VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

EFEKTÍVNE C++ ROZHRANIE PRE API VULKAN

EFFECTIVE C++ BINDING FOR VULKAN API

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **ADAM RUŽA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JAN PEČIVA, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

147856

Ústav:	Ústav počítačové grafiky a multimédií (UPGM)
Student:	**Ruža Adam**
Program:	Informační technologie
Specializace:	Informační technologie
Název:	**Efektivní C++ rozhraní pro API Vulkan**
Kategorie:	Počítačová grafika
Akademický rok:	2022/23

Zadání:

1. Nastudujte si teorii k jazyku C++ potřebnou pro návrh efektivního rozhraní pro API Vulkan, samotné API Vulkan a postupy v současné době používané pro generování Vulkan C a C++ rozhraní. Vzhledem k rozsáhlosti Vulkan C++ rozhraní Vulkan-HPP a jeho dlouhou dobu kompilace považujte za hlavní kritérium efektivity rychlost kompilace na různých kompilátorech.
2. Využijte projekt vkcpp-gen z předchozí projektové praxe pro generování C++ rozhraní pro Vulkan v různých konfiguracích a s různou funkcionalitou. Navrhněte systém pro měření efektivity vygenerovaného rozhraní. Navrhněte rozšíření projektu vkcpp-gen pro potřeby tohoto projektu.
3. Implementujte systém měření a požadovaná rozšíření projektu vkcpp-gen. Proveďte detailní analýzu efektivity vygenerovaných rozhraní vzhledem k době kompilace. Volbu různých konfigurací a funkcionality testovaného rozhraní konzultujte s vedoucím.
4. Na základě předchozích zkušeností a po domluvě s vedoucím vyberte jednu konfiguraci vygenerovaného Vulkan rozhraní a demonstrujte jeho použitelnost na jednoduché aplikaci.
5. Vyhodnoťte výsledky projektu. Diskutujte možné budoucí směry vývoje projektu. Projekt prezentujte na internetu.

Literatura:

- dle doporučení vedoucího

Při obhajobě semestrální části projektu je požadováno:
Prototyp funkčního testovacího systému s prvními výsledky.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Pečiva Jan, Ing., Ph.D.**
Vedoucí ústavu:	Černocký Jan, prof. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	10.5.2023
Datum schválení:	17.2.2023

Abstrakt

Mnoho aplikácií pre 3D grafiku využívajúce Vulkan sú vyvíjané v jazyku C++. Problém súčasného rozhrania pre C++ je náročná kompilácia. Cieľom tejto práce je vytvoriť C++ rozhranie pre Vulkan s rýchlejšou kompiláciou. Bol implementovaný vlastný nástroj pre generáciu rozhrania a systém pre meranie. Nový generátor umožňuje prispôsobenie rozhrania na mieru. Kompilácia Vulkan rozhrania bola odmeraná z pohľadu doby kompilácie. Redukciou nepotrebných častí v rozhraní pre špecifickú aplikáciu bolo dosiahnuté zrýchlenie kompilácie rozhrania o 50% až 60%. Výsledky tejto práce umožňujú použiť C++ Vulkan rozhranie poskytujúce lepšiu dobu kompilácie.

Abstract

Many 3D graphics applications are developed using Vulkan in C++ language. The major drawback of Vulkan C++ API is slow compilation time. Aim of this thesis is to create Vulkan C++ API with improved compilation time. A custom API generator was implemented for this purpose. This generator allows advanced customization of Vulkan C++ API. Using automated script, compilation times were measured in detail. By reducing unneccesary parts of code for a specific application, we measured improvement of about 50% to 60% in terms of Vulkan C++ API comilation time.

Kľúčové slová

počítačová grafika, C++, C++20, Khronos, Vulkan, API, moduly

Keywords

computer graphics, C++, C++20, Khronos, Vulkan, API, modules

Citácia

RUŽA, Adam. *Efektívne C++ rozhranie pre API Vulkan.* Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pečiva, Ph.D.

Efektívne C++ rozhranie pre API Vulkan

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Jana Pečivy Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Adam Ruža
5. mája 2023

Poďakovanie

Rád by som poďakoval pánu Ing. Janu Pečivovi, Ph.D., ktorý mi poskytol pôvodný námet k vytvoreniu tejto práce a v priebehu tvorby poskytoval odborné konzultácie a cennú zpätnú väzbu.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

V dnešnej dobe sú grafické karty dôležitou súčasťou počítačov. Poskytujú robustnú sadu funkcií nielen pre počítačovú grafiku, ale aj rozličné výpočetné úlohy. Ak programátor potrebuje vyvinúť aplikáciu využívajúcu grafickú kartu, musí použiť rozhranie, ktoré zabezpečuje komunikáciu s grafickou kartou. Aktuálne ich existuje celá rada: OpenGL[1], DirectX[2], Metal[3] a Vulkan[4].

Najmladší z týchto rozhraní je Vulkan, navrhnutý na nízkej úrovni pre maximálne využitie prostriedkov. Preto často programátor zvolí programovací jazyk C++ s ohľadom na rýchlosť. Množstvo herných štúdií zvolí Vulkan pre vývoj digitálnych hier.

Vulkan je vyvinutý v programovacom jazyku C[12] a má výbornú podporu na moderných grafických kartách, zvlášť od firmy NVIDIA. Hardware je navrhovaný s dôrazom na Vulkan, najnovšie technológie, ako napríklad Ray Tracing[5] a DLSS[5] – Deep Learning Super Sampling, sú podporované pre vývojárov čo najskoršie a majú dostupnú sadu nástrojov pre optimalizáciu a uľahčenie vývoja[2]. Vulkan je preto často zvolený pre náročné projekty.

Táto práca je zameraná na API – aplikačné programovacie rozhranie pre jazyk C++[15] (ďalej v kontexte práce *rozhranie*) pre Vulkan. Kvalitné a dobré rozhranie je veľmi dôležité. Uľahčí programátorovi prácu. Oproti rozhraniu pre jazyk C prináša určité výhody – prehľadnejší a bezpečnejší[17] kód – čo môžu byť pádne dôvody voľby rozhrania. Rovnako ako jazyk C++, Vulkan rozhranie pre jazyk C++ je veľmi komplexné. Hlavný problém je podstatne dlhšia doba kompilácie. Motivácia tejto práce je preto vytvorenie rozhrania pre jazyk C++ s nižšou dobou kompilácie.

Nové rozhranie bude navrhnuté predovšetkým pre moderný štandard jazyka C++. Štandard C++20[6] priniesol za posledné roky najväčšie množstvo zmien[4]. Nový spôsob členenia projektu do modulov teoreticky zrýchli dobu kompilácie[3], čo je v súlade s cieľom práce. Preto sa pokúsime vytvoriť variantu nového rozhrania s podporou modulov a overiť, či naozaj zrýchli kompiláciu.

V tejto práci vykonáme detailnú analýzu doby kompilácie Vulkan rozhrania v jazyku C++.

Optimalizácia rýchlosti kompilácie je zložitá úloha, pretože na túto tému existuje malé množstvo odbornej literatúry. Väčšina techník je zameraná na celkový projekt, a preto nie

[1] https://www.opengl.org/
[2] https://learn.microsoft.com/en-us/windows/win32/directx
[3] https://developer.apple.com/documentation/metal/
[4] https://www.vulkan.org/
[5] https://developer.nvidia.com/rtx/dlss
[6] https://en.cppreference.com/w/cpp/20

je aplikovateľná na samotné Vulkan rozhranie pre jazyk C++. Napriek tomu sa pokúsime odhaliť pomalé časti a nájsť alternatívne riešenia. Neexistuje definitívny spôsob ako odhadnúť rýchlosť kompilácie určitého kódu, pretože kompiláciu ovplyvňuje mnoho faktorov: konkrétny kompilátor a linker, operačný systém, samotný hardware.

Veľká časť práce bude venovaná vlastnej implementácií nástroja pre automatické generovanie rozhrania v jazyku C++. Vytvoríme systém pre meranie doby kompilácie, z ktorého získame výsledky. Merané budú nové varianty rozhrania a porovnané oproti súčasnému rozhraniu pre jazyk C++.

Kapitola 2

Súčasný ekosystém Vulkan

Ekosystém Vulkan bol založený v roku 2015 neziskovým konzorciom Khronos Group[1] skladajúcim sa zo 170 organizácií, medzi ktoré patrí napríklad AMD, Apple, Google, Intel, NVIDIA, Qualcomm, Sony a Valve. Celý ekosystém je rozsiahly, Vulkan rozhranie je len jedna zo súčastí.

Vulkan rozhranie pre jazyk C doposiaľ prešlo štyrmi revíziami:

- 2015: verzia 1.0
- 2016: verzia 1.1
- 2020: verzia 1.2
- 2022: verzia 1.3

Práca aktuálne používa verziu 1.3.239.0.

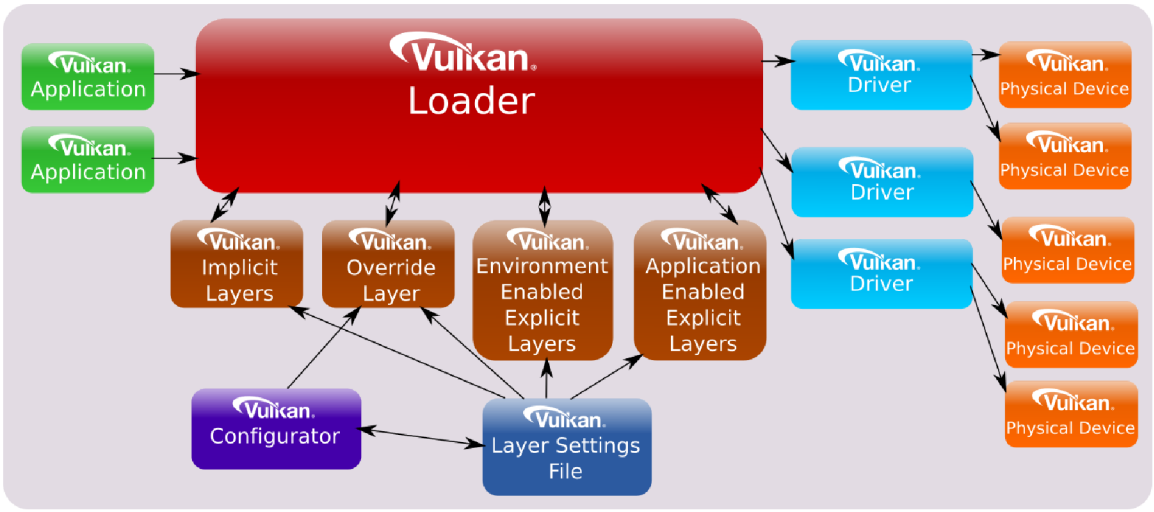
Súčasný ekosystém Vulkan sa skladá z niekoľkých voľne dostupných (open source) projektov:

- Vulkan-Loader[2]: popísaný v sekcii 2.1
- Vulkan-ValidationLayers [3]: implementácia validačných vrstiev, mimo rozsah tejto práce
- Vulkan-Tools[4]: dodatočné nástroje, mimo rozsah tejto práce
- Vulkan-Headers[5]: samotné rozhranie pre Vulkan
- Vulkan-Hpp[6]: popísaný v sekcii 2.2
- Vulkan-Profiles[7]: mimo rozsah tejto práce

[1] https://www.khronos.org/
[2] https://github.com/KhronosGroup/Vulkan-Loader
[3] https://github.com/KhronosGroup/Vulkan-ValidationLayers
[4] https://github.com/KhronosGroup/Vulkan-Tools
[5] https://github.com/KhronosGroup/Vulkan-Headers
[6] https://github.com/KhronosGroup/Vulkan-Hpp
[7] https://github.com/KhronosGroup/Vulkan-Profiles

V roku 2022 bol vydaný Vulkan SC pre kritické systémy, ktorý nebudeme uvažovať.

Kompletná dokumentácia je dostupná na stránke organizácie LunarG [8], vrátane špecifikácie[9] Vulkan rozhrania.



Obr. 2.1: Architektúra Vulkan. Prevzaté z [19].

Vulkan je viac vrstvová architektúra (pozri obrázok 2.1).

Vulkan Application – Aplikácia – je klientská časť kódu v jazyku C++ (prípadne v jazyku C, ktorý ale neuvažujeme). Predovšetkým sa jedná o grafické aplikácie.

Vulkan loader je knižnica. Komunikuje s nižšími vrstvami (pre nás nie sú veľmi dôležité) a zaisťuje zvyšok práce. Medzi aplikáciou a knižnicou existuje práve spomínané Vulkan rozhranie. Z pohľadu ABI – Application Binary Interface[10] – musí byť rozhranie kompatibilné s jazykom C.

2.1 Projekt Vulkan-Loader

Projekt Vulkan-Loader možno považovať za jadro systému. Stará sa o prácu s rozličnými vrstvami a radičmi ICD – Installable Client Driver – pre grafické karty.

Vulkan-Loader poskytuje dve funkcie pre získanie funkčných adries. Cez tieto funkcie je možné získať adresu funkcie, ktorá tvorí rozhranie, spôsobom nezávislým na platforme. Pritom Vulkan-Loader zaistí správne vloženie vrstiev[19], bližšie popísané na obrázkoch 2.2 a 2.3

Objektový model

Všetky objekty sú vo Vulkan rozhraní pre jazyk C reprezentované ako tzv. *handle*. Ak funkcia Vulkan rozhrania pracuje s objektom, tak je daný objekt uvedený ako prvý parameter. Vulkan objekty sú rozdelené do dvoch úrovní.

[8] https://vulkan.lunarg.com/doc/sdk
[9] https://vulkan.lunarg.com/doc/sdk/1.3.243.0/linux/1.3-extensions/vkspec.html
[10] https://itanium-cxx-abi.github.io/cxx-abi/

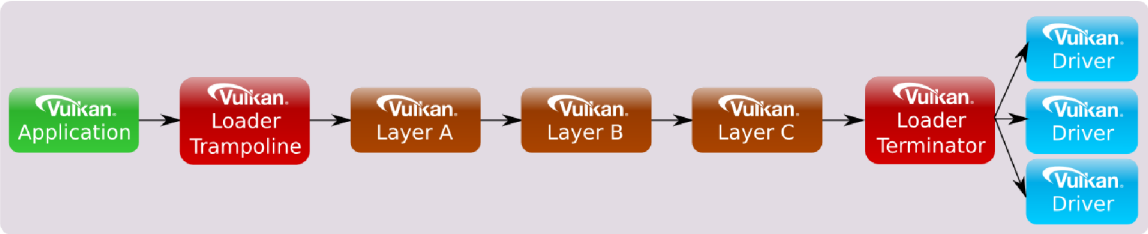
Objekty úrovne Instance

Vulkan nemá globálny stav. Objekt Instance reprezentuje stav aplikácie. Program môže pracovať s viac ako jednou inštanciou, avšak zvyčajne stačí jedna.

Určitá podmnožina objektov je vlastnená Instance objektom. Sem patrí napríklad objekt `VkPhysicalDevice`, ktorý reprezentuje fyzické zariadenie.

Funkcie úrovne Instance

Pomocou funkcie `vkGetInstanceProcAddr`[11] je možné získať akúkoľvek funkciu.



Obr. 2.2: Reťazec volaní úrovne Instance. Prevzaté z [18].

V tomto prípade je v reťazci prítomný Loader Terminator (pozri obrázok 2.2), ktorý zaistí agregáciu radičov. Vulkan Loader si pre Instance objekt udržiava tabuľku funkčných ukazovateľov.

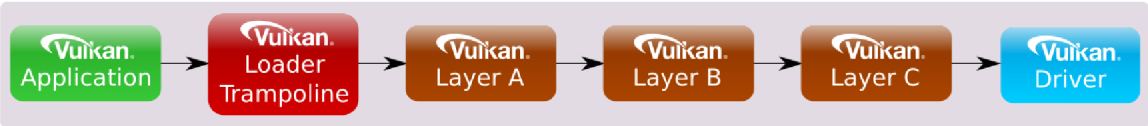
Objekty úrovne Device

Objekt Device je logické zariadenie spojené s konkrétnym fyzickým zariadením.

Určitá podmnožina objektov je vlastnená Device objektom.

Funkcie úrovne Device

Pomocou funkcie `vkGetDeviceProcAddr`[12] je možné získať funkciu Device úrovne pre špecifické zariadenie.



Obr. 2.3: Reťazec volaní úrovne Device. Prevzaté z [18].

Výhodou je rýchlejší reťazec volaní (pozri obrázok 2.3) oproti funkcii získanej z `vkGetDeviceProcAddr`. Vulkan Loader si pre Device objekt udržiava tabuľku funkčných ukazovateľov.

[11] https://registry.khronos.org/vulkan/specs/1.3-extensions/html/chap4.html#vkGetInstanceProcAddr
[12] https://registry.khronos.org/vulkan/specs/1.3-extensions/html/chap4.html#vkGetDeviceProcAddr

Linkovanie

Vulkan Loader je distribuovaný ako dynamická knižnica na významných platformách (samozrejme aj na ďalších, no sú mimo rozsah práce):

- Windows: **vulkan-1.dll**
- Linux: **libvulkan.so.1**
- macOS: **libvulkan.1.dylib**

Priame linkovanie

Aplikácia môže byť linkovaná s knižnicou dynamicky. V tomto prípade používajú volania Vulkan funkcií internú tabuľku funkčných ukazovateľov v knižnici Vulkan.

Statické linkovanie bolo zrušené z dôvodu niekoľkých problémov[18].

Nepriame linkovanie

Aplikácia nemusí byť linkovaná s knižnicou, ale za behu načíta symboly použitím mechanizmov závislých na platforme.

Obr. 2.4: Optimalizovaný reťazec volaní úrovne Device. Prevzaté z [18].

Veľké množstvo funkcií nemusí vstupovať cez Vulkan Trampoline vrstvu, čo vedie k vyššiemu výkonu (pozri obrázok 2.4).

Najrýchlejšia možnosť

Pre najrýchlejší výkon aplikácie je odporúčané použiť nepriame linkovanie [18]. V tomto prípade si aplikácia potrebuje udržiavať vlastné tabuľky funkčných ukazovateľov.

2.2 Projekt Vulkan-Hpp

Projekt Vulkan-Hpp vznikol ako rozšírenie rozhrania pre jazyk C++. Bol začlenený do ekosystému vo verzii 1.0.24. Oproti rozhraniu pre jazyk C poskytuje silnejšiu kontrolu chýb počas kompilácie [17] a integráciu s C++ funkcionalitou ako sú triedy, referencie[13], výnimky[14] a STL kontajnery[15].

Tento projekt sa zaoberá generáciou rozhrania (hlavičkové súbory jazyka C a C++). Samotné rozhranie je tiež dostupné v projekte **Vulkan-Headers**.

[13] https://en.cppreference.com/w/cpp/language/reference
[14] https://en.cppreference.com/w/cpp/language/exceptions
[15] https://en.cppreference.com/w/cpp/container

2.3 Vulkan register

Kompletné rozhranie Vulkan je špecifikované pre syntax C99 jazyka C. Špecifikácia je distribuovaná v súbore **vk.xml** formátu XML (v kontexte práce register). Celý súbor je veľmi rozsiahly, v momentálnej verzii má cez 23000 riadkov. Orientačne popíšeme výskyt dôležitých prvkov, ktoré sú pre prácu relevantné.

Štruktúra koreňa vyzerá následovne:

```
<?xml version="1.0" encoding="UTF-8"?>
<registry>
  <platforms>...</platforms>
  <tags>...</tags>
  <types comment="Vulkan type definitions">...</types>
  <enums name="API Constants">...</enums>
  ...
  <enums name=... type="enum">...</enums>
  ...
  <commands>...<commands>
  <feature api="vulkan" name="VK_VERSION_1_0" number="1.0">...</feature>
  <feature api="vulkan" name="VK_VERSION_1_1" number="1.1">...</feature>
  <feature api="vulkan" name="VK_VERSION_1_2" number="1.2">...</feature>
  <feature api="vulkan" name="VK_VERSION_1_3" number="1.3">...</feature>
  <extensions comment="Vulkan extension interface definitions">
  ...
  </extensions>
  <formats>...</formats>
</registry>
```

Uzol platforms

Obsahuje pole uzlov.

```
<platform name="..." protect="..."/>
```

Z týchto uzlov si potrebujeme uložiť informáciu o *názve* a reťazci *protect*. Výskyt časti kódu, ktorá patrí pod špecifickú platformu je chránená direktívom.

```
#if defined( ... )
#endif
```

Uzol tags

Obsahuje pole uzlov.

```
<tag name="..."/>
```

Každý uzol predstavuje príponu. Názvy určitých prvkov môžu mať práve jednu z prípon.

Uzol types

Obsahuje pole uzlov, kde atribút *category* určuje o aký typ sa jedná.

Uzol pre Vulkan objekt obsahuje názov a voliteľne rodiča.

```
<type category="handle" parent="...">
  <type>...</type><name>...</name>
</type>
```

Uzol pre Vulkan enumeráciu je typu enums alebo bitmask. Obsahuje iba názov, hodnoty sú uvedené v uzle Enums.

```
<type name="..." category="enum"/>
<type bitvalues="..." category="bitmask">
  <type>...</type> <name>.../name>
</type>
<type category="bitmask" name="..." alias="..."/>
```

Uzol kategórie funcpointer obsahuje definíciu PFN funkcie, ktoré sú popísané v 2.4.

```
<type category="funcpointer">...<name>...</name>)...</type>
```

Uzol pre štruktúru sa skladá z poduzlov member, ktoré popisujú jednotivé položky:

```
<type category="struct" name="...">
  <member>...</member>
</type>
<type category="union" name="...">
  <member>...</member>
</type>
```

Uzol enums

Koreň obsahuje ľubovoľné množstvo uzlov enums, kde každý predstavuje definíciu dátového typu enumerácia. Skladá sa z poduzlov:

```
<enum value="..." name="..."/>
```

Uzol commands

Obsahuje pole uzlov, kde každý uzol prestavuje definíciu funkcie, označenú ako *Vulkan príkaz*.

```
<command successcodes="..." errorcodes="..."/>
```

Uzol feature

Určuje množinu prvkov, ktoré sú zahrnuté pod určitú verziu.

2.4 Rozhranie pre jazyk C

Rozhranie pre jazyk C je automaticky generované z registra. Programovací jazyk C nemá menný priestor, preto má Vulkan definované pravidlá pre názvoslovie ako implicitný menný priestor [10]. Celé rozhranie je pomyselne rozdelené na niekoľko častí.

Definície preprocesora

Názvy začínajú predponou `VK_` [9]. Často používané je makro *VK_HEADER_VERSION* určujúce verziu rozhrania.

Definície enumerácií

Názvy enumerácií[16] začínajú predponou `Vk` [11].

```
typedef enum ... {
}
```

Hodnoty enumerácií majú veľké písmená a *snake_case* štýl. Začínajú predponou `VK_`, nasleduje nepovinná predpona názvu a zvyšok identifikátoru [7]. Napríklad *VK_STRUCTURE_TYPE_APPLICATION_INFO*.

Vo Vulkan rozhraní nájdeme dve kategórie enumerácií. Klasická enumerácia obsahuje vždy jednu konkrétnu hodnotu z množiny zadefinovaných hodnôt. Enumerácie s príponou *FlagBits* vyjadrujú pole individuálnych bitov. Hodnota môže byť kombinácia z množiny zadefinovaných hodnôt. Dátový typ *VkFlags* je 32 bitové číslo.

```
typedef enum ...FlagBits {

} ...FlagBits;
typedef VkFlags ...Flags;
```

Definície ukazovateľov na funkcie

Názvy ukazovateľov na funkcie začínajú predponou `PFN_` (Pointer to Function) [8].

```
typedef ... (VKAPI_PTR *PFN_...)(...);
```

Napríklad *PFN_vkCreateInstance*. Vyskytujú sa v RAII rozhraní pre jazyk C++.

Deklarácie Vulkan príkazov

Vulkan príkazy začínajú predponou `vk` [6].

```
VKAPI_ATTR ... VKAPI_CALL vk...(...);
```

Dajú sa vypnúť pomocou direktíva *VK_NO_PROTOTYPES*.

Definície Vulkan štruktúr

Vulkan štruktúry[17] začínajú predponou predponu `Vk` [11]. Až na pár výnimiek vždy obsahujú člen sType, ktorý musí byť programátorom nastavený na správnu hodnotu, a člen pNext umožňujúci reťazenie štruktúr, pričom povolené kombinácie sú uvedené v špecifikácií. Ak programátor nastaví nesprávne hodnoty môže nastať nepredvídateľná chyba za behu programu.

```
typedef struct ... {
  VkStructureType       sType;
  const void*           pNext;
} ...;
```

[16] https://en.cppreference.com/w/c/language/enum
[17] https://en.cppreference.com/w/c/language/struct

Do tejto časti spadajú aj dátové typy union[18].

```
typedef union... {
} ...;
```

Premenovanie (Alias)

Niektoré názvy boli nekonzistentné alebo chybné. Špecifikácia ich ponecháva ako zastaralé. Názvy z rozšírení presunuté do jadra sú tak isto odkazované cez *typedef*.

```
typedef ... ...;
```

2.5 Rozhranie pre jazyk C++

Rozhranie pre jazyk C++ je nadstavba nad rozhraním jazyka C, ktoré je implicitné zahrnuté. Je tak isto generované z Vulkan špecifikácie. Rozhranie je definované v mennom priestore `vk`. Názvy prvkov sú preto transformované podľa pravidiel [20]. Predpona `Vk` je odstránená v názvoch enumerácií, štruktúr, objektov a funkcií. Názvy hodnôt enumerácií začínajú písmenom `e` a sú v štýle camel. Predpona `VK_` a prípadná prípona sú odstránené. Rozhranie sa zahrnie do projektu cez hlavný súbor *vulkan.hpp*.

Definície preprocesora

V hlavnom súbore *vulkan.hpp* nájdeme mnoho direktív pre preprocesor[19]. Väčšina z nich zaisťuje podporu pre rôzne štandardy jazyka C++, pretože Vulkan rozhranie je spätne kompatibilné až po štandard C++11[20].

Názov menného priestoru je definovaný v direktíve *VULKAN_HPP_NAMESPACE* a dá sa zmeniť.

ArrayWrapper

Pomocná trieda okolo jednorozmerného a dvojrozmerného štandardného pola `std::array`. Poskytuje rozšírenú funkcionalitu, napríklad preťažené operátory. Používa sa vo Vulkan štruktúrach (súbor *vulkan_structs.hpp*).

Štruktúra FlagTraits

Táto štruktúra využíva template špecializáciu. Podľa konkrétneho typu určuje, či sa jedná o enumeráciu bitového pola zvanú *Bitmask*. Používa sa vo Vulkan enumeráciách (súbor *vulkan_enums.hpp*).

Trieda Flags

Poskytuje ekvivalentnú funkcionalitu ako bitové pole v rozhraní pre jazyk C, v ktorom sa jedná o primitívny číselný dátový typ takže operácie sú definované implicitne. Naopak v jazyku C++ sú enumerácie silne typované a preto potrebujeme pomocnú triedu na preťaženie operátorov logických operácií. Používa sa vo Vulkan enumeráciách (súbor *vulkan_enums.hpp*).

[18] https://en.cppreference.com/w/c/language/union
[19] https://en.cppreference.com/w/cpp/preprocessor
[20] https://en.cppreference.com/w/cpp/11

Trieda ArrayProxy

Táto trieda poskytuje pohľad na pole. V štandarde C++20 bol pridaný ekvivalent `std::span`. Avšak Vulkan rozhranie používa pre veľkosť pola dátový typ `uint32_t` a štandard `std::size_t`, čo by vyžadovalo explicitné pretypovanie. Z tohoto dôvodu používa Vulkan rozhranie stále vlastnú implementáciu. Používa sa vo Vulkan príkazoch (súbory *vulkan_handles.hpp, vulkan_funcs.hpp, vulkan_raii.hpp*).

Trieda ArrayProxyNoTemporaries

Poskytuje podobnú funkcionalitu ako *ArrayProxy*, no s obmedzením na životnosť zdrojového pola. Používa sa vo Vulkan štruktúrach (súbor *vulkan_structs.hpp*).

Trieda StridedArrayProxy

Poskytuje podobnú funkcionalitu ako *ArrayProxy*, počíta navyše aj s krokom pola. Používa sa vo Vulkan príkazoch (súbory *vulkan_handles.hpp, vulkan_funcs.hpp, vulkan_raii.hpp*).

Trieda Optional

Obaľuje ukazovateľ na dátový typ. Je možné priradiť hodnotu ako referenciu alebo nulový ukazovateľ. Používa sa vo Vulkan príkazoch ako voliteľný parameter (súbory *vulkan_handles.hpp, vulkan_funcs.hpp, vulkan_raii.hpp*).

Trieda StructureChain

Poskytuje reťazenie štruktúr s kontrolou počas kompilácie. Vulkan štruktúry sa dajú reťaziť cez ukazovateľ typu `void`, rovnako ako aj v rozhraní pre jazyka C. Každá štruktúra obsahuje položku *sType*, podľa ktorej Vulkan knižnica dedukuje typ štruktúry.

```
VULKAN_HPP_NAMESPACE::StructureType sType;
const void *pNext;
```

Vulkan register špecifikuje presné vzťahy medzi štruktúrami. Ak by sme používali reťazenie v kóde manuálne, kompilátor nevie skontrolovať správnosť. StructureChain v prípade nedovolenej kombinácie produkuje kompilačnú chybu. Ukážkový kód [17]:

```
vk::StructureChain<vk::AnchorStruct, vk::ChainedStruct> chain
(
  { /* set other values of anchor */ }
  { /* set other values of chained */ }
);
```

Trieda UniqueHandle

Táto trieda spoločne s triedami *UniqueHandleTraits, ObjectDestroy, ObjectFree a PoolFree* slúži pre implementáciu unikátnych objektov. Používa sa vo Vulkan objektoch (súbor *vulkan_handles.hpp*).

Trieda DispatchLoaderStatic

Slúži pre volanie priamo linkovaných funkcií. Používa sa vo Vulkan príkazoch (súbory *vulkan_handles.hpp, vulkan_funcs.hpp*).

Vulkan enumerácie

Vulkan enumerácie[21] sú definované v individuálnom súbore *vulkan_enums.hpp*. Podrobnejšie budú popísané v kapitole 3.

Prevod Vulkan enumerácie na reťazec

Definované v individuálnom súbore *vulkan_to_string.hpp*. Funkcie prevedú hodnotu enumerácie na užívateľom čitateľný reťazec. Funkcionalita sa dá vypnúť direktívom *VULKAN_HPP_NO_TO_STRING*.

Trieda Error

Táto trieda spoločné mnohými podtriedami slúži pre výnimky. Vulkan v predvolenom režime v prípade zlyhania príkazu vyvolá príslušnú výnimku. Návratová hodnota sa kontroluje pomocnou funkciou `resultCheck`, ktorá v prípade chyby deleguje tvorbu výnimky na funkciu `throwResultException`.

Podtriedy sú automaticky generované z enumerácie *Result*. Napríklad z *Result::eErrorOutOfHostMemory* vznikne trieda *OutOfHostMemoryError*.

Výnimky sa dajú vypnúť direktívom *VULKAN_HPP_NO_EXCEPTIONS*. Používajú sa vo funkciách (súbory *vulkan_funcs.hpp, vulkan_raii.hpp*).

Štruktúry ResultValue a ResultValueType

Používajú sa ako návratové typy z funkcií. `ResultValue` je pár *Result* a typu hodnoty T. Slúži podobne ako ekvivalent `std::pair<Result, T>`. Používa sa vo funkciách, ktoré okrem hodnoty vracajú aj výsledok pre kontrolu chýb (súbory *vulkan_handles.hpp, vulkan_funcs.hpp*).

`ResultValueType` je závislý na režime výnimiek. Ak sú výnimky zapnuté, tvári sa len ako typ T. Ak sú výnimky vypnuté, tvári sa ako *ResultValue*. Pre vytvorenie návratového typu slúži funkcie `createResultValueType`. Používa sa v ostatných funkciách (súbory *vulkan_handles.hpp, vulkan_funcs.hpp*).

Vulkan objekty

Vulkan objekty sú v jazyku C++ definované ako triedy[22] v individuálnom súbore *vulkan_handles.hpp*. Objekty obsahujú operátor pre implicitnú konverziu na ekvivalent v rozhraní pre jazyk C.

Vulkan štruktúry

Vulkan štruktúry sú definované v individuálnom súbore *vulkan_structs.hpp*. Dátové položky sú v prípade Vulkan enumerácií a objektov nahradené ekvivalentom v rozhraní pre jazyk C++. Poskytujú rozšírenú funkcionalitu oproti štruktúram v rozhraní pre jazyk C, ako napríklad konštruktor, setter funckie a operátor pre porovnanie. Štruktúry obsahujú operátor pre implicitnú konverziu na ekvivalent v rozhraní pre jazyk C.

[21] https://en.cppreference.com/w/cpp/language/enum
[22] https://en.cppreference.com/w/cpp/language/class

Vulkan príkazy

Vulkan príkazy sú definované v individuálnom súbore *vulkan_funcs.hpp*. Väčšina príkazov je implementovaná ako funkcia v triede. Štruktúra parametrov čo najviac odzrkadľuje parametre v rozhraní pre jazyk C. Ak je to možné, sú generované preťažené funkcie s parametrami obohatenými o funkcionalitu jazyka C++ (referencie atď.).

Štruktúra StructExtends

Špecifikuje, ktorá štruktúra sa môže reťaziť s ďalšou. Tieto vzťahy sú generované automaticky.

```
template <>
struct StructExtends<..., ...>
{
  enum
  {
    value = true
  };
};
```

Trieda DynamicLoader

Slúži pre načítanie Vulkan knižnice za behu. Používa sa v rozhraní RAII (súbor *vulkan_raii.hpp*).

Dá sa zapnúť direktívom *VULKAN_HPP_ENABLE_DYNAMIC_LOADER_TOOL*.

Trieda DispatchLoaderDynamic

Slúži pre volanie nepriamo linkovaných funkcií. Používa sa vo Vulkan príkazoch (súbory *vulkan_handles.hpp, vulkan_funcs.hpp*).

2.6 RAII rozhranie

Varianta rozhrania spĺňajúca RAII princíp[23]. Vulkan príkazy sú interne implementované cez PFN funkcie, čo poskytuje vyšší výkon. Správne použitie tohoto rozhrania prináša garanciu korektného uvoľnenia objektov. Objekty si musia navyše uchovať aj odkaz na rodiča a tabuľku funkčných ukazovateľov, takže réžia je trochu vyššia.

Rozhranie sa zahrnie do projektu cez súbor *vulkan_raii.hpp*, čo implicitne zahrnie *vulkan.hpp*.

[23] https://en.cppreference.com/w/cpp/language/raii

Tabuľky funkčných ukazovateľov

V rozhraní sú definované tabuľky funkčných ukazovateľov, ktoré sú spravované príslušným Vulkan objektom. Užívateľ rozhrania preto nemusí implementovať vlastnú tabuľku, ani používať explicitne tabuľky z rozhrania, aj keď sú verejne viditeľné.

- **ContextDispatcher**: Tabuľka je vlastnená objektom *Context*.
  Najvyššia úroveň, inicializuje sa buď interne cez *DynamicLoader*, prípadne externe užívateľom.
- **InstanceDispatcher**: Tabuľka je vlastnená objektom *Instance*.
  Inicializuje sa pri vzniku inštancie *Instance*, zdroj je *Context*.
- **DeviceDispatcher**: Tabuľka je vlastnená objektom *Device*.
  Inicializuje sa pri vzniku inštancie *Device*, zdroj je *Instance*.

Trieda Context

Rozhranie musí načítať Vulkan knižnicu a následne PFN funkcie, čo zabezpečuje trieda Context. Má dlhšiu životnosť ako ostatné Vulkan objekty.

Vulkan objekty

Vulkan objekty obaľujú objekty z rozhrania *vk*. Deštruktor sa stará o uvoľnenie objektu. Navyše poskytujú funkcie `clear()`, `release()`, `getDispatcher()` a `swap()`.

Kapitola 3

Návrh

Moderné kompilátory na platformách Windows a Linux podporujú štandard 20 jazyka C++. Preto je nové rozhranie navrhnuté prioritne pre C++ štandard 20 a vyššie. Užívateľský kód je prispôsobený pre oficiálne rozhranie pre jazyk C++ a mal by byť nekompatibilný len v okrajových prípadoch.

Primárny cieľ nového rozhrania je optimalizácia rýchlosti kompilácie. Vulkan rozhranie bolo predovšetkým navrhnuté pre maximálny výkon, preto nesieme na tento fakt zabudnúť a degradovať výkon aplikácie. Zároveň je žiadúce udržať úplnú kompatibilitu s doterajším rozhraním aspoň v predvolenej variante (konfigurácií).

Významná myšlienka pri návrhu je zúžiť všeobecnosť rozhrania a zaviesť experimentálne zmeny. Jeden z hlavných pilierov jazyka C++ je **zero-overhead princíp** [16]. Síce sa tento princíp priamo netýka náročnosti kompilácie, ale je možné ho do určitej miery aplikovať a navrhnúť efektívnejšie rozhranie. Bohužiaľ neexistuje svätý grál, ktorý by zrýchlil kompiláciu na zlomok sekundy. Môžeme predpokladať, že menej kódu sa kompiluje rýchlejšie, ale to nie vždy platí.

Existuje mnoho užitočných praktík, ktoré zlepšia dobu kompilácie v projekte využívajúce rozhranie. Keďže sa týkajú projektu mimo rozhrania, sú cez rozsah tejto práce.

S monolitickým Vulkan rozhraním pre jazyk C++ musíme zahrnúť do projektu celé rozhranie, tým pádom 'platíme' dobou kompilácie za časti ktoré nevyužívame. Ak bola pridaná do oficiálneho rozhrania funkcionalita s významným vplyvom na dobu kompilácie, tak bolo (nie vždy) zavedené direktívum pre voliteľné vypnutie danej funkcionality.

Súčasné Vulkan rozhranie pre jazyk C++ má okolo 200 000 riadkov kódu. Kompilátoru trvá spracovať rozhranie niekoľko sekúnd. V obsiahlom projekte môže byť zahrnuté v niekoľkých prekladových jednotkách, pričom spracovanie prebehne samostatne. Tento problém pomáhajú riešiť predkompilované hlavičkové súbory [1]. Táto metóda sa vzťahuje na konkrétny projekt využívajúci Vulkan rozhranie a je mimo rozsah práce.

Je odporúčané minimalizovať veľkosť hlavičkových súborov [13] (obvykle presunutím čo najviac kódu do zdrojových súborov), čím sa zníži čas spracovania hlavičkového súboru. Rozhranie je ale len vo forme hlavičkových súborov, takže kód nemáme kam presunúť. Mohli by sme teoreticky zaviesť konfiguráciu pre zdrojové súbory, ale užívateľ by musel upraviť projekt. Inštanciácia template kódu v hlavičkovom súbore môžu mať podstatne predĺžiť čas kompilácie[13]. Preto sa pokúsime obmedziť výskyt template kódu.

[1] https://learn.microsoft.com/en-us/cpp/build/creating-precompiled-header-files?view=msvc-170

Predvolený spôsob spracovania chýb v oficiálnom rozhraní sú výnimky. Napriek vyššej réžii výsledného programu ich považujeme za najlepší a preferovaný spôsob. Vývojári jazyka C++ pracujú na optimalizáciách výnimiek, vďaka ktorým by mohli mať takmer nulovú dodatočnú réžiu [2].

3.1 Rozhranie pre jazyk C++

Direktíva

Oficiálna implementácia Vulkan generátora opatrí segment kódu príslušným direktívom. Segment kódu je vygenerovaný nezávisle na okolitom kóde. Môže sa jednať o ochranné direktívum pre kód špecifický pre platformu, alebo direktívum pre povolenie alebo zakázanie určitých častí rozhrania. Samozrejme tieto direktíva sa kombinujú a vznikajú úrovne.

Vďaka vlastnej implementácií generátora máme kontrolu nad generáciou kódu a sme schopný združiť kód do skupín pre časti kódu kde je nezávislé poradie. Tým pádom vznikne menší počet direktív, čo znamená menej práce v preprocesorovej fáze.

Direktíva prispôsobujúce Vulkan rozhranie pre jazyk C++ začínajú predponou *VULKAN_HPP*.

Nové direktíva zavedené v tejto práci začínajú predponou *VULKAN_HPP_EXPERIMENTAL*.

Enumerácie

Enumerácie sú vygenerované v súbore *vulkan_enums.hpp*.

V C++ sú všetky enumerácie vo Vulkan rozhraní pre jazyk C++ deklarované s rozsahom. To súhlasí s bodom *Enum.3* v smernici[1] a prináša výhodu silného typovania. Nevýhoda je nadbytočný kód pre zaistenie logických operácií, ktoré sú v jazyku C implicitné. Hodnoty položiek sú odkazované na ekvivalent enumerácie v rozhraní pre jazyk C. Jediná možná zmena, ktorá nemá vplyv na vonkajšie rozhranie, je priame použitie číselných konštánt.

```
// C Vulkan:
typedef enum VkResult {
  VK_SUCCESS = 0,
  VK_NOT_READY = 1,
  VK_TIMEOUT = 2,
  ...
};
```

```
// C++ Vulkan:
enum class Result {
  eSuccess = VK_SUCCESS,
  eNotReady = VK_NOT_READY,
  eTimeout = VK_TIMEOUT,
  ...
};
```

Alternatíva:

```
enum class Result {
  eSuccess = 0,
  eNotReady = 1,
  eTimeout = 2,
  ...
};
```

[2] https://isocpp.org/blog/2019/09/cppcon-2019-de-fragmenting-cpp-making-exceptions-and-rtti-more-affordable-a

Je možné že táto zmena by nemala podstatný vplyv na dobu kompilácie. Pre zachovanie úplnej kompatibility musia ostať názvy všetkých prvkov rovnaké.

Enumerácie typu bitmask vykonávajú template inštanciu štruktúry `FlagTraits` (v rozhraní pre jazyk C++ sa využíva v implementácií logických operátorov), čo je ďalší kandidát na preskúmanie, preto pridáme pre tento kod direktívum *VULKAN_HPP_EXPERIMENTAL_NO_FLAG_TRAITS*.

Zdrojový kód 3.1: Pseudo-definícia Vulkan enumerácie

```
enum class ... {
  << values >>
};
```

Zdrojový kód 3.2: Pseudo-definícia Vulkan enumerácie typu Bitmask

```
#ifdef VULKAN_HPP_EXPERIMENTAL_NO_FLAG_TRAITS
template <>
struct FlagTraits<...> {
  static VULKAN_HPP_CONST_OR_CONSTEXPR bool isBitmask = true;
  static VULKAN_HPP_CONST_OR_CONSTEXPR << type >> allFlags = ...;
};
#endif // VULKAN_HPP_EXPERIMENTAL_NO_FLAG_TRAITS

enum class << type >> : << C type >> {
  << values >>
};
```

Prevod enumerácie na reťazec

Funkcie na prevod sú vygenerované v súbore *vulkan_to_string.hpp*.

Pre každú enumeráciu je deklarovaná funkcia pre prevod na štandardný reťazec. V rámci Vulkan rozhrania sú použité na chybový výpis v prípade výnimky (len enumerácia Result). Logická štruktúra je veľmi jednoduchá, na ktorej nie je priestor pre vylepšenie.

Tento súbor obsahuje funkciu `toHexString`, ktorá slúži na prevod hodnoty enumerácie na hexadecimálny reťazec. Súčasná implementácia využíva jednu zo štandardných knižníc podľa podpory kompilátora.

Zdrojový kód 3.3: Súčasné knižnice potrebné pre funkciu toHexString

```
#if __cpp_lib_format
#  include <format>   // std::format
#else
#  include <sstream>  // std::stringstream
#endif
```

Efektívnejšia alternatíva by využívala knižnicu s menšou réžiou, alebo prípadne nepoužívala žiadnu knižnicu. V jazyku C++ máme od štandardu C++11 k dispozícií vstupno-výstupné funkcie jazyka C v knižnici `cstdio`.[3]. Prevod na hexadecimálny reťazec implementujeme volaním funkcie `std::snprintf`, ktorá je bezpečná varianta funkcie `sprintf`. Túto funkcionalitu zavedieme pod voliteľné direktívum VULKAN_HPP_EXPERIMENTAL_HEX.

[3] https://en.cppreference.com/w/cpp/io/c/fprintf

Zdrojový kód 3.4: Implementácia funkcie toHexString

```
#ifdef VULKAN_HPP_EXPERIMENTAL_HEX
#  include <cstdio> // std::snprintf
#elif __cpp_lib_format
#  include <format>  // std::format
#else
#  include <sstream>  // std::stringstream
#endif


VULKAN_HPP_INLINE std::string toHexString( uint32_t value )
{
#ifdef VULKAN_HPP_EXPERIMENTAL_HEX
  std::string str;
  str.resize(6);
  int n = std::snprintf(str.data(), str.size(), "%x", value);
  VULKAN_HPP_ASSERT( n > 0 );
  return str;
#elif __cpp_lib_format
  return std::format( "{:x}", value );
#else
  std::stringstream stream;
  stream << std::hex << value;
  return stream.str();
#endif
}
```

Štruktúry

Štruktúry sú vygenerované v súbore *vulkan_structs.hpp*.

Jednaa sa o definície štruktúr vrátane dátového typu union. Dátové typy sú z rozhrania pre jazyk C++ v prípade štruktúr, enumerácií a objektov. Binárne usporiadanie štruktúry sa musí zhodovať s odpovedajúcou štruktúrou v rozhraní pre jazyk C. Štruktúra je navyše obohatená o konštruktory, setter funkcie a preťažené operátory. Táto funkcionalita pridáva veľké množstvo kódu a tým pádom je dobrý kandidát na preskúmanie.

Každá štruktúra navyše vykonáva template inštanciu štruktúry CppType, čo je ďalší kandidát na preskúmanie, preto pridáme pre tento kod direktívum *VULKAN_HPP_EXPERIMENTAL_NO_TEMPLATES*.

Zdrojový kód 3.5: Pseudo-definícia Vulkan štruktúry

```
struct ... {
  using NativeType = ...;
  << static members >>

#if !defined( VULKAN_HPP_NO_STRUCT_CONSTRUCTORS )
  << constructor >>
#  if !defined( VULKAN_HPP_DISABLE_ENHANCED_MODE )
  << enhanced constructor >>
#  endif // VULKAN_HPP_DISABLE_ENHANCED_MODE
#endif // VULKAN_HPP_NO_STRUCT_CONSTRUCTORS

#if !defined( VULKAN_HPP_NO_STRUCT_SETTERS )
  << setters >>
#  endif // VULKAN_HPP_NO_STRUCT_SETTERS

#if defined( VULKAN_HPP_USE_REFLECT )
  << reflect >>
#endif // VULKAN_HPP_USE_REFLECT
#ifndef VULKAN_HPP_EXPERIMENTAL_NO_STRUCT_COMPARE
#  if defined( VULKAN_HPP_HAS_SPACESHIP_OPERATOR )
  << operator<=> >>
#  else
  << operator== >>
  << operator!= >>
#  endif
#endif // VULKAN_HPP_EXPERIMENTAL_NO_STRUCT_COMPARE

  << operators >>
  << members >>
};

#ifndef VULKAN_HPP_EXPERIMENTAL_NO_TEMPLATES
template <>
struct CppType<StructureType, StructureType::...> {
  using Type = ...;
};
#endif // VULKAN_HPP_EXPERIMENTAL_NO_TEMPLATES
```

Konštruktory sa dajú vypnúť cez direktíva *VULKAN_HPP_NO_STRUCT_CONSTRUCTORS* a *VULKAN_HPP_NO_UNION_CONSTRUCTORS*.

Setter funkcie nápodobne dajú vypnúť cez direktíva *VULKAN_HPP_NO_STRUCT_SETTERS* a *VULKAN_HPP_NO_UNION_SETTERS*.

Operátory pre porovnanie dajú vypnúť cez direktívum *VULKAN_HPP_EXPERIMENTAL_NO_STRUCT_COMPARE*.

Od štandardu C++20 je podporovaná agregovaná inicializácia[4]. Štruktúra musí spĺňať podmienky, pričom najdôležitejšou z nich je absencia užívateľom definovaných konštruktorov. Ostatné podmienky Vulkan štruktúry našťastie spĺňajú. Oficiálne rozhranie už poskytuje vypnutie konštruktoruov cez direktívum.

Agregovanú štruktúru možno inicializovať veľmi podobnou syntaxou ako v jazyku C (designated inicializácia [5]). Výhodou je prehľadnejší kód, ktorý je takmer zhodný s ekvivalentom v jazyku C.

Zdrojový kód 3.6: Agregovaná inicializácia štruktúry

```
vk::ApplicationInfo appInfo = {
  .pApplicationName = "vulkan application",
  .apiVersion = VK_API_VERSION_1_3
};

vk::InstanceCreateInfo info = {
  .pApplicationInfo = &appInfo,
  .enabledLayerCount = (uint32_t)instanceLayers.size(),
  .ppEnabledLayerNames = instanceLayers.data(),
  .enabledExtensionCount = (uint32_t)extensions.size(),
  .ppEnabledExtensionNames = extensions.data()
};
```

Štruktúra StructExtends

Množina týchto štruktúr je vygenerovaná v hlavnom súbore *vulkan.hpp*, rovnako ako súčasné rozhranie pre jazyk C++.

Jedná sa o inštanciáciu template štruktúry, preto zavedieme direktívum *VULKAN_HPP_EXPERIMENTAL_NO_STRUCT_CHAIN*.

Triedy

Deklarácie tried sú narozdiel od súčasného rozhrania pre jazyk C++ vygenerované v súbore *vulkan_handles_forward.hpp*. Ak programátorovi stačí v niektorom z projektových súborov len deklarácia Vulkan objektov, nemusí ich manuálne písať, ale zahrnie tento súbor. Deklarácie sú nevyhnutné v rámci rozhrania, pretože v triede sa môže vyskytnúť iná trieda definovaná neskôr.

Definície tried sú vygenerované v súbore *vulkan_handles.hpp*. Každá trieda obaľuje Vulkan objekt rozhrania pre jazyk C v premennej (v generátore označovanú ako *handle*) a tým pádom daný objekt sama reprezentuje. Je dôležite, aby trieda obsahovala len *handle* premennú, inak by binárne usporiadanie nesedelo s odpovedajúcim objektom v rozhraní pre jazyk C, čo by malo za následok chybu počas behu programu. Triedy navyše obsahujú pevné časti kódu, ako sú konštuktory a operátory. Tiež obsahujú vygenerované funkcie (prípadne vygenerované konštruktory) pod direktívom *VULKAN_HPP_EXPERIMENTAL_NO_VK_FUNCS*.

Každá trieda navyše vykonáva template inštanciu štruktúr *CppType* a *isVulkanHandleType*, čo je ďalší kandidát na preskúmanie, preto pridáme pre tento kod direktívum *VULKAN_HPP_EXPERIMENTAL_NO_TEMPLATES*.

[4] https://en.cppreference.com/w/cpp/language/aggregate_initialization
[5] https://en.cppreference.com/w/cpp/language/aggregate_initialization#Designated_initializers

Zdrojový kód 3.7: Pseudo-definícia triedy pre Vulkan objekt

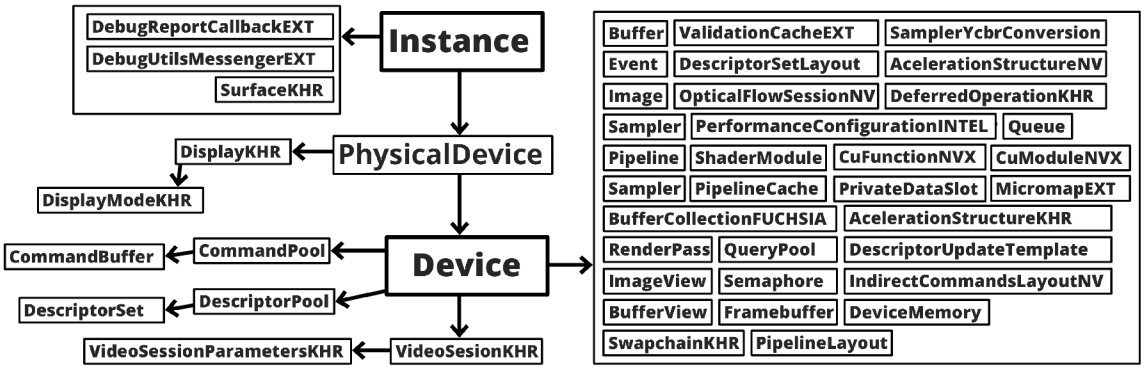
```
class ... {
public:
  using CType      = ...;
  using NativeType = ...;
  << static members >>
  << constructors >>
  << operators >>
#ifndef VULKAN_HPP_EXPERIMENTAL_NO_CLASS_COMPARE
  << compare operators >>
#endif // VULKAN_HPP_EXPERIMENTAL_NO_CLASS_COMPARE

#ifndef VULKAN_HPP_EXPERIMENTAL_NO_INTEROP
  << constructors >>
#endif // VULKAN_HPP_EXPERIMENTAL_NO_INTEROP
#ifndef VULKAN_HPP_EXPERIMENTAL_NO_VK_FUNCS
  << member functions >>
#endif // VULKAN_HPP_EXPERIMENTAL_NO_INTEROP
protected:
  Vk... m_... = {}; // handle
};
#ifndef VULKAN_HPP_EXPERIMENTAL_NO_TEMPLATES
  << template instantiation>
#endif // VULKAN_HPP_EXPERIMENTAL_NO_TEMPLATES
```

Kód operátorov pre porovnanie je voliteľný pod direktívom *VULKAN_HPP_EXPERIMENTAL_NO_CLASS_COMPARE*.

Voliteľná funkcionalita rozširujúca triedy o konštruktory kompatibilné s RAII variantou rozhrania je zavedená pod direktívom *VULKAN_HPP_EXPERIMENTAL_INTEROP*. V opačnom prípade je automaticky definované direktívum *VULKAN_HPP_EXPERIMENTAL_NO_INTEROP*.

Kompletná množina objektov je na obrázku 3.1. Zobrazuje hierarchiu vlastníctva objektov. Nadradený objekt musí byť vytvorený skorej, ako podradený. V momente uvoľnenia objektu z pamäte (príslušné Vulkan príkazy s predponou `vkDestroy` a `vkFree`) musia byť už uvoľnené všetky podradené objekty.



Obr. 3.1: Hierarchia objektov v novom rozhraní

Triedy kategórie SmartHandle

Triedy kategórie SmartHandle sú vygenerované v súbore *vulkan_handles.hpp*. Nazývajú sa tiež chytré triedy. Jedná sa o Vulkan objekty s podobnou sémantikou ako štandardný `std::unique_ptr`[6] spĺňajúce RAII princíp. Názov začína predponou `Unique`. Súčasné Vulkan rozhranie implementuje chytré triedy cez template inštanciáciu. Dajú sa vypnúť pomocou direktíva *VULKAN_HPP_NO_SMART_HANDLE*.

Zdrojový kód 3.8: Pseudo-definícia triedy UniqueHandle

```
template <typename Type, typename Dispatch>
class UniqueHandle : public UniqueHandleTraits<Type, Dispatch>::deleter
{
private:
  using Deleter = typename UniqueHandleTraits<Type, Dispatch>::deleter;
public:
  << constructors >>
  << destructor >>
  << operators >>
  << get() function >>
  << reset() function >>
  << release() function >>
  << destroy() function >>
  << swap() function >>
private:
  Type m_value;
};
```

Zdrojový kód 3.9: Pseudo-definícia triedy SmartHandle cez template

```
template <typename Dispatch>
class UniqueHandleTraits<..., Dispatch>
{
public:
  using deleter = ...;
};
using Unique... = UniqueHandle<..., VULKAN_HPP_DEFAULT_DISPATCHER_TYPE>;
```

Alternatívna možnosť je implementácia cez dedičnosť, ktorá síce zaberá viac riadkov kódu, ale môže byť rýchlejšie kontrolovateľná.

[6] https://en.cppreference.com/w/cpp/memory/unique_ptr

Zdrojový kód 3.10: Pseudo-definícia triedy SmartHandle cez dedičnosť

```
class Unique... : << public|private >> ... {
public:
  << constructors >>
  << destructor >>
  << operators >>
  << get() function >>
  << reset() function >>
  << release() function >>
  << destroy() function >>
  << swap() function >>
private:
  << members >>
};
```

Funkcie

Funkcie Vulkan objektov sú vygenerované v súbore *vulkan_funcs.hpp*. Poskytujú rozhranie medzi aplikáciou a Vulkan príkazmi. Pre jeden príkaz môže byť vygenerovaných viac variánt. Základná varianta funkcie ponecháva počet pôvodných parametrov. Enumerácie, štruktúry a objekty sú z rozhrania pre jazyk C++, funkcia vnútri pretypuje parametre na ekvivalent z rozhrania pre jazyk C.

Ostatné varianty je možné zakázať direktívom *VULKAN_HPP_DISABLE_ENHANCED_MODE*.

Návratová hodnota Vulkan príkazu v prípade dátového typu VkResult reprezentuje výsledok operácie. Ak je posledný parameter príkazu nekonštantný ukazovateľ, tak príkaz vráti hodnotu v tomto parametri. Táto kategória funkcií ma návratovú hodnotu podľa príkazu a výsledok operácie skontroluje interne volaním funkcie `resultCheck`, ktorá v prípade neúspechu vyvolá výnimku.

Nekonštantný parameter, ktorý je ukazovateľ a odkazuje na neho parameter vyjadrujúci počet prvkov, je výstupné pole. V tomto prípade sa návratový typ transformuje na `std::vector<T>` a funkcia sa interne stará o alokáciu štandardného vektora.

Ak je návratový typ funkcie `std::vector<T>`, je vygenerovaná varianta s parametrom pre štandardný alokátor. Ak by vznikol návratový typ `std::vector<void>`, je nahradený za `std::vector<uint8_t>` a veľkosť je interpretovaná ako počet bytov.

Konštantný parameter, ktorý je ukazovateľ a odkazuje na neho parameter vyjadrujúci počet prvkov, je vstupné pole. Dvojica týchto parametrov (počet a dátový typ T) sa transformuje na `ArrayProxy<T>`. Ak by vznikol parameter `ArrayProxy<void>`, je nahradený za `ArrayProxy<DataType>` a typ DataType uvedený ako `typename template` funkcie.

Ukazovateľ na Vulkan štruktúru sa transformuje na referenciu. Parameter, ktorý nie je ukazovateľ a je podľa registra poznačený ako voliteľný, je zabalený do dátového typu `Optional<T>`. Zvyčajne sa jedná o alokátor (dátový typ `AllocationCallbacks`).

Ak funkcia vytvára pole objektov, je vygenerovaná varianta pre vrátenie jediného objektu. Táto funkcia má odstránený plurál v názve.

Interné funkcie

Množstvo funkcií má návratový typ pole – `std::vector`. Vnútorný kód je v tomto prípade rovnaký boilerplate, závisí len na návratovom type Vulkan príkazu, ktorý je buď `VkResult` alebo `void`. Preto bola zavedená možnosť nahradiť výskyty redundantného kódu za pomocné funkcie v internom mennom priestore. Vedľajšia výhoda je, že výsledný súbor má menšiu veľkosť. Táto funkcionalita je dostupná cez konfiguráciu.

Zdrojový kód 3.11: Pomocné funkcie v internom mennom priestore

```
namespace internal {

  template<typename T, typename V, typename S, typename PFN, typename... Args>
  static inline std::vector<T> createArrayVoidPFN(const PFN pfn,
                                                  const char *const msg,
                                                  Args... args)
  {
    std::vector<T> data;
    S count;
    pfn(std::forward<Args>(args)..., &count, nullptr);
    data.resize( count );
    pfn(std::forward<Args>(args)..., &count, std::bit_cast<V*>(data.data()));
    if (count < data.size())
      data.resize( count );
    return data;
  }

  template<typename T, typename V, typename S, typename PFN, typename... Args>
  static inline typename ResultValueType<std::vector<T>>::type
    createArray(const PFN pfn,
    const char *const msg,
    Args... args)
  {
    std::vector<T> data;
    S count;
    VkResult result;
    do {
      result = pfn(std::forward<Args>(args)..., &count, nullptr);
      if (result == VK_SUCCESS && count) {
        data.resize( count );
        result = pfn(std::forward<Args>(args)..., &count,
                     std::bit_cast<V*>(data.data()));
      }
    } while (result == VK_INCOMPLETE);
    resultCheck(static_cast<Result>(result), msg);
    if (count < data.size())
      data.resize( count );
    return createResultValueType(static_cast<Result>(result), data);
  }

}  // namespace internal
```

Ostatné súbory

Ostatné súbory nie sú priamo zahrnuté v rozhraní jazyka C++. Preto ich nepovažujeme za potrebnú súčasť.

Súbor *vulkan_static_assertions.hpp*

Momentálne pozostáva z tisícky makier *VULKAN_HPP_STATIC_ASSERT* vykonávajúce kontrolu rozhrania pre jazyk C++ počas kompilácie. Predovšetkým sa jedná o kontrolu veľkostí dátových typov s ekvivalentom v rozhraní pre jazyk C.

Súbor *vulkan_hash.hpp*

Poskytuje podporu pre štandardnú funkciu `std::hash`[7].

Súbor *vulkan_profiles.hpp*

Popis je dostupný na GitHub stránke skupiny Khronos [8].

Súbor *vulkan_format_traits.hpp*

Informácie o formátoch[9].

3.2 RAII rozhranie

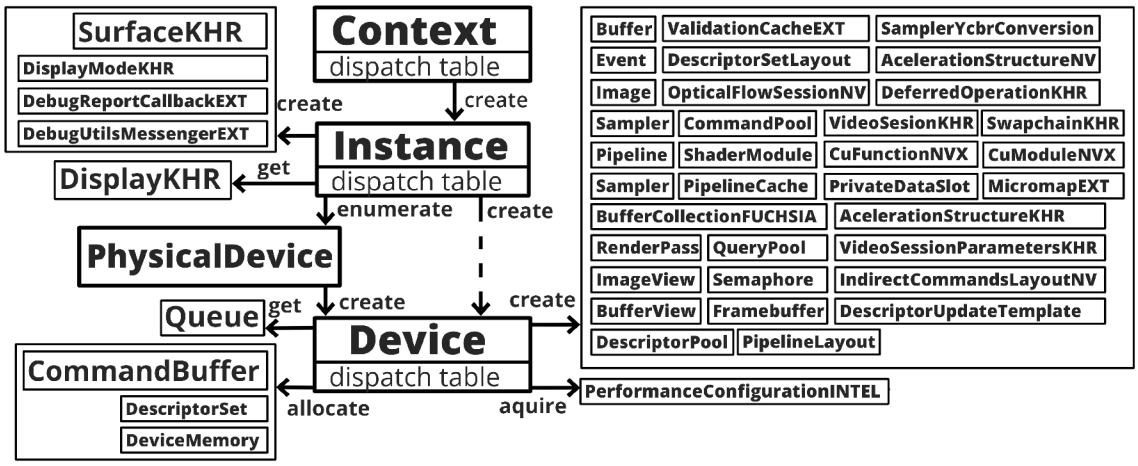
V tejto práci je kladený dôraz na RAII variantu Vulkan rozhrania z niekoľkých dôvodov. Hlavné výhody sú nepriame linkovanie (viď sekcia 2.1) a garantované uvoľnenie prostriedkov. Hla+vná nevýhoda je väčšie množstvo kódu v rozhraní. Triedy v RAII rozhraní obaľujú triedy z hlavného rozhrania pre jazyk C++. Pre uvoľnenie Vulkan objektu si musí objekt uložiť odkaz na rodičovský objekt najvyššej úrovne, ktorými sú objekty Instance a Device. To znamená dodatočné dátové položky v triede, tým pádom binárne usporiadanie objektu nie je zhodná s ekvivalentom v rozhraní pre jazyk C.

Kompletná množina objektov v RAII rozhraní je na obrázku 3.2. Zároveň sú zobrazené vzťahy vlastníctva a kategória Vulkan príkazu vytvárajúca podradený objekt. Tieto príkazy začínajú predponami.

- `vkCreate`
- `vkAllocate`
- `vkGet`

Vulkan knižnica a PFN ukazovatele sú inicializované počas behu programu v objekte Context, z ktorého sa vytvára objekt Instance. Trieda Context spravuje Vulkan knižnicu pomocou mechanizmov závislých na platforme a musí byť uvoľnená ako posledná.

[7] https://en.cppreference.com/w/cpp/utility/hash
[8] https://github.com/KhronosGroup/Vulkan-Profiles/blob/main/OVERVIEW.md
[9] https://github.com/KhronosGroup/Vulkan-Guide/blob/main/chapters/formats.adoc



Obr. 3.2: Hierarchia objektov v novom RAII rozhraní

Triedy

Definície tried RAII rozhrania sú vygenerované v súbore *vulkan_raii.hpp*. Deklarácie tried, podobne ako je to s triedami hlavného rozhrania, sú vygenerované v súbore *vulkan_raii_forward.hpp*. Je to hlavne z dôvodu rozšírenej funkcionality popísanej v sekcii 3.1, kedy je potrebná deklarácia RAII tried v hlavnom rozhraní.

Zdrojový kód 3.12: Pseudo-definícia triedy najvyššej úrovne v RAII rozhraní

```
class ... {
public:
  << static members >>
  << constructors >>
  << destructor >>
  << generated functions >>
#ifndef VULKAN_HPP_EXPERIMENTAL_NO_RAII_CREATE_CMDS
  << generated functions which create objects >>
#endif // VULKAN_HPP_EXPERIMENTAL_NO_RAII_CREATE_CMDS
#ifndef VULKAN_HPP_EXPERIMENTAL_NO_RAII_INDIRECT
  << generated functions from indirect category >>
#endif // VULKAN_HPP_EXPERIMENTAL_NO_RAII_INDIRECT
private:
   << data members >>
};
```

Podmnožina vygenerovaných funkcií vytvárajúce Vulkan objekt plnia rovnakú funkciu ako konštruktory Vulkan objektov, obzvlášť s experimentálnou rozšírenou funkcionalitou (pozri sekcia 3.1). Preto je možné tieto funkcie voliteľne vypnúť pomocou direktíva *VULKAN_HPP_EXPERIMENTAL_NO_RAII_CREATE_CMDS* a znížiť tak množstvo redundantného kódu.

Zdrojový kód 3.13: Pseudo-definícia podtriedy v RAII rozhraní

```
class ... {
public:
  << static members >>
  << constructors >>
  << destructor >>
#ifndef VULKAN_HPP_EXPERIMENTAL_NO_RAII_INDIRECT_SUB
  << generated functions >>
#endif // VULKAN_HPP_EXPERIMENTAL_NO_RAII_INDIRECT_SUB
private:
  << data members >>
};
```

Podmnožina Vulkan príkazov v súčasnom rozhraní pre jazyk C++ je priradená objektom nižšej úrovne. V novom rozhraní je možné vygenerovať tieto príkazy v triedach najvyššej úrovne (Instance a Device). Je to z dôvodu rozšírenej funkcionality, kedy objekty z nižšej úrovne sú z hlavného rozhrania pre jazyk C++ a nemajú prístup k daným príkazom (funkciám). Štýl volania príkazov cez objekt najvyššej úrovne je trochu viac zhodný s Vulkan rozhraním pre jazyk C. Táto funkcionalita sa dá zapnúť direktívom *VULKAN_HPP_EXPERIMENTAL_NO_INDIRECT_CMDS*.

```
#ifdef VULKAN_HPP_EXPERIMENTAL_NO_INDIRECT_CMDS
#   undef VULKAN_HPP_EXPERIMENTAL_NO_RAII_INDIRECT
#   define VULKAN_HPP_EXPERIMENTAL_NO_RAII_INDIRECT_SUB
#else
#   undef VULKAN_HPP_EXPERIMENTAL_NO_RAII_INDIRECT_SUB
#   define VULKAN_HPP_EXPERIMENTAL_NO_RAII_INDIRECT
#endif
```

Funkcie

Definícia funkcií RAII rozhrania sú narozdiel od súčasného rozhrania vygenerované v súbore *vulkan_raii_funcs.hpp*. Platí rovnaký princíp ako pre funkcie hlavného rozhrania popísané v sekcii 3.1.

3.3 Rozhranie vo forme modulov

Preskúmať nový systém modulov[10] v štandarde 20 je najviac experimentálna časť práce. Kľúčové slovo import nahradzuje direktívum preprocesora #include. Moduly poskytujú enkapsuláciu, viditeľný je len kód s kľúčovým slovom export, čo komplikuje situáciu z niekoľkých dôvodov:

- rozhranie pre jazyk C nie je exportované
  riešenie: rozhranie musí byť explicitne zahrnuté v kóde cez #include
- všetky definície preprocesora sú viditeľné len vnútri modulu
  riešenie: direktíva sú presunuté do samostatného hlavičkového súboru
- rozhranie sa nedá prispôsobiť definíciou direktíva pred importovaním
  riešenie: direktíva musia byť definované v projekte globálne

[10] https://en.cppreference.com/w/cpp/language/modules

Zdrojový kód 3.14: Rozhranie v module

```
// This header is generated from the Khronos Vulkan XML API Registry.

module;   // global module fragment, includes go here

#include "vulkan20_defines.hpp"
#include <vulkan/vulkan.h>

static_assert(VK_HEADER_VERSION == 239, "Wrong VK_HEADER_VERSION!");
#if VULKAN_HPP_ENABLE_DYNAMIC_LOADER_TOOL == 1
// DynamicLoader related definitions...
#endif

// #include of 14 standard libraries

export module VULKAN_HPP_NAMESPACE;   // primary module interface

// forward class declarations
#include "vulkan20_handles_forward.hpp"
#include "vulkan20_raii_forward.hpp"
export namespace VULKAN_HPP_NAMESPACE {
  // class ArrayProxy
  // class ArrayWrapper
#ifndef VULKAN_HPP_EXPERIMENTAL_NO_FLAG_TRAITS
  // struct FlagTraits
#endif
  // class Flags
  // class Optional
#ifndef VULKAN_HPP_EXPERIMENTAL_NO_STRUCT_CHAIN
  // struct StructExtends
#endif
  // contents of vulkan20_enums.hpp
#if !defined( VULKAN_HPP_NO_TO_STRING )
  // contents of vulkan20_to_string.hpp
#endif
  // class Error
  // struct ResultValue
  // struct forward declaration
  namespace internal {
    // helper functions
  }
  // contents of vulkan20_handles.hpp
  // contents of vulkan20_structs.hpp
  namespace VULKAN_HPP_NAMESPACE_RAII {
    namespace internal {
      // helper functions
    }
    // class Context
    // contents of vulkan20_raii.hpp ...
    // contents of vulkan20_raii_funcs.hpp ...
  }
}
```

Limitácie

Bohužiaľ kompilátory aj nástroje stále nie sú vyladené, dokonca nemajú plnú podporu, pre moduly. Súčasná verzia kompilátora clang na platforme Windows nie je schopná skompilovať štandardné knižnice. Preto ich musíme zahrnúť ako tradičný hlavičkový súbor v globálnej časti. Nadchádzajúci štandard C++23, ktorý ešte nie je dostupný, pridá štandardný modul std pre tento účel[11].

Pri konvertovaní rozhrania do modulov narazíme na problémy s linkovaním.

`error: declaration of 'getDispatchLoaderStatic' with internal linkage cannot be exported`

Statické funkcie v moduloch nie su viditeľné, preto stačí odstrániť kľúčové slovo a chyba sa odstráni.

`error: undefined symbol: vkResetQueryPoolEXT`

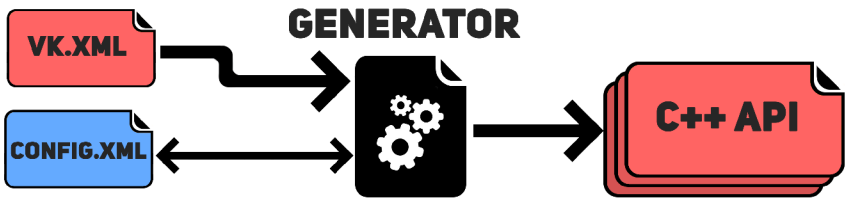
Rovnakú chybu dostaneme pre mnoho ďalších funkcií, ktoré nie sú exportované Vulkan knižnicou. Jedná sa o funkcie mimo Vulkan jadra. Jediné riešenie je tieto funkcie odstrániť. Preto je zatiaľ podporované len RAII rozhranie z dôvodu jednoduchšej implementácie.

3.4 Generátor rozhrania

Rozsiahle rozhranie pre Vulkan, či už pre programovací jazyk C alebo C++, je neprestaviteľné napísať ručne. Generuje sa automaticky cez nástroj (v kontexte práce generátor) zo špecifikácie (Vulkan register popísaný v sekcii 2.3).

Súčasný generátor Vulkan-Hpp (pozrie sekcia 2.2) zo špecifikácie vygeneruje monolitické rozhranie. Vlastný generátor je rozšírený o variabilnú konfiguráciu. Tento koncept umožní experimentovať s vygenerovaným rozhraním, ale možno sa stane v budúcnosti významnejším pre užívateľov Vulkan rozhrania.

Činnosť generátora je znázornená na obrázku 3.3. Užívateľ poskytne vstupný súbor obsahujúci Vulkan špecifikáciu (register). Prípadne môže poskytnúť konfiguračný súbor s nastaveniami. Generátor následne vytvorí rozhranie pre jazyk C++ a uloží ho do súborov v zvolenej cieľovej zložke.



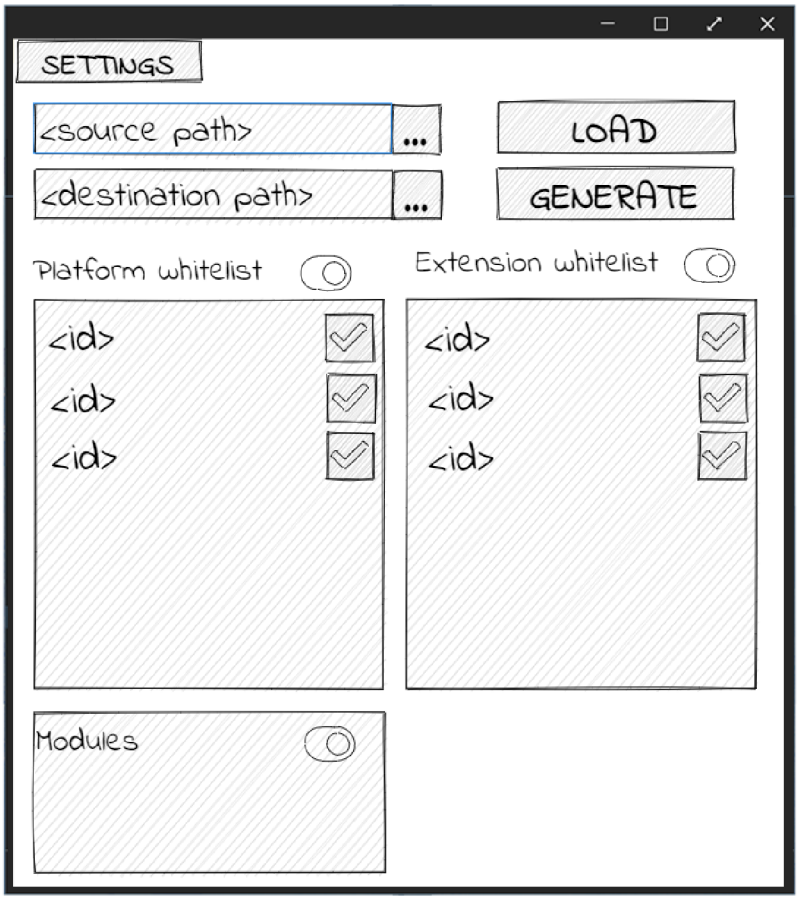
Obr. 3.3: Schéma generátora

[11] https://en.cppreference.com/w/cpp/standard_library#Importing_modules

Kapitola 4

Užívateľské rozhranie

Nový generátor poskytuje možnosť prispôsobiť vygenerované Vulkan rozhranie. Preto bolo za týmto účelom vytvorené GUI – grafické užívateľské rozhranie, poskytujúce interaktívnu konfiguráciu. Približný návrh na obrázku 4.1 ukazuje predovšetkým ovládacie prvky. Keďže nástroj je určený pre softvérových vývojárov, je kladený dôraz na praktickosť oproti vzhľadu.



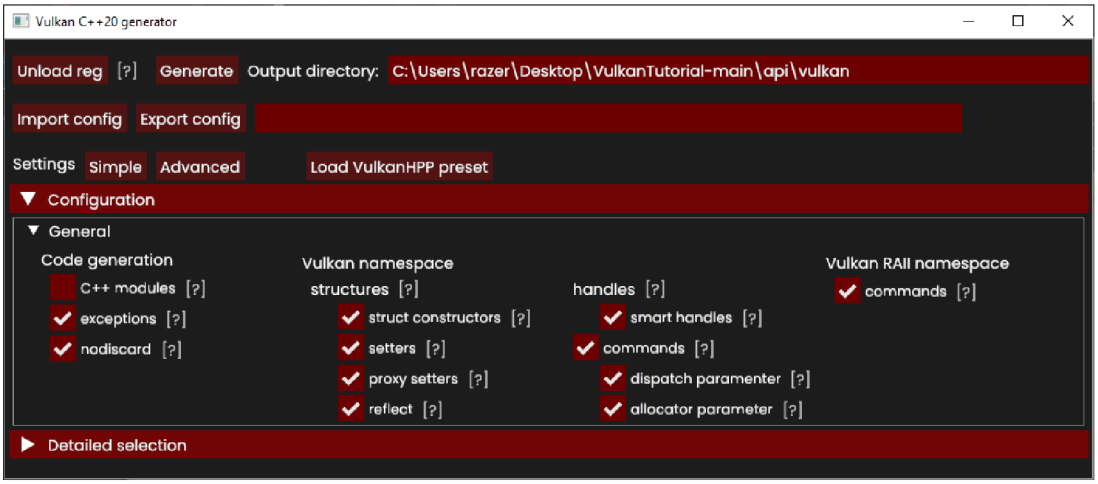
Obr. 4.1: Predbežný mockup aplikácie

Aplikácia po spustení zobrazí pohľad pre načítanie registra. Ak bol zadaný cez parameter príkazovej riadky, tak ho automaticky načíta a zobrazí hlavný pohľad.

Hlavná sekcia zobrazená v hornej časti na obrázku 4.2 obsahuje prevažne ovládacie prvky pre vstup a výstup. Užívateľ môže:

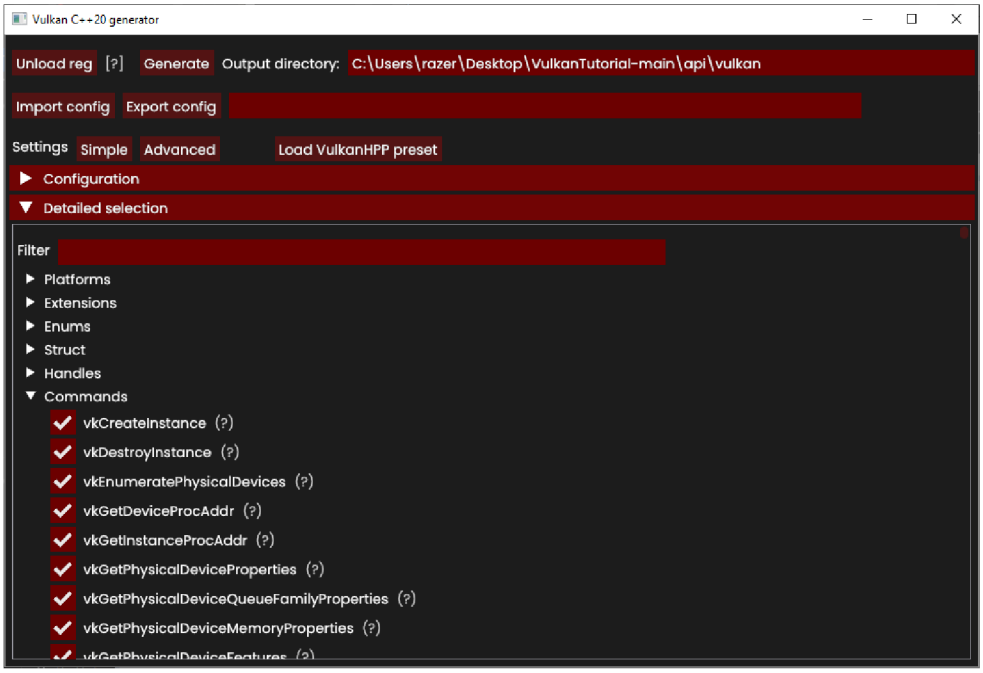
- načítať iný register cez tlačidlo Unload reg
- spustiť generáciu cez tlačidlo Generate
- zobraziť informácie o aktuálnom registri
- zadať výstupný adresár do textového pola (platnosť cesty je kontrolovaná)
- zadať cestu pre konfiguračný súbor do textového pola
- importovať konfiguračný súbor cez tlačidlo Import config
- exportovať konfiguračný súbor cez tlačidlo Export config
- prepnúť režim aplikácie na jednoduchý (zobrazí menej možností)
- načítať predvolenú konfiguráciu

Druhá sekcia zobrazená v spodnej časti na obrázku 4.2 obsahuje možnosti pre úpravu konfigurácie. Jedná sa o jednoduché bool nastavenia (checkbox). Najvýznamnejšie nastavenie je povolenie generácie modulov. Niektoré nastavenia majú rovnaký efekt ako definovanie direktív.

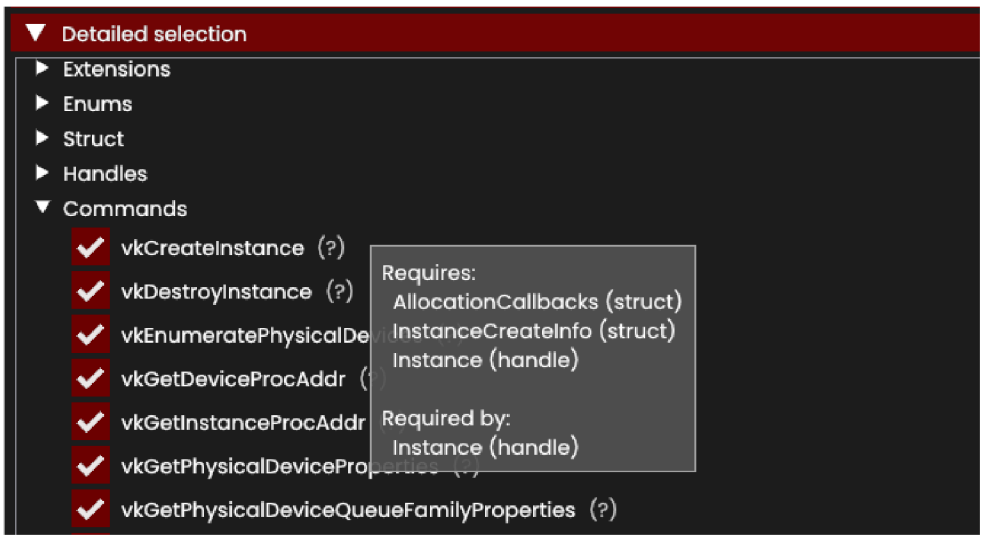


Obr. 4.2: Užívateľské rozhranie generátora (snímok 1)

Tretia sekcia je zobrazená v na obrázku 4.3. Obsahuje detailnú konfiguráciu prvkov Vulkan rozhrania na jednotlivých úrovniach. Aj keď môže pôsobiť odstrašujúco, predsa len pozostáva zo stoviek check-boxov, stačí konfigurovať len úroveň príkazov. Vďaka hierarchii závislostí vybudovanej pri načítaní špecifikácie sa automaticky povolia vyžadované prvky pre daný príkaz. Po nabehnutí kurzorom na nápovedu sú zobrazené závislosti (pozri obrázok 4.4).



Obr. 4.3: Užívateľské rozhranie generátora (snímok 2)



Obr. 4.4: Užívateľské rozhranie generátora (snímok 3)

4.1 Knižnica Dear ImGui

Zvolená knižnica Dear ImGui[1] je jednoduchá a nízko-úrovňová. Poskytuje len jednoduché prvky, napríklad `ImGui::Button`, `ImGui::Text` a `ImGui::CollapsingHeader`.

Výsledný generátor má tým pádom malú veľkosť (momentálne cca 3 MB) oproti robustnejším alternatívam. Bolo uvažované zvoliť Qt6[2], ale knižnice zaberajú desiatky MB.

[1] https://github.com/ocornut/imgui
[2] https://www.qt.io/product/qt6

Kapitola 5

Implementácia

V tejto práci bol implementovaný spustiteľný program pre generáciu rozhrania – generátor.

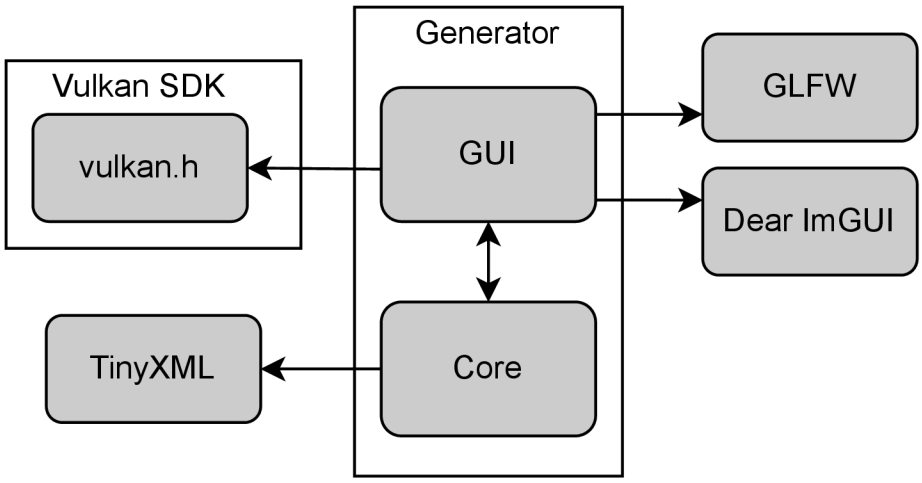
Projekt využíva nástroj **CMake** a bol implementovaný v jazyku C++. Knižnice tretích strán sú pribalené v projekte. Pre užívateľské rozhranie je vyžadovaná inštalácia **Vulkan SDK**.

Celá funkcionalita je z pohľadu užívateľa rozdelená do niekoľkých operácií (use-case):

1. Načítanie a spracovanie vstupu (súbor na lokálnom disku)
2. Generácia rozhrania (výstup)
3. Načítanie konfigurácie
4. Zápis konfigurácie
5. Úprava konfigurácie

Schéma generátora z pohľadu komponentov je na obrázku 5.1.

Jadro generátora je konzolová aplikácia. Nad jadrom je implementované užívateľské rozhranie umožňujúce pokročilú úpravu konfigurácie. Projekt je možné skompilovať len v režime konzolovej aplikácie – v takom prípade nie su potrebné externé závislosti.



Obr. 5.1: Aplikácia

Celá implementácia generátora presahuje 13000 riadkov kódu (oficiálny generátor má momentálne podobný rozsah). Vysvetlené budú dôležité koncepty.

5.1 Objektový prístup a dátové štruktúry

Jazyk C++ je objektovo orientovaný, preto prirodzene rozdelíme projekt do tried a štruktúr.

Trieda Container

Poskytuje abstrakciu kolekcie pre typy. Umožní vložiť a vyhľadať prvok, mazanie nie je potrebné. Pre existujúci prvok tiež umožní vytvoriť *alias* pre vyhľadávanie. Poradie prvkov je rovnaké ako vkladanie. Pre kolekciu sa dá vytvoriť zoradená kolekcia, čo bude v niektorých prípadoch nevyhnutné. Prvky sú uložené v poli. Vyhľadávanie využíva internú mapu inicializovanú cez funkciu prepare, po zavolaní sa kolekcia stane korektnou. Potom už nesmieme vkladať alebo mazať prvky, lebo by sa interná mapa stala neplatnou alebo zneplatnili ukazovatele. Kolekcie v generátore inicializujeme počas načítania registra a neskôr z nich už len čítame.

Štruktúra BaseType

V registri sa vyskytuje niekoľko typov, ktoré dedia zo štruktúry BaseType.

```
enum class Type {
  Unknown,
  Enum,
  Struct,
  Union,
  Handle,
  Command
};
```

Každý typ má názov, ktorý by mal byť implicitne unikátny v mennom priestore (zvyčajne nekontrolujeme). Typ sa dá priradiť do Vulkan rozšírenia. Medzi typmi existuje hierarchia závislostí, ktorá sa počas načítania registra vytvorí. Daný prvok je možné individuálne povoliť alebo zakázať, pričom sa rekurzívne automaticky povolia všetky závislosti.

Štruktúra PlatformData

Dedí z BaseType. Obsahuje údaje týkajúce sa Vulkan platformy: reťazec predstavujúci direktívum, ktorý sa využíva počas generovania. Vulkan rozšírenie môže byť priradené pod konkrétnu platformu.

Štruktúra EnumValue

Dedí z BaseType. Prestavuje hodnotu Vulkan enumerácie. Jednotlivé hodnoty môžu byť tiež priradené pod Vulkan platformu.

Štruktúra EnumData

Dedí z BaseType. Obsahuje údaje týkajúce sa typu Vulkan enumerácia.

Štruktúra ExtensionData

Dedí z BaseType. Obsahuje údaje týkajúce sa Vulkan rozšírenia.

Štruktúra StructData

Dedí z BaseType. Obsahuje údaje týkajúce sa typu Vulkan štruktúra.

Štruktúra CommandData

Dedí z BaseType. Obsahuje údaje týkajúce sa typu Vulkan príkaz (funkcia).

- Návratový typ
- Parametre
- Vulkan návratové kódy v prípade úspechu

Štruktúra ClassCommandData

Predstavuje Vulkan príkaz v priradený ku konkrétnemu objektu. Podľa názvu objektu je odvodený názov príkazu.

Štruktúra HandleData

Dedí z BaseType. Obsahuje údaje týkajúce sa typu Vulkan objekt.

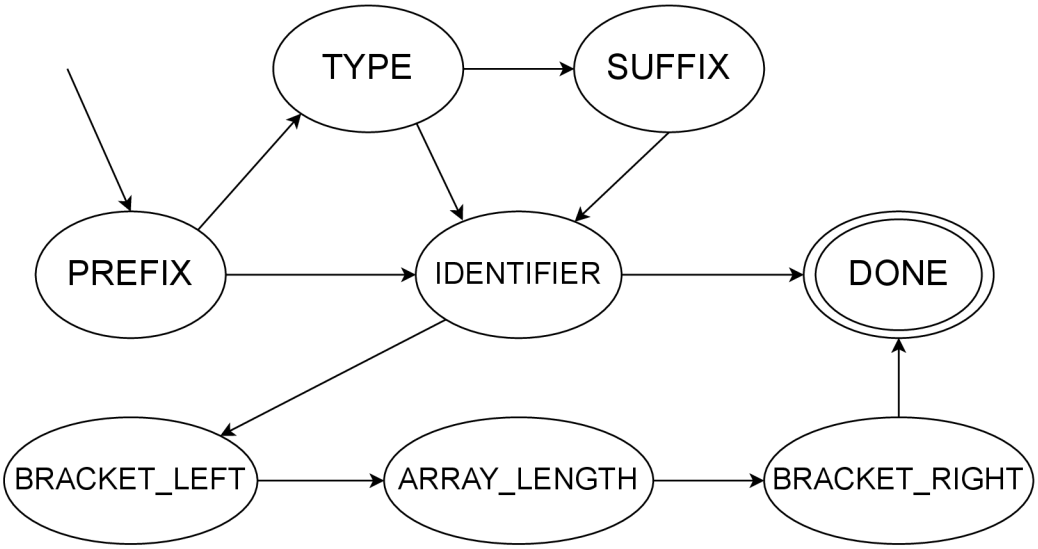
- Priamy rodičovský objekt (nie je povinný)
- Rodičovský objekt najvyššej úrovne
- Premenné
- Príkazy
- Príkazy, ktoré vytvoria daný objekt (konštruktor)
- Príkaz, ktorý uvoľní daný objekt (deštruktor)

Trieda XMLVariableParser

Pre spracovanie uzlov v registri vyjadrujúcich premenné používame triedu `XMLVariableParser`, ktorá dedí z triedy `XMLVisitor` v knižnici *tinyxml2*. Implementácia používa návrhový vzor *Visitor* [14]. Samotná logika je riadená konečným automatom na obrázku 5.2. Obsah uzlu v registri, pričom nie všetky prvky sú povinné.

```
.*<type>.+</type>.*<name>.+</name>(\[(\d+)|(<enum>.+</enum>)\])?
```

Prvá časť reprezentuje dátový typ. Nasleduje názov (identifikátor) a prípadne konštantná dĺžka pola. Uzol môže obsahovať atribút len, ktorý odkazuje na inú premennú a atribút *optional* označujúci nepovinný parameter z pohľadu Vulkan rozhrania. Činnosť spracovania uzlu je znázornená konečným automatom na obrázku 5.2.



Obr. 5.2: Konečný automat pre triedu XMLVariableParser

Trieda DependencySorter

Generický algoritmus pre zoradenie kolekcie typov. Je potrebný pre Vulkan štruktúry a triedy. Ak použijeme implicitné poradie podľa registra, narazíme počas kompilácie rozhrania na problém z dôvodu nesprávneho poradia. Niektoré prípady totiž vyžadujú definíciu typu. Implementovaný algoritmus 1 pracuje s abecedne zoradeným polom a presúva len nevyhnutné položky. Výsledné prvky sú zoradené čo najviac v abecednom poradí.

Algoritmus 1 Zoradenie závislostí

```
Vytvor pole P a naplň ho položkami
Pre každý položku v poli P:
  Nájdi závislosti
Dokým nie sú všetky prvky zoradené:
  Postupne prechádzaj pole P:
    Ak položka má splnené závislosti:
      Vlož položku do zoradeného pola
  Ak pole P nie je prázdne a nevložili sme žiadnu položku:
    Skonči chybou
```

Trieda Register

Tvorí prvú vrstvu. Je zodpovedná za načítanie vstupného registra.

Trieda Generator

Tvorí druhú vrstvu. Je zodpovedná za generovanie výstupu a správu konfigurácie, pričom využíva Register z prvej vrstvy.

5.2 Spracovanie argumentov príkazovej riadky

Hlavná funkcia programu začína spracovaním argumentov. Argumenty určujú vstupy a výstupy generátora.

- –help: vypíše nápovedu
- -r|–reg: cesta registrového súboru
- -c|–config: cesta konfiguračného súboru
- -d|–dest: cesta výstupnej zložky
- –nogui: zakáže režim užívateľského rozhrania

Uvedené argumenty nie su povinné ak je spustené užívateľské rozhranie, pretože je možné vstup a výstup nastaviť počas behu.

5.3 Načítanie Vulkan registra

Vulkan register (pozri sekcia 2.3) je načítaný ako súbor vo formáte XML pomocou knižnice *tinyxml2*.

Knižnica tinyxml2

S knižnicou tinyxml2 sa pracuje v štýle jazyka C, napriek tomu, že je kompatibilná s jazykom C++. Preto boli implementované pomocné funkcie a triedy rozširujúce rozhranie o štýl jazyka C++. Funkcie getRequiredAttrib a getAttrib vrátia atribút uzlu ako typ std::string_view. Trieda NodeContainer poskytuje štandardný iterátor a trieda ValueFilter filtruje uzly v kolekcii obdobným spôsobom ako štandardná knižnica std::ranges[1]. Použitie v generátore je následovné:

```
for (XMLElement *e : Elements(node) | ValueFilter("enum")) {
  ...
}
```

Trieda XMLDocument predstavuje XML súbor, ktorý sa načíta cez funkciu `LoadFile`. Následne spracujeme koreňové uzly (pozri sekcia 2.3) v správnom poradí.

Uzol platforms

Množinu uzlov typu platform načítame do kolekcie platforms. Jedná sa o mapovanie názvu plafromy na reťazec predstavujúci direktívum.

Uzol tags

Množinu uzlov typu tags načítame do neusporiadanej množiny tags, s ktorou pracujú pomocné funkcie pre úpravu reťazcov.

[1] https://en.cppreference.com/w/cpp/ranges

Uzol types

Množinu uzlov spracujeme podľa atribútu určujúci kategóriu. Kategória *'enum'* je spracovaná do prvku *EnumData*. Ak uzol obsahuje atribút alias, tak sa pridá k prvku na ktorý je viazaný.

Kategória *'bitmask'* je tiež spracovaná do prvku *EnumData*. Názov sa prípadne upraví, aby obsahoval príponu FlagBits. Rovnaká upráva sa vykoná aj pre atribút alias.

Kategória *'handle'* je spracovaná do prvku *HandleData*.

Kategória *'struct'* a *'union'* je spracovaná do prvku *StructData*.

Z kategórie *'define'* potrebujeme načítať jedine hodnotu pre VK_HEADER_VERSION, zvyšok sa týka rozhrania pre jazyk C.

Uzol enums

Každý uzol obsahuje hodnoty pre už deklarovanú enumeráciu. Hodnoty sa spracujú a priradia danej enumerácii.

Uzol Commands

Každý uzol predstavuje Vulkan príkaz pozostávajúci z hlavičky (prototyp) a parametrov. Uzly sú spracované do prvkov *CommandData*.

Uzol Feature

Uzol popisuje, ktoré prvky patria do určitej revízie. Podobne ako uzol *'enums'* obsahuje dodatočné hodnoty pre enumerácie.

Uzol Extensions

Každý uzol predstavuje Vulkan rozšírenie a množinu prvkov, ktoré patria do daného rozšírenia. Uzly sú spracované do prvkov *ExtensionData*.

5.4 Vrstva register

Po načítaní Vulkan špecifikácie je stav pevný a nesmie sa meniť.

Funkcie pre úpravu reťazcov

Je definovaná sada funkcií pre prevod názov vo Vulkan špecifikácií.

- –help: vypíše nápovedu
- -r|–reg: cesta registrového súboru
- -c|–config: cesta konfiguračného súboru
- -d|–dest: cesta výstupnej zložky
- –nogui: zakáže režim užívateľského rozhrania

5.5 Vrstva generátor

Počas generácie sa mení vnútorný stav.

Konfigurácia

Samotná konfigurácia je uložená v štruktúre Config. Každá položka je obalená štruktúrou ConfigWrapper, ktorá zaistí logiku import a export operácií.

Konfigurácia obsahuje prevažne množinu nastavení typu bool. Zvyšné nastavenia sa týkajú určitých direktív z Vulkan rozhrania. Informácia o samostatnom direktíve je uložená v štruktúre Macro.

Konfigurácia sa dá exportovať a importovať do XML súboru cez funkcie loadConfigFile a saveConfigFile. Import a export je implementovaný pomocou reflexie, Čo má za následok jednoduché pridávanie ďalších nastavení. Súčasťou XML súboru môže byť uzol whitelist určujúci množinu povolených prvkov Vulkan rozhrania na úrovniach (uzly):

- funkcia `strStripPrefix`: odstráni predponu
- funkcia `strStripSuffix`: odstráni príponu
- funkcie `camelToSnake` a `snakeToCamel`: prevedie štýl
- funkcie `enumConvertCamel` a `snakeToCamel`: prevedie hodnotu enumerácie

Inštanciu konfigurácie spravuje trieda generator.

Funkcia format

Funkcia `format` poskytuje špeciálne formátovanie reťazca pre potreby generátora. Je inšpirovaná štandardnou knižnicou `std::format` [2] zo štandardu C++20. Výhoda vlastnej implementácie je spätná kompatibilita so starším štandardom. Navyše poskytuje náhradu vybraných kľúčových slov za direktíva.

Zdrojový kód 5.1: Deklarácia funkcie format

```
template <class... Args>
std::string Generator::format(const std::string &format, const Args&&... args)
  const;
```

Funkcie pre generáciu kódu

Generátor intenzívne využíva lambda funkcie[3] volané cez funkciu `genOptional`. Týmto spôsobom je automaticky zabezpečené obalenie príslušným direktívom. Vypnuté časti kódu jednoducho nezavolajú lambda funkciu.

Zdrojový kód 5.2: Deklarácia funkcie genOptional

```
std::string genOptional(const BaseType &type,
                        std::function<void(std::string &)> function) const;
```

[2] https://en.cppreference.com/w/cpp/utility/format/format
[3] https://en.cppreference.com/w/cpp/language/lambda

Kapitola 6

Výsledky, meranie a zhodnotenie

6.1 Skript

V tejto práci je metrika doba kompilácie, čím nižšie tým lepšie. Keďže testujeme mnoho konfigurácií budeme potrebovať automatizovaný skript. Skript je napísaný v jazyku Python.

Potrebné nástroje:

- Python 3.10.2+ [1]
- CMake 3.26.3+ [2]
- Ninja 1.11.1+ [3]
- vcperf 2.2.22080401+ [4]
- VulkanSDK 1.3.239.0+ [5]

Testované kompilátory:

- Clang 16.0.1
- MSVC 19.35.32215

Adresárová štruktúra

V koreňovej zložke sa nachádza skript run.py a zložka src obsahujúca projekty pre kompiláciu. Každý projekt môže navyše obsahovať konfiguračné súbory. Nové rozhranie je generované buď v globálnej ceste *${VulkanSDK}/Include/gen-vulkan20* alebo lokálne v projektovej zložke podľa potreby.

Konfigurácia prostredia

Skript po spustení s prepínačom –**configure** vytvorí pre každý projekt zložku pripravenú na kompiláciu.

[1] https://www.python.org/downloads/
[2] https://cmake.org/download/
[3] https://ninja-build.org/
[4] https://github.com/microsoft/vcperf
[5] https://vulkan.lunarg.com/sdk/home

Algoritmus 2 Konfigurácia

```
Pre každý režim kompilácie:
  Pre každý kompilátor:
    Vytvor zložku
    Spusti príkaz 'cmake' s konkrétnymi parametrami
```

Kompilácia a zber údajov

Kompilácia sa spustí univerzálnym príkazom *cmake –build* ako podproces v jazyku Python. Zmeraním času trvania podprocesu získame celkovú dobu kompilácie, avšak aj s celou réžiou. Výsledok je potom skreslený a je ťažké s ním pracovať.

Preto potrebujeme presnejšiu metódu. Našťastie niektoré moderné kompilátory sú schopné exportovať súbor s podrobnou štatistikou. Obsahuje čas spracovania hlavičkových súborov, presne to čo potrebujeme. Z tohoto dôvodu boli zvolené spomenuté kompilátory, nakoľko u ostatných nie sme schopný zmysluplne získať štatistiky.

Kompilátor clang podporuje prepínač *-ftime-trace*, ktorý vytvorí vo výslednej zložke súbor formátu json (pre každú prekladovú jednotku).

Kompilátor msvc podporuje podobnú funkcionalitu cez externý nástroj *vcperf*. Tento nástroj funguje len v administrátorskom režime a zbiera štatistiky v celom systéme. Pred spustením kompilácie je nutné začať zber. Po skončení kompilácie ukončiť zber, čo následne vytvorí súbor formátu json. Ak sa z dôvodu chyby alebo vnúteného ukončenia skriptu neukončí zber tak sa automaticky znovu nespustí. Preto na začiatku skriptu prebieha reset.

Algoritmus 3 Kompilácia všetkých projektov

```
Rekurzívne prechádzaj zložku 'build'
  Ak je zložka pripravená pre kompiláciu:
    Ak odpovedajúca zdrojová zložka obsahuje konfigurácie:
      Pre každú konfiguráciu:
        Vygeneruj rozhranie
        Spusti kompiláciu a zber
    Inak:
        Spusti kompiláciu a zber
```

Algoritmus 4 Kompilácia projektu a zber

```
Vykonaj N krát:
  Vyčisti cieľovú zložku
  Ak je kompilátor 'MSVC':
    Spusti nástroj 'vcperf'
  Spusti príkaz 'cmake -build .'
  Ak je kompilátor 'MSVC':
    Ukonči nástroj 'vcperf'
  Ulož získané hodnoty
Zo získaných hodnôt vypočítaj agregované údaje
```

Každá kompilácia projektu je vykonaná N krát a zo získaných časov sa vypočíta medián. Predvolená hodnota je 25 meraní, no je možné ju zmeniť prepínačom **–samples N**. Získané údaje sa uložia v koreňovej zložke do súborov **results.json** a **results.csv**.

Spracovanie údajov

Predpokladá sa, že každá zdrojová zložka musí obsahovať zdrojový súbor main.cpp. Tým pádom kompilátor clang po dokončení kompilácie vygeneruje v zložke súbor main.cpp.json. Tento súbor obsahuje predovšetkým dobu kompilácie pre hlavičkové súbory. Skript prečíta informácie zo súboru a uloží relevantné údaje podľa filtra popísanom v algoritme 5. Súborov s dobou kompilácie nižšou ako 50ms je veľký počet, čo by bolo neprehľadné v tabuľke. Preto ich filtrujeme a zaujímajú nás tzv. horúce miesta (súbory s najvyššou dobou kompilácie).

Algoritmus 5 Filtrovanie údajov

```
Ak je názov súboru z projektu:
  Ulož výsledok
Ak je názov súboru z Vulkan rozhrania:
  Ulož výsledok a uprav názov súboru
V ostatných prípadoch:
  Ak je doba kompilácie menšia ako 50ms:
    Ulož výsledok
  Inak:
    Zahoď výsledok
```

Skript vyrobí z dostupných údajov tabuľku vo forme `csv` súboru. Každá konfigurácia má v prvom stĺpci označenom 'n' počet vzorkov (celkový počet vykonaných kompilácií). Druhý stĺpec označený 'name' predstavuje jednoznačný názov konfigurácie vo formáte <režim optimalizácie>-<názov kompilátora>-<názov projektu>, kde režim optimalizácie: je skratka D pre *debug* a skratka R pre *release*. Ak projekt používa viac ako jeden konfiguračný súbor, tak je uvedený v názve. Tretí stĺpec označený 'total' predstavuje celkovú dobu kompilácie v sekundách. Ostatné stĺpce predstavujú jednotivé súbory alebo fázy kompilátora získane z `json` súboru.

6.2 Merané konfigurácie

Prázdny projekt

Odmeriame dobu kompilácie úplne prázdneho projektu za účelom zistenia réžie celého procesu kompilácie.

Zdrojový kód 6.1: Prázdny program

```
// main.cpp
int main() {}
```

Zdrojové zložky:

- empty

Vytvorené konfigurácie:

- D-clang-empty
- D-msvc-empty

Oficiálne rozhranie

Odmeriame dobu kompilácie oficiálneho rozhrania ako referenčný bod. Zdrojový súbor zahŕňa oficiálne rozhranie vrátane RAII.

Zdrojový kód 6.2: Referenčný program

```
// api.hpp
#pragma once
// optional vulkan API directives go here
#include <vulkan/vulkan_raii.hpp>
// main.cpp
#include "api.hpp"
int main() {}
```

Výsledky nového rozhrania porovnáme oproti referencii.
Zdrojové zložky:

- **reference**
- **reference-no-constructors**
- **reference-no-enhanced**
- **reference-no-setters**

Nové rozhranie

Projekt zahŕňa nové rozhranie namiesto oficiálneho rozhrania, generované v zložke gen-vulkan20.

Zdrojový kód 6.3: Program s novým Vulkan rozhraním

```
// api.hpp
#pragma once
// optional vulkan API directives go here
#include <gen-vulkan20/vulkan20_raii.hpp>

// main.cpp
#include "api.hpp"
int main() {}
```

Zdrojové zložky:

- **default**: Obsahuje konfigurácie:
  - **default.xml**: predvolená základná konfigurácia
  - **internal-funcs.xml**: varianta s internými funkciami v rozhraní
  - **minimal.xml**: varianta s minimálnou funkcionalitou pre projekt
  - **no-allocator.xml**: varianta bez parametra pre alokátor
  - **no-dispatch.xml**: varianta bez parametra pre dispatch

- **experimental**: v projekte je definovaná sada direktív pre odstránenie nepotrebnej funkcionality (hlavne sa jedná o časti kódu s podozrením na vysoký vplyv na dobu kompilácie).
  Celý zoznam direktív bude uvedený v sekcii 6.4.
- **hex**: alternatívna implementácia prevodu na reťazec (spomenuté v sekcii 3.1).
  Projekt definuje direktívum *VULKAN_HPP_EXPERIMENTAL_HEX*
- **interop**: rozšírená funkcionalita medzi rozhraniami.
  Projekt definuje direktívum *VULKAN_HPP_EXPERIMENTAL_INTEROP*
- **local**: rovnaká konfigurácia ako default, ale súbory sú v lokálnej zložke
- **no-compare**: vypnuté operátory pre porovnávanie. Projekt definuje direktíva:
  - *VULKAN_HPP_EXPERIMENTAL_NO_STRUCT_COMPARE*
  - *VULKAN_HPP_EXPERIMENTAL_NO_CLASS_COMPARE*
- **no-funcs**: vypnuté Vulkan funkcie v hlavnom rozhraní (vrátane chytrých objektov). Projekt definuje direktíva:
  - *VK_NO_PROTOTYPES*
  - *VULKAN_HPP_NO_SMART_HANDLE*
  - *VULKAN_HPP_EXPERIMENTAL_NO_VK_FUNCS*
- **no-chain**: vypnutá funkcionalita StructureChain. Projekt definuje direktívum *VULKAN_HPP_EXPERIMENTAL_NO_STRUCT_CHAIN*.
- **no-templates** čiastočne vypnutá funkcionalita spojená s inštanciovaním template. Projekt definuje direktívum *VULKAN_HPP_EXPERIMENTAL_NO_TEMPLATES*.
- **no-traits**: vypnutá funkcionalita FlagTraits. Projekt definuje direktívum *VULKAN_HPP_EXPERIMENTAL_NO_FLAG_TRAITS*.

Aplikácia

Účelom je testovať dobu kompilácie v zložitejšom projekte. Je odmeraná doba kompilácie celej aplikácie, ktorá je popísaná v sekcii 6.4. Navyše je meraná implementácia aplikácie v C++20 moduloch.

Zdrojové zložky:

- **example**: ukážková aplikácia
- **modules**: ukážková aplikácia s Vulkan rozhraním ako modul
- **modules-full**: ukážková aplikácia so samotným kódom aj rozhraním v moduloch

Testované konfigurácie:

- **full.xml**: východzia konfigurácia rozhrania
- **experimental.xml**: nové rozhranie s experimentálnymi zmenami
- **project-minimized.xml**: rozhranie minimalizované pre špecifickú aplikáciu

6.3 Vyhodnotenie výsledkov

Kompletná tabuľka je vložená v prílohe A.

Kompilátor clang je rýchlejší o zhruba 70% (rozdiel 0,6 sekundy) v kompilácií prázdneho súboru. Pravdepodobne zapríčinené rýchlejším linkovacím nástrojom. Tento čas tvorí dodatočnú réžiu v stĺpci *total*.

Oficiálne rozhranie

Prehľad nameraných časov je v tabuľke 6.1.

name	total	api. hpp	vk20_ raii	vk20. hpp	vk.h	std libs
D-clang-reference	5,006	4,686	4,686	2,514	0,069	1,068
D-clang-reference-no-constructors	4,827	4,506	4,506	2,321	0,064	1,06
D-clang-reference-no-setters	4,867	4,544	4,544	2,353	0,063	1,049
D-clang-reference-no-enhanced	2,784	2,486	2,486	2,075	0,074	1,055

Tabuľka 6.1: Súhrnná tabuľka nameraných výsledkov pre oficiálne rozhranie

Vypnutie konštruktorov cez direktíva
VULKAN_HPP_NO_STRUCT_CONSTRUCTORS
a *VULKAN_HPP_NO_UNION_CONSTRUCTORS* zrýchlilo kompiláciu o 4%.

Zhruba rovnaký rozdiel bol odmeraný pre vypnutie setter funkcií cez direktíva
VULKAN_HPP_NO_STRUCT_SETTERS
a *VULKAN_HPP_NO_UNION_SETTERS*.

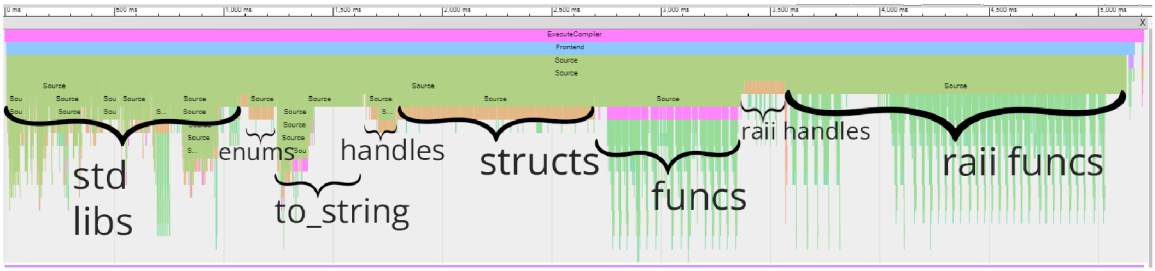
Na druhú stranu najväčší vplyv malo vypnutie rozšírených funkcií v rozhraní cez direktívum *VULKAN_HPP_DISABLE_ENHANCED_MODE*. Doba kompilácie sa znížila skoro o 50% (rozdiel 2,15 sekúnd) pre kompilátor clang a skoro o 30% (rozdiel 1,4 sekundy) pre kompilátor MSVC. Dôvod je veľmi pravdepodobne použitie template funkcionality v rozšírených funkciách.

Zapnutie reflexie cez direktívum *VULKAN_HPP_USE_REFLECT* predĺžilo dobu kompilácie až o 185%, preto túto funkcionalitu neodporúčame používať.

Nové rozhranie

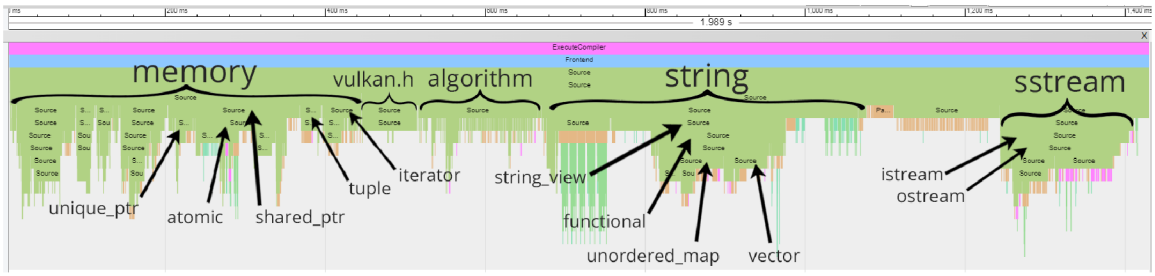
Webové prehliadače na báze chrómia vedia vizualizovať informácie o kompilácií vytvorené kompilátormi (popísané v sekcii 6.1). Zadaním url `chrome://tracing/` do adresového riadku sa dostaneme na internú stránku, na ktorej otvoríme `.json` súbor cez tlačidlo `load`.

Na obrázku 6.1 je graf jednej kompilácie konfigurácie default. RAII rozhranie je veľmi náročné na kompiláciu. Z hlavného rozhrania sú to štruktúry a funkcie. Taktiež kompilátor strávil nemalú dobu na štandardných knižniciach.



Obr. 6.1: Vizualizácia doby kompilácie (snímok 1)

Na obrázku 6.2 je priblíženie ľavej časť grafu z obrázku 6.1. Zobrazené sú podstatné štandardné knižnice. Väčšina z nich je pre Vulkan rozhranie jazyka C++ nevyhnutná. No zároveň sú implicitne zahrnuté nepotrebné časti, ale to nemôžeme ovplyvniť.



Obr. 6.2: Vizualizácia doby kompilácie (snímok 2)

Prehľad nameraných časov je v tabuľke 6.2.

name	total	api. hpp	vk20_ raii	vk20. hpp	vk.h	std libs
D-clang-default	4,883	4,559	4,558	2,621	0,063	1,039
D-clang-experimental	3,912	2,523	2,523	1,122	0,053	0,881
D-clang-no-enhanced	3,223	2,919	2,918	1,916	0,066	1,022
D-clang-no-enhanced-exp	2,435	2,111	2,111	1,156	0,053	0,873
D-clang-default(minimal.xml)	2,544	2,244	2,244	1,325	0,063	1,041
D-clang-minimal-exp	1,806	1,515	1,515	0,747	0,053	0,889

Tabuľka 6.2: Súhrnná tabuľka nameraných výsledkov pre nové rozhranie

Doba kompilácie nového rozhrania je v predvolenej konfigurácií nižšia o necelé 2,5% (rozdiel 0,1 sekundy pre kompilátor clang) oproti referencii.

Zahrnutie rozhrania z lokánej zložky projektu predĺžilo dobu kompiacie o 2,5%(rozdiel 0,1 sekundy pre kompilátor clang). Je to pravdepodobne zapríčinené súborovým systémom.

V konfigurácií `experimental` bola definovaná sada direktív pre vypnutie nepotrebnej funkcionality. Doba kompilácie sa znížila o 20% (rozdiel 1 sekunda pre kompilátor clang). Doba kompilácie hlavného rozhrania klesla až o 55%.

V konfigurácií `no-enhanced` vypnutie rozšírených funkcií zrýchlilo dobu kompilácie o 33% (rozdiel 1,6 sekundy pre kompilátor clang).

V konfigurácií `no-enhanced-exp` boli spojené `experimental` a `no-enhanced`. Doba kompilácie sa nížila o 50% (rozdiel 2,4 sekundy pre kompilátor clang).

Konfigurácia `default(minimal.xml)` demonštruje odstránenie zbytočného kódu pre špecifický projekt. Maximálne možné zrýchlenie týmto spôsobom je skoro o 50%.

Konfigurácia `minimal-exp` kombinuje `default(minimal.xml)` a `experimental`. Doba kompilácie sa nížila o 65% (rozdiel 3 sekundy pre kompilátor clang). Dosiahli sme minimum medzi testovanými konfiguráciami. Polovicu doby kompilácie už začínajú tvoriť štandardné knižnice, ktorých sa nemôžeme zbaviť bez drastických zmien v rozhraní pre jazyk C++.

V konfigurácií `no-hex` alternatívna implementácia urýchlila dobu kompilácie o 3%(rozdiel 0,2 sekundy pre kompilátor clang). Pre ostatné neuvedené konfigurácie bol rovnako rozdiel pár %.

Aplikácia

Prehľad nameraných časov je v tabuľke 6.3.

name	total	api .hpp	vk20 .hpp	vk20_ raii	vk.h	std libs
D-clang-example(full.xml)	10,018	4,625	2,275	4,617	0,085	1,555
D-clang-example(experimental.xml)	8,938	3,555	1,398	3,546	0,078	1,241
D-clang-example(minimal.xml)	6,626	2,478	0,895	2,474	0,09	1,264

Tabuľka 6.3: Súhrnná tabuľka nameraných výsledkov pre apliká̆ciu

Konfigurácia `experimental.xml` s vypnutou nepotrebnou funkcionalitou znížila dobu kompilácie aplikácie o 11% (rozdiel 1,1 sekundy pre kompilátor clang).

Konfigurácia minimalizovaná špecificky pre aplikáciu `minimal.xml` skrátila dobu kompilácie rozhrania o 47% (rozdiel 2,2 sekundy pre kompilátor clang). Celkový čas kompilácie aplikácie klesol o 35% (rozdiel 3,4 sekundy).

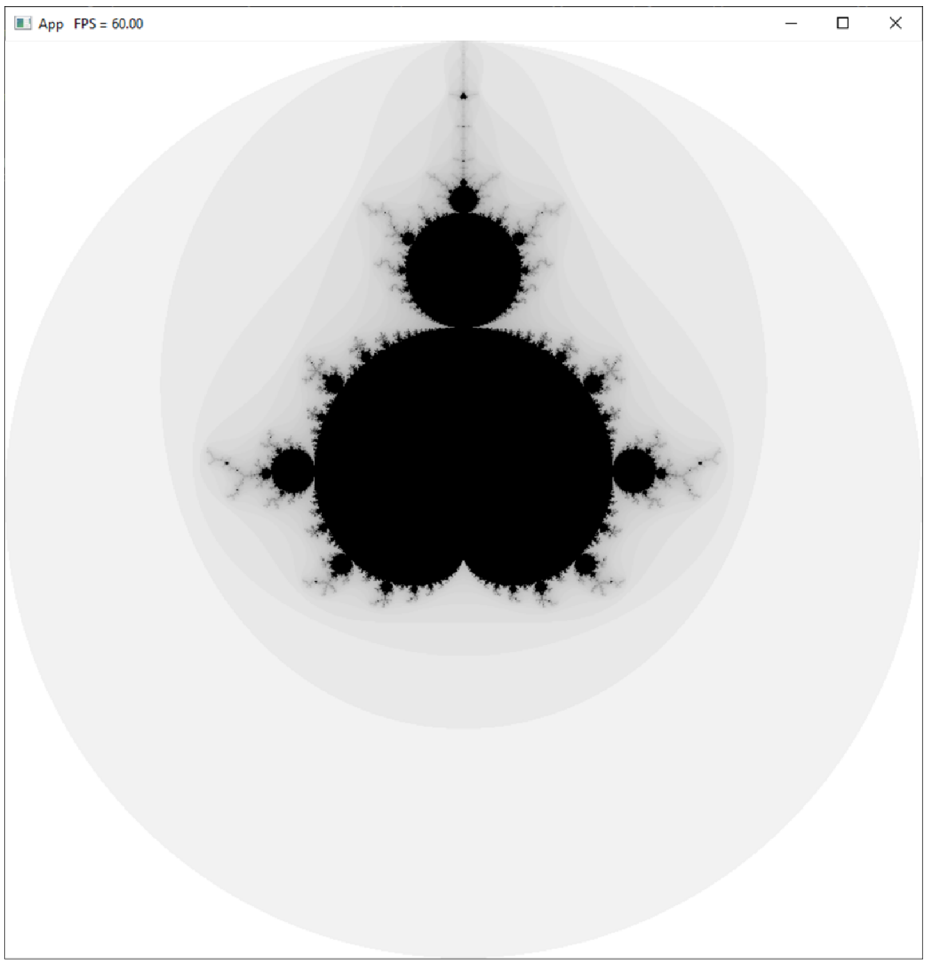
6.4 Ukážková aplikácia

Ukážková aplikácia demonštruje použiteľnosť a funkčnosť nového rozhrania. Implementácia vychádza z tutoriálu od pána Ing. Jana Pečivy Ph.D na webstránke root.cz [6]. Jedná sa o program pre vykreslenie Julia množiny (Mandelbrotovej množiny)[7]. Pôvodný kód je dostupný na GitHube [8]. Aplikácia je upravená pre RAII variantu rozhrania. Využíva knižnicu GLFW pre vytvorenie grafického okna. Aplikácia je rozdelená do súborov:

- main.cpp: hlavný súbor zodpovedný za chod programu
- app.cpp: vykresľovanie a logika programu
- vulkanwindow.cpp: správa grafického okna
- vulkaninstance.cpp: správa objektu Vulkan Instance
- vulkanphysical.cpp: správa objektu Vulkan PhysicalDevice
- vulkandevice.cpp: správa objektu Vulkan Device

[6] https://www.root.cz/serialy/tutorial-vulkan/
[7] https://users.math.yale.edu/public_html/People/frame/Fractals/MandelSet/welcome.html
[8] https://github.com/pc-john/VulkanTutorial/tree/main/15-julia

Aplikácia je súčasťou meraní pre skript, tiež sa dá manuálne zostaviť aj v zložke build cez príkazový riadok. Pre manuálne zostavenie sú poskytnuté súbory configure a compile.



Obr. 6.3: Ukážková aplikácia

Zvolená konfigurácia rozhrania

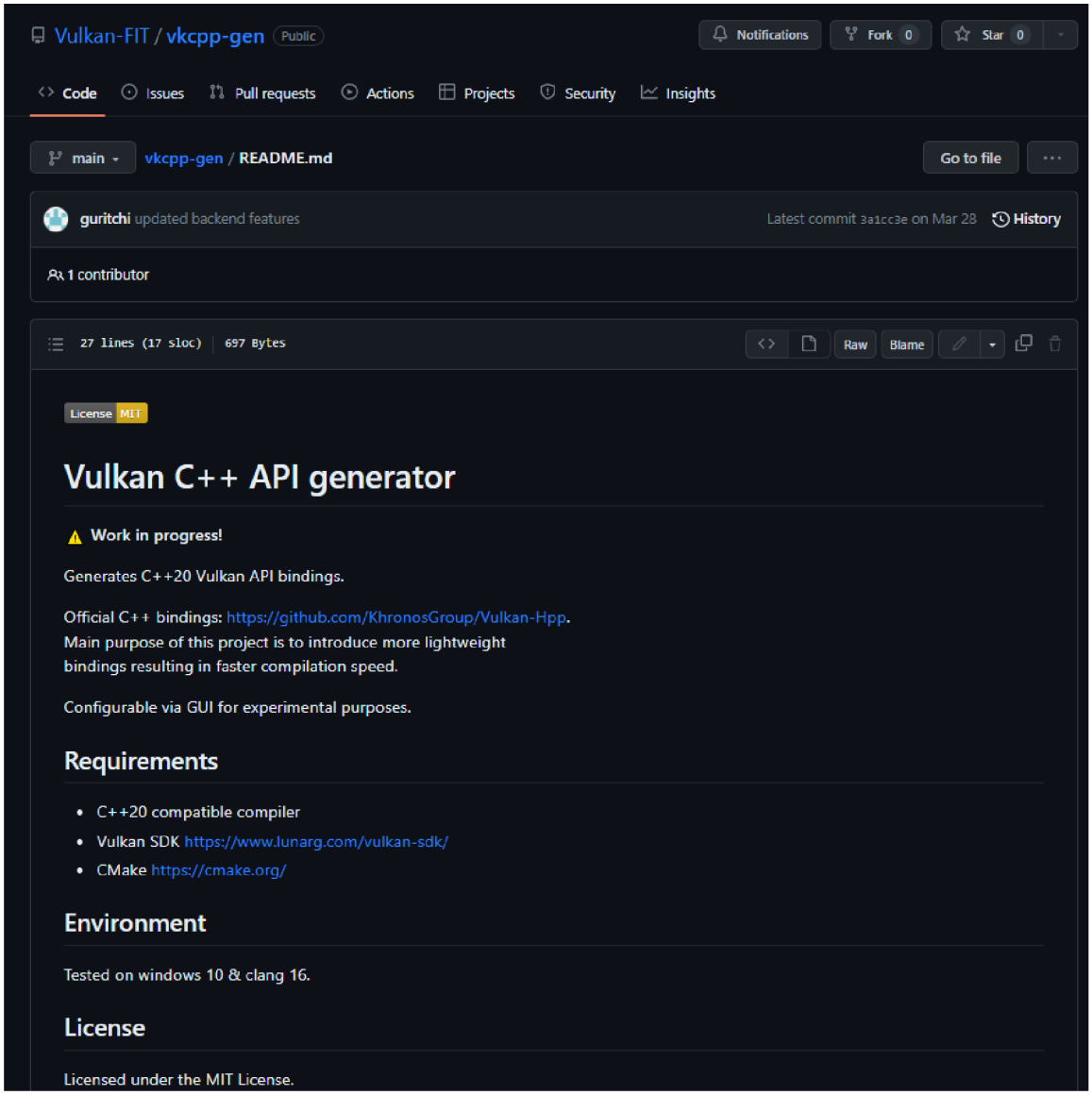
Implementácia aplikácie kladie dôraz na využitie PFN funkcií cez nepriame linkovanie (sekcia 2.1), ktoré sú dostupné cez RAII rozhranie. Nevyhnutné sú len objekty Context, Instance a Device, ktoré spravujú tabuľky funkčných ukazovateľov (pozri sekcia 2.6). Vulkan funkcie sú tak isto volané len cez objekty Instance a Device s novým direktívom *VULKAN_HPP_EXPERIMENTAL_NO_INDIRECT_CMDS*.
Ostatné Vulkan objekty je možno využiť z Vulkan rozhrania s novým direktívom *VULKAN_HPP_EXPERIMENTAL_INTEROP*.
Štruktúry využívajú agregovanú inicializáciu (sekcia 3.1), pre ktorú je nutné definovať direktíva *VULKAN_HPP_NO_STRUCT_CONSTRUCTORS*
a *VULKAN_HPP_NO_UNION_CONSTRUCTORS*.

Pre ušetrenie času kompilácie je vypnutá nepotrebná funkcionalita pre projekt cez následovné direktíva:

- *VK_NO_PROTOTYPES*: pozri sekcia 2.4
- *VULKAN_HPP_NO_UNION_SETTERS*: pozri sekcia 3.1
- *VULKAN_HPP_NO_STRUCT_SETTERS*: pozri sekcia 3.1
- *VULKAN_HPP_NO_SMART_HANDLE*: pozri sekcia 3.1
- *VULKAN_HPP_EXPERIMENTAL_NO_FLAG_TRAITS*: pozri sekcia 3.1
- *VULKAN_HPP_EXPERIMENTAL_NO_TEMPLATES*: pozri sekcia 3.1
- *VULKAN_HPP_EXPERIMENTAL_NO_CLASS_COMPARE*: pozri sekcia 3.1
- *VULKAN_HPP_EXPERIMENTAL_NO_VK_FUNCS*: pozri sekcia 3.1
- *VULKAN_HPP_EXPERIMENTAL_NO_STRUCT_COMPARE* pozri sekcia 3.1
- *VULKAN_HPP_EXPERIMENTAL_NO_STRUCT_CHAIN*: pozri sekcia 3.1
- *VULKAN_HPP_EXPERIMENTAL_HEX*: pozri sekcia 3.1

6.5 Repozitár

Implementovaný generátor je zverejnený vo verejnom repozitári na GitHube[9] pod open-source licenciou MIT[10] (pozri obrázok 6.4).



Obr. 6.4: Repozitár projektu

[9] https://github.com/Vulkan-FIT/vkcpp-gen
[10] https://opensource.org/license/mit/

Kapitola 7

Záver

Rozhranie pre Vulkan je využívané pre programovanie aplikácií komunikujúcich s grafickou kartou. Práca prezentuje upravené Vulkan rozhranie pre jazyk C++ dosahujúci lepšiu dobu kompilácie.

Vulkan rozhranie pre jazyk C, ktoré je veľmi jednoduché, trvá skompilovať len desiatky milisekúnd. Problém nastáva v použití rozhrania pre jazyk C++, ktoré je robustné rozšírenie nad rozhraním pre jazyk C. Kompilácia trvá niekoľko sekúnd. V testoch sme namerali 2,6 sekúnd oproti 0,065 sekundy – až 40-násobne dlhší čas. RAII verzia rozhrania prináša výhody, ale zároveň rozširuje rozhranie o ďalšie množstvo kódu a tým pádom trvá ešte dlhšie skompilovať. Namerali sme 4,6 – až 70-násobne dlhší čas.

V tejto práci bolo navrhnuté a implementované rozhranie pre Vulkan so zámerom znížiť dobu kompilácie. Ako bolo spomenuté v sekcii 3.4, pre potreby práce bol implementovaný generátor rozhrania[1], čomu bola venovaná veľká časť práce. Rovnako ako samotné rozhranie pre Vulkan, tak aj súčasný generátor vyvíjaný konzorciom Khronos sa rýchlo mení. Momentálna verzia generátora stále poskytuje priestor pre inováciu. Generátor je zverejnený pod open-source licenciou (pozri sekcia 6.5).

Bol vytvorený skript v programovacom jazyku Python pre automatické meranie doby kompilácie sady projektov (konfigurácií), pomocou ktorého boli získané výsledky. Skript je schopný, vďaka podpore kompilátorov, poskytnúť dobu kompilácie na úrovni jednotlivých súborov. Postupnou analýzou boli nájdené súbory s najdlhšou dobou a zvážené kroky pre zlepšenie a opätovné meranie.

Generátor poskytuje tvorbu konfigurácie, pričom niektoré nastavenia zásadne zmenia vygenerované rozhranie a porušia kompatibilitu s terajším kódom. Ostatné nastavenia menia rozhranie len interne a zavádzajú nové direktíva pre vypnutie funkcionality.

Vulkan rozhranie sa bohužiaľ nedá významne vylepšiť bez obetovania funkcionality. Väčšina konfigurácií priniesla zlomkové zlepšenie doby kompilácie okolo 3%. Niektoré konfigurácie sa ale dajú kombinovať a začnú prispievať k nižšej dobe kompilácie.

Dobrá správa je, že ak sme ochotný vypnúť funkcionalitu môžeme dosiahnuť viditeľné zlepšenie kompilácie. Konfigurácia *experimental* bez mnohých nadbytočnej funkcionality zrýchlila dobu kompilácie Vulkan rozhrania o minimálne 20%. Ak sme dodatočne vypli rozšírené funkcie, tak klesla doba kompilácie Vulkan rozhrania až o 50%.

Ešte lepší výsledok priniesli zmeny vypínajúce nepotrebné prvky vo Vulkan rozhraní. Tento prístup je vytváranie konfigurácie minimalizovanej pre projekt. Stačí zoznam použitých príkazov a generátor odstráni všetky nepoužité prvky v rozhraní. Takéto rozhranie

[1] https://github.com/Vulkan-FIT/vkcpp-gen

je potom zahrnuté v lokálnej zložke projektu. Vytvorením minimálnej konfigurácie pre aplikáciu používajúcu množinu príkazov nevyhnutnú pre vykresľovanie bola znížená doba kompilácie skoro o 50%.

V kombinácií s konfiguráciu *experimental* sme dosiahli zrýchlenie kompilácie Vulkan rozhrania o 65%, čo je momentálny limit. Kompilácia demonštračnej aplikácie sa zrýchlila o 47%.

Budúci smer vývoja

Čiastočným vyhnutím sa funkcionality jazyka C++ bolo dosiahnuté rozhranie s podstatne nižšou dobou kompilácie. Avšak stále je priestor na zlepšenie. Nové experimentálne rozhranie, ktoré by už naisto porušovalo kompatibilitu s doterajším (užívateľským) aplikačným kódom, by teoreticky mohlo priniesť ešte výraznejšie zlepšenie doby kompilácie. Čím bližšie je rozhranie k jazyku C, tým rýchlejší čas uvidíme.

Už len samotné štandardné knižnice zaberú 15-násobne dlhšie oproti Vulkan rozhraniu pre jazyk C. Avšak C++20 moduly môžu zlepšiť situáciu. Bohužiaľ momentálna verzia kompilátoru *clang* neukázala zlepšenie. S novým štandardom C++23[2] budú štandardné knižnice modularizované.

Ako bolo spomenuté, oficiálne rozhranie a generátor sa neustále vyvíjajú. Preto je potrebné vlastný generátor udržiavať a aktualizovať. Implementácia Vulkan rozhrania v moduloch je stále v počiatkoch.

[2] https://en.cppreference.com/w/cpp/23

Literatúra

[1] Bjarne Stroustrup, H. S. *C++ Core Guidelines* [online]. 13. apríla 2023 [cit. 2023-4-27]. Dostupné z: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.

[2] EINHORN, E. *Vulkan Fan? Six Reasons to Run It on NVIDIA* [online]. 25. januára 2022 [cit. 2023-4-17]. Dostupné z: https://blogs.nvidia.com/blog/2022/01/25/vulkan-nvidia/.

[3] Fertig, A. *C++20 Modules: The possible speedup* [online]. 7. septembra 2021 [cit. 2023-5-2]. Dostupné z: https://andreasfertig.blog/2021/09/cpp20-modules-the-possible-speedup/.

[4] Grimm, R. *The Next Big Thing: C++20* [online]. 18. októbra 2019 [cit. 2023-5-2]. Dostupné z: http://www.modernescpp.com/index.php/c-20-an-overview.

[5] Haines, E. a Akenine Möller, T. *Ray Tracing Gems*. 1. vyd. Apress Berkeley, CA, 2019. ISBN 978-1-4842-4426-5.

[6] Jon Leech, T. H. *Vulkan® Documentation and Extensions: Command Names* [online]. 13. apríla 2023 [cit. 2023-4-17]. Dostupné z: https://registry.khronos.org/vulkan/specs/1.3/styleguide.html#_command_names.

[7] Jon Leech, T. H. *Vulkan® Documentation and Extensions: Enumerant Names* [online]. 13. apríla 2023 [cit. 2023-4-17]. Dostupné z: https://registry.khronos.org/vulkan/specs/1.3/styleguide.html#_enumerant_names.

[8] Jon Leech, T. H. *Vulkan® Documentation and Extensions: Function Pointer Type Names* [online]. 13. apríla 2023 [cit. 2023-4-17]. Dostupné z: https://registry.khronos.org/vulkan/specs/1.3/styleguide.html#naming-funcpointers.

[9] Jon Leech, T. H. *Vulkan® Documentation and Extensions: Preprocessor Defines* [online]. 13. apríla 2023 [cit. 2023-4-17]. Dostupné z: https://registry.khronos.org/vulkan/specs/1.3/styleguide.html#naming-preprocessor.

[10] Jon Leech, T. H. *Vulkan® Documentation and Extensions: Procedures and Conventions* [online]. 13. apríla 2023 [cit. 2023-4-17]. Dostupné z: https://registry.khronos.org/vulkan/specs/1.3/styleguide.html.

[11] Jon Leech, T. H. *Vulkan® Documentation and Extensions: Type Names* [online]. 13. apríla 2023 [cit. 2023-4-17]. Dostupné z: https://registry.khronos.org/vulkan/specs/1.3/styleguide.html#_type_names.

[12] Kernighan, B. W. a Ritchie, D. M. *C Programming Language, 2nd Edition*. 2. vyd. Pearson, 1988. ISBN 0-13-110362-8.

[13] Mivelli, D. *Analyzing and Reducing Compilation Times for C++ Programs*. Sweden, SE, 2022. Master's thesis. Linköping University. Dostupné z: https://www.diva-portal.org/smash/get/diva2:1671584/FULLTEXT01.pdf.

[14] Rogowski, K. *Design Patterns: The Visitor Pattern* [online]. 9. júna 2022 [cit. 2023-4-26]. Dostupné z: https://softwarehut.com/blog/tech/design-patterns-visitor-pattern.

[15] Stroustrup, B. *The C++ Programming Language, First Edition*. Addison-Wesley, 1986. ISBN 0-201-12078-X.

[16] Stroustrup, B. *Zero-overhead principle* [online]. 25. januára 2023 [cit. 2023-4-16]. Dostupné z: https://en.cppreference.com/w/cpp/language/Zero-overhead_principle.

[17] Süssenbach, A. *Preferring Compile-time Errors over Runtime Errors with Vulkan-hpp* [online]. 16. októbra 2020 [cit. 2023-4-15]. Dostupné z: https://developer.nvidia.com/blog/preferring-compile-time-errors-over-runtime-errors-with-vulkan-hpp/.

[18] The Khronos® Vulkan Working Group. *Application Interface to Loader* [online]. 27. marca 2023 [cit. 2023-4-17]. Dostupné z: https://vulkan.lunarg.com/doc/view/1.3.243.0/windows/LoaderApplicationInterface.html.

[19] The Khronos® Vulkan Working Group. *Architecture of the Vulkan Loader Interfaces* [online]. 27. marca 2023 [cit. 2023-4-17]. Dostupné z: https://vulkan.lunarg.com/doc/sdk/1.3.243.0/windows/LoaderInterfaceArchitecture.html.

[20] The Khronos® Vulkan Working Group. *Vulkan-Hpp: C++ Bindings for Vulkan* [online]. 27. marca 2023 [cit. 2023-4-19]. Dostupné z: https://github.com/KhronosGroup/Vulkan-Hpp/blob/main/README.md#namespace-vk.

Príloha A

Tabuľky meraní

Význam jednotlivých stĺpcov je popísaný v sekcii 6.1.

name	total
D-clang-empty	0,237
D-msvc-empty	0,87

name	total
D-clang17-modules(full.xml)	11,213
D-clang17-modules(minimal.xml)	9,797
D-clang17-modules-full(full.xml)	18,811
D-clang17-modules-full(minimal.xml)	15,369

name	total	vk_ enums	vk_ funcs	vk_ handles	vk_ structs	std libs
D-msvc-reference	5,952	0,192	0,259	0,119	1,118	0,111
D-msvc-reference-no-constructors	5,206	0,117	0,263	0,118	0,832	0,083
D-msvc-reference-no-setters	5,408	0,119	0,256	0,115	0,996	0,087
D-msvc-reference-no-enhanced	3,929	0,122	0,07	0,076	0,917	0,082

name	total	api .hpp	vk20 .hpp	vk20_ enums	vk20_ handles	vk20_ raii	vk20_raii_ funcs	vk20_ structs	vk20_to_ string	vk.h	std libs
D-clang-example(full.xml)	10,018	4,625	2,275	0,187	0,09	4,617	1,171	0,862	0,577	0,085	1,555
D-clang-example(experimental.xml)	8,938	3,555	1,398	0,036	0,038	3,546	1,051	0,427	0,377	0,078	1,241
D-clang-example(minimal.xml)	6,626	2,478	0,895	0,019	0,019	2,474	0,526	0,082	0,225	0,09	1,264

name	total	vk20_ enums	vk20_ funcs	vk20_ raii_ funcs	vk20_ smart	vk20_ structs	std libs
D-msvc-default	5,291	0,117	0,137	0,443	0,07	1,044	0,097
D-msvc-default(internal-funcs.xml)	5,277	0,112	0,102	0,451	0,083	1,05	0,081
D-msvc-default(no-allocator.xml)	5,218	0,113	0,128	0,417	0,105	1,051	0,081
D-msvc-default(no-dispatch.xml)	5,41	0,112	0,224	0,404	0,086	1,042	0,082
D-msvc-local	5,288	0,114	0,136	0,439	0,084	1,03	0,085
D-msvc-interop	5,231	0,116	0,137	0,392	0,083	1,048	0,082
D-msvc-hex	4,524	0,116	0,131	0,433	0,068	1,099	0,083
D-msvc-no-chain	5,516	0,123	0,144	0,456	0,075	1,078	0,087
D-msvc-no-templates	5,446	0,122	0,144	0,46	0,073	1,045	0,084
D-msvc-no-traits	5,5	0,011	0,142	0,444	0,073	1,063	0,101
D-msvc-no-compare	4,993	0,117	0,142	0,446	0,078	0,639	0,093
D-msvc-no-setters	5,273	0,115	0,14	0,449	0,078	0,977	0,083
D-msvc-no-constructors	5,338	0,117	0,167	0,467	0,074	0,94	0,086
D-msvc-no-funcs	5,161	0,119				1,05	0,084
D-msvc-no-enhanced	4,418	0,12	0,021	0,229		1,075	0,085
D-msvc-no-enhanced-exp	2,691	0,008		0,298		0,262	0,083
D-msvc-experimental	3,432	0,008		0,494		0,258	0,082
D-msvc-default(minimal.xml)	3,239	0,051	0,056	0,133	0,045	0,268	0,09
D-msvc-minimal-exp	2,222	0,006		0,083		0,048	0,1

name	total	api.hpp	vk20.hpp	vk20_enums	vk20_funcs	vk20_handles	vk20_raii	vk20_raii_funcs	vk20_smart	vk20_structs	vk20_to_string	vk.h	std libs
D-clang-default	4,883	4,559	2,621	0,113	0,608	0,126	4,558	1,382	0,073	0,787	0,351	0,063	1,039
D-clang-default(no-allocator.xml)	4,843	4,515	2,597	0,113	0,596	0,117	4,514	1,361	0,069	0,795	0,358	0,062	1,055
D-clang-default(no-dispatch.xml)	5,048	4,594	2,632	0,112	0,631	0,113	4,594	1,403	0,068	0,799	0,362	0,063	1,05
D-clang-default(internal-funcs.xml)	4,866	3,498	2,215	0,113	0,208	0,126	3,498	0,731	0,072	0,801	0,369	0,062	1,041
D-clang-interop	4,816	4,494	2,618	0,113	0,609	0,128	4,494	1,314	0,072	0,794	0,36	0,062	1,055
D-clang-hex	4,748	4,424	2,476	0,116	0,604	0,126	4,424	1,38	0,072	0,829	0,217	0,061	0,874
D-clang-local	5,187	4,863	2,667	0,131	0,615	0,138	4,863	1,42	0,079	0,891	0,393	0,069	1,089
D-clang-no-chain	4,879	4,556	2,611	0,113	0,615	0,129	4,556	1,38	0,074	0,807	0,36	0,063	1,042
D-clang-no-compare	4,683	4,358	2,408	0,112	0,612	0,121	4,358	1,378	0,071	0,602	0,358	0,063	1,051
D-clang-no-funcs	4,667	4,347	1,903	0,114		0,04	4,347	1,892		0,805	0,363	0,055	1,047
D-clang-no-constructors	4,817	4,48	2,538	0,116	0,608	0,125	4,479	1,377	0,072	0,716	0,357	0,061	1,065
D-clang-no-setters	4,76	4,424	2,472	0,113	0,61	0,129	4,423	1,387	0,071	0,656	0,362	0,063	1,047
D-clang-no-templates	4,87	4,535	2,598	0,114	0,616	0,121	4,535	1,375	0,071	0,778	0,371	0,062	1,049
D-clang-no-traits	4,844	4,515	2,563	0,022	0,602	0,131	4,514	1,384	0,077	0,81	0,394	0,066	1,039
D-clang-no-enhanced	3,223	2,919	1,916	0,111	0,054	0,038	2,918	0,454		0,783	0,356	0,066	1,022
D-clang-no-enhanced-exp	2,435	2,111	1,156	0,02		0,025	2,111	0,407		0,314	0,258	0,053	0,873
D-clang-experimental	3,912	2,523	1,122	0,021		0,032	2,523	0,824		0,29	0,245	0,053	0,881
D-clang-default(minimal.xml)	2,544	2,244	1,325	0,055	0,197	0,072	2,244	0,431	0,044	0,16	0,283	0,063	1,041
D-clang-minimal-exp	1,806	1,515	0,747	0,013		0,009	1,515	0,263		0,053	0,143	0,053	0,889

name	total	api.hpp	vk20.hpp	vk20_enums	vk20_funcs	vk20_handles	vk20_raii	vk20_raii_funcs	vk20_smart	vk20_structs	vk20_to_string	vk.h	std libs
D-clang-reference	5,006	4,686	2,514	0,113	0,358	0,098	4,686			0,875	0,4	0,069	1,068
D-clang-reference-no-constructors	4,827	4,506	2,321	0,112	0,36	0,094	4,506			0,69	0,407	0,064	1,06
D-clang-reference-no-setters	4,867	4,544	2,353	0,111	0,357	0,098	4,544			0,734	0,403	0,063	1,049
D-clang-reference-no-enhanced	2,784	2,486	2,075	0,113	0,074	0,051	2,486			0,79	0,407	0,074	1,055