 2D z warstwami nie jest obrazem 3D.

Flagi tworzenia pozwalają na definicję dodatkowych funkcjonalności. Przykładowo flaga *CUBE_COMPATIBLE* pozwala na traktowanie obrazu z 6 kwadratowymi warstwami jako obrazu sześciennego.

Mipmapa to kopia obrazu z każdym wymiarem zmniejszonym dwukrotnie. W obrazie z mipmapami pierwotny obraz traktuowany jest jako pierwszej mipmapa poziomu zerowego, z której generowane są mipmapy następonych poziomów aż do osiągnięcia wymiaru 1x1x1. Przykładowo poniższy obraz 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,



Rysunek 2.3: Obraz 2D 1024x1024 z modelu Sponza [18] i jego 10 mipmap

które jest popularną techniką zwiększającej prędkość i jakość renderowania poprzez wprowadzenie nowych metod filtrowania podczas próbkowania obrazu.

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 rozkład tekseli obrazu jest decydowany całkowicie przez sterownik - wybrane kafelkowanie ma na celu zwiększenie prędkości dostępu poprzez poprawę lokalności przestrzennej w pamięci podręcznej GPU.

// HIRO próbkowanie, filtrowanie mag, min, mipmap // HIRO tekstury pozeekranowe

// HIRO pamięć // HIRO widok // HIRO bufory uniform i bufory magazynowe, obrazy, próbniki, obrazy magazynowe

Format i przestrzeń kolorów Format (*VkFormat*) definiuje rozmiar, strukturę i sposób kodowania podczas zapisu i dekodowania podczas próbkowania pojedyńczego 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 powszechnia 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 zmienno przecinkowa pomiędzy 0 i 1), używany przez teksturę pozekranowa będącą dołączeniem koloru,
- *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 bufory głębi.

2.2.5. 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()*.

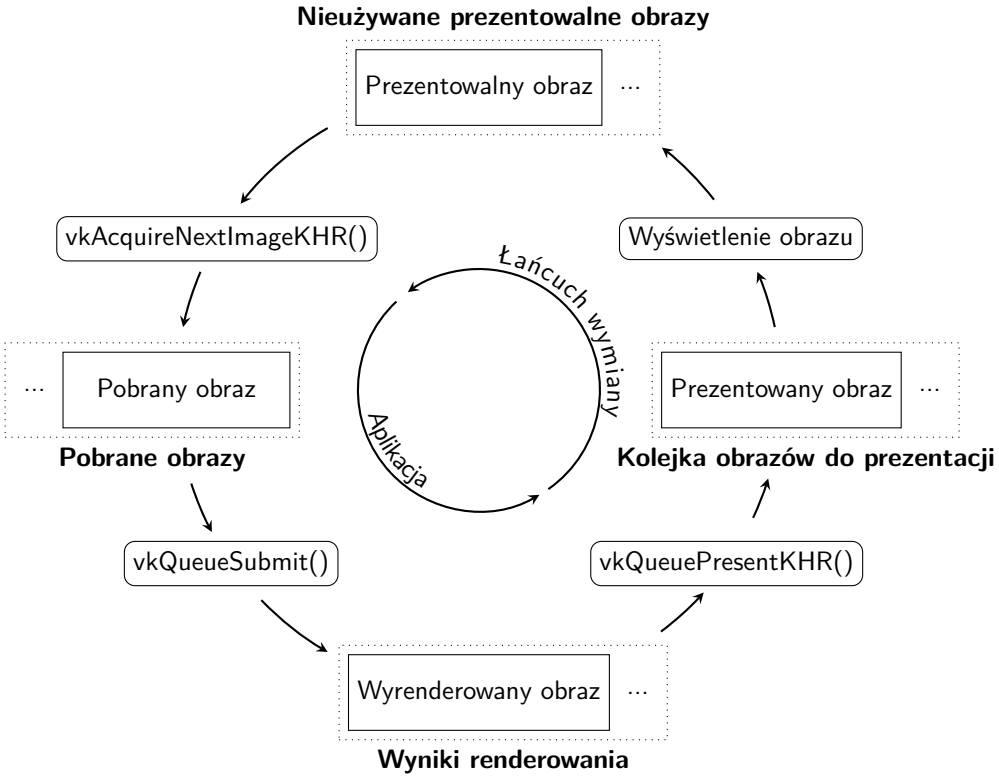
Każdy z tych kroków wymaga synchronizacji z krokiem następnym przy użyciu semaforów. Funkcja *vkAcquireNextImageKHR()* sygnalizuje *semafor dostępności obrazu*, na który czeka funkcja *vkQueueSubmit()* - GPU zaczyna renderowanie dopiero wtedy, gdy prezentowany obraz nie jest używany przez okno. Funkcja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

Prezentacja funkcją *vkQueuePresentKHR()* wymaga użycia uchwytu kolejki prezentacji, która jest dowolną kolejką wspierająca prezentację. Nie istnieje osobna rodzina kolejek prezentacji, zwykle 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.

Poniższy diagram ilustruje cykl życia obrazu łańcucha wymiany:



Rysunek 2.4: Cykl życia obrazu łańcucha wymiany

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

Trybu 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 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.

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 zwrócony przez funkcję *vkAcquireNextImageKHR()*. Następnie należy utworzyć widoki prezentowalnych obrazów, które będą później używane do renderowania do nich.

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.

2.2.6. Bufory poleceń

// TODO diagram stanu,
// TODO Rodzaje poleceń: zmiana stanu (dowiązywanie), rysowanie
// TODO ponowne użycie vs nagrywanie co klatkę

2.2.7. Synchronizacja

// TODO wstęp o synchronizacji

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.

2.2.8. 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 [19], 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.

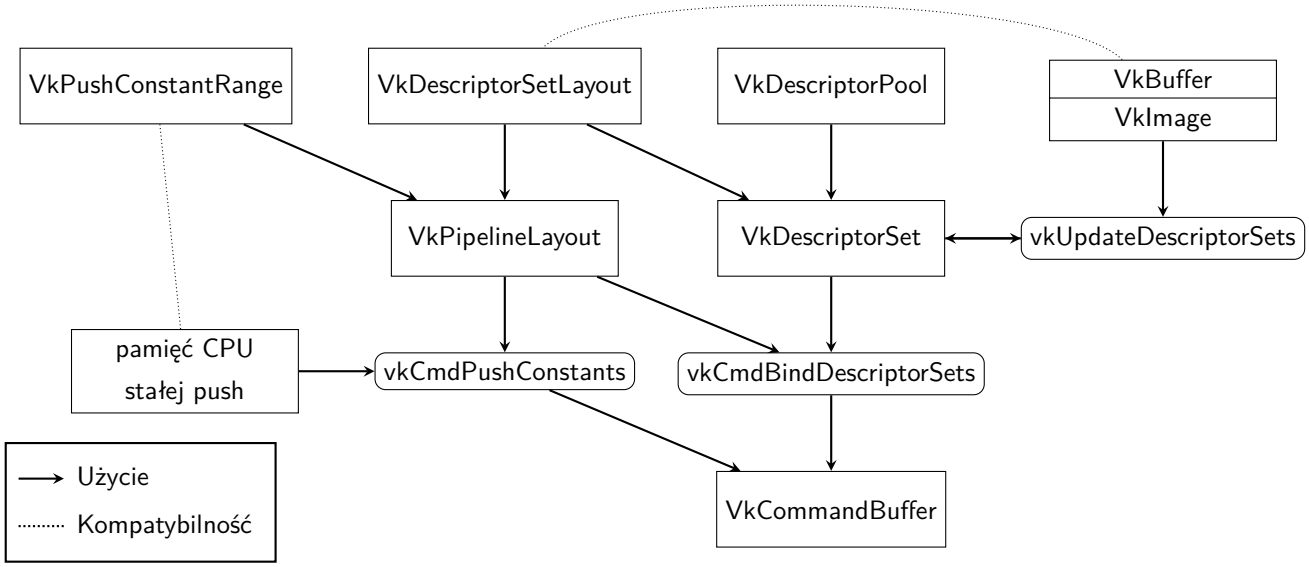
Liczba poszczególnych typów deskryptorów uzwględnionych w potoku renderowania jest ograniczona limitami urządzenia fizycznego. Limit *maxPerStageResources* to maksymalna liczba zasobów, które mogą być dostępne dla pojedyńczego etapu cieniowania. Rodzina limitów *maxDescriptorSet**, gdzie *** to typ deskryptora, 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 2.5: 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];
```

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

Układ pamięci scalar wymaga ona 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 następującą dyrektywę preprocesora:

```
#extension GL_EXT_scalar_block_layout : require
```

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.

Powyższe 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.

2.3. 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.

2.3.1. 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 [20].

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 [21].

2.3.2. 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 [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.

2.3.3. 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 [20].

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*.

2.3.4. 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 [20].

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

2.3.5. 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 [20].

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.

2.4. Przebiegi renderowania i potoki

// TODO vkPipeline - potok, potok graficzny

// HIRO globalna macierz przekształceń (macierz model-widok-rzutowanie), przestrzenie współrzędnych, Reverse-Z

Topologia definiuje sposób renderowania wierzchołków.

Format wierzchołka (w tym rozdzielanie lub separacja atrybutów) ... Wierzchołek składa się z kilku różnych atrybutów wierzchołka, np. pozycji, normalnej, i koloru.

Indeksów wierzchołków pozwalających na kompresję bufora geometrii używanego przez polecenia rysowania poprzez użycie bufora indeksów definiującego

// TODO

2.4.1. Moduły shaderów

Shader to program specyfikujący operacje wykonywane podczas etapu potoku dla każdego przetwarzanego przez niego elementu (np. wierzchołka, punktu sterującego, fragmentu lub grupy roboczej).

Kod źródłowy GLSL reprezentuje shader w formie tekstowej przy użyciu języka programowania potoku graficznego składniowo zbliżonego do jezyka C.

Kod bajtowy SPIR-V reprezentuje shader w formie ustandaryzowanej binarnej reprezentacji pośredniej. Jest on zwykle uzyskiwany poprzez kompilację kodu źródłowego w języku wyższego poziomu takiego jak GLSL.

Moduł shaderów to obiekt *vkShaderModule* uzyskiwany poprzez kompilację kodu bajtowego SPIR-V do kodu binarnego w ISA GPU używając funkcją *vkCreateShaderModule()*. Po kompilacji moduł shadera jest w formie, która może być używana bezpośrednio przez potok.

// HIRO diagram spirv->moduł

// TODO vkRenderPass - przebieg renderowania, dołączenia koloru

2.4.2. Rozszerzenie VK_EXT_dynamic_rendering

// TODO dynamiczne przebiegi renderowania

2.5. 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.

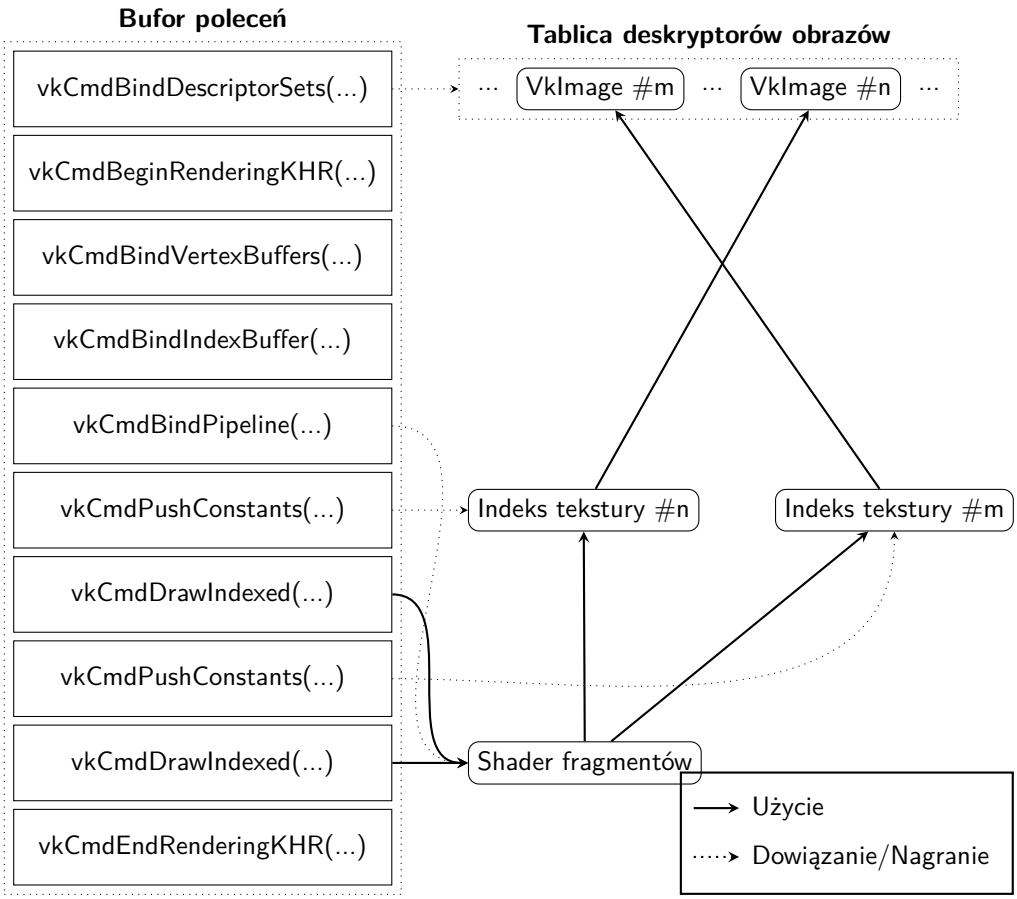
// TODO unified geometry buffer
// TODO unified uniform buffer
// TODO instancjonowanie, polecenia rysowania pośredniego,

2.5.1. 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.

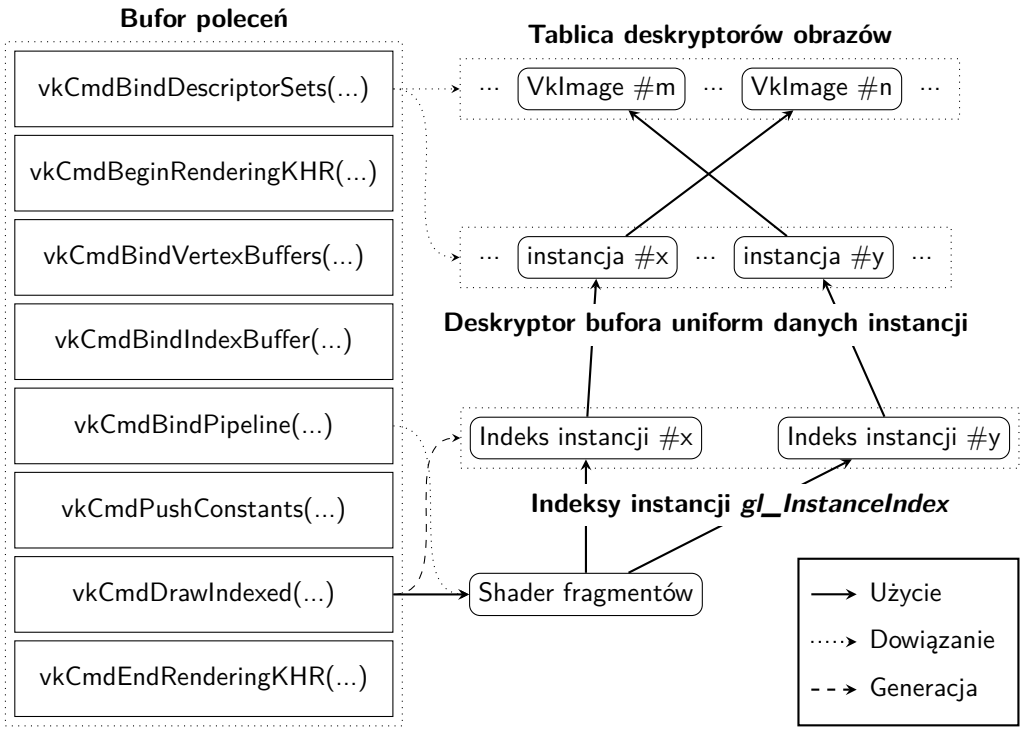
Poniższy diagram przedstawia tradycyjne tekstury z dowiązaniami używające jednolitego indeksu w stałej push:



Rysunek 2.6: 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 2.7: Tekstury bez dowiązań używając niejednolitych indeksów instancji

2.5.2. Geometria bez dowiązań

// TODO draw call batching używając jednolitego bufora geometrii i poleceń rysowania pośredniego

2.6. Mapowanie tekstur

// TODO filtrowanie w samplerze (powiększający, pomniejszający)

// TODO mapowanie nierówności

// TODO teksturowanie sześcienne Skybox to sześcian oteksturowany przy pomocy tekstury sześciennej przedstawiającej niebo.

2.7. Oświetlenie

// TODO model PBR

Materiał to zbiór parameterów i tekstur używanych do przez shadery do renderowania powierzchni prymitywów.

W formacie glTF materiał jest zdefiniowany przy użyciu modelu PBR metaliczności-chropowatości i posiada 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 bę-

dące wartościami w przedziale $[0,1]$ mnożonymi przez tekstury metaliczności-chropowatości - wartości metaliczńści są próbkowane z komponenu B, a wartości metaliczńści z komponentu G.

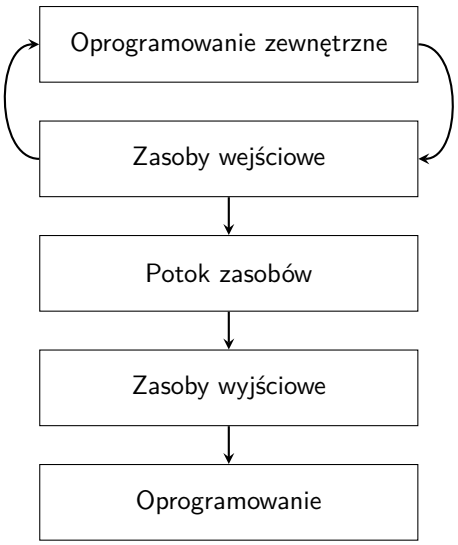
// TODO glTF shader

2.8. Cieniowanie odroczone

// TODO

2.9. Potok zasobów

W kontekście silnika zasób (ang. asset) można zdefiniować jako ogół 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 2.8: Przepływ pracy dla zasobów

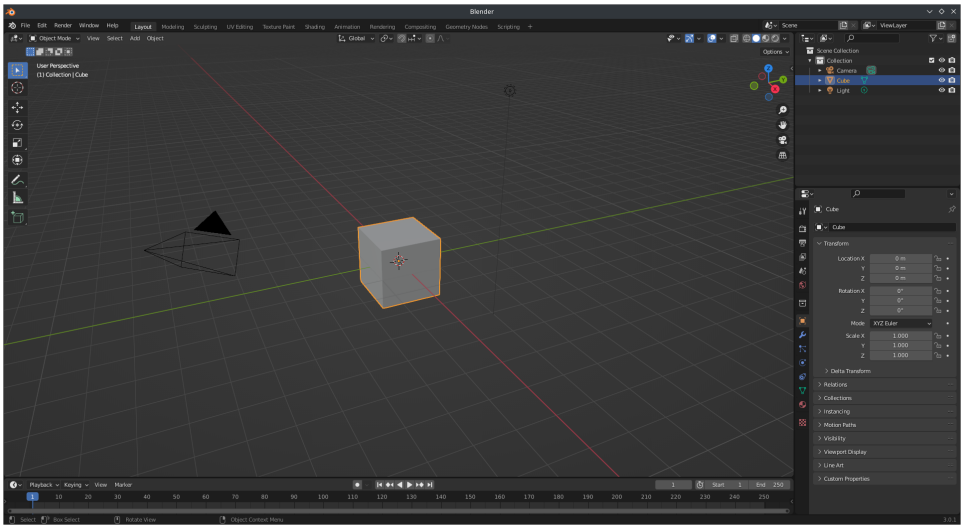
2.9.1. Zasoby wejściowe

Zasoby wejściowe mają formę plików w formacie przystosowanym do edycji przy użyciu zewnętrznego oprogramowania. Oczywistymi 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.

Formaty opisu sceny glTF

Liczba możliwych elementów składającymi się na zasób 3D skłoniła branżę do opracowania ustandaryzowanych 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 reimplementacji wymaganych części formatu inżynierią wsteczną.



Rysunek 2.9: Interfejs programu Blender [22] używanego do modelowania 3D

- *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 wyrenderowania bądź dalszego przetworzenia. Wersja 2.0 wydana w 2021 zerwała kompatybilność wsteczną z wersją 1.0 i całkowicie ją zastąpiła.

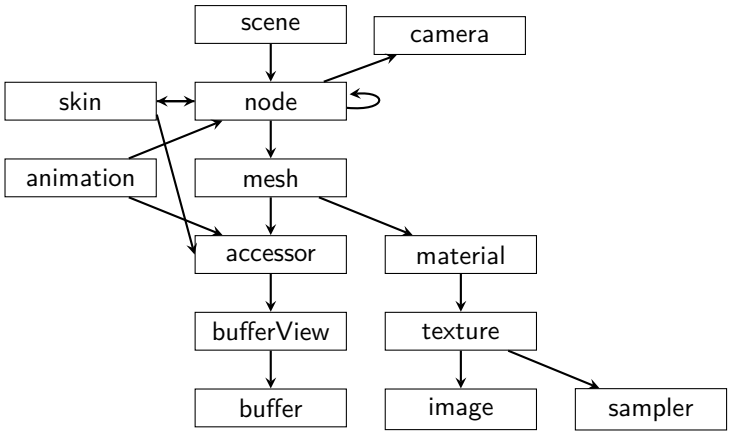
Wśród wymienionych formatów jedynie glTF jest w pełni otwarty i łatwo importowany, dlatego jest on dobrym wyborem jako format zasobów 3D używanych przez silnik graficzny. Na przykład Godot [4] wspiera glTF jako główny i rekomendowany format opisy sceny.

Zgodnie ze specyfikacją glTF [23] 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 obiecie 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 2.10: Relacje pomiędzy rożnymi typami elementów w formacie glTF

// TODO Opisz poszczególne typy elementów?

Przykładowo plik *triangle.gltf* opisujący scenę zawierającą pojedyńczy trójkątą z geometrią składającą się z wierzchołków pozycji zawartych w pliku *triangle.bin* wygląda następująco:

```
{
  "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 ]
  } ]
}
```

2.9.2. Zasoby wyjściowe

Zasoby wyjściowe mają formę plików w formacie przystosowanym do manipulacji przez aplikację. Przykładami zasobów wyjściowych są pliki z rozszerzeniami *.uasset* i *.umap* używane przez silnik Unreal Engine [5] do przechowywania zasobów w zoptymalizowanym formacie binarnym czy pliki *.streamdb* i

.resources, które według twórców narzędzia do ekstrakcji zasobów z gier UnArch [24] są używane w grze Doom Eternal i zastąpiły pliki *.pak* wcześniejszych gier od id Software.

2.9.3. 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.

2.9.4. Baza zasobów

Baza zasobów (ang. asset database) to zasób wyjściowy mający na celu zgromadzenie informacji o wszystkich zasobach używanych przez silnik. // HIRO

// TODO

2.10. Graf sceny

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

Wyemitowanie pojedyńczego 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 modelu:

- geometria: *vkCmdBindVertexBuffer()*, *vkCmdBindIndexBuffer()*;
- materiał: *vkCmdBindPipeline()*, *vkCmdBindDescriptorSets()*;
- globalne przekształcenie (macierz model-widok-rzutowanie): *vkCmdPushConstants()*, *vkCmdBindDescriptorSets()*.

Proces konwersji sceny na polecenia rysowania zależy od jej formatu. // HIRO lista vs hierarchiczne modelowanie

Hierarchiczne modelowanie to technika reprezentowania złożonych obiektów polegająca na podzieleniu modelu 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 potomne. Jeden z wezłów jest oznaczony jako korzeń. Emitowanie poleceń rysowania sprowadza się do rekurencyjengo przemierzania grafu sceny zaczynając od korzenia [25]. Pseudokod:

```
RenderujGrafSceny(węzeł):
  jeżeli węzeł.rodzic istnieje:
    węzeł.globalnaTransformacja = węzeł.rodzic.globalnaTransformacja * węzeł.lokalnaTransformacja
  w przeciwnym wypadku:
    węzeł.globalnaTransformacja = węzeł.lokalnaTransformacja
  WyemitujPolecenieRysowania(węzeł.geometria, węzeł.materiał, węzeł.globalnaTransformacja)
```

```
dla każdego węzła dziecko w węzeł.dzieci:
  RenderujGrafSceny(dziecko)
```

// HIRO diagram graf sceny -> polecenia rysowania

// HIRO wydajność, draw call optimization

2.11. Graf renderowania

// TODO

3. NARZĘDZIA, ARCHITEKTURA I IMPLEMENTACJA

3.1. 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.

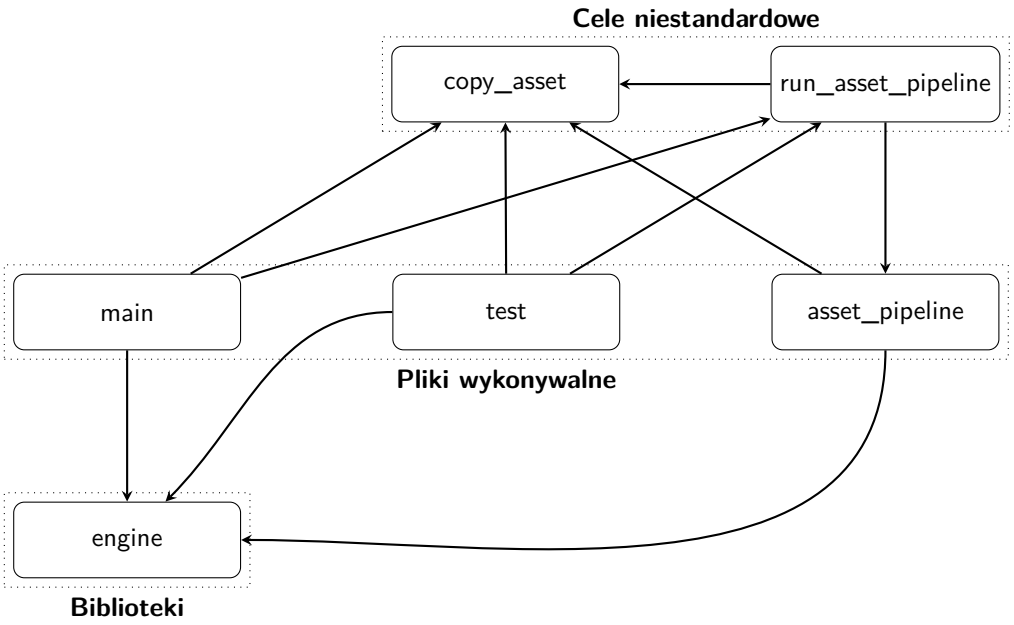
Skrypty w języku Python są obsługiwane pośrednio przez CMake, które wykrywa zainstalowany interpreter języka Python i używa go do stworzenia tzw. środowiska wirtualnego w tymczasowym katalogu venv/ w głównym katalogu projektu. Podczas procesu budowania środowisko wirtualne jest używane do zainstalowania 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 [26].

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.

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,
- *stb_image 2.27*: wczytywanie obrazów,
- *stb_truetype 1.26*: rasteryzacja tekstu czcionek,
- biblioteka standardowa języka C,
- API systemu operacyjnego: pliki nagłówkowe POSIX albo WinAPI,
- biblioteka standardowa języka 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

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

Silnik był rozwijany metodą *bottom-up* - jego pierwsza iteracja była pojedyńczym plikiem źródłowym wyświetlającym trójkąt [11], który w procesie dekompozycji i refaktoryzacji organicznie rozrósł się do 7 modułów znajdujących się w osobnych podkatalogach zawierających łącznie 9 skryptów Python *.py*, 65 nagłówków **.h* i 63 plików źródłowych **.c*.

Obecnie silnik składa się z następujących modułów: // TODO lista modułów, podkatalog i przeznaczenie

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

Pliki nagłówowe 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 plika 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.

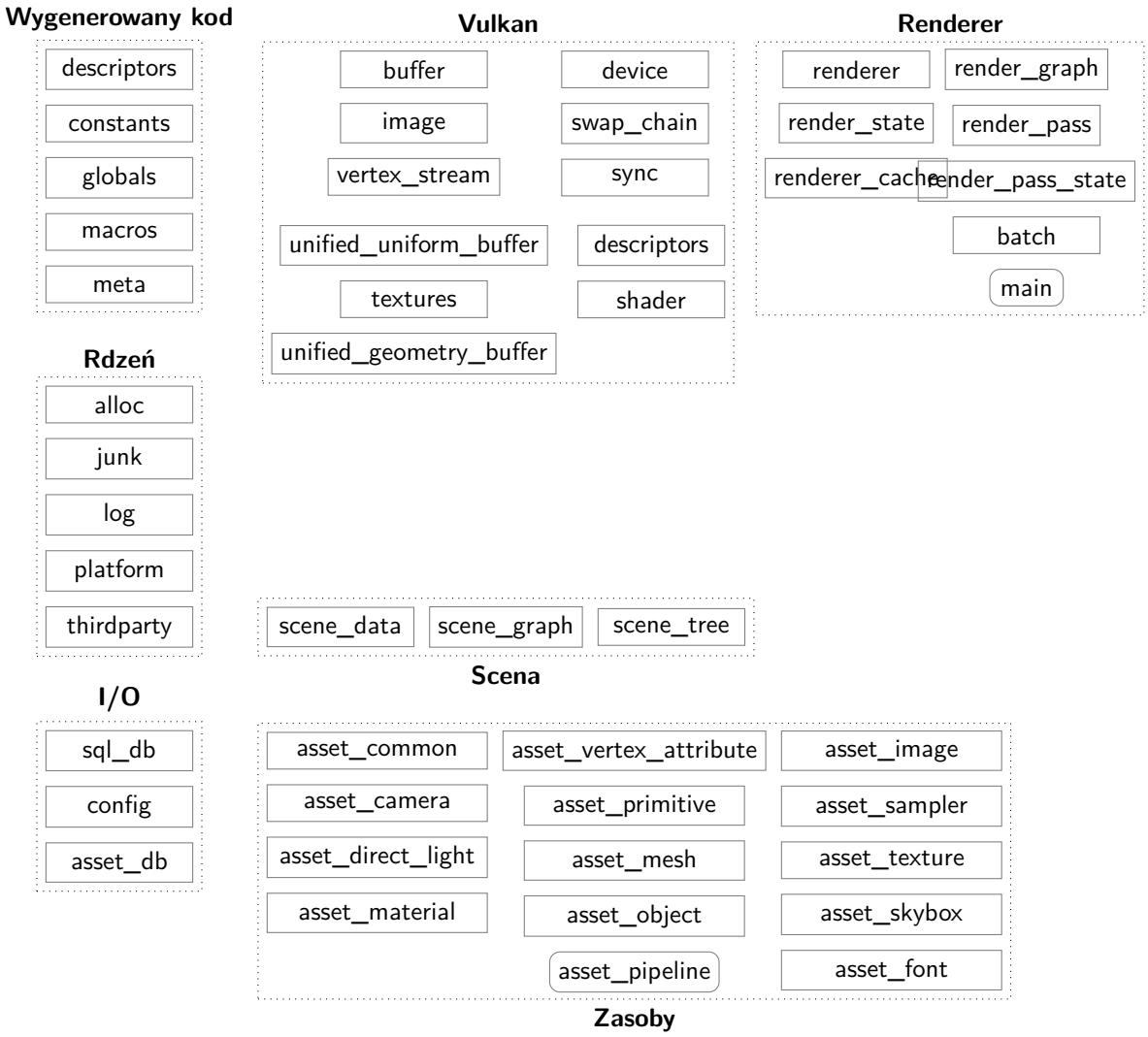
Struktury są zorganizowane w sposób obiektowy. Język C nie posiada wbudowanej koncepcji klasy, ale w projekcie przyjęto założenie, że dla klasy *struct* jej stan jest reprezentowany przez strukturę *struct*, która może posiadać metodę *func()*, jeśli istnieje funkcja *struct_func()* przyjmująca wskaźnik 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 dealokują instancję 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 wewnętrznym stanie instancji użyteczne podczas debugowania.

Relacje pomiędzy modułami silnika i ich najważniejszymi jednostkami są przedstawione na poniższym diagramie:

// TODO: Ładniejszy diagram, relacje.



Rysunek 3.2: Relacje pomiędzy modułam silnika i ich najważniejszymi klasami

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 kompilacji

właściwego kodu źródłowego biblioteki.

Język C nie posiada mechanizmów pozwalających na metaprogramowanie z wyjątkiem makr preprocessora, które mogą zaspokoić część potrzeb programisty chcącego przykładowo dodać nowy rodzaje pętli [27], 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* [28]) 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 liczbę 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*: $MAX_RENDER_TARGET_COUNT + 1 + 1$, 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 liczbę świateł kierunkowych na scenie;
- *MAX_POINT_LIGHT_COUNT*: 128, maksymalna liczbę świateł punktowych na scenie;
- *MAX_TEXT_CHARACTER_COUNT*: 256, maksymalną liczbę znaków w renderowanym ciągu znaków;
- *MIN_DELTA_TIME*: $(1.0/60.0)$, minimalny czas pomiędzy wywołaniami funkcji zwrotnej *update* w pętli głównej, domyślnie $\frac{1}{60}$ sekundy (60 FPS);
- *WORLD_UP*: $0, 1, 0$; wektor interpretowany jako "w górę"w przestrzeni świata.

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

X makro to przydatna technika preprocessora pozwalająca na pisanie kodu, który jest automatycznie aktualizowany po zmienie danych opisywanych przez X makro [29]. Przykładowo poniższa funkcja 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;
        }
}
```

Ten sam kod używający X makro:

```
#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;
        }
}
```

X makra ułatwiają utrzymywanie kodu poprzez deklaratywnego "jedynego źródła prawdy"i są intensywnie używane na wskroś silnika.

Jednostka globals

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

Silnik używa poniższej sekcji konfiguracji generacji do opisania ścieżkek dla kolejno 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.glsl
logFileName = log.txt
```

Powyższa konfiguracja generuje poniższą 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.glsl");
        globals.logFileName =
                get_executable_dir_file_path("", "log.txt");
}
```

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). Przykładowy fragment konfiguracji generatora opisujący konfiguracji globalnej:

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

Powyższa konfiguracja pozwala silnikowi na sparsowanie poniższego pliku INI:

```
[settings]
StartScene = MetalRoughSpheresNoTextures

[graphics]
WindowWidth = 1024

[controls]
Enabled = 1
```

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

```
[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"
```

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 poniższych funkcji:

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

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

Funkcja **_debug_str()* jest generowana tylko wtedy, jeśli literały 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.

Przykłady powyższych struktur:

```
// łstaa 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;

// łstaa push 'global'
typedef struct global_uniform_buffer_struct {
```

```
        mat4 viewMat;
        mat4 projMat;
        ...
        offscreen_texture_helper_struct offscreenTextures;
} global_uniform_buffer_struct;
```

Układ pamięci struktur zdefiniowanych w języku C nie są koniecznie kompatybilne układem pamięci wymaganymi 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. 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");
}
```

Generacja jest kończona X makraami wyliczającymi nazwy wszystkich sparsowanych rodzin struktur.

Dzięki automatycznej generacji kodu modyfikacja sposobu organizacji pamięci GPU buforów sprowadza 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* [30].

Wygenerowane struktury, funkcje i X makra są używane podczas kopiowania 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 pomocniczych 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 sę z plikach źródłowych jest uzyskiwany poprzez ponowne dołączenie nagłówka przy użyciu dyrektywy *#include* po wcześniejszym zdefiniowaniu odpowiedniego symbolu preprocesora. Przykładowo biblioteka *cgltf* wymaga ponownego dołączenia nagłówka w następujący sposób:

```
#define CGLTF_IMPLEMENTATION
#include "cgltf.h"
```

Jednostka alloc

Funkcje pomocnicze wspomagające zarządzanie pamięcią CPU, co objemuje 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).

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 bibliteki *uthash*.

Jednostka log

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

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 zaostała zdefiniowana w wygenrowanych zmiennych (domyślnie *log.txt*).

Logowanie komuniaktu odbywa się poprzez grupę funkcji *log_*()*, 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.

Przykładowy kod demonstrujący logowanie:

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

Powyższy kod powinien zapisać do pliku *log.txt* w katalogu z plikiem wykonywalnym komunikaty podobne do poniższych:

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

Jednostka junk

Proste funkcje i makra które mogą być potencjalnie używane na wskroś wszystkich modułów w całej bibliotece, ale nie zostały uznane za wystarczająca skomplikowane, aby uzasadnić wydzielenia do osobnej jednostki.

Jednostka definiuje stałe preprocesora *PLATFORM_** 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
```

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

Funkcja *count_bits()* zlicza bity w liczbie całkowitej używając metody Briana Kernighana [31]. Przykładem użycia jest określenie liczby flag ustawionych w wyliczeniu.

Makra *HASH_** ukrywają detale użycie funkcji skrótu biblioteki *xxHash*:

```
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);
```

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 nieosiągnalności przypadku domyślny instrukcji switch bądź bloku else informuje o kompletności sprawdzanych warunków:

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

Makro pomocnicze *MACRO_FOREACH()* jest używane w X makrach. // TODO więcej

Jednostka definiuje też makra używające formy metaprogramowania w celu dodania nowych struktur kontrolnych [27] upraszczających iterowanie po strukturach danych bibliteki *uthash*:

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

Jednostka platform

Głowna 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 przeciwieństwie 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 tekstory 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 kopiowane przez potok zasobów do katalogu budowania stając się zasobami wyjściowymi. Reszta zasobów wejściowych staje się częścią bazy zasobów będącej plikiem bazy danych SQLite.

SQLite [32] 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 wydajności, bogata wewnętrzna struktura oferowana przez relacyjną bazę danych i formę łatwego do dystrybucji pojedyńczego samodzielnego pliku na dysku [33].

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 *config* reprezentujący pojedyńczy 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 klucze 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. Przykładowo poniższy funkcja kod wybiera wartość FLOAT z tabeli *directLight*, kolumny *intensity* i klucza *key*:

```
float value =
        asset_db_select_directLight_intensity_float(assetDb, key).value;
```

Analogicznie wygląda umieszczenie nowej wartości:

```
asset_db_insert_directLight_intensity_float(assetDb, key,
        data_float_temp(value));
```

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

3.3.4. Zasoby

Moduł zawierajaca 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 współdzielą 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()*.

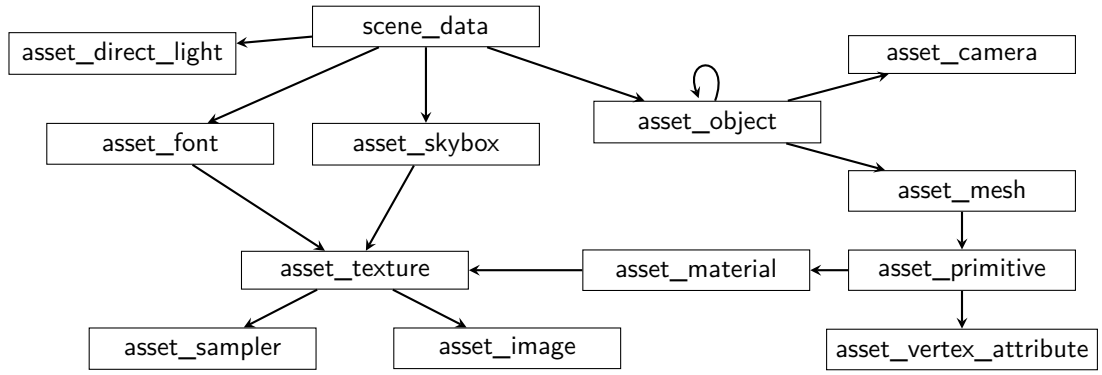
Powyższy interfejs jest zdefiniowany przez makra w jednostce *common* i jest używany w reszcie jednostek modułu do implementacji obiektów zasobów.

Wspieranych jest następujących 12 rodzajów 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 poniższym diagramie:



Rysunek 3.3: Relacje pomiędzy obiektami zasobów w silniku

Zasób węzła *asset_object*

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

Węzeł zawiera macierz 4x4 z lokalną transformację 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 zawiarać 4 identyczne koła renderowanych 4 razy tym samym prymitywem. Siatka jest tożsama z geometrią pojedyńczego modelu przygotowanego w programie do modelowania 3D.

Zasób siatki *asset_primitive*

Zasób prymitywu *asset_primitive* reprezentuje część siatku 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 rozdzielanie 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 pojedyńczego 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 $+Z$.

Powyższe 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 świateł:

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

Wszystkie rodzaje świateł 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*

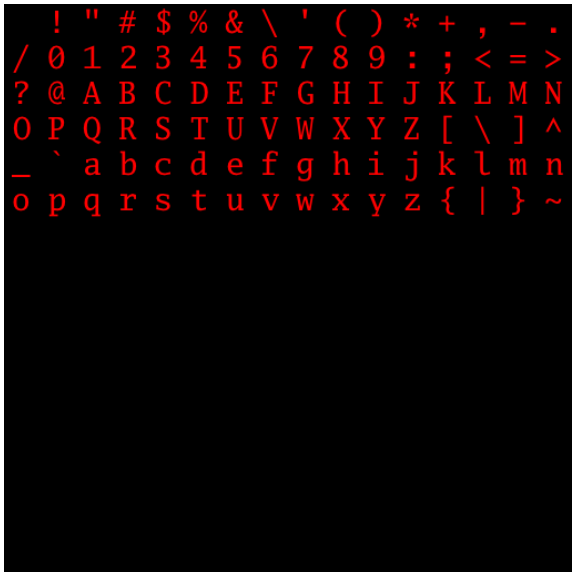
Zasób skybox *asset_skybox* opisuje składa się z zasobu tekstury oraz nazwy używanej przez konfigurację globalną.

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 tekstury: sk,
- alfabet: ciąg znaków ASCII,
- rozmiar znaku: rozmiar jednego glifu w pikselach (uint32_t).

Poniższy obraz przedstawia przykładową teksturę dla czcionki Go-Mono [34]:



Rysunek 3.4: Przykładowa tekstura dla czcionki Go-Mono [34]

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 tekstury *asset_texture*

Zasób tekstury *asset_texture* reprezentuje próbkowalny 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 pomniejszający (VkFilter),
- *minFilter*: filtr powiększający (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:

- narzędzia 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 poniższy potok zasobów wywołuje narzędzie 5 razy 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"
```

3.3.5. Vulkan

// TODO Vulkan to moduł zawierajacy

Jednostka shader

Obiekt *shader* reprezentuje pojedyńczy shader i jest odpowiedzialny za kompilację ich do formy używalnej przez Vulkan.

Obiekt składa się z następujących elementów:

- typ shadera,
- kod źródłowy GLSL,
- kod bajtowy SPIR-V,
- moduł shadera,
- 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 must być znany podczas tworzenia - jest on uzyskiwany poprzez użycie obiektu *shader_generator*.

Kod bajtowy SPIR-V jest uzyskiwany poprzez kompilację kodu źródłowego GLSL biblioteką *shaderc*. Jej użycie ilustruje poniższy kod:

```
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)
```

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

Jednostka 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.

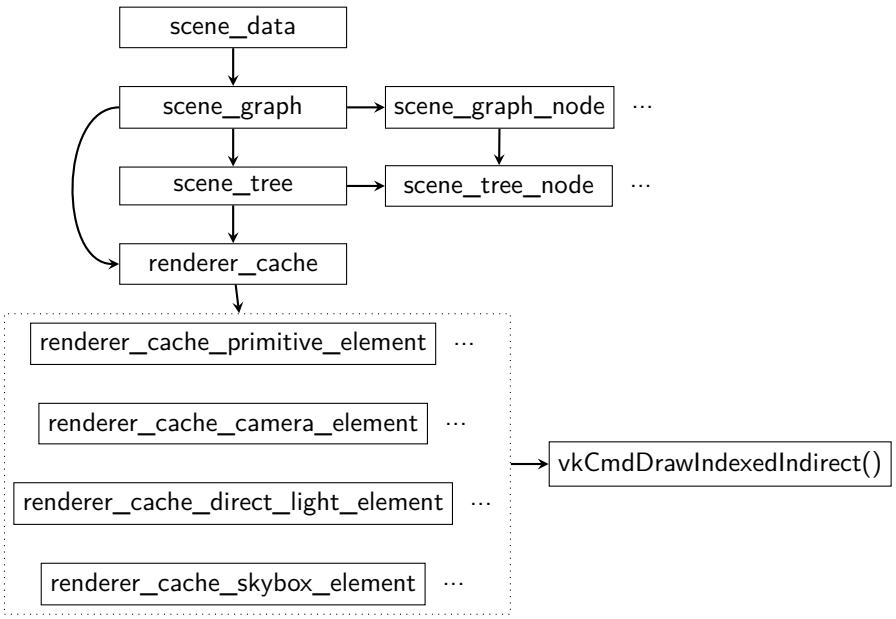
3.3.6. Scena

Moduł sceny jest odpowiedzialny za konwersję sceny z wysokopoziomowej formy używanej przez bazę zasobów do niskopoziomowej formy łatwo używalnej przez renderer do wyemitowania polecenia rysowania pośredniego.

Implementacja została zainspirowana techniką modelowania hierarchicznego opracowana w firmie Nvidia [35].

Scena jest opisywana przy pomocy grafu sceny, którego węzeł zawieraja zasób prymitywu, lokalną trasformację przestrzeni świata oraz węzły potomne. W tradycyjnym podejściu do modelowania hierarchicznego znajdują się w przy użyciu grafu sceny // HIRO

Obiekty biorące udział w procesie konwersji sceny wraz z przepływem danych są przedstawione na poniższym diagramie:



Rysunek 3.5: Obiekty biorące udział w procesie konwersji sceny wraz z przepływem danych

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 on dla każdego typu obiektu zasobu osobną listę dwukierunkową zawierającą wszystkie obiekty używane przez scenę oraz 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 wywoływane 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

Obiekt *scene_graph* reprezentuje // HIRO

Drzewo sceny scene_tree

Obiekt *scene_tree* reprezentuje // HIRO

3.3.7. Renderer

// TODO Renderer to moduł zawierajacy
// HIRO renderer_cache
// HIRO Stan renderowania renderer_state
// HIRO Graf renderowania render_graph
// HIRO renderer

4. BADANIA

// TODO: Prezentacja renderdoc // TODO: Profiling różne sceny. // TODO: test: technika filtrowanie anizotropowe, usuń TRANSIENT // TODO: test: usuń mipmapy, usuń multidraw

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] LunarG, *Vulkan SDK*, 2022. adr.: https://www.lunarg.com/vulkan-sdk/.

[7] 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/.

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

[9] K. Group, *SPIR-V Specification*, 2022. adr.: https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html.

[10] J. Barczak, *OpenGL Is Broken*, 2022. adr.: http://www.joshbarczak.com/blog/?p=154.

[11] A. Overvoorde, *Vulkan SDK*, 2022. adr.: https://vulkan-tutorial.com/.

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

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

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

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

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

[17] Baldur Karlsson, *Debugger graficzny RenderDoc*, 2022. adr.: https://renderdoc.org/.

[18] K. Group, *glTF Sample Models*, 2022. adr.: https://github.com/KhronosGroup/glTF-Sample-Models.

[19] J. Ekstrand, *Descriptors are hard*, 2022. adr.: https://www.jlekstrand.net/jason/blog/2022/08/descriptors-are-hard/.

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

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

[22] Blender Foundation, *Modeler 3D Blender*, 2022. adr.: https://www.blender.org/.

[23] K. Group, *glTF 2.0 Specification*, 2022. adr.: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html.

[24] ArwgLacyProgramming, *Narzędzie do ekstrakcji archiw gier UnArch*, 2022. adr.: www.alprogramming.com.

[25] 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.

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

[27] S. Tatham, *Metaprogramming custom control structures in C*, 2022. adr.: https://www.chiark.greenend.org.uk/~sgtatham/mp/.

[28] *configparser — Configuration file parser*, 2022. adr.: https://docs.python.org/3/library/configparser.html.

[29] W. Bright, *The X Macro*, 2010. adr.: https://digitalmars.com/articles/b51.html.

[30] J. E. E. Mansouri, *Rendering 'Rainbow Six | Siege'*, 2018. adr.: https://www.gdcvault.com/play/1022990/Rendering-Rainbow-Six-Siege.

[31] S. E. Anderson, *Bit Twiddling Hacks*, 2022. adr.: http://graphics.stanford.edu/~seander/bithacks.html.

[32] D. Richard Hipp, *Baza danych SQLite*, 2022. adr.: https://www.sqlite.org.

[33] *SQLite As An Application File Format*, 2022. adr.: https://www.sqlite.org/appfileformat.html.

[34] R. P. Nigel Tao Chuck Bigelow, *Go fonts*, 2022. adr.: https://go.dev/blog/go-fonts.

[35] C. K. Markus Tavenrath, *Advanced Scenegraph Rendering Pipeline*, 2013. adr.: https://on-demand.gputechconf.com/gtc/2013/presentations/S3032-Advanced-Scenegraph-Rendering-Pipeline.pdf.

SPIS RYSUNKÓW

SPIS TABLIC