Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Vlastní vykreslovací engine pracující v reálném čase

2024

Martin Šlachta

Vedoucí práce:
Mgr. Markéta Trnečková, Ph.D.

Studijní program: Informatika,
Specializace: Obecná informatika

Bibliografické údaje

Autor:	Martin Šlachta
Název práce:	Vlastní vykreslovací engine pracující v reálném čase
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2024
Studijní program:	Informatika, Specializace: Obecná informatika
Vedoucí práce:	Mgr. Markéta Trnečková, Ph.D.
Počet stran:	34
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Martin Šlachta
Title:	Real-time rendering engine
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2024
Study program:	Computer Science, Specialization: General Computer Science
Supervisor:	Mgr. Markéta Trnečková, Ph.D.
Page count:	34
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Tato práce se zaměřuje na vykreslovací enginy, které jsou dnes kritické pro filmovou produkci nebo herní průmysl. Ukazuje způsoby možnosti akcelerace a jaké problémy to přináší. Je představeno řešení problému synchronizace a jeho implementace v ukázkovém vykreslovacím enginu.

Synopsis

This thesis focuses on rendering engines, which are critical for today's film production and game industry. It demostrates methods of acceleration and challenges they present. A solution to problem of synchronization is presented along with it's implementation in a sample rendering engine.

Klíčová slova: Vulkan; vykreslovací engine; frame graf; render graf; synchronizace

Keywords: Vulkan; rendering engine; frame graph; render graph; synchronization

Děkuji paní Mgr. Markétě Trnečkové, Ph.D. za věnovaný čas a přátelům, kteří mě podporovali.

Odevzdáním tohoto textu jeho autor místopřísežně prohlašuje, že celou práci včetně příloh vypracoval samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

Seznam tabulek

1 Úvod

Vykreslovací enginy v reálném čase představují klíčovou technologii v oblasti počítačové grafiky, která umožňuje vytváření interaktivních prostředí. Tyto enginy jsou široce využívány v herním průmyslu, virtuální realitě, filmovém průmyslu a simulacích. Jejich hlavní předností je schopnost generovat obraz v reálném čase, což umožňuje uživatelům okamžitou interakci s virtuálním světem. Jenom herní průmysl se dlouhodobě stává velkou částí zábavního průmyslu. V poslední době je také velmi populární virtuální realita pro hry nebo vzdělávání. Podle Entertainment software association[1] v Americe hraje hry alespoň jednou týdně více jak polovina američanů všeho věku. V České republice generuje herní průmysl každoročně vyšší zisky a vytváří spoustu nových pracovních míst, jak informuje Asociace českých herních vývojářů [2]. Mezi populární vykreslovací enginy, které jsou součástí herních enginů, patří například CryEngine, Unreal Engine a Unity.

Cílem práce je představit vykreslovací enginy, jak fungují dnes a jaké problémy řeší. V praktické části potom vytvořit ukázkový vykreslovací engine, který bude načítat objekty ze souboru a vykreslovat je v reálném čase.

Práce je rozdělena do pěti kapitol a doplněna o vlastní ilustrace pro lepší pochopení problematiky.

Druhá kapitola slouží jako úvod do problematiky. Představuje základní pojmy týkající se vykreslovacího enginu a metody vykreslování, které budeme využívat v následujících kapitolách.

Ve třetí kapitole akcelerujeme výpočty pro vykreslování pomocí grafické karty. Kapitola představuje různá rozhraní, které pro akceleraci existují. Především si představíme rozhraní Vulkan API. Ukážeme si problém synchronizace, který vzniká pokud máme vícero operací, které mohou běžet souběžně. V takovém případě je potřeba definovat závislosti pomocí synchronizačních objektů.

Ve čtvrté kapitole se podíváme na render graf, který řeší zmíněný problém synchronizace. Vytvoříme si implementaci render grafu speciálně pro Vulkan API. Render graf je populární řešení ve velkých vykreslovacích enginech, protože vykreslování dokáže snadno upravit. Implementace bude schopna synchronizovat libovolné operace. Řekneme, co k tomu budeme potřebovat.

V páté kapitole vytvoříme základní ukázkovou aplikaci s vykreslovacím enginem. Zároveň ověříme funkčnost naší implementace render grafu.

2 Vykreslovací engine

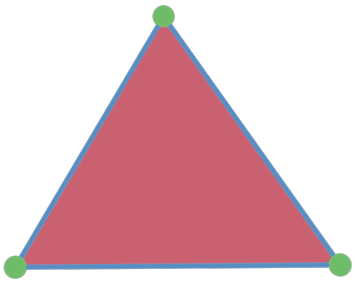
Tato kapitola se zabývá základními pojmy týkající se vykreslování a vykreslovacích enginů. Vykreslovací engine je software, který zobrazuje v počítači reprezentované objekty do obrázku nebo přímo do roviny monitoru. Vykreslovací enginy se tak používají například v CAD softwarech, ve videohrách nebo pro filmové efekty. Vykreslovací engine je většinou součástí většího systému, například herního enginu. Herní engine je sada nástrojů určených pro tvorbu her. V této kapitole budeme čerpat ze zdrojů: [3], [4], [5] a [6].

2.1 Vstupní data

Uvedeme si vstupní data pro vykreslovací engine. V zásadě se jedná o reprezentaci 3D objektu v počítači. Každý typ reprezentace má své výhody a nevýhody. Uvedeme si nejpoužívanější reprezentaci pro vykreslování objektů v reálném čase.

2.1.1 Polygonální síť

Pro vykreslování v reálném čase 3D objekty nejčastěji reprezenuje tzv. *polygonální síť*, dále jen *síť*. Ta popisuje pouze porvch objektu a skládá se z tzv. *polygonů*. Ten je tvořen *vertexy*, *hranami* a *plochou*. Vertex si můžeme představit jako bod v 3D prostoru a mezi dvěma vertexy může vést hrana. Pokud spojíme alespoň tři vertexy hranami, které nejsou na stejné přímce, vznikne plocha, která představuje výplň prostoru mezi hranami. Na obrázku 1 vidíme zeleně znázorněné vertexy, modře hrany a červeně plocha.



Obrázek 1: Ilustrace polygonu

Pro přesnější reprezentaci (především zakřivených povrchů) je třeba použít více polygonů. Na obrázku 2 vidíme dvě polygonální sítě, přičemž oběma se snažíme reprezentovat velmi zakřivený povrch. Nalevo je síť s nižším počtem polygonů a napravo s vyšším počtem polygonů. Síť nalevo má velmi ostré hrany a v tomto konkrétním případě tak špatně reprezentuje zakřivený povrch. Síť napravo obsahuje více polygonů a daleko lépe reprezentuje zakřivený povrch.

Je třeba dát pozor na výpočetní náročnost vykreslování, protože každý přidaný polygon ji zvyšuje.

(a) Polygonální síť s málo polygony (b) Polygonální síť s více polygony

Obrázek 2: Polygonální sítě

Ukážeme si, jak polygonální síť reprezentovat v počítači. Prozatím budeme definovat vertex jako pozici v 3D euklidovském prostoru tj. vektorem o třech složkách. Můžeme vytvořit list vertexů, do kterého pro každý polygon vložíme vertexy, které ho tvoří, vedle sebe. List vertexů sám o sobě nestačí pro reprezentaci polygonů. Potřebujeme ještě znát počet vertexů, které polygon tvoří. Pro získání všech polygonů z listu bychom začali na začátku, přečetli právě tolik vertexů, kolik jich polygon má, a v tomto pořadí, v jakém jsou v listu, je spojovali hranami do ploch. Poté bychom se posunuli na další a stejně pokračujeme, dokud nedojdeme na konec listu. Nevýhoda tohoto způsobu reprezentace je, že ukládá vícekrát vertex, který je pro více polygonů společný. Proto vytvoříme druhý list, který bude tvořen indexy do listu vertexů. Pro získání polygonu pak přečteme indexy jeho vertexů. To nám umožní společný vertex uložit jen jednou a vícekrát jej indexovat. Reprezentaci ještě vylepšíme a omezíme se pouze na trojuhelníky. Díky tomu si nemusíme ukládat, kolik má který polygon vertexů. Pro vykreslení polygonální sítě nám stačí jen list vertexů, list indexů a počet polygonů, které list indexů reprezentuje. Toto omezení nám nevadí, protože i grafická karta, kterou si představíme ve třetí kapitole, pracuje výhradně s trojuhelníky.

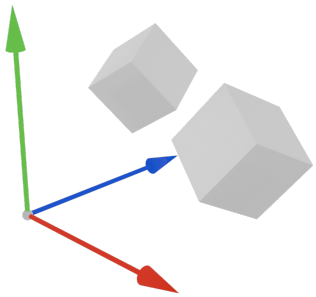
2.2 Textury

Více polygonů znamená více náročných výpočtů. Abychom 3D objektům dodali malé detaily, které jsou pro realističnost kritické, použijeme tzv. *textury*. To je společně obrázek a funkce, zobrazující pozici (většinou dvourozměrnou) na bod na polygonální síti. Tato funkce slouží k promítnutí obrázku na 3D povrch objektu.

2.3 Scéna

Často je třeba polygonálních sítí vykreslit více, například dvě vedle sebe, jako vidíme na obrázku 3. Struktura, která spravuje více polygonálních sítí, se nazývá *scéna*. Pokud bychom chtěli vykreslit dva stejné tvary, každý posunutý do jiné

pozice, bylo by nutné mít tuto síť uloženou dvakrát, přičemž každá by měla posunuté všechny vertexy. Úspornějším řešením je každý vertex transformovat až při vykreslování. Transformaci definujeme maticí afinní transformace. Scéna bude reprezentovaná stromovou strukturou, kde vrchol je *objekt*. Každý objekt má *lokální transformaci* a může mít přiřazenou polygonální síť. Před rasterizací projdeme strom scény od kořene a pokud objekt obsahuje polygonální síť, aplikujeme *globální transformaci* objektu a následně ji rasterizujeme. Globální transformaci získáme aplikováním lokální transformace na globální transformaci rodiče. Pokud se jedná o kořen, je globální transformace stejná jako lokální transformace. To způsobí, že transformace objektu transformuje stejným způsobem i všechny jeho potomky. Tímto způsobem má scéna jasně danou hierarchii a umožňuje přiřadit jednu polygonální síť vícekrát, aniž by byla vícekrát uložena v paměti.



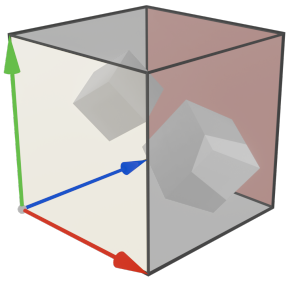
Obrázek 3: Ilustrace objektů ve scéně

2.3.1 Projekce

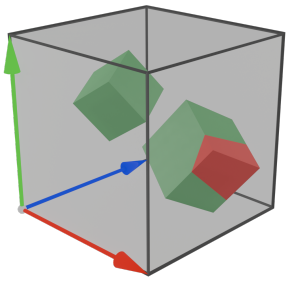
V této části si ukážeme, jak promítnout scénu do 2D prostoru tak, abychom ji mohli zobrazit do výsledného obrázku. Zatím nevíme, co a jak přesně vykreslit. Proto si nejprve zadefinujeme tzv. *frustum*. Ten se skládá ze dvou ploch: *přední plocha* a *zadní plocha*. Frustum pak tvoří prostor mezi nimi, který promítneme do výsledného obrázku. Na obrázku 4a vidíme frustum a žlutě zvýrazněnou přední plochu a červeně zadní plochu.

Všechny vertexy promítneme na přední stěnu, ale ještě před tím ořízneme části objektů, které nejsou ve frustum. Na obrázku 4b vidíme červeně vyznačené části objektů, které se odřežou a zeleně objekty, které nám po oříznutí vzniknou. Po oříznutí můžeme promítnout objekty na přední plochu, jako vidíme na obrázku 5a.

Výsledný obrázek tak bude vypadat jako na obrázku 5b. Na obrázku stále vidíme červenou a zelenou šipku, které jsou čistě ilustrativní a slouží pro lepší orientaci v prostoru.

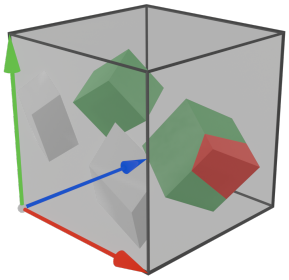


(a) Ilustrace frustum

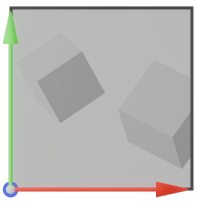


(b) Ilustrace oříznutých objektů

Obrázek 4: Ilustrace projekce 1



(a) Ilustrace promítnutých objektů



(b) Ilustrace finálního prostoru obrázku

Obrázek 5: Ilustrace projekce 2

2.4 Kamera

Ve scéně se chceme volně pohybovat. Vytvoříme proto *transformaci kamery*, která bude simulovat kameru a kterou všechny vertexy před vykreslením transformujeme. Pro pohyb ve scéně je ale třeba si uvědomit, že transformace kamery se musí transformovat opačně, než je pohyb kamery, tzn. pro transformaci kamery dozadu musíme transformovat transformaci kamery dopředu.

2.5 Výstupní data

Výstup vykreslovacího enginu je zpravidla *obrázek*. Obrázek představuje matici s dimenzí $w \times h$, kde w je *šířka* a h *výška* obrázku. Prvku matice říkáme *pixel*, který by měl být z předem určené množiny, které budeme říkat *formát*.

2.6 Snímek

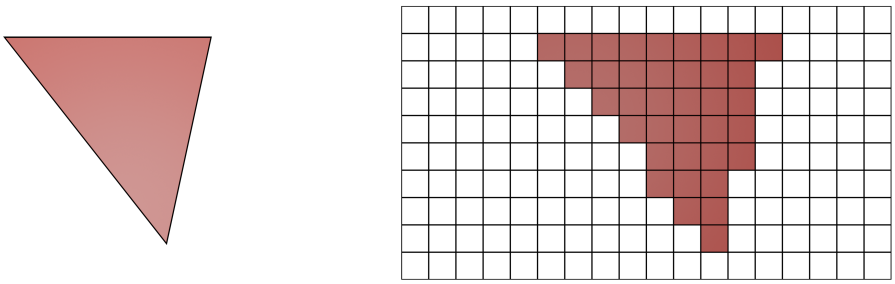
Veškeré výpočty, které jsou potřeba k vykreslení jednoho obrázku, se nazývají *snímek*. Známý může být pro čtenáře pojem „počet snímků za sekundu“, který nám říká, kolik snímků jsme schopni za jednu sekundu vypočítat.

Následující slovní spojení „engine pracující v reálném čase“ znamená, že engine dokáže vypočítat a zobrazovat snímky takovou rychlostí, že při zobrazování

nejnovějšího snímku na monitoru, připomíná plynulý pohyb. Pro lidské oko je potřeba alespoň 24 snímků za sekundu, ale pro hry je ideální alespoň 60 snímků za sekundu.

2.7 Metody vykreslování do obrázku

Představíme si, co to znamená, vykreslit něco do obrázku. Na obrázku 6 vidíme nalevo polygon, který chceme vykreslit. Napravo pak vidíme žádaný obrázek, kde každému pixelu je přiřazena správná barva podle toho, jestli jej polygon protíná.



(a) Polygon (b) Polygon zobrazený do obrázku

Obrázek 6: Síť

Tento proces se skládá z více fází. Nejprve je potřeba převést 3D pozice vertexů z euklidovského prostoru scény do *normalizovaných souřadnic zařízení*. Je více způsobů, jak toho docílit. Uvedeme si dva konkrétní způsoby: *ray tracing* a *rasterizaci*.

2.7.1 Ray tracing

Ray tracing je metoda zobrazení scény do obrázku, která „střílí“ paprsky od obrazovky do scény. Nejprve si vytvoříme obrázek. Poté zadefinujeme projekci, transformaci kamery a objekty ve scéně. Následně pro každý pixel vytvoříme paprsek, pošleme ho směrem do scény a hledáme objekt a *bod nárazu*. Pokud paprsek narazil na více objektů, vybereme ten objekt, který má bod nárazu nejblíže obrazovce. Nakonec by měla implementace přiřadit pixelu hodnotu. Základní implementace by mohla na pixel, ze kterého jsme paprsek vyslali, nastavit barvu, kterou má objekt v bodu nárazu. V případě, že paprsek na žádný objekt nenarazil, nastaví v daném pixelu černou barvu. Výhodou je, že implementace může po nárazu paprsek odrazit a zjistit na co následně narazil. Tímto můžeme simulovat odraz světla v reálném světě a výsledek vypadá daleko realističtěji. Nevýhodou je, že takový postup je výpočetně náročný. Ray tracing je velmi používaný ve filmové produkci, kde čas není velké omezení a realističnost důležitá.

2.7.2 Rasterizace

Nejčastěji používaná metoda vykreslování v reálném čase je tzv. *rasterizace*. Při rasterizaci pro každý pixel zjišťujeme, který polygon ho překrývá. Na rozdíl od

raytracingu ale není možné pro tento bod sbírat další informace o scéně, jako to umožňoval ray tracing odražením paprsku. Tento proces je také výpočetně náročný. Proto vznikly tzv. *grafické akcelerátory*, dnes nazývané *grafické karty*, které jsou schopné vykreslovat daleko rychleji. Z toho důvodu, je velmi vhodný pro vykreslovací enginy pracující v reálném čase. Dnes jsou grafické karty optimalizované i pro jiné výpočty, například vědecké, a nebo právě ray tracing.

3 Grafická karta

V této kapitole si uvedeme, jak můžeme využít grafickou kartu pro akceleraci rasterizace a podobných výpočtů. Podíváme se, jaké máme možnosti programování na grafické kartě a na rozdíly od programování na centrálním procesoru. Dále budeme pro grafickou kartu používat zkratku GPU a pro centrální procesor zkratku CPU.

3.1 Ovladače

Ovladač nám poskytuje rozhraní, tzv. *grafické API*, pomocí které můžeme akcelerovat rasterizaci a jiné výpočty. Ovladač abstrahuje konkrétní firmware na grafické kartě a poskytuje jednotné rozhraní.

3.2 Grafická API

K dispozici máme několik grafických API, a to OpenGL, Vulkan, DirectX a Metal. Ovladač ale zpravidla nepodporuje všechny, proto je třeba podle specializace vykreslovacího enginu vybrat tu správnou.

DirectX je dostupné pouze na operačních systémech Windows. To je dostačující například pro herní vykreslovací enginy, protože většina hráčů používá platformu Windows. Navíc herní konzole XBox je také platforma Windows.

Metal je dostupný výhradně na hardwaru od společnosti Apple. Na platformě Apple je menší počet hráčů, takže vykreslování v reálném čase je zde spíše zaměřeno na grafický software, například pro modelování.

OpenGL je nejstarší grafické API, které vzniklo ještě předtím, než existovaly grafické karty. To způsobilo velký technologický dluh, protože se OpenGL nebylo schopné adaptovat na nový a jinak fungující hardware. Navíc má velmi abstraktní rozhraní, které neumožňuje tolik optimalizací. Přesto je OpenGL vhodné třeba pro *hardwarovou akceleraci*, tedy využití grafické karty pro vykreslování uživatelského rozhraní a efektů v něm.

3.3 Vulkan API

Pro akceleraci rasterizace použijeme Vulkan API rozhraní, protože umožňuje do velkých detailů definovat jednotlivé operace. Ovladač má daleko více informací, které může využít pro lepší optimalizace. Starší API tyto detaily nevyžadovaly a ovladač byl nucen hodně optimalizací dělat heuristicky. Tato vlastnost je u vykreslovacích enginů pracujících v reálném čase velmi žádaná. V této kapitole budeme čerpat z [7].

Vývoj Vulkan API započala společnost AMD pod pracovním názvem Mantle. Poté vývoj převzalo koncorcium Khronos Group, které je zodpovědné i za OpenGL, OpenCL a mnoho dalších standardů ze světa grafických akcelerací. Má tedy s vývojem grafických API bohaté zkušenosti.

Vulkan API je multiplatformní, lze ho tedy použít na Windows, Linux a nebo Android. Fungování grafické karty na stolním a mobilním zařízení nebo herní konzoli se ale dost liší a API je musí podporovat všechny s jednotným rozhraním a zároveň s ne moc velkou abstrakcí. To má za následek občas neintuitivní rozhraní. Vulkan API proto podporuje tzv. *rozšíření*, což jsou nadstandardní rozšíření API, například pro úpravu rozhraní nebo novou funkcionalitu. Rozšíření mohou vydávat členové koncorcia a výrobci grafických karet je mohou, ale nemusí, podporovat. Proto je nutné před spuštěním ověřit, zda uživatelské zařízení veškerá potřebná rozšíření podporuje.

Pracovat s Vulkan budeme pomocí Vulkan objektů. Hlavním takovým objektem je *instance*. Ta realizuje spojení mezi naší aplikací a Vulkan knihovnou. Dále je třeba vybrat konkrétní grafickou kartu, kterou zařízení poskytuje. To obnáší i kontroly, zda grafická karta podporuje veškerou funkcionalitu, kterou aplikace vyžaduje.

3.3.1 Zdroje ve Vulkan

Vulkan podporuje dva typy zdrojů: buffery a obrázky. Zdroje představují pohledy na paměť s přiřazeným formátováním. Buffer reprezentuje lineární pole struktur, ke kterému můžeme přistupovat během rasterizace na grafické kartě pomocí tzv. *přiřazování*. Obrázky představují vícerozměrná pole dat. Před použitím je třeba zdrojům přiřadit paměť. V grafické pipeline, o které se dozvíme v další kapitole, můžeme nadefinovat množinu vstupů a výstupů, ke které před rasterizací přiřadíme zdroj.

Existují dva typy paměti: *hostitelská* a *na zařízení*. Hostitelská paměť je dostupná jak z GPU, tak z CPU a používá se především pro ukládání dat, která se často mění nebo jsou čtena z CPU. Paměť zařízení je viditelná pouze z GPU, je mnohem rychlejší pro přístup z GPU, a proto se snažíme co nejvíce dat ukládat právě tam. Je důležité si uvědomit, že dvěma různým zdrojům lze přiřadit stejnou paměť, což se nazývá *paměťový aliasing*.

3.3.2 Rasterizace ve Vulkan

Hlavním účelem pro použití grafické karty je akcelerace rasterizace. Pro ovládání rasterizace ve Vulkan API, potřebujeme vytvořit Vulkan objekty *renderpass*, *framebuffer* a *grafickou pipeline*.

Render Pass
: specifikuje počet vůstupů rasterizace a jak budou vypadat. Běžně je třeba mít render passů více, protože výsledný obrázek skládáme postupně aplikací různých efektů. Zároveň jeden obrázek často nestačí, protože si chceme o scéně uložit více informací. Těmto výstupům budeme dále říkat *attachmenty*.

Framebuffer
: každému attachmentu z render passu přiřazuje zdroj. Pro jeden render pass

můžeme mít více framebufferů a provést rasterizaci do různých zdrojů.

3.3.3 Grafická Pipeline

Grafická pipeline přímo ovlivňuje rasterizaci. Pro správné fungování potřebuje mít předem určený render pass a při použití určený framebuffer. Grafickou pipeline si více přiblížíme.

Je to objekt, který se skládá z *shaderů* a *fixních fází*, které jdou postupně po sobě. Shader je programovatelná fáze, která umožňuje transformovat data z předchozí fáze podle potřeby a poslat je dál. Definuje své vstupní a výstupní proměnné, které slouží pro posílání dat mezi jednotlivými shadery v grafické pipeline nebo může být ke vstupní proměnné přiřazen zdroj. Shader může číst ze vstupních a výstupních proměnných a také zapisovat do výstupních proměnných. Fixní fáze nám umožňuje upravit konfiguraci grafické pipeline pouze pomocí atributů.

3.3.4 Vertex Input

Ve Vulkan API není vertex definován pouze jako pozice, ale jako n-tice atributů, kde každý reprezentuje informaci o vertexu. První krok je fixní fáze jménem *vertex input*, která čte atributy z vertexů ve vertex bufferu a přiřazuje je jako vstupní proměnné ve *vertex shaderu*. Příkladem atributu může být pozice, normála nebo barva. V tabulce 1 vidíme tabulku vertexů s dvě atributy: „pozice“ a „barva“ s ilustrativními hodnotami.

Vertexy					
Pozice			Barva		
X_1	Y_1	Z_1	R_1	G_1	B_1
X_2	Y_2	Z_2	R_2	G_2	B_2
...			...		

Tabulka 1: Ilustrativní tabulka vertexů.

Při vykreslování je ale třeba mít tabulku reprezentovanou v souvislém bloku paměti, který pak fixní fáze *vertex input* čte. Této fázi tak musíme nastavit správnou konfiguraci tak, aby souvislý blok přečetla správně. Příklad reprezentace tabulky 1 v souvislém bloku paměti je znázorněna v tabulce 2.

Vertex 1						Vertex 2						...
Pozice 1			Barva 1			Pozice 2			Barva 2			...
X_1	Y_1	Z_1	R_1	G_1	B_1	X_2	Y_2	Z_2	R_2	G_2	B_2	...

Tabulka 2: Ilustrativní reprezentace tabulky vertexů optimalizovaně.

Tato reprezentace ale není vhodná, pokud by některá grafická pipeline nepoužívala atribut „barva“, protože by čtecí hlava musela tento atribut přeskakovat. Lepší je ve vertex bufferu nejprve souvisle uložit všechny pozice a až poté souvisle uložit barvy. Poté budeme mít dvě přiřazení, první bude ukazovat, kde začínají pozice a druhá, kde začínají barvy. Ilustrativní příklad této reprezentace je v tabulce 3

Vertex 1						...	Vertex 2						...
Pozice 1			Pozice 2			...	Barva 1			Barva 2			...
X_1	Y_1	Z_1	X_2	Y_2	Z_2	...	R_1	G_1	B_1	R_2	G_2	B_2	...

Tabulka 3: Ilustrativní reprezentace tabulky vertexů.

Takovému souvislému bloku paměti budeme říkat *vertex buffer*. Pro vertex input definujeme jak číst vertex buffer pomocí lokací a přiřazení. Přiřazení určuje místo ve vertex bufferu, kde má vertex input začít číst a velikost bloku. Je definováno pouze tím, jak veliké bloky popisuje. Lokace určuje, jak konkrétně číst daný blok. Lokace je konkrétně definována přiřazením, kterému náleží, počtem a velikostí prvků, které čte a odsazením od začátku bloku. V případě tabulky 2 by přiřazení bylo pouze jedno, a ukazovalo na začátek. Lokace pro toto přiřazení by byly dvě, první by měla tři prvky, každý o velikost čtyři byty, a odsazení nula. Druhá by také měla tři prvky, každý o velikost čtyři byty, a odsazení 12 bytů.

3.3.5 Vertex shader

Vertex shader má na vstupu vstupní a výstupní proměnné a aplikuje se na každý vertex, který vertex input přečte. Mezi vstupními proměnnými jsou takové, které mají podle lokace přiřazené některý vertex atribut. Účelem shaderu je nastavit pozici vertexu v normalizovaném prostoru obrázku. V tomto shaderu se tedy aplikuje transformace, transformace kamery a projekce.

3.3.6 Rasterizer

Fixní fáze, ve které můžeme ovlivnit rasterizaci, se nazývá *rasterový stav*. Můžeme nastavit atributy jako *culling*, kterým můžeme ovlivnit, jestli vykreslovat i polygony orientované od kamery. Takové polygony uživatel na daném snímku neuvidí, a tím pádem není třeba je posílat do dalšího kroku pro ušetření výpočetního výkonu. Také lze nastavit způsob, jak se polygony mají zobrazovat. Existují tři způsoby: „výplň“, „body“ a „hrany“. Ve většině případů budeme používat mód „výplň“. Mód „body“ vykreslí pouze pozice vertexů jako tečky a mód „hrany“ vykreslí jen hrany polygonů. Tyto dva módy slouží například k diagnóze.

3.3.7 Fragment Shader

Shader, který se aplikuje na každý pixel (fragment) v rasterovém obrázku, se nazývá *fragment shader*. Na vstupní proměnné jsou přiřazené hodnoty daného

fragmentu, na který se shader aplikoval. Při vykreslování polygonálních sítí se fragment shader používá pro výpočet množství odraženého světla, pro simulaci stínování nebo pro načítání hodnot z textur. Tento shader je mimo jiné schopný získat i hodnoty z jiných fragmentů. Tato vlastnost se využívá hlavně pro tzv. *post-processing* efekty, které tak umožňují efekt rozmazání apod.

3.4 Příkazový buffer

Grafické kartě zadáme práci přes příkazy a jejich parametry. Ve Vulkan API je nutné příkazy shlukovat do tzv. *příkazových bufferů*. Procesu, kdy plníme příkazový buffer, se říká *recording*, neboli *nahrávání*. Jakmile příkazový buffer nahrajeme, změní se jeho stav na *nahraný* a můžeme ho poslat do *fronty*. Tímto způsobem si můžeme příkazové buffery předpřipravit a ušetřit výpočetní výkon. Navíc lze příkazový buffer poslat vícekrát, bez nutnosti jej připravovat znova.

3.5 Fronty

Vulkan využívá fronty, do kterých příkazové buffery posíláme. Ovladač příkazový buffer rozbalí a jednotlivé příkazy začne vykonávat. Ve Vulkan je každá fronta z určité rodiny a při vytváření fronty je třeba zvolit tu vhodnou. Každá rodina má jiné možnosti a můžeme mít více front ze stejné rodiny. Každá má jednu nebo více z těchto vlastností:

1. Transfer – Především umožňuje kopírovat a přesouvat data na paměti grafické karty.
2. Graphics – Slouží především pro vykreslovací příkazy, ale umí obecně i transfer.
3. Compute – Slouží pro vlastní výpočty, mimo rasterizaci. Většinou pro vědecké výpočty, ale Compute si získává velké zastoupení i v herním průmyslu, například pro výpočet optimalizací pro scénu.

Je možné vytvořit jen jednu frontu, která by se starala o všechno - grafickou frontu. Problém ale je, že na grafické kartě je více částí procesoru, kde se každý stará o něco jiného. Jedna část se stará jen o vertex shader, druhá jen o fragment shader, další jen o přesouvání v paměti atd. Pokud bychom měli jen jednu grafickou frontu, mohlo by se stát, že ji zahltíme vykreslovacími příkazy a následně pošleme příkaz pro přesun paměti. To by způsobilo, že příkaz pro přesun paměti by musel čekat na dokončení všech vykreslovacích příkazů i přesto, že by mohl běžet paralelně. Z toho důvodu je lepší používat více front z různých rodin.

3.6 Posílání do front

Máme-li nahraný příkazový buffer, můžeme ho poslat do fronty. Ve Vulkan API je zaručeno, že se příkazy začnou vykonávat v pořadí, ve kterém přišly do fronty.

Není ale zaručeno, že v tomto pořadí skončí. To může vést k desynchronizaci hlavně v případě, že používáme více render passů, kde druhý závisí na prvním. Občas je třeba čekat na CPU, než GPU dokončí výpočty, a to vyžaduje další synchronizaci. V této podkapitole budeme čerpat ze zdrojů [8] a [9].

3.6.1 Synchronizace mezi GPU a CPU

Synchronizace mezi GPU a CPU je důležitá, protože nemůžeme začít znova posílat příkazy, pokud nebyly dokončeny příkazy z předchozího snímku. Mohlo by se stát, že nové příkazy začnou zapisovat do paměti, kterou předchozí snímek ještě využívá.

K této synchronizaci nám Vulkan poskytuje objekt *fence*. Ten obsahuje dva stavy: *čekající* a *signalizovaný*. Na CPU pak na fence můžeme v klidu čekat a nebo se periodicky ptát, zda je ve stavu signalizovaný. Stav se změní na signalizovaný poté, co se všechny příkazy, na které čeká, dokončily.

3.6.2 Synchronizace ve frontách

Další užitečná synchronizace je přímo ve frontě. Jak již bylo zmíněno, příkazy se vykonávají ve stejném pořadí, v jakém do fronty přišly, ale nemusí se dokončit v tomto pořadí. Pro synchronizaci ve frontě můžeme využít *bariéru*, což je příkaz, který zablokuje vykonávání všech dalších příkazů. Bariéry ve Vulkan API umožňují například blokovat jen určitý typ příkazů nebo konkrétní krok v grafické pipeline. Optimalizace bariér ve Vulkan je klíčová, protože chceme blokovat co nejméně času.

3.6.3 Synchronizace mezi frontami

Vulkan API nám dává možnost použít více front a je třeba synchronizovat i příkazy mezi nimi. K tomu nám dává synchronizační objekt *semafor*. S ním iteragují dva aktéři, kde jeden čeká na semafor a druhý ho signalizuje. Při odesílání nahraných příkazových bufferů do fronty máme možnost určit, na které semafory počkat a které po dokončení signalizovat.

3.7 Vykreslování

Ukážeme si, jak dát dohromady render pass, framebuffer a grafickou pipeline pro vykreslování. Nejprve je třeba vytvořit a začít nahrávat příkazový buffer. Veškeré vykreslovací příkazy se musí nacházet uvnitř render passu. To znamená, že na začátku je třeba poslat příkaz oznamující začátek render passu a na konci poslat příkaz oznamující konec render passu. Při oznamování začátku render passu zároveň přiřadíme framebuffer. Před samotným vykreslením uvnitř render passu, je ještě třeba přiřadit grafickou pipeline, vertex buffer a index buffer. Poté už můžeme pomocí příkazu `vkCmdDraw` vykreslovat polygonální sítě z

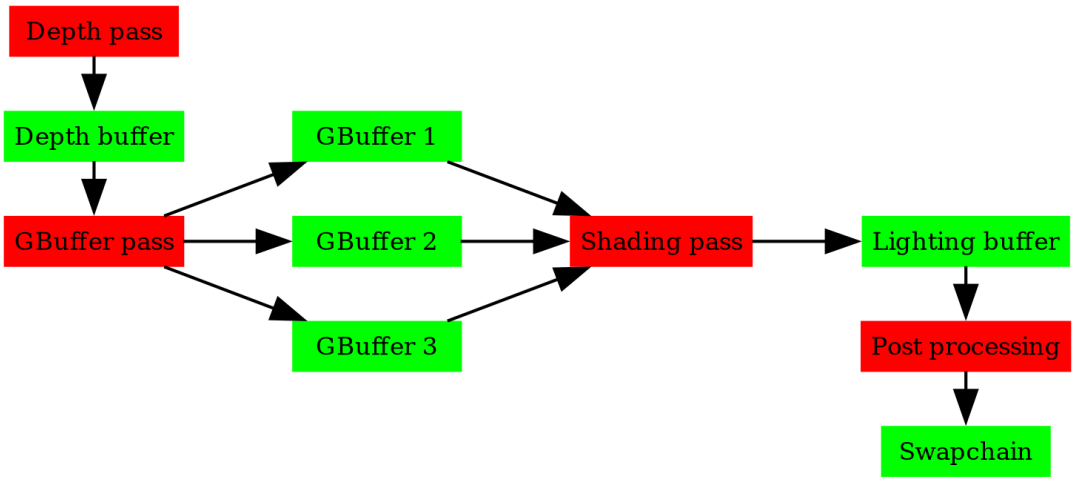
přiřazeného vertex bufferu. Tento příkaz obsahuje dva parametry, první je pro počet vertexů a druhý pro odsazení od začátku vertex bufferu.

4 Render graf

Moderní hry mohou mít i stovky render passů, které se musí vykonat a synchronizovat. Navíc se mohou dynamicky měnit, například podle nastavení nebo podle scény, ve které se uživatel nachází. Synchronizovat vše ručně by bylo časově náročné a způsobovalo by spoustu chyb. Většina velkých herních enginů využívá tzv. *render graf*. To je datová struktura, která reprezentuje závislosti mezi zdroji a vykreslovacími operacemi. Pomáhá organizovat a optimalizovat pořadí těchto operací a automaticky řeší i ideální synchronizaci mezi nimi. Navíc umožňuje *virtualizovat zdroje*. To znamená, že dvěma zdrojům, u kterých nemůže nastat, že by se oba používaly zároveň, lze přiřadit stejný úsek paměti.

4.1 Inspirace

Inspirujeme se prezentací od společnosti EA, která prezentovala jejich použití render grafu v herním enginu FrostBite (zdroj [10]) pro její snadno rozšiřitelnou architekturu. Implementace každý snímek poskládá množinu všech render passů a množinu všech zdrojů, které se v aktuálním snímku budou využívat. Každý render pass pak do některých zdrojů vykresluje a z jiných čte. Poté z množiny „poskládá“ render graf. Nakonec podle grafu efektivně spouští jen ty operace, které jsou opravdu třeba. Ostatní operace neprovádí pro ušetření výkonu. Ilustrační graf můžeme vidět například na obrázku 7, kde zeleně jsou znázorněny zdroje a červeně render passy.



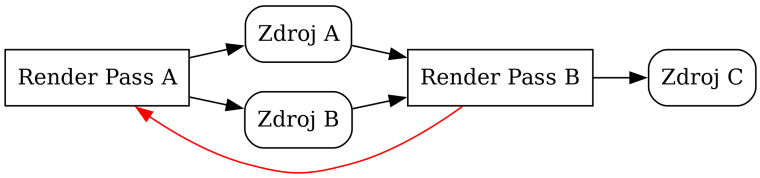
Obrázek 7: Ilustrační render graf

Z množiny render passů a zdrojů vytvoří graf, který optimalizuje, a následně vytvoří synchronizační objekty. Tento přístup je výborný pro herní enginy, protože pro každý snímek můžeme kompletně změnit vykreslování. Další výhoda je, že zdroje jsou virtuální a jsou přiřazovány líně. To znamená, že implementace render grafu může vícero zdrojům přiřadit stejnou paměť, pokud se nepoužívají

ve stejný moment. Tuto paměť ale přiřazuje až v momentě, kdy se zdroj používá. Zdroje, které se nikdy nepoužijí, nezabírají žádnou paměť.

4.2 Předpřipravená implementace

Vytvoříme implementaci, která render graf připraví předem do spustitelného stavu a každý snímek ho bude spouštět. Životní cyklus render grafu rozdělíme na tři fáze: příprava, sestavení a spuštění. Ve fázi přípravy definujeme render passy a jejich vstupní a výstupní zdroje. To je znázorněno na obrázku 8. Vidíme, že render pass A má na výstupu zdroj A a zdroj B, které jsou zároveň vstupy render passu B. Render pass B má na výstupu zdroj C.



Obrázek 8: Ilustrace render passů a zdrojů

Navíc pro render pass definujeme *funkci pro přípravu* a *funkci pro nahrávání*. Funkce pro přípravu je funkce definovaná uživatelem, která bude zavolána render grafem po úspěšné přípravě a po přiřazení zdrojů k výstupům. Tato funkce by měla vytvořit všechny ostatní Vulkan API objekty, které budou ve funkci pro nahrávání potřeba, jako například grafická pipeline. Funkce pro nahrávání je funkce definovaná uživatelem, která bude také zavolána render grafem a jejím účelem je nahrát do příkazového bufferu získaného na vstupu veškeré příkazy, které se mají splnit. Render graf tedy od uživatele kompletně abstrahuje i spouštění a správu příkazových bufferů.

Implementace, ze které jsme brali inspiraci, měla tu výhodu, že spouštěla pouze render passy, které byly třeba. Tuto optimalizaci přidáme bez toho, aniž bychom render graf stavěli znovu. Zároveň přidáme možnost nahrávat jen příkazové buffery, které jsou třeba. Pro tuto optimalizaci definujeme *platnost* render passu. Render pass se stává neplatný právě, když je neplatná některá z jeho závislostí a nebo je označen za neplatný pomocí metody `invalidate`. Tuto metodu definujeme pro render graf a bude mít jediný argument, a to název render passu.

Nahrát příkazový buffer znova je potřeba právě tehdy, když příkazy v něm jsou závislé na datech na CPU, a tato data se změnila. Například mějme příkazový buffer, který bude mít příkaz pro vykreslení pro každou polygonální síť ve scéně. Je třeba ho znovu nahrát právě, když do scény přidáme polygonální síť. Přidáme možnost přiřadit render passu *závislost nahrávky*. Render pass nahrajeme právě, když se závislost nahrávky stane neplatnou. Výhoda závislosti nahrávky je ta, že víme, které dva render passy se budou nahrávat vždy spolu. Toho využijeme k optimalizaci v další podkapitole.

4.3 Stavitel

V druhé fázi sestavení sestavíme spustitelný render graf. Algoritmu, který z množiny render pass objektů a výstupního zdroje vytvoří render graf, budeme říkat *stavitel*. Zároveň využijeme návrhový vzor stavitele a vytvoříme si pro algoritmus třídu `RenderGraphBuilder`. Na rozhraní bude mít třída metodu pro přidání render passu `add_render_pass` a metodu pro stavbu render grafu `build`. Pomocí metody `add_render_pass` do stavitele přidáme všechny render passy, které chceme ve výsledném render grafu. Metoda `build` spustí algoritmus, který postaví render graf do instance třídy `RenderGraph`, kterou vrátí.

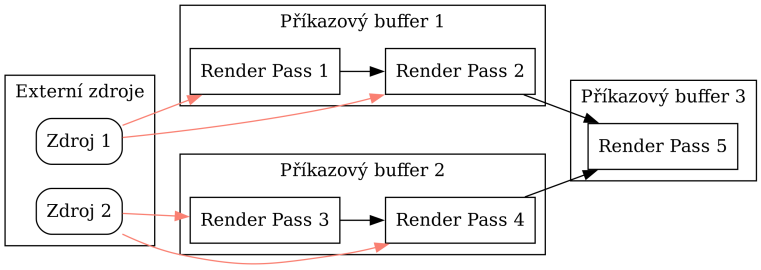
Popíšeme si podrobněji, jak metoda pro stavbu render grafu funguje. Render graf sám definuje, které render passy na sobě *závisí*. Render pass B závisí na render passu A právě, když render pass A má na výstupu zdroj, který má render pass B jako vstup. Tuto závislost vidíme na obrázku 8 znázorněnou červenou šipkou. Tyto závislosti reprezentujeme orientovaným grafem. Každý vrchol reprezentuje render pass a hrana z vrcholu A do vrcholu B reprezentuje, že render pass na vrcholu B závisí na výstupu render passu na vrcholu A. V grafu zvolíme kořen, který bude výstupní zdroj. V případě obrázku 8 to je zdroj C. Graf budeme reprezentovat pomocí matice, protože takovou matici lze rychle vytvořit a umožňuje snadno zjistit, jestli dva render passy na sobě závisí. Netrápí nás ani vyšší paměťová náročnost, protože render passů nikdy nebude více jak pár set.

Na grafu navíc provedeme *tranzitivní redukci*, protože chceme minimalizovat počet závislostí. Tranzitivní redukce odstraní všechny hrany, které nejsou nutné pro zachování dosažitelnosti v grafu. Například pokud A závisí na B a B závisí na C, můžeme odstranit hranu mezi A a C, aniž by se změnily závislosti reprezentované grafem.

Stavitel musí pro každou zbývající hranu zvolit vhodný synchronizační objekt. Uvedeme si, podle čeho se bude rozhodovat. Nejprve se budeme dívat na každý render pass jako na samostatný příkazový buffer. To znamená, že každý render pass v render grafu bude obsahovat Vulkan API objekty pro příkazový buffer, render pass a framebuffer. V tomto případě jediný synchronizační objekt, který využijeme, je semafor. Stačí, aby každý render pass měl jeden, který po provedení signalizuje, a množinu těch, na které čeká. Semafory, na které čeká, získá tak, že posbírá všechny signalizační semafory render passů, na kterých závisí.

AMD[11] i Nvidia[12] ale doporučují používat co nejméně příkazových bufferů. Zkusíme tedy spojit některé render passy do jednoho příkazového bufferu, tzv. *bloku*. Dva render passy můžeme mít ve stejném příkazovém bufferu právě tehdy, když nahrávka obou má stejnou závislost. Na obrázku 9 můžeme vidět ilustraci shlukování render passů do příkazových bufferů. Červená šipka značí závislost nahrávky. Vidíme, že „Render Pass 1“ a „Render Pass 2“ jsou v jednom bloku, protože mají stejnou závislost nahrávky a sousedí spolu. „Render Pass 5“ nemá žádnou závislost nahrávky, a to pro render graf znamená, že stačí render pass nahrát jednou.

Upravíme reprezentaci grafu tak, aby shlukovala render passy do příkazových bufferů. Vrchol grafu nyní budou tvořit právě bloky tvořené render passy.



Obrázek 9: Ilustrace shlukování render passů

Struktura bloku bude obsahovat vektor všech render passů, které obsahuje, signalizační semafor a množinu bloků, na kterých závisí. Dále potřebujeme ukázat, kde je třeba vložit bariéru. Proto bude sloužit druhý vektor, kde prvek bude obsahovat index do vektoru render pass a bariéru.

4.4 Spouštění

Nyní si uvedeme, jak se bude render graf spouštět a ovládát. Pomocí metody `invalidate_dependency` s argumentem pro název závislosti, můžeme nastavit závislost jako neplatnou. Pokud existuje nahrávka, která na ní závisí, můžeme nahrávku rovnou nahrát znovu. Poté každý render pass, který na ní závisí, vložíme do fronty, pokud nezáleží tranzitivně na žádném jiném render passu z fronty. Při spuštění render grafu pomocí metody `run` projdeme všechny prvky ve frontě a každý spustíme. To ale způsobí neplatnost jejich výstupů, takže opět musíme spustit ty render passy, které na nich závisí. Takhle pokračujeme dokud není žádný další render pass s neplatnou závislostí.

5 Ukázková aplikace

Představili jsme si problémy, které vykreslovací engine musí řešit, a v této kapitole vše aplikujeme na ukázkovém projektu. Použijeme render graf pro řešení synchronizace, ověříme přívětivost jejího rozhraní a jak moc vývojáře omezuje. Ukázková aplikace je spustitelný soubor, který při spuštění načte objekty ze souboru a začne je v reálném čase vykreslovat do okna. Navíc umožňuje interaktivní pohyb ve scéně.

Celý projekt je napsán v jazyce C++ a sestavuje se pomocí CMake. V projektu jsou tři moduly, které jsou reprezentovány knihovnami. Slouží čistě pro abstrakci rozhraní, které se týká vykreslování a do produkční aplikace nepatří.

Modul „base“
: obsahuje základní abstrakci nad Vulkan API, aby nám usnadnil práci s ní. Umožňuje alokovat paměť, vytvářet zdroje nebo posílat příkazové buffery do front. Také obsahuje třídy, s návrhovým vzorem „stavitel“, které budou usnadňovat tvorbu některých Vulkan API objektů.

Modul „render_graph“
: obsahuje naši implementaci render grafu.

Modul „window“
: obsahuje rozhraní pro tvorbu a správu oken. Umožňuje při spuštění aplikace otevřít okno a vykreslovat do něj.

Dále přiblížíme si knihovny, mimo našich modulů, které budeme používat.

Volk
: pomáhá načítat funkce pro Vulkan API. Vulkan API nelze staticky přiřadit jako knihovnu, ale musí se načítat dynamicky, protože je poskytován až ovladačem grafické karty. Ten se ale na různých zařízení může lišit. Zároveň knihovna usnadňuje načítání funkcí z rozšíření pro Vulkan API.

VMA
: je knihovna zvaná Vulkan Memory Allocator od společnosti AMD. Vulkan API má omezení na počet alokací v paměti, které je snadné překročit, pokud nebudeme opatrní. Zároveň každá alokace je výpočetně náročná operace. Knihovna VMA tento problém řeší tak, že alokuje po větších blocích paměti, takže vícero zdrojům přiřadí stejnou alokaci, pouze jiný blok v ní. Tím značně redukuje počet alokací.

SDL2
: je knihovna pro správu okenních aplikací. Umožní nám vytvořit okno pro naši aplikaci a vykreslovat do něj. Výhoda je, že knihovna funguje na více platformách.

ImGui
: je populární knihovna pro ladění grafických aplikací, protože umožňuje její vykreslování přímo přidat do našeho vykreslování. Navíc lze knihovnu přiřadit staticky.

cgltf
: je malá knihovna (pouze jeden soubor), která lze staticky přiřadit a usnadňuje načítání souborů ve formátu *glTF*. Tento formát reprezentuje scénu, včetně stromové struktury objektů, polygonální sítě, materiály, textury i transformace. Formát spravuje koncorcium Khronos Group a umožňuje snadno data načíst na grafickou kartu. Navíc formát používá strukturu JSON a je tedy i snadno čitelný.

stb_image
: je opět malá knihovna, kterou tvoří jeden soubor, která umí načítat obrázky v různých formátech jako PNG, JPEG apod. Knihovnu použijeme pro načítání textur.

5.1 Načítání scény

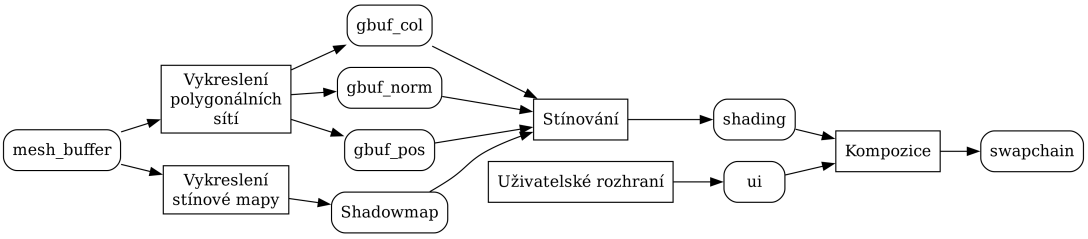
Aplikace nejprve načte scénu ze souboru. Následně vytvoří dva buffery, jeden pro vertexy a druhý pro indexy, do kterých následně načtená data nahraje.

5.2 Materiály a Textury

Datová struktura materiálu se skládá pouze z parametrů a ukazatelů na textury. Všechny materiály si uložíme do bufferu, ke kterému budeme přistupovat z fragment shaderu. Textury pak budeme ukládat v poli, do kterého budeme přistupovat přes ukazatele na textury v materiálu. Pro kompletní správu materiálů, včetně správy paměti a bufferů, jsme vytvořili třídu `MaterialBuffer`. Ta na rozhraní bude obsahovat metodu pro přidání textury, která vrátí index textury v poli textur, abychom ho mohli přiřadit materiálu. Aplikace po načtení scény vytvoří pole textur a buffer pro materiály, do kterých poté nahraje patřičná data.

5.3 Vykreslování

Vykreslování v aplikaci spravuje render graf. Ten tvorbu aplikace značně usnadnil, protože nebylo třeba psát spousty správy zdrojů a synchronizace. Ke zkrácení kódu došlo i v případě enginu FrostBite, jak je zmíněno ve zdroji [10]. V případě této aplikace jsme nenarazili na případ, který by nešel pomocí naší implementace v render grafu zvládnout. Vizualizovaný render graf vidíme na obrázku 10. Jednotlivé render passy a jejich vstupy a výstupy si popíšeme.



Obrázek 10: Vizualizace render grafu

5.3.1 Vykreslování polygonálních sítí

První render pass bude pro vykreslení polygonálních sítí. Použijeme techniku zvanou *deferred rendering*, která se snaží co nejvíce výpočtů ve fragment shaderu odložit na později. Fragment shader se spouští pro každý pixel, který se ve frustum nachází, včetně těch, které nejsou viděť. Je to z toho důvodu, že ve fragment shaderu můžeme nastavit hloubku nebo průhlednost, a grafická karta tak zatím nemůže zjistit, pro které pixely výpočet má smysl a pro které ne. Pokud nejprve vykreslíme do obrázku informace, které jsou pro výpočty potřeba, můžeme v další iteraci informace z obrázku číst a dělat výpočty jen pro výsledné, tedy viditelné, pixely. Těmto obrázkům se říká *g-buffer*.

Pro naše účely budeme potřebovat vypočítat množství odraženého světla, které přidává dojem zdroje světla ve scéně. Vykreslená scéna pak působí daleko realističtěji. Pro tento výpočet je třeba pozice, normála a barva. Proto budeme mít tři výstupy, a to `gbuf_pos` pro pozici, `gbuf_norm` pro normálu a `gbuf_col` pro barvu.

Tento render pass nebude mít žádnou závislost výstupu, ale bude mít závislost nahrávky na `mesh_buffer`. Tato závislost reprezentuje změnu objektů, které chceme vykreslit.

5.3.2 Stíny

Dalším efektem, který výsledný obrázek udělá realističtější, jsou stíny vrhané objekty. V případě rasterizace si musíme pomoci a informace o vrhajících objektech získat. Používá se technika, kdy se vykreslí hloubka z pohledu zdroje světla. Takový výstup se nazývá *stínové zobrazení*. Pro získání informace o tom, jestli je na bod vrhaný stín, najdeme na který pixel se bod v shadow mapě zobrazí. Následně porovnáme informaci na pixelu ve shadow mapě se vzdáleností bodu od zdroje světla. Je-li vzdálenost stejná jako hloubka, s malou tolerancí, pak na bod není vrhán stín. V opačném případě je na bod vrhán stín. Výstup render passu pojmenujeme `shadowmap`. Nahrávku bude mít render pass závislou na `mesh_buffer`.

5.3.3 Shading

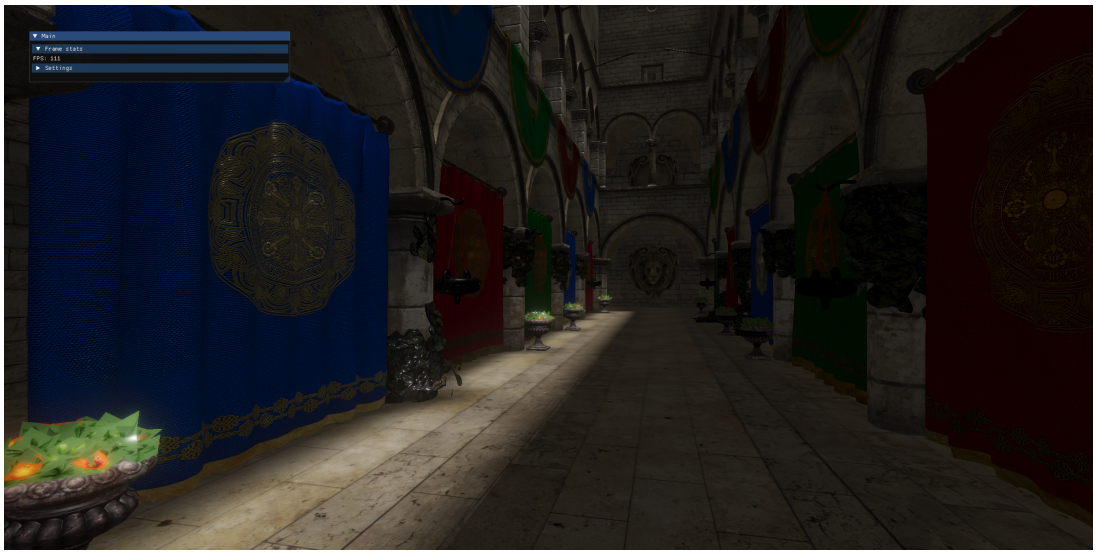
Tento render pass využije informace z g-bufferu a shadow mapy a provedeme tzv. *stínování*. To znamená, že barva bodu ve výsledném obrázku už bude brát v potaz to, kolik světla je z bodu odraženo. Pro tento výpočet se používá tzv. *vykreslovací rovnice*, její vysvětlení je ale nad rámec této práce. Render pass bude záviset na `gbuf_pos`, `gbuf_norm` a `shadowmap` a na výstupu bude mít `shading`.

5.3.4 Užitelské rozhraní

V předposledním render passu vykreslíme základní uživatelské rozhraní, které zobrazí základní informace o vykreslování, jako například počet snímků za sekundu. Použili jsme knihovnu ImGui, která je otevřená a je možné ji staticky odkazovat. Navíc vykreslování můžeme snadno přidat do render grafu. Tento render pass bude mít nahrávku závislou na `ui_draw_list` a pouze jeden výstup `ui`.

5.3.5 Kompozice

Poslední render pass spojí vykreslenou scénu a uživatelské rozhraní. Bude záviset na `ui` a `shading` a výstup budeme psát do finálního zdroje `swapchain`. Render pass na `shading` vykreslí `ui`, podle alfa kanálu `ui`. Výsledek můžeme vidět v obrázku 11.



Obrázek 11: Výsledný obrázek

5.4 Běh programu

Pro každý snímek podle vstupů od uživatele změníme transformaci kamery. Následně definujeme uživatelské rozhraní, které se má na daný snímek zobrazit. V

našem případě se vykreslí podokno, které ukazuje počet snímků za sekundu a jednoduché nastavení. Poté se v render grafu automaticky nastaví `ui_draw_list` a `gbuf_col` jako neplatné. To způsobí, že se nám celá scéna překreslí. Není třeba měnit platnost `mesh_buffer`, protože scéna se pouze překreslila, ale nezměnila.

5.5 Sestavení a spuštění

Nakonec potřebujeme aplikaci sestavit do spustitelného souboru, který budeme spouštět. K sestavení použijeme CMake, který tento proces usnadňuje. Všechny shadery jsou ve složce shaders/, kterou je třeba i s shadery distribuovat se spustitelným souborem.

Pro spuštění je třeba jako první argument použít cestu ke glTF souboru, který se načte. Aplikace po spuštění otevře okno a začně do něj vykreslovat načtenou scénu. Pro změnu scény je třeba okno aplikace zavřít a spustit znovu s jinou hodnotou argumentu pro cestu.

Pro pohyb ve scéně používáme klávesy W, A, S a D a otáčíme se pohybem myší.

Závěr

Cílem práce bylo představit vykreslovací enginy pracující v reálném čase a vytvořit vlastní implementaci. Ve druhé kapitole jsme popsali, co je vykreslovací engine, a definovali jsme důležité pojmy, které se ho týkají. Ukázali jsme si výpočty, které jsou pro vykreslování třeba. Ve třetí kapitole jsem tyto výpočty akcelerovali akcelerovali pomocí grafické karty. Pro akceleraci jsme si ukázali několik rozhraní, především Vulkan API, a jak jeho rozhraní funguje. Zejména jsme představili problém synchronizace. Ve čtvrté kapitole jsme ukázali populární řešení tohoto problému zvané render graf. Ten jsme upravili tak, aby správně pasoval na Vulkan API. Nakonec jsme si vytvořili ukázkovou aplikaci, která načte objekty ze souboru a vykreslí je v reálném čase. Aplikace využívá dříve představenou implementaci render grafu, která zásadně zjednodušila kód a udělala ho snadno rozšiřitelný.

Na práci by bylo možné navázat vylepšením render grafu, který zatím umí používat pouze jednu frontu. Více front by mohlo umožnit vykreslovat více snímků za sekundu. Také by implementace mohla lépe optimalizovat bariéry tak, aby více operací mohlo běžet souběžně. Na rozhraní render grafu by bylo dobré ověřit více krajních případů, které se v průmyslu objevují, například různé efekty nebo techniky vykreslování.

Conclusions

The aim of this thesis was to present real-time rendering engines and create our own implementation. In the second chapter, we described what a rendering engine is and defined important related terms. We demostrated the calculations necessary for rendering. In the third chapter, we accelerated these calculations using a graphics card. For acceleration, we explored several interfaces, primarily the Vulkan API and how it's interface works. In particular, we introduced the issue of synchronization. In the fourth chapter, we demonstrated a popular solution to this problem called the render graph. We adapted it to fit the Vulkan API correctly. Finally, we created a sample application that loads objects from a file and renders them in real-time. The application utilizes the previous introduced render graph implementation, which significantly simplified the code and made it easily extendable.

Future work could involve improving the render graph, which currently can only use a single queue. Multiple queues could allow for rendering more frames per second. The implementation could also better optimize barriers so that more operations can run concurrently. It would be beneficial to test the render graph interface with more edge cases that occur in the industry, such as various effects or rendering techniques.

A Obsah elektronických dat

kidiplom/
: Adresář s textem práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech (textových) příloh, a všechny soubory potřebné pro bezproblémové vytvoření PDF dokumentu textu (případně v ZIP archivu), tj. zdrojový text textu a příloh, vložené obrázky, apod.

README.*
: Textový soubor obsahující instrukce ke kompilaci a spuštění praktické části.

src/
: Adresář obsahující zdrojové kódy k ukázkové aplikaci.

build/
: Adresář obsahující sestavené spustitelné soubory pro platformu Windows a Linux. Spustitelné soubory jsou zabalené, s dalšími potřebnými soubory, v ZIP archívu.

data/
: Adresář obsahující soubory ve formátu glTF pro vykreslení přes ukázkovou aplikaci.

Literatura

[1] *2024 Essential Facts About the U.S. Video Game Industry*. [online]. 2024 [cit. 2024-3-8]. Dostupný z: ⟨https://www.theesa.com/resources/essential-facts-about-the-us-video-game-industry/2024-data/⟩.

[2] *ČR 2023 HERNÍ PRŮMYSL*. [online]. 2023 [cit. 2024-3-8]. Dostupný z: ⟨https://gda.cz/wp-content/uploads/2023/12/Infografika2023.pdf⟩.

[3] *Polygon mesh*. [online]. [cit. 2024-3-8]. Dostupný z: ⟨https://en.wikipedia.org/wiki/Polygon_mesh⟩.

[4] *3D Computer Graphics Primer: Ray-Tracing as an Example*. [online]. [cit. 2024-3-8]. Dostupný z: ⟨https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/implementing-the-raytracing-algorithm.html⟩.

[5] *Rasterization*. [online]. [cit. 2024-3-8]. Dostupný z: ⟨https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/overview-rasterization-algorithm.html⟩.

[6] Ahn, Song Ho. *OpenGL Projection Matrix* [online]. [cit. 2024-3-8]. Dostupný z: ⟨https://songho.ca/opengl/gl_projectionmatrix.html⟩.

[7] *Vulkan Documentation*. [online]. [cit. 2024-3-8]. Dostupný z: ⟨https://docs.vulkan.org/⟩.

[8] Raphael Mun John Zulauf, Jeremy Gebben. *Understanding Vulkan Synchronization* [online]. 2021 [cit. 2024-3-8]. Dostupný z: ⟨https://www.khronos.org/blog/understanding-vulkan-synchronization⟩.

[9] Kristian, Hans. *Yet another blog explaining Vulkan synchronization* [online]. 2019 [cit. 2024-3-8]. Dostupný z: ⟨https://themaister.net/blog/2019/08/14/yet-another-blog-explaining-vulkan-synchronization/⟩.

[10] O'Donnell, Yuriy. *FrameGraph: Extensible Rendering Architecture in Frostbite* [online]. 2017 [cit. 2024-3-8]. Dostupný z: ⟨https://gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-Architecture-in⟩.

[11] AMD. *RDNA Performance Guide* [online]. [cit. 2024-3-8]. Dostupný z: ⟨https://gpuopen.com/learn/rdna-performance-guide/⟩.

[12] Nuno Subtil Matthew Rusch, Ivan Fedorov. *Tips and Tricks: Vulkan Dos and Don'ts* [online]. 2019 [cit. 2024-3-8]. Dostupný z: ⟨https://developer.nvidia.com/blog/vulkan-dos-donts/⟩.