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

CLUSTERED DEFERRED SHADING VO VULKAN API

CLUSTERED DEFERRED SHADING IN VULKAN API

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MATEJ KARAS**
AUTHOR

VEDOUCÍ PRÁCE **Ing. TOMÁŠ MILET**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Karas Matej**
Program: Informační technologie
Název: **Clustered deferred shading ve Vulkan API**
Clustered Deferred Shading in Vulkan API
Kategorie: Počítačová grafika

Zadání:

1. Nastudujte metody zobrazování scén s mnoha světly a API Vulkan.
2. Naimplementujte zobrazování scény s texturami pomocí API Vulkan.
3. Naimplementujte metodu clustered deferred shading do stávající aplikace.
4. Změřte a zhodnoťte naimplementovanou metodu a vytvořte demonstrační video.

Literatura:

- Dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2 + implementovaná metoda tiled shading.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Milet Tomáš, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Práca sa zaoberá tvorbou aplikácie pre vykresľovanie väčšieho počtu svetiel v reálnom čase použitím novej generácie grafického API. V texte je popísaný dôvod vzniku novej generácie grafických API, rozobrané optimalizačné metódy na vykreslenie väčšieho počtu svetiel v reálnom čase a v závere práce sú popísané dosiahnuté výsledky, možné vylepšenia a budúce pokračovanie na práci.

Abstract

This thesis deals with application development for rendering many lights in real-time using next generation graphics API. Text of thesis contains reasoning of new generation graphics API and theory for methods used for rendering many lights in real-time. The conclusion discusses performance of implementation, possibilities for improvements and options of future work.

Kľúčové slová

Vulkan, Clustered shading, Deferred rendering, Grafika, Veľa svetiel, Vykreslovanie v reálnom čase, C++

Keywords

Vulkan, Clustered shading, Deferred rendering, Graphics, Many lights, Real-time rendering, C++

Citácia

KARAS, Matej. *Clustered deferred shading vo Vulkan API.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Milet

Clustered deferred shading vo Vulkan API

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Tomáša Mileta. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Matej Karas
12. mája 2019

Poďakovanie

Ďakujem pánovi Ing. Tomášovi Miletovi za skvelé vedenie práce, doslova hodiny konzultácií a neskutočný objem informácií z oblasti počítačovej grafiky, ktoré mi poskytol. Taktiež vďaka anonymným recenzentom môjho článku z konferencie Excel@FIT 2019. Taktiež ďakujem všetkým svojim neurónom, ktoré boli ochotné túto prácu napísať a rozsahu 27.5Hz až 4186Hz za udržiavanie môjho psychického zdravia, El Psy Kongroo!

Obsah

Kapitola 1

Úvod

Výpočet osvetlenia je jednou z najdôležitejších fází vo vykresľovacích aplikáciách, resp. počítačových hrách. Počítačové hry sa snažia napodobňovať realitu a vtiahnuť používateľa do virtuálneho sveta. Na to však zvyčajne potrebujú niekoľko tisíc svetiel. Výpočet osvetlenia je náročná operácia a pri naivných technikách vzniká množstvo redundantných kalkulácií.

Táto práca sa zaoberá optimalizačnou metódou clustered deferred shading a jej implementáciou v modernom grafickom aplikačnom rozhraní Vulkan. Metóda **clustered deferred shading** efektívne redukuje počet redundantných kalkulácií a prináša určité lepšie vlastnosti oproti predchádzajúcej metóde. Obe metódy sú popísané v kapitole 2, ktorá okrem nich rozoberá teóriu v oblasti tvorby svetla a dôvody vzniku novej generácie grafických API.

V kapitole 3 je popísaný návrh aplikácie a dôvody uskutočnenia niektorých krokov. V nasledujúcej kapitole (kapitola 4) je popísaná implementácia aplikácie v modernom grafickom API Vulkan za použitia jazyka C++. Dosiahnuté výsledky a ich zhodnotenie je zhrnuté v kapitole 5.

Kapitola 2

Teória

V tejto kapitole bude popísaná teória potrebná na vypracovanie tejto práce. Najskôr bude prebratá problematika, prečo vznikla nová generácia grafických API (aplikačných rozhraní) a aké kroky je potrebné pri tvorbe jednoduchej aplikácie vo Vulkan API. Ďalej bude popísaná tvorba svetla v počítačovej grafike a so štandartnými osvetľovacími modelmi a nakoniec budú popísané optimalizačné metódy na vykresľovanie väčšieho počtu svetiel v reálnom čase, *tiled* a *clustered deferred shading*.

2.1 Nová generácia grafických API

Problém, prečo je potrebná ďalšia generácia grafických API je, že zaužívané grafické API (ako napr. OpenGL), ktoré vznikli približne pred 20 rokmi, boli určené pre grafické karty s fixnou funkcionalitou. Programátor musel dodať vertex dáta v štandartnom formáte a bolo na výrobcovi grafickej karty, ako bude prebiehať tieňovanie a výpočet osvetlenia. Avšak, ako sa grafické karty postupom času vyvíjali, ich použitie nie je len na grafické výpočty, ale zasahuje do širokého spektra rôznych odvetví.

2.1.1 Vývoj grafických kariet

Počas niekoľkých desaťročí sa grafické karty vyvinuli z obyčajného jednoduchého koprocesoru s fixnou funkcionalitou na skupinu programovateľných paralelných jadier [17, 25, 15].

V roku 1976 spoločnosť RCA vytvorila grafický urýchľovač **Pixie**, ktorý bol schopný vytvárať obraz o rozlíšení 62×128. V roku 1979 vznikol prvý systém, ktorý bol schopný produkovať farebný obraz a podporoval viacfarebné *sprity* a pozadie typu *tilemap*. V roku 1981 začala spoločnosť IBM používať vo svojich počítačoch monochromatické a farebné urýchľovače. V roku 1983 spoločnosť Intel vytvorila revolučný grafický urýchľovač, **ISBX 275**, ktorý bol schopný vykresľovať 8 farieb pri rozlíšení 256×256 alebo 512×512 pri monochromatickom obraze. V roku 1985 založili traja hongkongský imigranti z Kanady spoločnosť Array Technology Inc, neskôr premenovanú na ATI Technology, ktorá na dlhú dobu podmanila trh svojimi grafickými kartami. Do roku 1985 všetci veľkí výrobcovia grafických kariet pridali 2D akceleráciu do svojich výrobkov. Počas deväťdesiatych rokov vznikli prvé grafické API (OpenGL – 1992, DirectX – 1995). Koncom roku 1997 ovládala spoločnosť Nvidia jednu štvrtinu svetového trhu so svojimi grafickými kartami.

Prvé „skutočné“ grafické karty, ktoré umožňovali 3D, začali vznikať v roku 1995. Prvou takouto kartou bola karta **Voodoo** od spoločnosti 3DFX, avšak podporovala len 3D akceleráciu a na 2D akceleráciu sa musel použiť iný urýchľovač. Karta umožňovala urých-

ľovať mapovanie textúr, z-buffering a rasterizáciu. Jej nasledujúca verzia **Voodoo2**, ktorá vznikla v roku 1998, obsahovala tri procesory a bola vôbec prvou grafickou kartou, ktorá umožňovala paralelné výpočty 2 kariet v jednom systéme. V roku 1999 spoločnosť Nvidia vytvorila kartu **GeForce 256**, neskôr karta **Radeon 7500** od spoločnosti ATI, ktorá implementovala celú grafickú pipeline, umožňovala transformácie, výpočet osvetlenia, atď. V roku 2000 sa začala rivalita medzi spoločnosťami ATI a Nvidia.

V roku 2001 spoločnosť Nvidia vydáva kartu **GeForce 3**, alternatíva **Radeon 8500** od spoločnosti ATI, ktorá obsahuje už programovateľný vertex shader. V roku 2002 prichádzajú plne programovateľné karty: **Nvidia Geforce FX**, **ATI Radeon 9700**, ktoré umožňujú plne programovateľný vertex a fragment shader programy.

V roku 2004 vychádzajú karty **GeForce 6** a **Radeon X800**, ktoré používajú zbernicu PCI-express. Dovtedy bola používaná zbernica PCI, resp. neskôr AGP (Accelerated Graphics Port).

V roku 2007 začala éra GPGPU (general purpose GPU). V tomto roku Nvidia vydala ich *CUDA development environment* a o 2 roky neskôr sa začína vysoká podpora OpenCL. V roku 2010 prichádza architektúra **Fermi** od Nvidie, ktorá je špecificky dizajnovaná na použitie pre GPGPU a prináša funkčnosť ako unifikovaný adresový priestor, ECC, dvojitý plánovač warpov, atď.

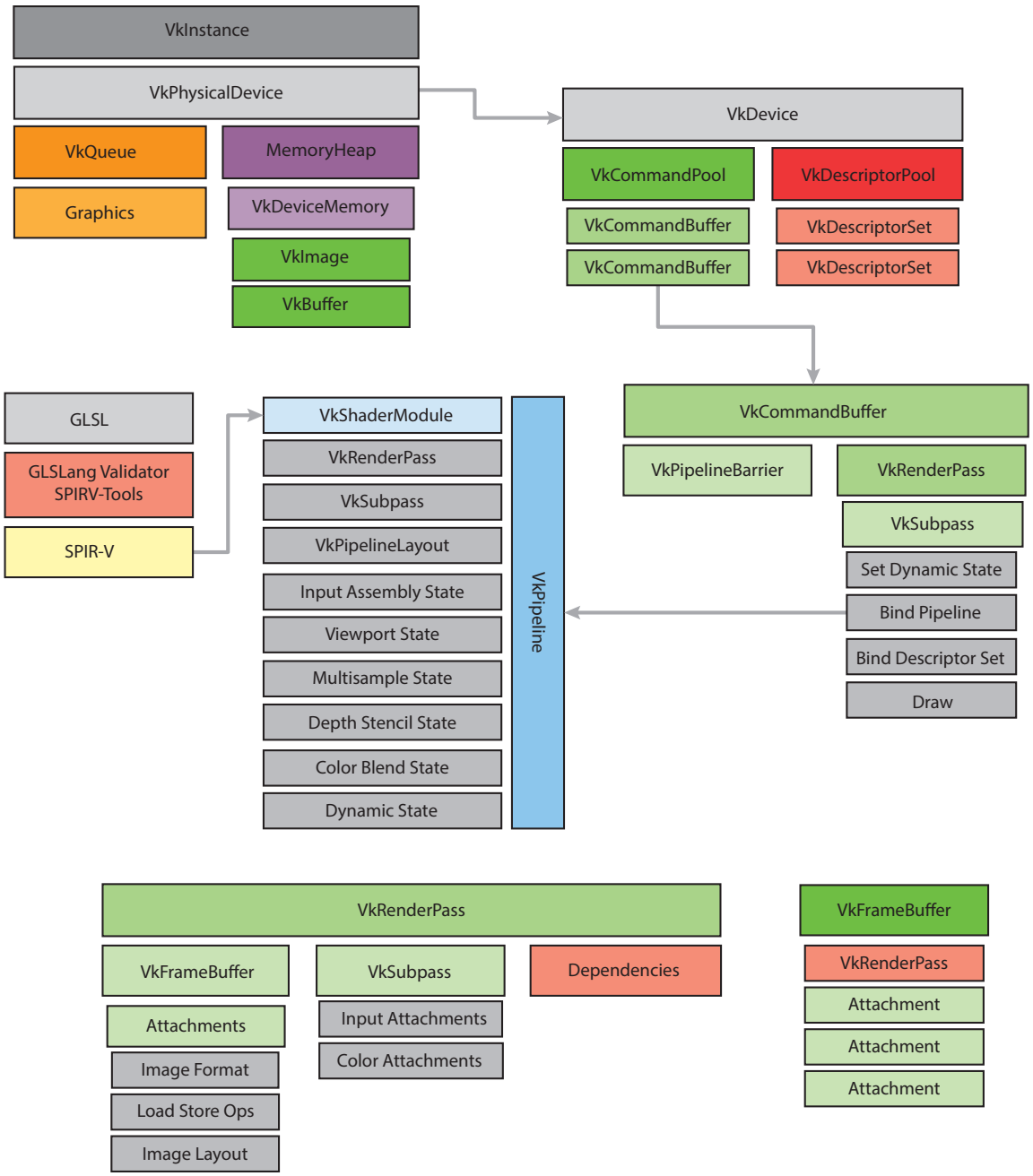
V roku 2018 vydáva Nvidia radu **GeForce 20 series**, ktorá obsahuje *tensor cores*, ktoré dokážu vykonať súčin 4×4 matíc s presnosťou fp16 (floating point 16 bitov) a pripočítaním matice s presnosťou fp32 (rovnica 2.1) v jednom GPU cykle.

$$\mathbf{A} \cdot \mathbf{B} + \mathbf{C} \tag{2.1}$$

2.1.2 Novodobé API

Ako sa grafické karty postupom času vyvíjali z jednoduchého koprocesoru fixnej funkcionality vytvorenej špeciálne pre vykresľovanie až do sady vysoko paralelných a programovateľných jadier, poskytovali viac a viac možností na konfigurovanie stavu. Všetka táto funkcionalita musela byť nabalená na už existujúce API, kvôli spätnej kompatibilite. Toto viedlo k neideálnej abstrakcii hardvéru, pretože nie všetko sa dá vyriešiť pridávaním novej funkcionality, ktorá reflektuje možnosti nových grafických kariet. Preto ovládač grafickej karty musí v dnešnej dobe robiť veľa „odhadov“, aby dokázal namapovať programátorov kód na dnešné grafické karty. Ďalším limitujúcim faktom je, že z doby, keď vznikali tieto grafické API, sa nepočítalo s paralelným vytváraním príkazov na vykresľovanie a preto je takmer nemožné vykresľovať paralelne pomocou knižnice OpenGL.

Najmä z týchto dôvodov vznikli nové grafické API ako Vulkan, DirectX 12, Metal, Mantle. Vulkan je tzv. nástupca OpenGL, vznikol z Mantle API a stojí za ním nezisková organizácia Khronos, ktorá je tvorcom OpenGL. Spôsob, akým Vulkan odľahčuje záťaž na ovládač grafickej karty je, že všetky príkazy musia byť jasne vytvorené vopred (vytvorené command buffre, ktoré sa len nahrajú na grafickú kartu počas kreslenia), musia byť vopred vytvorené všetky primitíva (pipeline, rasterizér, descriptor sety, ...), taktiež vopred preložené shader programy, alokovaná pamäť s jasným úmyslom, ako sa bude využívať. Programátor musí zaručiť správnu synchronizáciu pomocou synchronizačných primitív (bariéra, semafor, *fence*) a ovládač sa potom stará najmä o komunikáciu s grafickou kartou, kdežto pri OpenGL programátor nadeklaruje, čo chce robiť, ako to chce robiť a ovládač grafickej karty sa postará o zbytok riešením veľa úloh na pozadí (napr. prekladanie shader programov, alokovanie pamäte, vloženie synchronizačných primitív, atď).



Obr. 2.1: Kroky potrebné k vytvoreniu jednoduchej aplikácie vo Vulkane. Z obrázku je vidieť, že Vulkán je skutočne *verbose* API a je nutné spraviť veľa krokov, aby sa dokázalo vykresliť pár vertexov.

2.1.3 Schéma jednoduchej aplikácie vo Vulkan API

Vytvorenie jednoduchej aplikácie vo Vulkane nie je na prvý pohľad jednoduché, a preto budú popísané jednotlivé kroky k jej vytvoreniu. Celý postup je zobrazený na obrázku 2.1. Nasledujúci text neslúži ako kompletný návod, skôr ako stručná referencia na vytvorenie a popis objektov nutných pre správnu funkciu jednoduchej aplikácie.

Ako prvý krok je nutné vytvoriť inštanciu Vulkanu **VkInstance**, ktorá uchováva stav aplikácie (názov, verzia, ...), pretože Vulkan neuchováva žiaden globálny stav. Po inicializovaní Vulkanu sa musí vybrať vhodné fyzické zariadenie **VkPhysicalDevice**, ktorým sa bude vykresľovať. List dostupných zariadení sa získa pomocou funkcie *vkEnumeratePhysicalDevices*. Z fyzického zariadenia je nutné vytvoriť logické zariadenie[1] **VkDevice**. Pri vytváraní logického zariadenia je nutné zadať informácie o frontách, ktoré sa budú používať. Fronta **VkQueue** je abstrakcia, používaná na zadávanie príkazov do grafickej karty.

Po vytvorení logického zariadenia, treba alokovať **VkCommandBuffer**, slúžiace na nahrávanie príkazov, ktoré sa môžu neskôr zadať grafickej karte na spracovanie. Na ich alokáciu je potrebný objekt **VkCommandPool**.

Ďalej je nutné vytvoriť plochu na kreslenie **VkSurfaceKHR**, ktorá slúži ako abstrakcia natívneho surface danej platformy, resp. uchováva informácie o ploche okna, na ktorú sa bude vykresľovať. Ďalej je potrebný objekt **VkSwapchainKHR**, ktorý slúži ako abstrakcia poľa snímkov (**VkImage**), do ktorých sa bude vykresľovať a následne prezentovať. Dnešné grafické karty podporujú minimálne 2 snímky, resp. double buffering. K objektom snímkov sa nepristupuje priamo, ale používajú sa **VkImageView**, ktorými sa špecifikuje ako sa plánuje daný snímok využívať. Prípona KHR indikuje, že daná funkcionalita je rozšírenie (extension), ktoré treba aktivovať počas vytvárania logického zariadenia.

Pokiaľ aplikácia plánuje využívať akýkoľvek druh pamäte (depth, depthstencil, SSBO, textúry, atď), je nutné ich alokovať. Aby sa dali tieto zdroje použiť v shaderoch, je nutné ich zadať do **VkDescriptorSet**. Descriptor je dátová štruktúra, ktorá uchováva zdroje ako buffer, buffer view, image view, sampler, atď. Descriptory sa zhlukujú do descriptor setov, ktoré sa viažu počas nahrávania príkazov. Na alokáciu descriptor setov je potrebné vytvoriť **VkDescriptorPool**.

Každé vykresľovanie musí prebiehať v **VkRenderPass** a každý *render pass* musí obsahovať aspoň jeden *subpass*. *Render pass* je kolekcia príloh (depth/stencil, color, ...), *subpassov* a závislostí medzi nimi. Popisuje, ako sú použité prílohy počas vykresľovania. Každý *render pass* používa framebuffer **VkFrameBuffer**, ktorý predstavuje kolekciu pamätí príloh, ktoré bude používať.

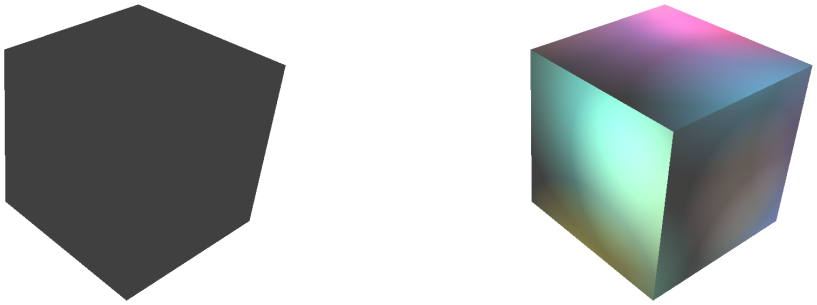
Ako posledný krok sa vytvorí grafická pipeline **VkPipeline**. Grafická pipeline je definovaná pomocou **VkPipelineLayout** a môže pozostávať z niekoľkých **VkShaderModule** a fixnej funkcionality. Shader moduly sú vytvorené zo `SPIR-V` kódu. Fixná funkcionalita pozostáva z častí ako *vertex input assembly*, rasterizér, miešanie farieb, viewport, depthstencil, multismapling, atď.

V tomto štádiu je už možné vykresľovať. Pred každým začatím kreslenia sa musí získať snímka zo *swapchain*, do ktorej sa bude vykresľovať.

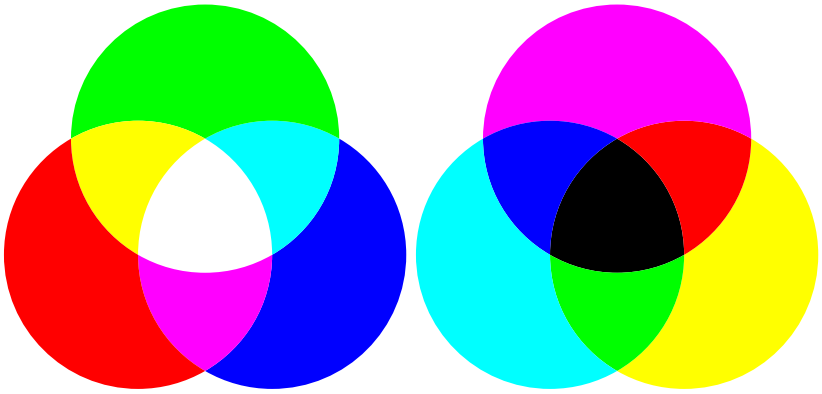
2.2 Svetlo v počítačovej grafike

Výpočet osvetlenia je veľmi dôležitá časť v počítačovej grafike, pretože pridáva „vizuálnu nápovedu“ pre ľudí (obr. 2.2). V tejto sekcii bude popísaná problematika výpočtu osvetlenia v počítačovej grafike. V skratke budú načrtnuté farebné modely, predstavené typy štandartne používaných svetiel a osvetľovacie modely.

[1] Logické zariadenie predstavuje inštanciu jedného fyzického zariadenia



Obr. 2.2: **Vľavo** kocka bez osvetlenia. **Vpravo** kocka s osvetlením. Informácia o tom, že sa jedná o kocku bez osvetlenia sa úplne stratila.



Obr. 2.3: Miešanie farieb podľa farebného modelu. **Vľavo** Aditívne miešanie – pridávaním farieb sa zvyšuje intenzita svetla. **Vpravo** Substraktívne miešanie – pridávaním farieb sa odčíta farebná zložka. Žltá odčítava modrú, magenta odčítava zelenú a tyrkysová odčíta červenú.

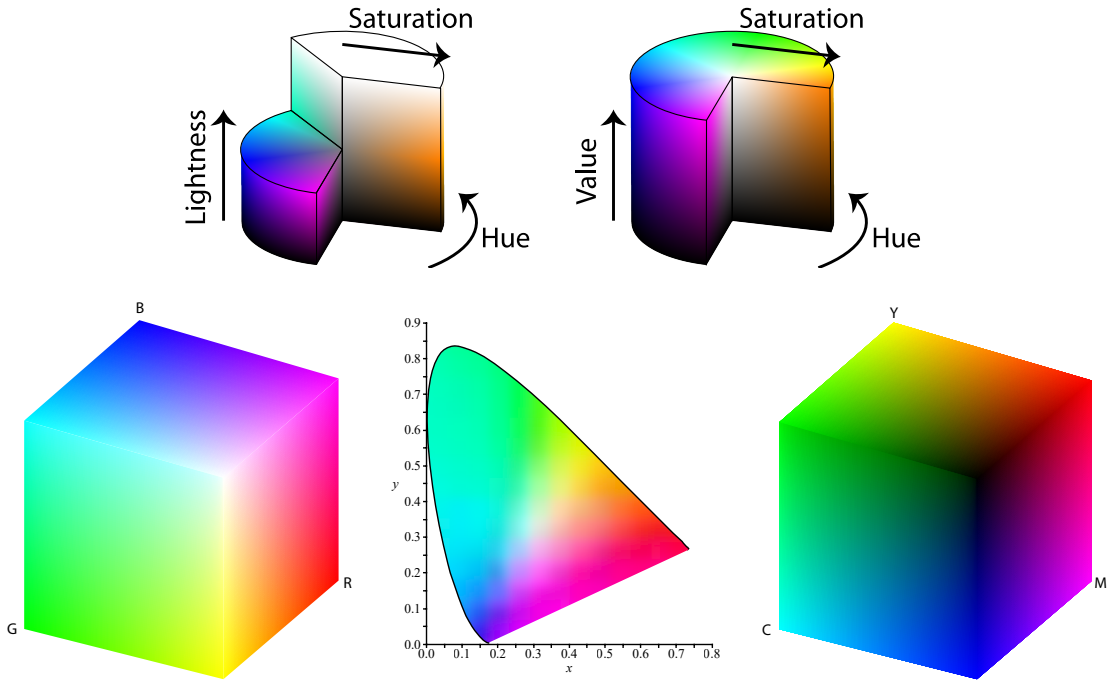
2.2.1 Farebné modely

Aby človek dokázal vidieť farbu, je nutná prítomnosť svetla, ktoré osvetlí objekty okolo. Svetlo je elektromagnetické žiarenie s určitou vlnovou dĺžkou. Pri dopade svetla na objekt je podľa jeho povrchových vlastností časť odrazená a časť pohltená. Farba objektu je potom spektrum odrazených vlnových dĺžok. Okrem farby je možné vnímať jas, sýtosť, svetlosť a odtieň.

Kombináciou rôznych farieb a ich intenzít sa dokáže vytvoriť veľké množstvo farieb. V praxi stačia tri farby dominantných vlnových dĺžok na vytvorenie nekonečne[2] veľa farebných kombinácií. Sú to najčastejšie RGB (red, green, blue) alebo CMY (cyan, magenta, yellow) farby. Podľa výberu základných farieb sa rozlišujú dva druhy miešania farieb: aditívne a substraktívne (obr. 2.3).

Aditívne miešanie farieb je charakteristické pre zobrazovacie jednotky (monitor, displej, projektor, ...). Pri prekrytí všetkých troch zložiek pri ich maximálnej intenzite vzniká

[2] V skutočnosti je možné vytvoriť len 2^{24} kombinácií, pri 8-bitoch na jeden kanál. Druhým faktorom je, či zobrazovacia jednotka dokáže takéto spektrum zobraziť.



Obr. 2.4: Na obrázku sú predstavené vybrané farebné modely. **Vľavo hore – HLS** Užívateľsky orientovaný model, ktorý umožňuje pracovať s farbou, vyberať sýtosť (saturáciu), farebný tón (hue) a svetlosť. Obrázok prebratý z wikipedia.org. **Vpravo hore – HSV** Podobne ako HLS sa jedná o užíveteľsky orientovaný model, ktorý umožňuje pracovať s farbou, vyberať sýtosť, farebný tón a jas. Obrázok prebratý z wikipedia.org. **Vľavo dole – RGB kocka** Odpovedá aditívnemu miešaniu farieb. **V strede dole – CIE** Takzvaný chromatický diagram, používaný ako medzinárodný štandart na porovnávanie farieb a kalibráciu zariadení. Bol navrhnutý tak, aby jeho y-súradnica predstavovala svetlosť farby. **Vpravo dole – CMY kocka** Odpovedá substraktívnemu miešaniu farieb.

biele svetlo. Aditívne miešanie farieb je konceptuálne jednoduchšie, pretože sa jedná len o pridávanie energie o rôznych rozsahoch viditeľného spektra.

Substraktívne miešanie farieb je charakteristické pre prácu s pigmentom, a teda pre zariadenia, ktoré tlačia farbou. Pri prekrytí všetkých troch zložiek, pri ich maximálnej intenzite vzniká čierna farba[3].

Existuje množstvo farebných modelov, ktoré umožňujú pracovať s farbami, ktoré v princípe uľahčujú miešanie farieb. Každý model je prispôsobený na účel svojho použitia – pre určitú skupinu používateľov, rôzne oblasti použitia, so svojimi výhodami a nevýhodami. Vybrané modely sú popísané na obrázku 2.4.

2.2.2 Typy svetiel

V počítačovej grafike sa používa viacero typov zdroja svetla. Líšia sa predovšetkým spôsobom akým osvetľujú, dosahom, prípadne stratou na intenzite (atenuáciou). Na obrázku 2.5 sú predstavené štandartné typy zdrojov svetla: ambientné (ambient), smerové (directional), bodové (point) a reflektor (spot).

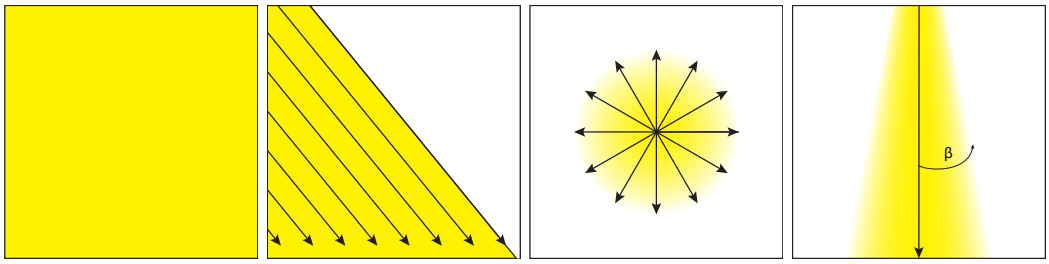
[3]Dnešné tlačiarne používajú farebný model CMYK, ktorý pridáva ešte čiernu farebnú zložku, pretože pri zmiešaní troch základných farieb CMY sa dostane tmavohnedá farba, namiesto čiernej.

Ambientné svetlo Prítomné všade v scéne pred tým, ako sa pomocou osvetľovacieho modelu pridajú ďalšie typy zdrojov svetla. Zvyčajne reprezentuje prírodné svetlo.

Smerové svetlo Zdroj svetla, ktorý vyžaruje v špecifickom smere. Má charakter akoby bolo nekonečne ďaleko a vyžarované lúče sú všetky rovnobežné. Simuluje sa ním žiarenie denného svetla, pretože slnko môžeme považovať akoby bolo nekonečne ďaleko a jeho lúče sú rovnobežné.

Bodové svetlo Vyžaruje z jedného bodu do všetkých strán. So vzdialenosťou intenzita žiarenia klesá (atenuácia). Typické použitie pre bodové svetlá je simulovanie svetla žiarovky.

Reflektor Vyžaruje svetlo z jedného bodu v jednom smere pozdĺž kužeľa, ktorý narastá so vzdialenosťou od zdroju svetla. Intenzita svetla je maximálna v smere, v ktorom vyžaruje a exponenciálne klesá kolmo na tento smer (atenuácia). Typicky sa tento zdroj používa na simulovanie, ako už názov napovedá, reflektoru.



Obr. 2.5: Štandartné typy zdrojov svetla používané v počítačovej grafike. Postupne zľava doprava: ambientné svetlo, smerové svetlo, bodové svetlo, reflektor.

2.2.3 Osvetľovacie modely

Výpočet osvetlenia v počítačovej grafike je simulácia interakcie svetla s jednotlivými typmi povrchov v scéne. Simulácia môže byť veľmi presná a výkonnostne náročná, založená na fyzikálnych zákonoch zo skutočného sveta alebo menej presná, ktorá tieto zákony aproximuje. V minulosti, keď ešte výkonnosť hardvéru nepostačovala na realistickejšie modely, sa na vykresľovanie v reálnom čase používali zjednodušené modely, ako napr. Blinn Phong, ktorý sa používa do dnes. Avšak, so zvyšovaním výkonu grafických kariet je dnes trend používania realistickejších modelov.

Lambertov difúzny model

Lambertov difúzny osvetľovací model bol jeden z najpoužívanejších, ak nie aj najpoužívanejší osvetľovací model. Tento model je nezávislý na pozícii kamery, čo má za následok, že povrch objektu sa nemení voči zmene pozície kamery. Výpočetne to je veľmi jednoduchý osvetľovací model.

Blinn-Phong

Blinn-Phong osvetľovací model je optimalizácia klasického Phongovho osvetľovacieho modelu, vytvorená Jimom Blinnom. Klasický Phongov osvetľovací model sa používa na vytvorenie spekulárnych odleskov. Je založený na tom, že odraz svetla od povrchu je kombinácia difúznej zložky drsného povrchu a spekulárneho odlesku lesklého povrchu. Modifikácia, ktorú prináša Blinn je, že namiesto počítania odrazového vektoru medzi smerom svetla a kamery, sa vypočíta polovičný vektor. To má za dôsledok, že spekulárne odlesky sú oveľa

hladšie a krajšie. Tento model bol použitý v skorých grafických kartách, ktoré mali fixnú funkcionalitu.

Physically based rendering

Physically based rendering (PBR) je kolekcia vykresľovacích techník, ktoré sa zakladajú na teórii svetla z reálneho sveta, a teda svetlo vypočítané týmto modelom vyzerá realistickejšie v porovnaní s napr. Blinn-Phong osvetľovacím modelom. Avšak, stále sa jedná len o aproximáciu reality založenej na fyzikálnych vlastnostiach materiálov. Veľkou výhodou tohoto modelu je, že dizajnéri môžu navrhovať povrchy materiálov na základe ich fyzikálnych vlastností tak, aby reflektovali materiály zo skutočného sveta. Azda najväčšou výhodou je, že osvetlené materiály budú vyzerať správne, nezávislé od svetelných podmienok[4].

2.3 Metódy vykresľovania

V tejto sekcii sú popísané štandartné metódy vykresľovania v reálnom čase: *Forward rendering* a *Deferred shading* [21].

2.3.1 Forward rendering

Forward rendering je azda najpoužívanejšia metóda. Pri tejto vykresľovacej metóde prebieha hneď po rasterizácii geometrie výpočet osvetlenia. Pre všetky vzniknuté fragmenty sa počíta zvolený osvetľovací model, a teda zložitosť tejto metódy je $\mathcal{O}(\textit{počet fragmentov} \cdot \textit{počet svetiel})$. Metóda Forward rendering je veľmi triviálna na implementáciu a môže byť veľmi rýchla[5]. Oproti metóde deferred shading podporuje priehľadnosť (color blending) a taktiež je v nej možné jednoducho použiť techniky, ako napr. *multi-sample anti-aliasing* (MSAA), ktoré by sa zložito implementovali v metóde deferred shading.

2.3.2 Deferred shading

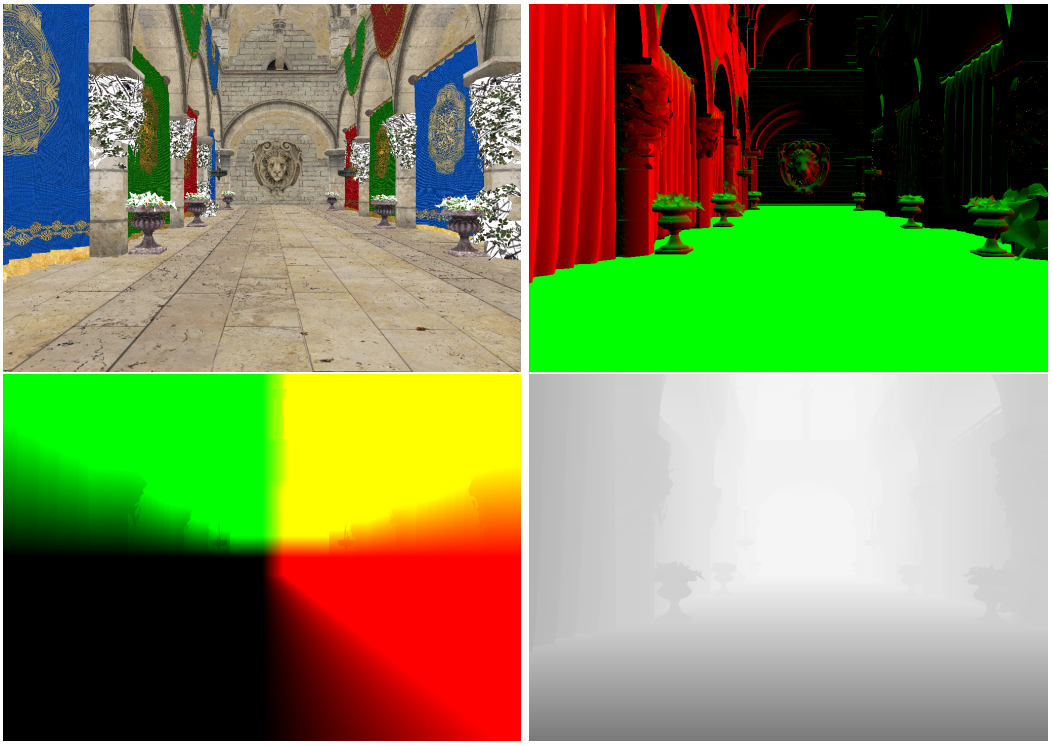
Deferred shading (možný preklad ako odložené tieňovanie) je metóda, ktorá redukuje počet objektov, pre ktoré sa musí počítať osvetľovací model. Všetky objekty prejdú prvým priechodom, v ktorom sa vykreslia za použitia viacnásobných vykresľovacích cieľov do jednotlivých geometrických bufferov alebo v skratke G-bufferov. Do G-bufferov (obr. 2.6) sa ukladajú farby daných fragmentov, normály, spekulárna zložka svetla, pozícia fragmentu, hĺbková mapa a prípadne ďalšie informácie. Týmto spôsobom sa redukuje počet fragmentov, pre ktorý sa musí počítať osvetlenie na $\mathcal{O}(\textit{rozlíšenie} \cdot \textit{počet svetiel})$. Samotný výpočet osvetlenia sa vykonáva ako 2D post proces.

Nevýhodou tejto metódy je náročnosť na veľkosť grafickej pamäte (rastúcou s rozlíšením a počtom použitých materiálov, resp. počtom G-bufferov), a teda aj s nárokmi na priepustnosť grafickej pamäte. Ďalšími nevýhodami, ako bolo spomínané v predchádzajúcej sekcii, sú nepodporované miešanie farieb[6] (alpha blending) a zložitejšie použitie techniky MSAA.

[4] V klasických osvetľovacích modeloch, ako napr. Blinn-Phong, museli dizajnéri upravovať (napr. zosvetlovať/stmavovať) povrchy pre rôzne svetelné podmienky, aby osvetlenie vyzeralo správne za všetkých podmienok.

[5] Hardwarové nároky tejto metódy sú nižšie, ako pri metóde deferred shading

[6] Miešanie farieb môže byť vyriešené použitím samostatného priechodu typu forward rendering pre transparentné objekty



Obr. 2.6: Príklad obsahu G-bufferov postupne zľava: albedo, normály, pozícia, hĺbková mapa.

Kompresia normál

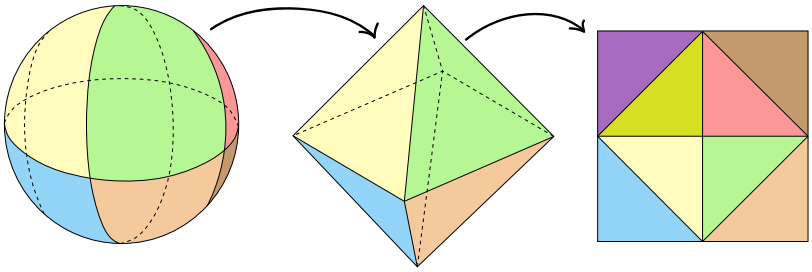
Článok *A Survey of Efficient Representations for Independent Unit Vectors* [6] popisuje a porovnáva množstvo komprimačných algoritmov, z ktorých najefektívnejší je algoritmus mapovania jednotkových vektorov na oktahedrón. Tento algoritmus má dobré rovnomerné rozloženie a je veľmi rýchly, s relatívne malou chybou oproti ostatným porovnávaným algoritmom.

Algoritmus funguje na princípe mapovania jednotkových vektorov (ktoré majú guľový charakter) na oktahedrón obyčajnou zmenou definície vzdialenosti z Euklidovskej (L^2) do Manhattanskej (L^1). Následne sú jednotlivé steny oktahedrónu premietnuté na rovinu. Tento spôsob mapovania zaručí jednoduchú a efektívnu kompresiu, resp. dekompresiu. Postup je znázornený na obrázku 2.7.

2.4 Tiled deferred shading

V tejto sekcii bude predstavená metóda tiled deferred shading [18]. Tiled deferred shading je optimalizácia metódy deferred shading, ktorá sa snaží zredukovať počet svetelných kalkulácií a hlavne znížiť nároky na priepustnosť pamäte, čo predstavuje najväčší problém metódy deferred shading, pretože pre každý fragment zasiahnutý svetlom musia byť prečítané informácie z G-bufferov. Jej optimalizácia spočíva v radení svetiel do *screen-space tiles* (dlaždíc), a teda výpočet osvetlenia prebieha len nad listom svetiel zasahujúcich danú *tile*. Tiled deferred shading má hneď niekoľko výhod voči klasickému deferred shading:

- Pre každý fragment sú G-buffre prečítané práve jedenkrát



Obr. 2.7: Na obrázku je znázornené mapovanie gule na oktahedrón a následne jej premietnutie na rovinu.

- Frame buffer je zapísaný práve jedenkrát
- Pre fragmenty v rovnakých *tiles* prebieha výpočet osvetlenia nad rovnakým zoznamom svetiel (dobrá pamäťová koherencia)
- Jednoduchý prechod medzi *deferred* a *forward* technikou

Jedna veľká nevýhoda je, že pri scénach s vysokou frekvenciou zmeny hĺbky je citeľná degradácia výkonu, pretože tu vzniká problém nazývaný *diskontinuita hĺbky*, ktorý bude popísaný v sekcii 2.4.3.

Tiled shading má podobnosti so starou metódou tiled rendering [9], a to, ako už názov napovedá, aplikovanie *tiles*. V metóde *tiled rendering* sa *tily* aplikujú na geometrické primitíva, na rozdiel od metódy tiled shading, v ktorej sa *tily* aplikujú na svetlá. *Tiled rendering* je v dnešnej dobe už nepoužiteľný, pretože scény majú veľkú komplexnosť (milióny geometrických primitív). Tiled deferred shading bol popísaný v mnohých článkoch [1, 3, 13, 26].

Metóda tiled shading je úspešne používaná v hernom priemysle. Objavila sa v hernej konzole Microsoft XBOX 360 a Sony Playstation 3 [26] s cieľom znížiť nároky na priepustnosť pamäte. S touto metódou sú spojené tituly ako **Battlefield 3** (grafický engine Frostbite) [1], **Uncharted: Drake's Fortune** [3].

Princíp metódy tiled deferred shading sa môže zrhnúť do nasledujúcich štyroch krokov, ktoré budú popísané v nasledujúcich sekciách:

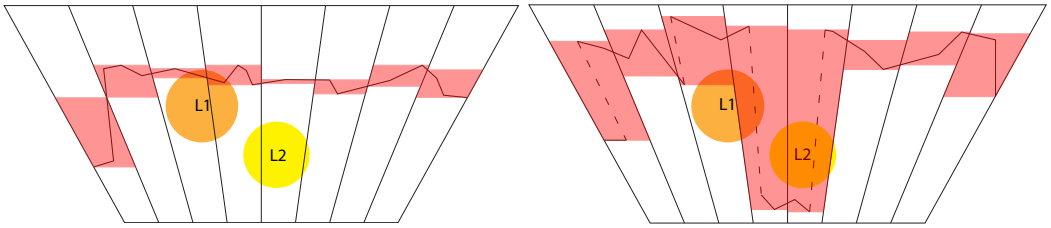
1. Vykreslenie geometrie do G-bufferov
2. Vytvorenie screen-space *tiles*
3. Priradenie svetiel
4. Výpočet osvetlenia a vykreslenie scény

2.4.1 Vykreslenie geometrie do G-bufferov

Tak ako pri klasickej metóde deferred shading, popísanej v sekcii 2.3.2, sa pomocou viacnásobných vykresľovacích cieľov vykreslí geometria do G-bufferov.

2.4.2 Vytvorenie screen-space tiles

Screen-space tiles sa dostanú rozsegmentovaním screen-space s pevnou veľkosťou kroku. Každá *tile* bude uchovávať list svetiel, ktoré ju zasahujú. Pri výbere veľkosti *tile* je nutné myslieť na to, že veľkosť kroku bude ovplyvňovať počet výpočtov osvetlenia, resp. počet



(a) (b)

Obr. 2.8: Vytvorenie subfrusta pre danú *tile* vo *view-space*. **Vľavo** Priradenie svetiel je efektívne, pokiaľ scéna obsahuje malú frekvenciu zmeny hĺbky. **Vpravo** Pokiaľ scéna obsahuje veľké hĺbkové nesúvislosti, vzniká tzv. hĺbková diskontinuita. Do *tiles* sa začnú priradzovať svetlá, ktoré neosvetľujú žiadne fragmenty.

svetiel na jednu *tile* a nároky na priepustnosť pamäte. Typická veľkosť *tile* je 32×32 alebo 64×64 pixelov.

2.4.3 Priradenie svetiel

Priradenie svetiel je najdôležitejší krok algoritmu. Prakticky sa *screen-space tiles* vytvárajú až v tomto kroku.

Na to, aby sa dalo rozhodnúť či svetlo potenciálne zasahuje fragmenty v danej *tile*, je nutné vytvoriť jej frustum (obr. 2.8a). Steny frusta sa získajú priamo z pozície tile vo *view-space* a minimálnej a maximálnej hĺbky fragmentov, ktoré sa v nej nachádzajú. Pokiaľ je medzi minimálnou a maximálnou hĺbkou veľký rozdiel, resp. scéna obsahuje veľké hĺbkové nesúvislosti, vzniká tzv. hĺbková diskontinuita (obr. 2.8b). Pokiaľ tento problém nastane, ostane frustum natiahnuté a priradia sa do neho svetlá, ktoré nezasahujú žiaden fragment, čo má negatívny dopad na výkonnosť metódy.

Po vytvorení frusta sa otestujú všetky svetlá a tie, ktorých obalová plocha (guľa pre bodové svetlá) zasahuje frustum, sa priradia do listu svetiel danej *tile*.

2.4.4 Výpočet osvetlenia

Výpočet osvetlenia sa voči klasickému deferred shading líši len v tom, že pre každý fragment sa musí získať list svetiel, ktoré zasahujú danú *tile*. Výpočet osvetlenia je naznačený pomocou pseudokódu 2.1.

```
1  for each G-Buffer fragment
2  {
3      samples = load samples from G-Buffers
4      affecting_lights = get lights affecting tile
5
6      for each light in affecting_lights
7      {
8          out color += calculateLighting(samples, light)
9      }
10 }
```

Výpis 2.1: Pseudokód výpočtu osvetlenia pre metódu tiled deferred shading

2.5 Clustered deferred shading

Clustered deferred shading vychádza z predchádzajúcej metódy, tiled deferred shading. Pri predchádzajúcej metóde sa *screen-space* delil na 2D *tile*, čo predstavovalo problém pri scénach s veľkou frekvenciou zmeny hĺbky - hĺbkovú diskontinuitu 2.8b. Aby sa tento problém eliminoval a zároveň sa vylepšila konzistencia rýchlosti vykresľovania, rozdelí sa *view-space* na 3D *clustre* (zhluky) [19].

Clustered shading je momentálny *state of the art*[7]. Najpoužívanejšia varianta je clustered forward (Forward++), ktorá našla uplatnenie napr. v **DOOM (2016)** [7] od spoločnosti *id Software* a je vôbec prvá hra napísaná vo Vulkan API. Varianta s metódou clustered deferred bola úspešne použitá v grafickom engine spoločnosti *Avalanche Studios* [22].

Algoritmus možno popísať nasledovnými krokmi:

1. Vykreslenie geometrie do G-Bufferov
2. Priradenie clustrov
3. Vytvorenie unikátnych clustrov
4. Zoradenie svetiel a vytvorenie BVH
5. Priradenie svetiel clustrom
6. Výpočet osvetlenia a vykreslenie scény

Krok 1, vykreslenie geometrie do G-Bufferov, je ten istý ako pri predchádzajúcej metóde a je popísaný v sekcii 2.3.2.

2.5.1 Priradenie clustrov

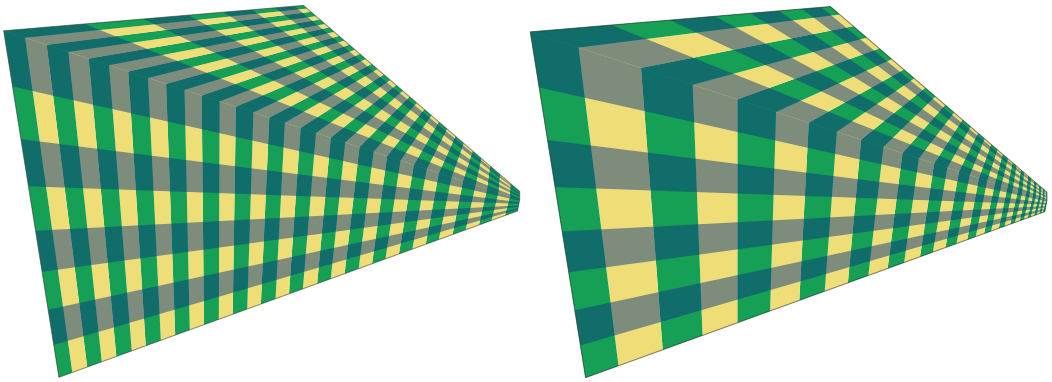
Cieľom tohoto kroku je na základe informácií z G-Bufferov deterministicky a najmä efektívne priradiť kľúč *clustru* danému fragmentu so snahou, aby vzorky, ktoré sú blízko pri sebe, boli v rovnakom clustry (zhluku). Existuje množstvo zhlukovacích algoritmov, z nich asi najznámejší je *k-means*, avšak žiadne z nich nie sú dostatočne efektívne pri miliónoch[8] prvkov (fragmentov). Z tohoto dôvodu je vhodné deliť vzorky do clustrov na základe ich pozície tak, aby výsledné clustre obsahovali čo najviac vzoriek, ale zároveň ich zasahovalo čo najmenej svetiel.

Najjednoduchšie by bolo rozdeliť world-space do pravidelnej mriežky, avšak to by vyžadovalo manuálne nastavenie parametrov mriežky pre každú scénu. Taktiež, pri hrách s otvoreným svetom (open world), sú scény zvyčajne veľmi veľké, a preto kľúč clustra by musel obsahovať veľa bitov na zakódovanie pozície vzorku. Z týchto dôvodov je lepšie vytvoriť delenie vo *view-space*.

Pri delení vo *view-space* sa môže vychádzať z delenia predchádzajúcej metódy, ktorá delí view-space na 2D *tile* a to pridaním ďalšej dimenzie – delenie pozdĺž z-osi. Toto delenie vytvorí pozdĺž z-osi malé subfrustá. Najintuitívnejšie je lineárne rozdeliť z-os na pravidelné segmenty. Avšak to by zapríčinilo, že segmenty blízko near plane by boli dlhé a úzke a naopak, pri far plane by boli široké a sploštené. Na vytvorenie čo najkubickejších subfrúst sa použije exponenciálne delenie (obr. 2.9).

[7] https://www.reddit.com/r/gamedev/comments/8klygv/

[8] Pri Full HD rozlíšení, ktoré je dnes už štandartné, to predstavuje viac ako 2 milióny vzorkov



Obr. 2.9: **Vľavo** Frustum vytvorené lineárnou segmentáciou pozdĺž z-osi. Clustre blízko near plane sú dlhé a úzke, a naopak pri far plane sú široké a sploštené. **Vpravo** Frustum vytvorené exponenciálnym delením pozdĺž z-osi. Toto delenie vytvára clustre s približne rovnakými dĺžkami strán.

Zaujímavé delenie použili v *Avalanche Engine*. Pretože ich scény mali veľkú hĺbku (0.1m - 50km), klasické exponenciálne delenie malo veľmi malú účinnosť. Z toho dôvodu limitovali far plane len do 500m. Pre fragmenty, ktoré sa nachádzali za posledným možným clustrom, používali iný systém na vykresľovanie svetiel v diaľke [22].

Tvorba kľúča

Na vytvorenie kľúča je nutné zistiť trojicu (i, j, k), ktorá jednoznačne identifikuje dané subfrustum, resp. zistiť, do ktorej *tile* spadá, a v ktorom segmente na z-osi sa nachádza. Pozícia tile sa získa priamo z pozície fragmentu a veľkosti tile $[t_x, t_y]$ ako

$$(i, j) = (x_{ss}/t_x, y_{ss}/t_y). \tag{2.2}$$

Exponenciálne delenie pozdĺž z-osi nemusí byť na prvý pohľad zrejmé, a preto bude postup podrobne popísaný. Pre k-ty segment sa získa pozícia near plane ako

$$near_k = near_{k-1} + h_{k-1} \tag{2.3}$$

a teda prvé delenie má hodnotu near plane *view-space* frusta, $near_0 = near$. Pri počte *tile* pozdĺž y-osi S_y a zornom poli (field of view; FOV) 2θ, sa získa krok pozdĺž z-osi ako

$$h_k = \frac{2\ near_k \tan\theta}{S_y}. \tag{2.4}$$

Po dosadení do rovnice 2.3 pre $k = 1$ sa získa vzťah

$$near_1 = near_0 + \frac{2\ near_0 \tan\theta}{S_y} \tag{2.5}$$

$$near_1 = near_0 \left(1 + \frac{2\tan\theta}{S_y}\right) \tag{2.6}$$

pre $k = 2$

$$near_2 = near_1 + h_1 \tag{2.7}$$

$$near_2 = near_1 + \frac{2\ near_1 \tan\theta}{S_y} \tag{2.8}$$

$$near_2 = near_1 \left(1 + \frac{2\tan\theta}{S_y}\right) \tag{2.9}$$

$$near_2 = near_0 \left(1 + \frac{2\tan\theta}{S_y}\right)\left(1 + \frac{2\tan\theta}{S_y}\right) \tag{2.10}$$

Z rovnice 2.10 je zrejmé, že vzdialenosť near plane pre k-ty segment bude

$$near_k = near_0 \left(1 + \frac{2\tan\theta}{S_y}\right)^k \tag{2.11}$$

a vyriešením rovnice pre k sa dostane vzťah

$$\frac{near_k}{near_0} = \left(1 + \frac{2\tan\theta}{S_y}\right)^k \tag{2.12}$$

$$\log\left(\frac{near_k}{near_0}\right) = k \log\left(1 + \frac{2\tan\theta}{S_y}\right) \tag{2.13}$$

$$k = \frac{\log(\frac{near_k}{near_0})}{\log\left(1 + \frac{2\tan\theta}{S_y}\right)} \tag{2.14}$$

Upravením rovnice 2.14 na celočíselné hodnoty a dosadením hĺbky z G-Bufferu z_{vs} sa dostane konečný vzťah na získanie segmentu z danej hĺbky

$$k = \left\lfloor \frac{\log(\frac{-z_{vs}}{near_0})}{\log\left(1 + \frac{2\tan\theta}{S_y}\right)} \right\rfloor \tag{2.15}$$

2.5.2 Vytvorenie unikátnych clustrov

Pretože mapovanie fragmentov k jednotlivým clustrom môže byť typu *M:1*, je vhodné vytvoriť list unikátnych clustrov, v ktorých sa fragmenty nachádzajú a až potom im priraďovať list svetiel.

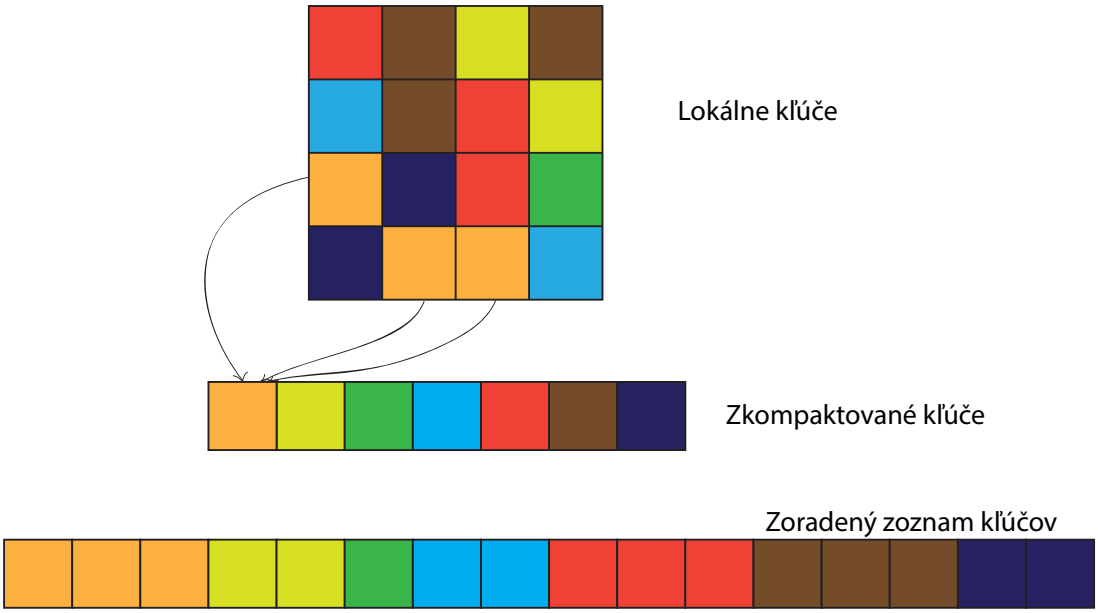
Najjednoduchší postup ako nájsť unikátne clustre paralelne, je zoradiť list clustrov a následne z nich vybrať unikátne clustre (tzv. kompakcia). Avšak radenie je stále výpočetne náročná operácia, a preto budú popísané efektívnejšie alternatívne metódy.

Lokálne zoradenie

V tejto technike sa síce radenie používa, ale radia sa len clustre spadajúce do jednotlivých *screen-space tiles* a vďaka tomu môže radenie prebiehať v rýchlej zdieľanej (shared) pamäti[9]. Po zoradení sa pomocou paralelnej kompakcie [4] vytvorí list unikátnych clustrov spadajúcej do danej *tile*.

Počas kompakcie sa pre každý fragment uloží ukazovateľ na unikátny cluster (obr. 2.10). Kompakcia zaručí vytvorenie listu globálne unikátnych clustrov.

[9]Pri veľkosti tile 32×32 to predstavuje len 1024 elementov na jednu *Work Group*, resp. 4kB zdielanej pamäte pri 32b kľúčoch



Obr. 2.10: Na obrázku je zobrazený priebeh lokálneho zoradenia a následnej kompakcie clustrov v *screen-space tile*. Po kompakcii sú ku kľúčom jednotlivých fragmentov pribalené ukazatele na unikátny cluster, do ktorého patria.

Tabuľky stránok

Technika s tabuľkami stránok má podobnosti s metódou virtuálnych textúr [14], avšak pretože počet všetkých možných clustrov je príliš veľký na to, aby sa zmestil do pamäte, je mapovanie kľúča na fyzickú pamäť nemožné[10].

Na prekonanie tohoto problému sa použijú tabuľky stránok, ktoré zaručia alokáciu fyzického miesta pre potrebné kľúče. Pretože stránky majú fixnú veľkosť, musia byť zo začiatku naplnené hodnotami, ktoré budú indikovať neprítomnosť kľúča. Tabuľky stránok alokujú potrebné stránky z lineárnej globálnej pamäte, čo prináša jednu veľkú výhodu a to, že alokované stránky sú už v kompaktnom rozsahu. Na získanie listu globálne unikátnych clustrov stačí zkompaktovať tento rozsah (obr. 2.11).

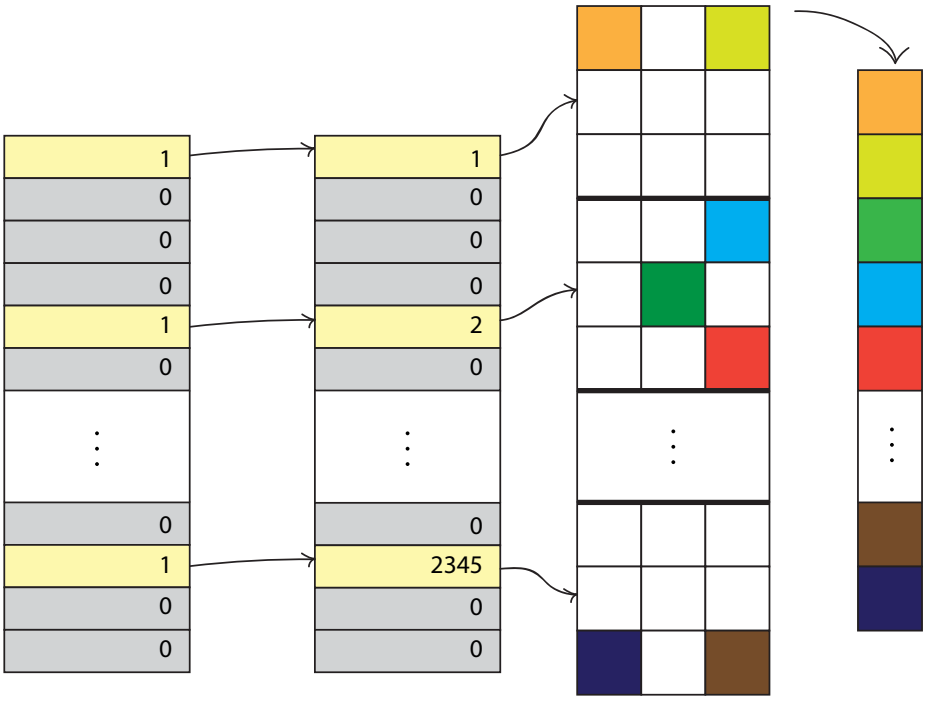
2.5.3 Zoradenie svetiel a vytvorenie BVH

Pri predchádzajúcej metóde tiled shading sa na priradenie svetiel do *tiles* prechádzal zoznam všetkých svetiel. To je postačujúce pre menší počet svetiel, avšak s narastajúcim množstvom je pokles výkonu citeľný a tento spôsob nevhodný. A pretože počet clustrov môže byť n-násobne väčší ako počet *tiles*, je nutné použiť optimalizačnú štruktúru.

Cieľom tohoto kroku je vytvoriť optimalizačnú stromovú štruktúru *bounding volume hierarchy* (BVH), ktorá umožní jednoduchý priechod a priradenie svetiel. Na to, aby bolo možné jednoducho skonštruovať BVH strom, je nutné svetlá zoradiť.

Efektívne radenie na GPU nie je jednoduché kvôli vysokej paralelite grafických kariet. Dmitri I. A. v článku *Sorting with GPUs: A Survey* [2] popisuje *state of the art* radiacich algoritmov.

[10] Pri dnešnom hardvéri by mapovanie bolo teoreticky možné, avšak bolo by nežiadúce alokovať väčšinu pamäte len pre túto operáciu



Obr. 2.11: Ukážka alokácie stránok. Stránky sa alokujú v lineárnom rozsahu, čo prináša žiadúci efekt – čiastočnú kompakciu. Zkompaktovaním tohoto rozsahu sa získa list globálne unikátnych clustrov.

Mortonova krivka

Na to, aby bolo možné svetlá radiť, je nutné namapovať ich 3D súradnice pozície na jednu dimenziu. Existuje množstvo mapovacích algoritmov [16]. Pre túto prácu bola vybratá Mortonova krivka (Z-krivka).

Mortonova krivka je definovaná tzv. Mortonovým kódom. Mortonov kód sa získa postupným prekladaním bitov jednotlivých dimenzií.

$$morton(x, y, z, \dots) = x_0, y_0, z_0, \dots, x_1, y_1, z_1, \dots, x_2, y_2, z_2, \dots$$

Bitonic-warp merge sort

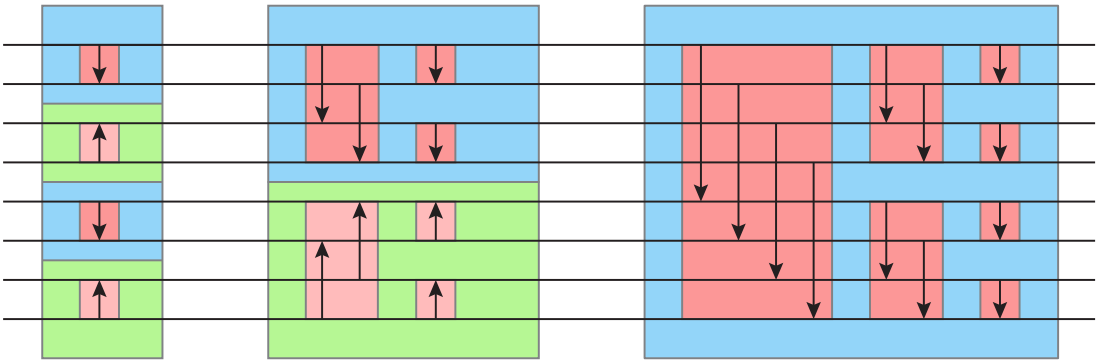
Bitonic-warp merge sort je porovnávací radiaci algoritmus, ktorý využíva bitonickú sieť na radenie menšieho[11] počtu elementov, ktoré sa postupne spájajú klasickým *merge sortom* pokiaľ sa nezotriedi celý list. Jeho efektívnosť spočíva v použití bezbariérovej a nedivergentnej bitonickej siete a rýchlym merge sortom [29].

Bitonická sieť funguje na princípe postupného radenia bitonických sekvencií. Bitonická sekvencia je postupnosť prvkov, ktorej elementy narastajú a potom klesajú (rovnica 2.16). Bitonická sieť pre n-prvkov pozostáva z $\log n$ fází, z ktorej každá fáza k pozostáva z $k+1$ krokov. Každý krok pozostáva z porovnania 2 prvkov, a teda bitonická sieť sa vytvorí postupnou rekurziou komparátorov (obr. 2.12).

$$x_0 \leq x_1 \leq \cdots \leq x_i > x_{i+1} > \cdots > x_{n-1} \tag{2.16}$$

Algoritmus možno popísať nasledovnými krokmi:

[11] Bitonická sieť je kvôli jej zložitosti efektívna len pre menší počet prvkov



Obr. 2.12: Bitonická sieť pre 8 prvkov. Šípky predstavujú jednotlivé komparátory, ktoré zoraďujú väčšie prvky v smere šípky. Komparátory v červených štvorcoch porovnávajú vždy vrchnú polovicu vstupných prvkov so spodnou. Červené štvorce sú použité na vytvorenie zelených a modrých, z ktorých každý tvorí tzv. *butterfly network* – všetky komparátory zoraďujú celú sekvenciu v jednom smere a po každom kroku rekurzívne s polovičným počtom prvkov.

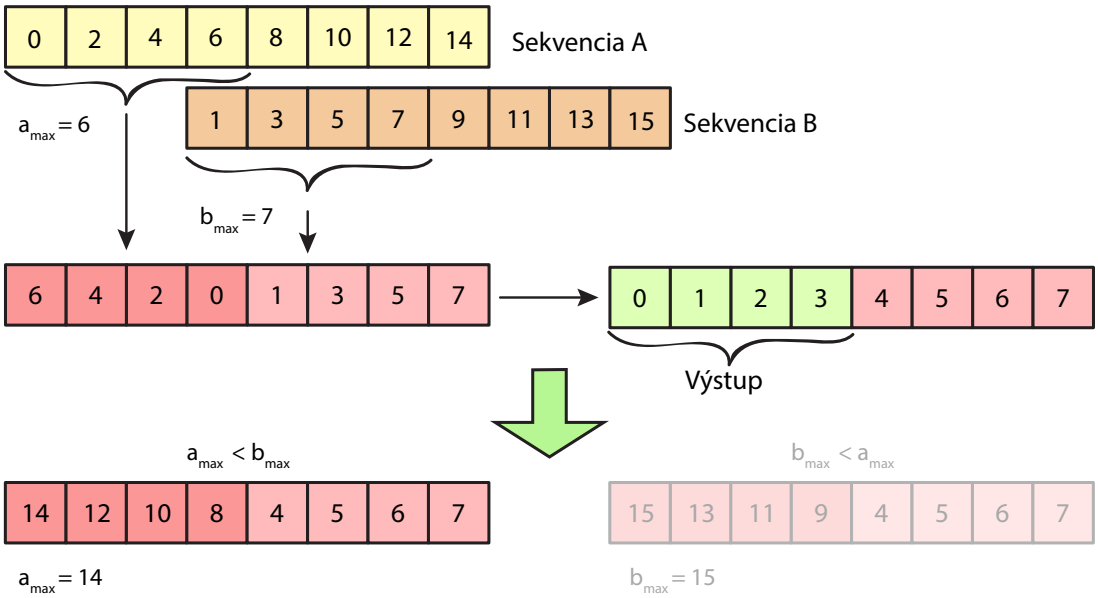
1. Rozdelenie vstupnej sekvencie na sekvencie pre bitonickú sieť
2. Spojenie vzniknutých sekvencií z kroku 1
3. Rozdelenie veľkých sekvencií z kroku 2 na menšie
4. Spojenie vzniknutých sekvencií z kroku 3

Pokiaľ je vstupná sekvencia v lineárnom poli, jej delenie na subsekvencie je triviálne. Na zefektívnenie sa využije fakt synchronizácie vláken vo warpe [10] – každú subsekvenciu radí jeden warp. Týmto spôsobom sa získa bezbariérový *bitonic-warp sort*, ktorý vytvára zoradené subsekvencie o počte n prvkov.

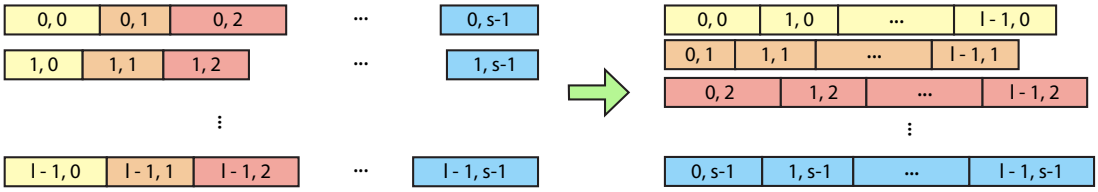
Vzniknuté sekvencie sa spájajú nasledovným spôsobom: dve vstupné sekvencie A a B, pri veľkosti bitonickej siete o n prvkoch, sa budú spájať tak, že z každej sekvencie sa zoberie spodných $n/2$ prvkov a uložia sa ich maximálne prvky a_{max}, b_{max}. Pokiaľ sa vybrané prvky zo sekvencie A usporiadajú s vybranými prvkami zo sekvencie B tak, aby tvorili bitonickú sekvenciu, stačí použiť posledný krok bitonickej siete na ich zoradenie (obr. 2.13). Po zoradení sa uloží spodných $n/2$ prvkov a vyberie sa ďalších $n/2$ prvkov z listu, ktorého najväčší prvok, a_{max} alebo b_{max}, bol väčší až pokiaľ sa nezlúčia obe sekvencie.

Počet zlučovaných sekvencií sa každým krokom geometricky znižuje. Keď je počet sekvencií nízky, paralelizmus zlyháva a algoritmus sa stáva neefektívny. Na obnovenie paralelizmu sa každá zvyšná sekvencia l rozdelí na s sublistov podľa oddelovačov (obr. 2.14. Oddelovače sa získajú náhodným vzorkovaním vstupnej sekvencie ešte pred prvým krokom a to tak, že sa vyberie náhodne $s \cdot k$ vzorkov, ktoré sa zoradia a následne sa z nich vyberie každý k-ty prvok. Väčšie hodnoty k tvoria kvalitnejšie oddelovače, avšak ich tvorba je pomalšia.

Po vykonaní kroku 3, vznikli veľké subsekvencie s, z ktorých každá obsahuje l sublistov. Vzniknuté sublisty $(l_0, s_i), (l_1, s_i), \ldots, (l_{n-1}, s_i)$ je opäť možné radiť paralelne. Vďaka dobre zvolenému počtu sublistov s je paralelizmus postačujúci počas celého radenia. Takto zoradené sekvencie sublistov S je možné jednoducho zlúčiť do finálne zoradenej sekvencie, pretože platí vzťah 2.17.



Obr. 2.13: Spájanie dvoch subsekvencií pri bitonickej sieti o veľkosti 8 prvkov. Bitonická sieť sa vytvorí pomocou 4 najmenších prvkov zo sekvencie A a B, ktoré sa usporiadajú tak, aby tvorili bitonickú sekvenciu. V ďalšej iterácii spájania sa odoberú 4 najmenšie prvky zo sekvencie, ktorej maximálny prvok bol menší.



Obr. 2.14: Vytvorenie menších sublistov z väčších listov podľa oddeľovačov. Oddeľovače sa získajú z počiatočnej nezoradenej sekvencie. Vytvorené sublisty sa opäť dajú radiť paralelne.

$$\forall a \in S_i, \forall b \in S_j, i < j : a < b \tag{2.17}$$

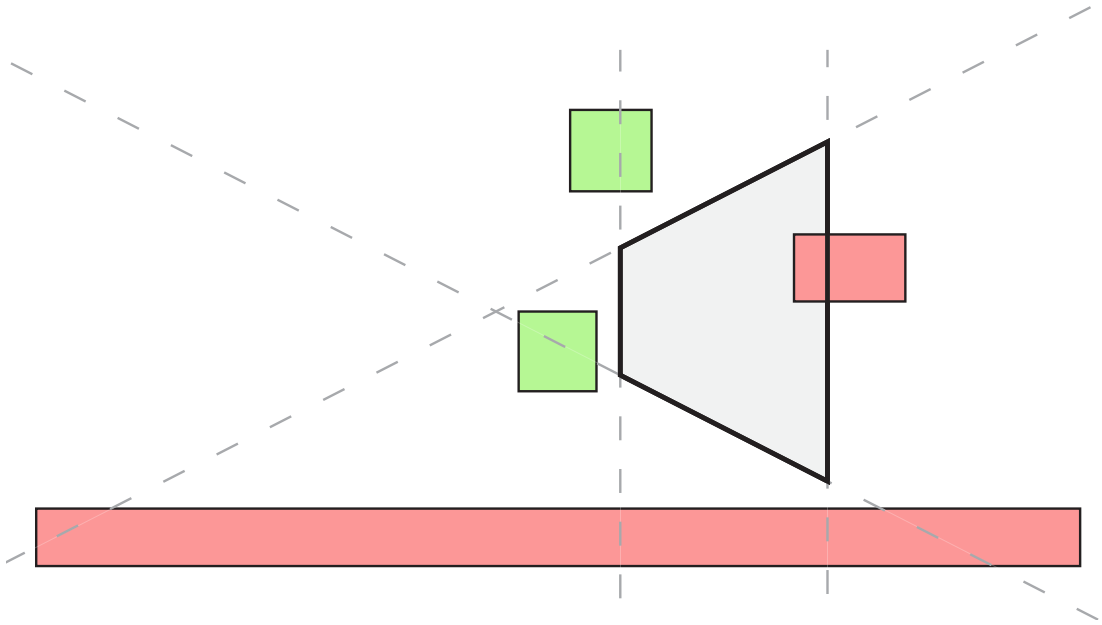
BVH strom svetiel

Bounding volume hierarchy je stromová akceleračná štruktúra, ktorá zhlukuje geometrické objekty do *bounding volumes* (obalových telies). Zhlukuje sa dovtedy, kým nevznikne koreňový uzol, ktorý obaľuje všetky objekty. Po zoradení svetiel na základe mortonovho kódu ich pozície je vytvorenie BVH stromu triviálne. Listy stromu budú tvoriť samotné zoradené svetlá a jednotlivé svetlá sa budú zhlukovať na základe vetviaceho faktora.

2.5.4 Priradenie svetiel clustrom

Na priradenie svetiel clustrom sa rekurzívne prechádza BVH strom do hĺbky. Na každej úrovni stromu sa testuje frustum daného clustra voči AABB (*axis-aligned bounding box*) daného uzla stromu. Pri listoch sa použije priamo obalová plocha daného svetla.

Množstvo algoritmov na test kolízie frusta s objektom je nedokonalých, avšak sú veľmi efektívne [23]. Klasický test frusta voči AABB predstavuje test, či sa všetky z ôsmich vrcholov AABB nachádzajú na vonkajšej strane steny (vo smere normály) frusta a pokiaľ áno, prehlási sa, že objekt je mimo frusta. Nedokonalosť tohoto testu spočíva v tom, že pokiaľ je frustum malé, resp. objekt príliš veľký, algoritmus môže v niektorých situáciách zlyhať, ako je naznačené na obrázku 2.15.



Obr. 2.15: Klasický test kolízie frusta s AABB zlyháva, pretože jeho návrh počíta s tým, že testované objekty sú mnohonásbne menšie ako samotné frustum. Pokiaľ je objekt oveľa väčší ako frustum, je označený, že zasahuje frustum, hoci tomu tak nie je.

Pretože sú svetlá zhlukované do obalových telies a zároveň sa testujú voči malému subfrustu, je nutné použiť pomalší, avšak perfektný test kolízie frusta. Test frusta a obalového telesa je založený na tvrdení, že pokiaľ pre dva konvexné polyhedrony A a B existuje rovina, ktorá ich oddeľuje a zároveň je paralelná so stranou A, alebo paralelná so stranou B, alebo paralelná s hranou A a hranou B, tak potom tieto dva polyhedrony nekolidujú [11].

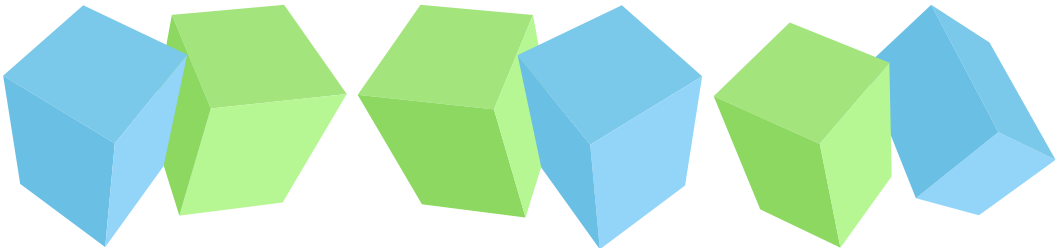
Zjednodušene povedané, prvý prípad testuje, či hrana objektu B koliduje so stranou objektu A, resp. test, či sa objekt B nachádza vnútri objektu A. Druhý prípad testuje opačnú situáciu, či hrana objektu A koliduje so stranou objektu B, resp. test, či sa objekt A nachádza vnútri objektu B. Posledný prípad je test medzi hranami objektov A a B a zároveň je výpočetne najnáročnejší zo spomínaných troch. Popísané typy kolízií sú zobrazené na obrázku 2.16.

Avšak, pokiaľ bude priraďovanie svetiel prebiehať vo *view-space*, možno posledný prípad úplne vynechať, pretože pri AABB vo *view-space* nemôže nikdy tento prípad kolízie nastať.

2.5.5 Vykresľovanie

Výpočet osvetlenia sa líši voči metóde tiled shading (sekcia 2.4.4 len v tom, ako sa získa list svetiel zasahujúcich cluster, do ktorého fragment spadá.

Pri použití lokálneho radenia sa pre každý fragment ukladá ukazateľ na *cluster* , do ktorého spadá. Tento postup je možné vidieť na obrázku 2.10.



Obr. 2.16: Vizualizácia jednotlivých prípadov testov. **Vľavo** Test kolízie hrany B so stranou A. **V strede** Test kolízie hrany A so stranou B. **Vpravo** Test kolízie hrany A s hranou B.

Pri použití tabuliek stránok, sa pri kompakcii, resp. vytvorení unikátnych kľúčov, uloží index unikátneho clustru na miesto v stránke, kde bol uložený kľúč daného clustru. Tento postup má žiadúci efekt – virtuálne vyhľadanie kľúča vráti index unikátneho clustra.

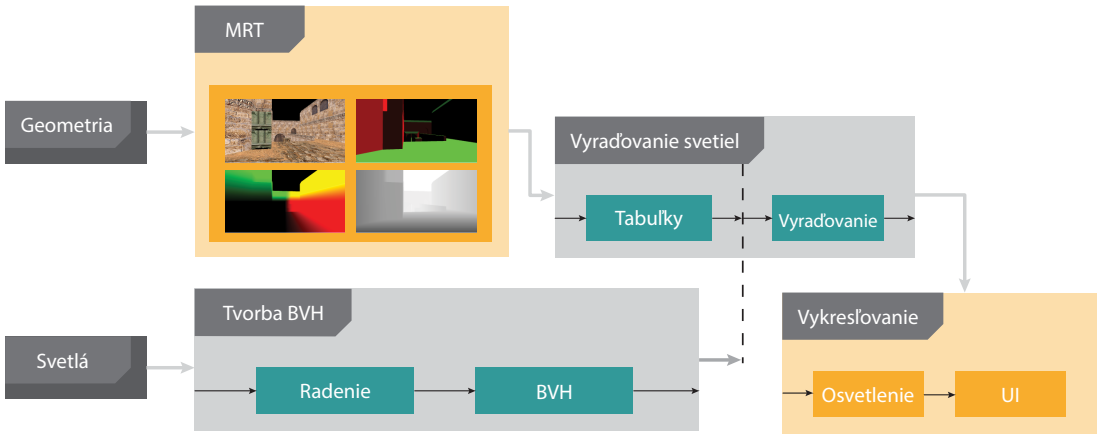
Kapitola 3

Návrh aplikácie

Táto kapitola bude preberať návrh aplikácie a odôvodnenie zvolenia niektorých navrhnutých krokov.

3.1 Vykresľovacia pipeline

Pri navrhovaní aplikácie bol kladený dôraz na to, aby všetky dáta, s ktorými sa bude pracovať, boli uchované v pamäti grafickej karty a minimalizoval sa prenos dát medzi GPU a CPU. Celá vykresľovacia pipeline je zobrazené na obrázku 3.1. Jednotlivé fázy algoritmu budú rozobrané v nasledujúcich sekciách.

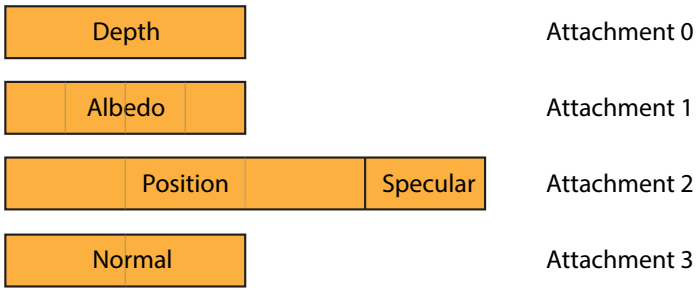


Obr. 3.1: Obrázok vizualizuje navrhnutú schému vykresľovacej pipeline.

3.2 Návrh G-bufferov

Štruktúru G-bufferov tvoria celkovo 4 textúry. Pri navrhovaní bol kladený dôraz na to, aby sa čo najviac znížili pamäťové nároky a zároveň sa zlepšila dátová koherencia, čo bude viesť k urýchleniu výpočtu.

Vizualizácia štruktúry G-bufferu je znázornená na obrázku 3.2. Tým, že technika deferred shading sama osebe neumožňuje vykresľovanie priehľadných objektov, ostal alfa kanál z albedo textúry nevyužitý. Taktiež sú normály transformované do *view-space*, čo prináša dve výhody: kompresiu normál a zjednodušenie výpočtu osvetlenia.



Obr. 3.2: Vizualizácia G-bufferov. Na zvýšenie presnosti je pre hĺbku použitých 32 bitov. Ostatné textúry majú presnosť 16 bitov na jeden kanál, okrem albedo textúry, pre ktorú je použitých 8 bitov na jeden kanál. Na šetrenie miesta je k textúre pozícií pribalená spekulárna zložka svetla. Taktiež vďaka kompresii stačia na uloženie normál len 2 kanály textúry.

Väčšina používaných textúr vyžaduje buď tri alebo len jeden farebný kanál, a pretože väčšina grafických kariet nepodporuje takéto druhy textúr[1], je vhodné pribaliť textúry s jedným farebným kanálom (napríklad spekulárna intenzita svetla pre daný materiál) k textúre, ktorej posledný farebný kanál by ostal nevyužitý. Druhým spôsobom, ako je možné znížiť pamäťové nároky, je kompresia. Kompresia bola použitá len pre textúry normál a je popísaná v sekcii 2.3.2 a pseudokód 3.1 zobrazuje komprimačný algoritmus.

```
oct_distance = 1.0 / (abs(normal.x) + abs(normal.y) + abs(normal.z))
plane = normal.xy * oct_distance

// Reflect the folds of the lower hemisphere over the diagonals
if normal.z <= 0
    plane = (1.0 - abs(plane.yx)) * sign_not_zero(plane)
```

Výpis 3.1: Pseudokód výpočtu kompresie normál

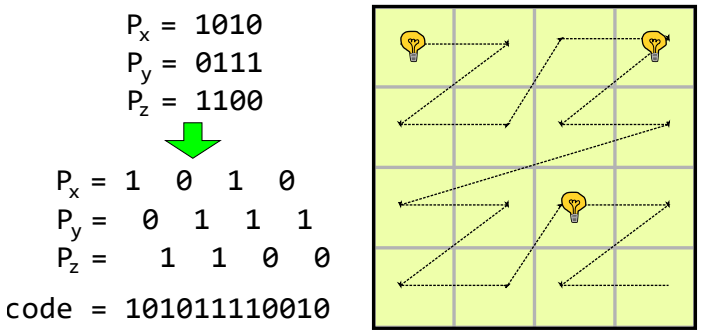
3.3 Tvorba BVH stromu svetiel

Tvorba BVH stromu prebieha asynchrónne (pokiaľ to daná platforma podporuje) a kompletne na grafickej karte. Na vytvorenie BVH stromu je nutné mať list svetiel zoradený. Bolo dokázané, že *radix sort* a jeho varianty je najrýchlejší radiaci algoritmus pre GPU [24]. Avšak na vytvorenie kvalitného radix sortu je nutné mať dostatok skúseností z viacerých oblastí. Z tohoto dôvodu a nedostatku času som sa rozhodol pre menej efektívnu variantu, *bitonic-warp-merge sort*, ktorá bola popísaná v sekcii 2.5.3.

3.3.1 Mortonov kód

Mortonov kód sa vytvára postupným prekladaním bitov jednotlivých dimenzí pozície daného svetla (obr. 3.3). Na mortonov kód sa používa dátový typ obsahujúci 32 bitov, resp. 10 bitov na jednu dimenziu. Z toho plynie, že maximálna rozloha svetla pokrýva ±512 jednotiek v každej dimenzii, a preto je vhodnejšie mať svetlá transformované do *view-space*.

[1] Moderné grafické karty z výkonnostných dôvodov nepoužívajú 24 bitové textúry, resp. textúry, pri ktorých by vznikal problém so zarovnaním na 32 bitov



Obr. 3.3: Znázornenie Mortonovej krivky pre pozíciu svetiel. Mortonova krivka mapuje súradnice bodu do jednej dimenzie postupným prekladaním jednotlivých bitov súradníc daného bodu.

Ako optimalizácia by sa dal využiť fakt, že svetlá, ktoré sú relevantné, budú len v zápornej z-rovine, a teda z-dimenzia by mohla byť v intervale $[-1023, 0]$.

Na efektívne vytvorenie mortonovho kódu sa využijú vlastnosti celočíselného delenia, ktoré je znázornené v pseudokóde 3.2 [12].

```
function expandBits(vector)
{
    vector = (vector * 0x00010001) & 0xFF0000FF
    vector = (vector * 0x00000101) & 0x0F00F00F
    vector = (vector * 0x00000011) & 0xC30C30C3
    vector = (vector * 0x00000005) & 0x49249249
    return vector
}

function morton3D(position)
{
    // clip to 10 bits
    position = min(max(position + 512.0f, 0.0f), 1023.0f)
    position = expandBits(position)
    return (position.x << 2) + (position.y << 1) + position.z
}
```

Výpis 3.2: Pseudokód tvorby mortonovho kódu

Na zaručenie práce pre všetky vlákna a nedivergentný priechod sa priradí každému warpu 128 svetiel na zoradenie. Hodnota 128 je daná tým, že radenie n-prvkov pozostáva z $n/2$ porovnávaní a zároveň sa prvá polovica porovnáva v smere zhora nadol a druhá zdola nahor. Z toho vyplýva, že list svetiel musí byť zarovnaný na násobky 128, kvôli charakteru bitonickej siete a na doplnenie, resp. zarovnanie, sa používa špeciálny mortonov kód `0xFFFFFFFF`, ktorý indikuje neplatný mortonov kľúč[2].

Ďalším krokom je fáza spájania vzniknutých 128 prvkových zoradených listov. Postup spájania bol popísaný v sekcii 2.5.3 a je zobrazený v pseudokóde 3.3.

[2]Vrchné 2 bity mortonovho kódu sú nevyužité, resp. vždy budú nulové.

```
sublist_A = move_lower_64_reversed(list_A)
sublist_B = move_lower_64(list_B)
max_A = max(sublist_A)
max_B = max(sublist_B)

while not merged
{
    // insitu sort sublists, sublist_A contains lowest 64
    bitonic_last_phase(sublist_A, sublist_B)

    merged_list.pushback_lower_64(sublist_A)

    if max_A > max_B
    {
        sublist_A = move_lower_64_reversed(list_A)
        max_A = max(sublist_A)
    }
    else
    {
        sublist_A = move_lower_64_reversed(list_B)
        max_B = max(sublist_A)
    }
}
```

Výpis 3.3: Pseudokód spájania dvoch zoradených zoznamov svetiel

Postupným spájaním zoznamov svetiel l sa ich počet geometricky znižuje. To má za následok úbytok na paralelizme, a s tým spojený úbytok výkonu. Na obnovenie paralelizmu sa zvyšné zoznamy svetiel l rozdelia na menšie podzoznamy s (sekcia 2.5.3). Vytvorenie oddeľovačov prebieha na CPU strane vybratím 1024 náhodnych svetiel[3], ktoré sú potom zoradené na základe ich mortonovho kódu pozície. Zo zoradených prvkov je vybratý každý ôsmy, čím sa vytvorí 128 oddeľovačov. Poslednému oddeľovaču je priradená maximálna hodnota `0xFFFFFFFF`, ktorá je používaná na zarovnanie zoznamov.

Po vytvorení podzoznamov je možné opäť radiť efektívne paralelne. Algoritmus radenia je taký istý ako pseudokód 3.3 s tým rozdielom, že jednotlivé podzoznamy je nutné zarovnať na 64 prvkov. Po zoradení podzoznamov vznikajú sekvencie zoradených svetiel, ktoré je možné jednoducho poskladať do finálne zoradeného zoznamu.

3.3.2 BVH strom

Po zoradení zoznamu svetiel je vytvorenie BVH stromu jednoduché. Listy stromu tvoria už zoradené svetlá. Pre efektívny priechod stromom je vhodné zhlukovať svetlá podľa počtu vláken v jednom warpe[4]. Bezprostredných 32, resp. 64 svetiel bude tvoriť jeden uzol v ďalšej úrovni stromu. Uzle v danej úrovni budú opäť rovnakým spôsobom zhlukované, až pokiaľ sa nedostane počet menší alebo rovnaký veľkosti vetviaceho faktoru.

[3] Výber prebieha na CPU strane kvôli nutnosti náhodného výberu a zároveň nízkemu paralelizmu.
[4] Nvidia má 32 vláken na jeden warp, AMD 64, Intel 32, ...

3.4 Tabuľky stránok

Na tvorbu unikátnych clustrov bol vybratý algoritmus s tabuľkami stránok, popísaný v sekcii 2.5.2, pretože sa ukázal byť efektívnejší ako algoritmus lokálneho radenia [19]. Algoritmus tabuliek možno rozdeliť do 4 krokov, resp. spúšťaných kernelov, ktoré budú popísané v nasledujúcich sekciách.

Tabuľka stránok je obyčajné pole prvkov, v ktorom jednotlivé prvky odpovedajú skupine clustrov. Identifikátor clustru tvorí trojica (c, j, i), kde c je segment, v ktorom sa daný cluster nachádza (sekcia 2.5.1), a dvojica (j, i) sú koordináty screen-space tile. Táto trojica je potom zabalená do 32 bitového integeru tak, že dvojica (j, i) tvorí spodných 14 bitov a pre pozíciu segmentu c je vyhradených 9 bitov. Pri veľkosti tile 32×32 postačuje 7 bitov pre jednu súradnicu na 4K rozlíšenie. Počet potrebných bitov pre parameter c je odvodený z rovnice 2.15, a teda pri parametroch near plane 0.05 a far plane 100.0 preň postačuje 9 bitov. Veľkosť stránky bola nastavená na 4096 možných clustrov, a teda tabuľka bude mať veľkosť 8kB. Skupina clustrov, resp. index do tabuľky sa získa ako

$$\left\lfloor \frac{c}{4096} \right\rfloor .$$

Veľkosť buffera stránok je experimentálne nastavená na maximálny počet 512 stránok (štvrtina všetkých možných clustrov pri danom nastavení scény).

3.4.1 Označenie stránok

V tomto kroku sa označia vstupy tabuľky stránok, pre ktoré sa musí alokovať pamäť. Tabuľka stránok sa na začiatku každého vykresľovania vynuluje, pretože kľúč s hodnotou 0 indikuje, že daný vstup nebol použitý. Fakt, že skupina clustrov s hodnotou 0 môže vzniknúť, je zohľadnený v poslednom kroku algoritmu.

Označenie nutnosti alokácie danej stránky je obyčajné zapísanie konštanty (hodnota 1) do príslušného vstupu tabuľky. A pretože táto operácia nevyžaduje atomický prístup, je veľmi rýchla.

Ako optimalizácia bolo vyskúšané ukladať značky do bitového poľa. Avšak táto operácia vyžadovala atomický prístup do pamäte a ukázala sa menej efektívna.

3.4.2 Alokácia potrebných stránok

Alokácia stránky predstavuje zapísanie unikátneho indexu stránky, číslovaného od 1, namiesto značky v tabuľke stránok. Priradenie značiek s konštantou 1 so sebou nesie výhodu a to, že unikátny index stránky sa získa inkluzívnou prefixovou sumou. Jednotlivé indexy stránok predstavujú ofset do globálneho poľa stránok.

3.4.3 Uloženie clustrov

Zbalené kľúče vzniknutých clustrov sa ukladajú do alokovaných stránok. Prekladom virtuálnej adresy (identifikátor clustra) sa získa ofset do globálneho poľa stránok, do ktorého sa uloží identifikátor clustra, ktorý bude spotrebovaný v ďalšom kroku. Preklad virtuálnej adresy na adresu v stránke je zobrazený v pseudokóde 3.4.

```
page_number = virtual_address / page_size
page_address = table[page_number] * page_size
offset = virtual_address % page_size

address = page_address + offset
```

Výpis 3.4: Pseudokód virtuálneho vyhľadania indexu unikátneho clustra. Adresa sa získa ako adresa v danej stránke a globálny ofset stránky

3.4.4 Vytvorenie zoznamu unikátnych clustrov

Na vytvorenie unikátnych clustrov sa využíva fakt, že stránky sú alokované v kompaktnom rozsahu, avšak nie všetky miesta v stránke sú platné identifikátory clustru. Za neplatný identifikátor sa používa hodnota 0, avšak cluster s hodnotou 0 vzniknúť môže. Z toho dôvodu sa cluster s týmto identifikátorom automaticky považuje za unikátny.

Na odstránenie neplatných clustrov z poľa obsahujúceho alokované stránky je použitá technika „*stream compaction*“ [5]. Pomocou nej sa vyextrahujú všetky platné identifikátory clustrov, čím vznikne pole unikátnych clustrov a na ich miesto v stránke sa zapíše ukazateľ do tohoto poľa.

3.5 Vyraďovanie svetiel

Pod vyraďovaním svetiel sa rozumie priradenie svetiel unikátnym clustrom, vytvoreným algoritmom tabuliek stránok. Na zistenie, ktoré svetlá zasahujú daný cluster, sa vytvorí subfrustum daného clustra z informácií získaných z G-Bufferov a identifikátoru clustra.

Subfrustum sa vytvorí v NDC (normalized device coordinates) a je transformované do *view-space*. Near a far plane sa získa inverznou funkciou funkcie 2.15. Tvorba subfrusta je zobrazená v nasledujúcom pseudokóde.

```
step = 2.0 * TILE_SIZE / (TILE_COUNT * TILE_SIZE)
ndc_points = create_ndc_points(step, tiledID)

near = inverse_cluster_ID(cluster_ID)
far = inverse_cluster_ID(cluster_ID + 1)
points_near = create_view_frustum_points(ndc_points, near)
points_far = create_view_frustum_points(ndc_points, far)

planes = create_frustum_planes(points_near, points_far)
```

Výpis 3.5: Pseudokód tvorby subfrusta. Subfrustum je vytvorené v NDC a následne transformované do view-space.

Po vytvorení subfrusta sa do hĺbky prechádza BVH strom vytvorený v predchádzajúcom kroku. Vďaka vhodne zvolenému vetviacemu faktoru (32 pri Nvidia kartách, 64 pri AMD) môže každý warp bezbariérovo prechádzať jednotlivé uzle. Prehľadávanie do hĺbky vyžaduje použitie rekurzie, avšak vo väčšine jazykoch pre GPGPU nie je rekurzia povolená, a preto je nutné prepísať rekurziu do cyklu. Na to sa využije dátová štruktúra zásobník. Na testovanie kolízií bol navrhnutý perfektný test kolízie subfrusta, popísaný v sekcii 2.5.4.

```
stack.push(bvh.nodes.root)

while stack not empty
{
    node = stack.top()
    stack.pop()

    if node is leaf
    {
        if collides_light(frustum, node)
            frustum.collisions.push(node)
    }
    else
    {
        if collides_aabb(frustum, node)
            stack.push_all(node.children)
    }
}
```

Výpis 3.6: Pseudokód rekurzívneho priechodu BVH stromom implementovaného pomocou cyklu a zásobníku.

Po vytvorení zoznamu svetiel, ktoré zasahujú daný cluster, sa uloží ich ukazateľ na miesto ukazateľa unikátneho clustra v tabuľke stránok.

3.6 Vykresľovanie

Vykresľovanie prebieha takmer tým istým spôsobom ako by prebiehalo pri klasickom *deferred* vykresľovaní. Jediné odlišnosti sú v tom, ako sa získa list svetiel a nutnosť vykonať dekompresiu normál.

List svetiel sa získa virtuálnym prekladom (pseudokód 3.4) identifikátoru clustra a na výpočet osvetlenia je použitý Blinn-Phong osvetľovací model. Pokiaľ výpočet osvetlenia prebieha vo view-space, sa pri výpočte osvetlenia nemusí počítať s kamerou, a teda, vedie to k čiastočnému zjednodušeniu rovníc.

Kapitola 4

Implementácia

V tejto kapitole budú rozobraté implementačné detaily návrhu popísaného v predchádzajúcej kapitole. Aplikácia je implementovaná v jazyku C++ s využitím štandartu 17 a moderných techník, ktoré prináša. Na vykresľovanie je použité Vulkan API, resp. jeho C++ wrapper, vo verzii 1.1. Preklad je implementovaný nástrojom Cmake a zdrojové kódy aplikácie sú verejne dostupné na github stránke[1].

4.1 Použité knižnice

Na tvorbu aplikačného okna a spracovanie vstupu z klávesnice a myši je použitá knižnica `GLFW`. Jej vybratie bolo podmienené najmä jednoduchosťou a multiplatformnosťou. Na vykresľovanie užívateľského rozhrania a nastavovanie parametrov je použitá knižnica `Dear ImGui`. Maticové, vektorové a iné výpočty zabezpečuje knižnica `GLM`. Na testovanie a ladenie aplikácie boli použité štandartné validačné vrstvy pre Vulkan. Načítavanie textúr zabezpečuje knižnica `LodePNG`. Táto knižnica dokáže načítať len PNG súbory. Ďalej boli použité knižnice `tinyobjloader` a `GLSLang` s použitím knižnice `spirv-tools`, ktorým budú venované nasledujúce sekcie.

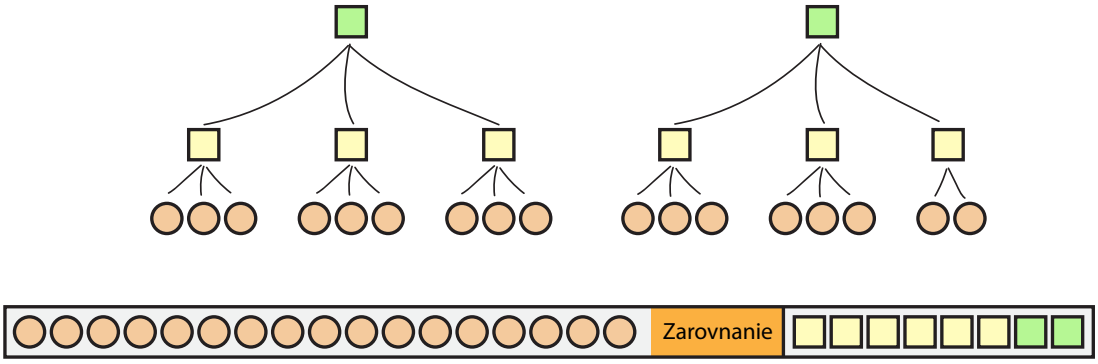
4.2 Načítanie modelov

Aplikácia dokáže načítať iba *wavefront* (`.obj`) modely a na ich načítanie je použitá knižnica `tinyobjloader`. Dôvodom použitia tejto knižnice je jednoduchosť jej použitia a zároveň je tenká (jeden hlavičkový súbor). Avšak táto knižnica interne nepočíta tangent vektory a na správne fungovanie osvetlenia bolo nutné ich dopočítať [27].

Na odľahčenie pamäte a zároveň potenciálne zrýchlenie vykresľovania je využité indexované vykresľovanie. Avšak *wavefront* súbory tento typ nepodporujú, resp. ich vertexi sú v nich duplikované. Na odstránenie duplikátov je využitá tabuľka s rozptýlenými položkami (*hash table*), do ktorej sa vložia všetky vertexi a zároveň sa ukladajú jednotlivé indexy pre kreslenie. Táto operácia je výpočetne náročná. Z tohto dôvodu sú takto spracované vawefront súbory uložené do mnou vytvoreného *.asd* cache súboru. Použitie tabuľky s rozptýlenými prvkami so sebou nesie väčšie nároky na pamäť, resp, pri komplexnejších scnénach proces odstraňovania duplikátov zaberá rádovo gigabajty pamäte RAM.

Na načítanie modelov paralelne bol použitý návrhový algoritmus *thread pool*. V tomto algoritme sa vytvorí pole hardvérovo dostupných vláken, ktorým sa priradí fronta s prácou,

[1] https://github.com/CaptainUnitaco/Clustered-Deferred-shading-in-Vulkan



Obr. 4.1: Obrázok ukazuje rozloženie BVH stromu v pamäti. Pre jednoduchosť je na obrázku strom s vetviacim faktorom 3 (typický vetviaci faktor je 32 alebo 64). Po vytvorení prvej úrovne musia byť indexy svetiel zarovnané na veľkosť AABB štruktúry. Strom sa vytvára až pokiaľ nezostane počet prvkov v najvyššej úrovni menší alebo rovný, ako je vetviaci faktor.

ktorú budú paralelne vykonávať. Znovupoužívaním vytvorených vláken sa zníži latencia vykonávania, ktorá vzniká pri vytváraní, resp. zanikaní vláken.

Každé vlákno paralelne nahráva command buffre s vertexami a indexami modelu. Okrem načítania modelu bol *thread pool* použitý na aktualizaovanie pozície svetiel pri ich animácii.

4.3 Preklad shader programov

Každý shader program musí byť pred použitím preložený do `SPIR-V` reprezentácie. Preklad prebieha nástrojom *GLSLang Validator*. Na zjednodušenie vývoja aplikácie je implementovaný preklad shader programov za behu aplikácie. Na preklad je použitá knižnica `glslang`, ktorá validuje prekladaný GLSL kód a následne z neho generuje `SPIR-V`. Taktiež umožňuje použitie jazyka HLSL.

S použitím knižnice `spirv-tools` je možné vytvorený `SPIR-V` kód optimalizovať. Knižnica ponúka 2 preddefinované druhy optimalizácie – v prospech rýchlosti alebo veľkosti a zároveň obsahuje funkciu na legalizáciu HLSL kódu [8]. Avšak, použitím optimalizátora sa nepreukázalo žiadne viditeľné zrýchlenie aplikácie.

4.4 Tvorba BVH stromu svetiel

Pred samotnou tvorbou BVH stromu sa list svetiel zoradí. Na radenie je použitý spomínaný *bitonic-warp merge* radiaci algoritmus. Tento radiaci algoritmus je implementovaný v sade kernelov s príponov „*sort_*“. Tvorba stromu prebieha v asynchrónnej compute fronte.

V prvej fáze algoritmu sú priradené zoznamy o veľkosti 128 prvkov jednotlivým warpom. Toto radenie prebieha in situ. Zároveň sa v tejto fáze transformujú pozície svetiel do view-space. Druhá fáza algoritmu, spájanie vzniknutých listov, už nemôže prebiehať in situ. Z tohoto dôvodu sú využité tzv. *ping pong* buffre – spájané listy sú odkladané do tzv. *swap* buffera a dispatch, ktorý bude spájať vyššiu úroveň, má tieto dva buffre vymenené.

Ďalšia fáza by bola vytvorenie podzoznamov, resp. obnovenie paralelizmu. Avšak, kvôli nedostatku času ostala táto fáza nedoimplementovaná. V posledných úrovniach spájania, keď pár warpov spája veľké zoznamy svetiel, je paralelizmus mizerný, čo sa odzrkadlilo na výkone (sekcia 5.3).

Po zoradení sa vytvorí zdola nahor BVH strom, ktorý je v pamäti uložený tak, že prvá úroveň obsahuje indexy do globálneho poľa svetiel a ďalšie úrovne obsahujú už AABB ich potomkov (obr. 4.1). Strom sa vytvára postupným spúšťaním kernelu, jeho vetviaci faktor je veľkosť warpu danej grafickej karty a je riadený na základe informácií (ofset na začiatok predchádzajúcej a terajšej úrovne a počet prvkov v terajšej úrovni) odovzdaných pomocou *push constants*.

4.5 Vyraďovanie svetiel

Vyraďovanie svetiel bolo implementované so snahou maximálnej utilizácie kariet od Nvidie a zároveň Amd. K tomu bolo použité rozšírenie `GL_KHR_shader_subgroup`, čo sú intrinsic funkcie na prácu s warpami. Použitie tohoto rozšírenia viedlo k viditeľnému zrýchleniu aplikácie.

Tvorba unikátnych clustrov je implementovaná v sade kernelov s predponou „*pt_*“. Po ich vytvorení prichádza gro celej aplikácie – priradenie svetiel unikátnym clustrom. Na spustenie kernelu je využitý *indirect dispatch*, ktorý vytvorí a spustí počet blokov dynamicky spočítaných rovnicou 4.1.

$$\left\lceil \frac{unikátne\ clustre}{počet\ warpov\ na\ blok} \right\rceil \tag{4.1}$$

Každému warpu sa priradí jeden unikátny cluster, ktorý bude testovať voči jednotlivým úrovniam BVH stromu. Na zníženie počtu prístupov do globálnej pamäte sa využíva rýchla zdieľaná pamäť, ktorá sa po dávkach zapisuje do globálnej pamäte. Aby mohol mať zoznam svetiel zasahujúcich cluster premenlivú dĺžku[2], je použitá jedna úroveň nepriameho odkazovania.

Výstupný buffer, do ktorého sa zapisujú zoznamy svetiel, je rozsegmentovaný následným spôsobom: na začiatku buffera sa vyhradí $64 \cdot (počet\ unikátnych\ clustrov)$ miest na nepriame odkazovanie a zvyšok buffera slúži na zoznamy svetiel. Z týchto 64 miest slúži prvé miesto ako počítadlo použitých nepriamych odkazov, druhé miesto na počet zabraných miest v poslednom nepriamom zozname a zvyšných 62 slúži na uchovávanie ofsetov na začiatky zoznamov. Jeden nepriamy zoznam má veľkosť 192 miest (768B). Je to dané tým, že jeden blok obsahuje 16 warpov a je preň vyhradených 16kB[3] rýchlej zdieľanej pamäte. Obrázok 4.2 zobrazuje popísané rozloženie pamäte.

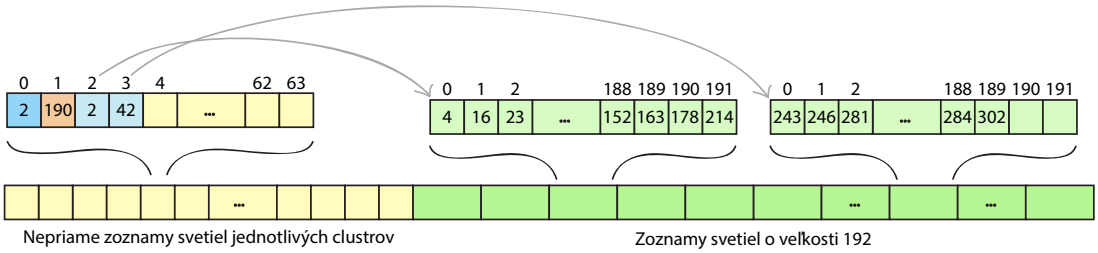
Na priechod BVH stromom je implementovaná rekurzia cyklom. Pretože zvolený vetviaci faktor BVH stromu je veľkosť jedného warpu, každé vlákno testuje jeden uzol z nižšej úrovne. Pokiaľ testovaná úroveň nie sú listy stromu, uloží sa bitová maska testu kolízií na zásobník a vyberie sa prvý uzol, ktorý koliduje s clustrom na testovanie. Takto sa cyklicky pokračuje, pokiaľ existujú nejaké hodnoty na zásobníku.

4.6 Debugging

Na implementáciu bolo použité *Visual Studio* od spoločnosti *Microsoft* s použitím nástroja *Resharper* od spoločnosti *JetBrains*, ktoré spolu tvoria sadu kvalitných prostriedkov na

[2] Pri implementácii metódy tiled shading bola veľkosť zoznamu fixne daná tak, že jedna *tile* mohla obsahovať maximálne 1024 svetiel. Pri prekročení tejto hranice sa zvyšné svetlá zahadzovali, čo viedlo k viditeľným artefaktom.

[3] Väčšina grafických kariet má 48kB zdielanej pamäte pre jeden blok, ale aby sa maximalizovala okupancia jednotlivých multiprocesorov, bola táto hodnota nastavená na 16kB. Okupancia bola odhadovaná nástrojom CUDA Occupancy Calculator.



Obr. 4.2: Na obrázku je zobrazené rozloženie pamäte zoznamu svetiel zasahujúce jednotlivé clustre. Každý unikátny cluster môže mať až 62 nepriamych zoznamov svetiel o veľkosti 192, resp. jeden cluster môže obsahovať maximálne 11904 svetiel.

vývoj a ladenie. Na skúmanie a ladenie výpočtov na grafickej karte boli použité nástroje *RenderDoc*, *Nsight Graphics* od spoločnosti Nvidia a *GPU PerfStudio* od spoločnosti Amd. *RenderDoc* poskytuje nástroje na prezeranie textúr, bufferov, descriptor settov, jednotlivé drawcalls, ale aj compute dispatch a veľa iných. Jeho veľká výhoda je, že funguje s Amd aj Nvidia kartami a je open-source. *Nsight Graphics* je pokročilejší nástroj, ktorý obsahuje takisto debugger a naviac v terajšej verzii aj profiler na skúmanie a ladenie výkonu aplikácie. Avšak, v momentálnej verzii funguje profiler len na najvyššiu radu grafických kariet[4], a pretože som v dobe písania práce nemal prístup k týmto kartám, nedokázal som aplikáciu vyladiť.

[4]Po konzultácii so senior technickým manažérom v Nvidii, by koncom roku 2019 mal byť profiler dostupný aj pre nižšie rady grafických kariet.

Kapitola 5

Zhodnotenie výsledkov

V tejto kapitole budú popísané vykonané experimenty a následne popísané dosiahnuté výsledky a ich zhodnotenie. Snímky z testovaných scén sú k dispozícii v prílohe A. Merania boli uskutočnené pomocou programov Renderdoc, Nvidia Nsight Graphics a GPU PerfStudio. Vzniknuté grafy sú ohraničené na maximálnu hodnotu $33ms$, resp. 30 FPS.

5.1 Použité architektúry

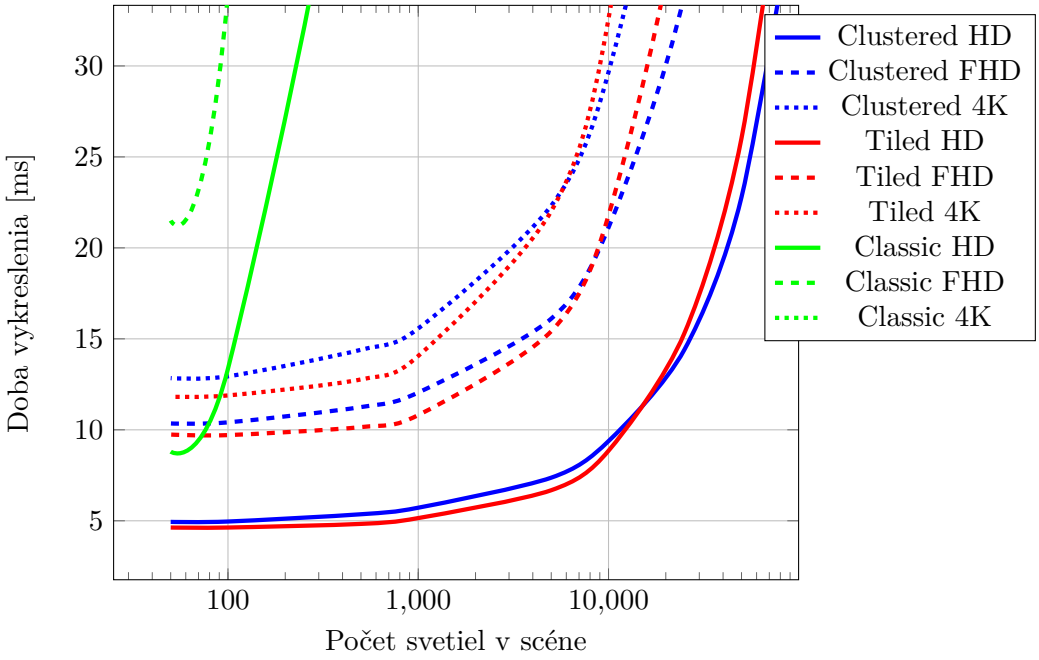
Na testovanie boli použité dve architektúry zobrazené v tabuľke 5.1 a prebiehalo pod operačným systémom Windows 10. Prvá uvedená zostava obsahuje mobilné verzie procesoru a grafickej karty. Druhá zostava je relatívne stará (jedná sa o najnižšiu architektúru grafickej karty od Amd, ktorá podporuje Vulkan), avšak výkonnosť grafickej karty je porovnateľná s prvou zostavou.

		Zostava 1	Zostava 2
Procesor		Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz	AMD Phenom(tm) II X4 B60 @ 3.31GHz
Operačný systém		Windows 10 Pro	Windows 10 Pro
Verzia vulkánu		1.1.101.0	1.1.101.0
Grafická karta	Typ	NVIDIA GeForce GTX 960m	AMD Radeon R9 270
	Architektúra	Maxwell	GCN1
	Rýchlosť jadra	1096MHz	900MHz
	Priepustnosť pamäte	80GB/s	179GB/s
	Veľkosť zbernice	128bit	256bit
	Počet shaderov	640	1280
	Driver	430.39	19.4.1

Tabuľka 5.1: Tabuľka zobrazuje použité zostavy počas testovania a vývoja aplikácie.

5.2 Porovnanie s predchádzajúcimi metódami

Porovnanie s predchádzajúcou metódou bolo vykonané len na prvej architektúre. Na porovnanie bola použitá scéna *Crytek Sponza*, ktorá obsahuje minimum hĺbkových nesúvislostí.



Obr. 5.1: Graf nameraných hodnôt pre scénu Sponza. Minimum hĺbkových nesúvislostí spôsobilo, že clustre zdegenerovali na dlaždice a pri vyššom počte svetiel sa prejavilo vyraďovanie svetiel pomocou BVH stromu.

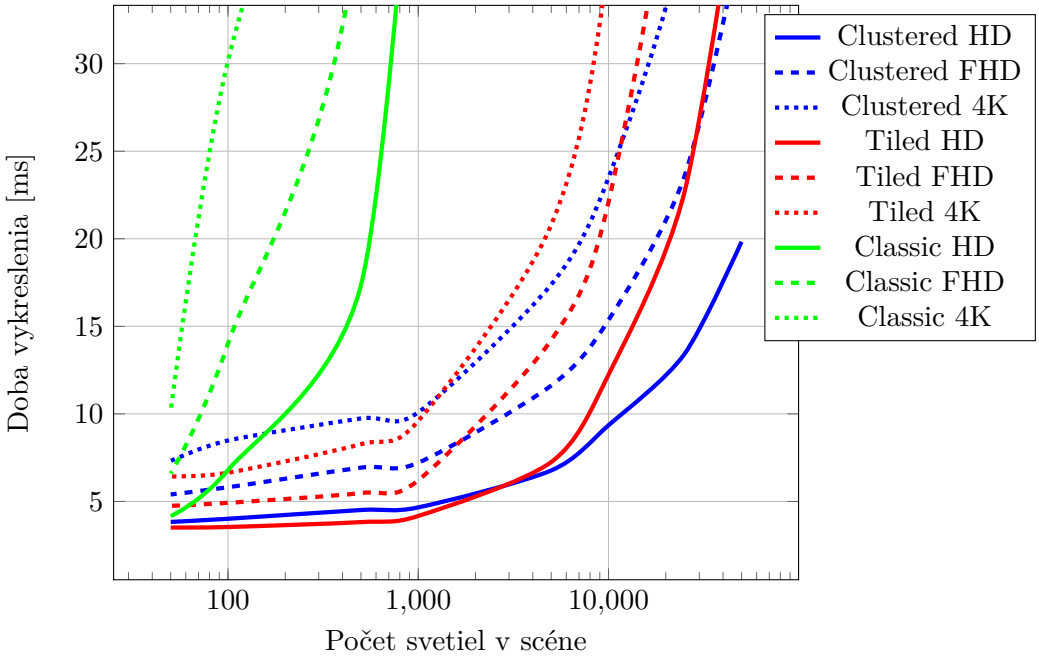
Na zobrazenie skutočného potenciálu a robustnosti metódy clustered shading bola použitá scéna *Monbulk Creek Trestle bridge*[1], ktorej vegetácia a piliére mostu vytvárajú v určitých pohľadoch kamery množstvo hĺbkových nesúvislostí. Testy sa vykonávali pri rôznych rozlíšeniach s veľkosťou *tile* 32×32. Namerané hodnoty sú zobrazené v grafoch 5.1 a 5.2. Porovnávaný bol klasický deferred shading, tiled deferred shading a nakoniec clustered deferred shading. Husté rozmiestenie svetiel v scéne spôsobuje, že tiles, resp. clustre, sú nimi vysoko saturované a teda je nutné počítať veľké množstvo svetelných kalkulácií.

Je nutné podotknúť, že implementácia metódy tiled shading má veľkosť zoznamu svetiel pre jednotlivé *tiles* fixne nastavených na 1024 svetiel. To zapríčiní horné ohraničenie maximálneho množstva svetelných kalkulácií a zároveň spôsobí artefakty, pri vysokej saturácii *tiles* svetlami.

Z grafu 5.2 je možné pozorovať prejavenie hĺbkových nesúvislostí v metóde Tiled shading. Frustum dlaždíc zdegenerovalo na obyčajný 2D test a počet svetiel zasahújúcich toto objemné frustum je priveľký. To zapríčiní množstvo redundantných svetelných kalkulácií (aj napriek hornému ohraničeniu počtu svetiel) a tým výrazný pokles výkonu.

Avšak, pokiaľ scéna obsahuje malý počet týchto nesúvislostí, je metóda tiled shading efektívnejšia (graf 5.1). V metóde clustered shading sa prejavil len efektívnejší priechod zoznamom svetiel v scéne – pokiaľ scéna obsahuje dostatočne veľký počet svetiel, je metóda clustered shading efetívnejšia.

[1]Voľne dostupná na stránke `https://skfb.ly/6JOMC`



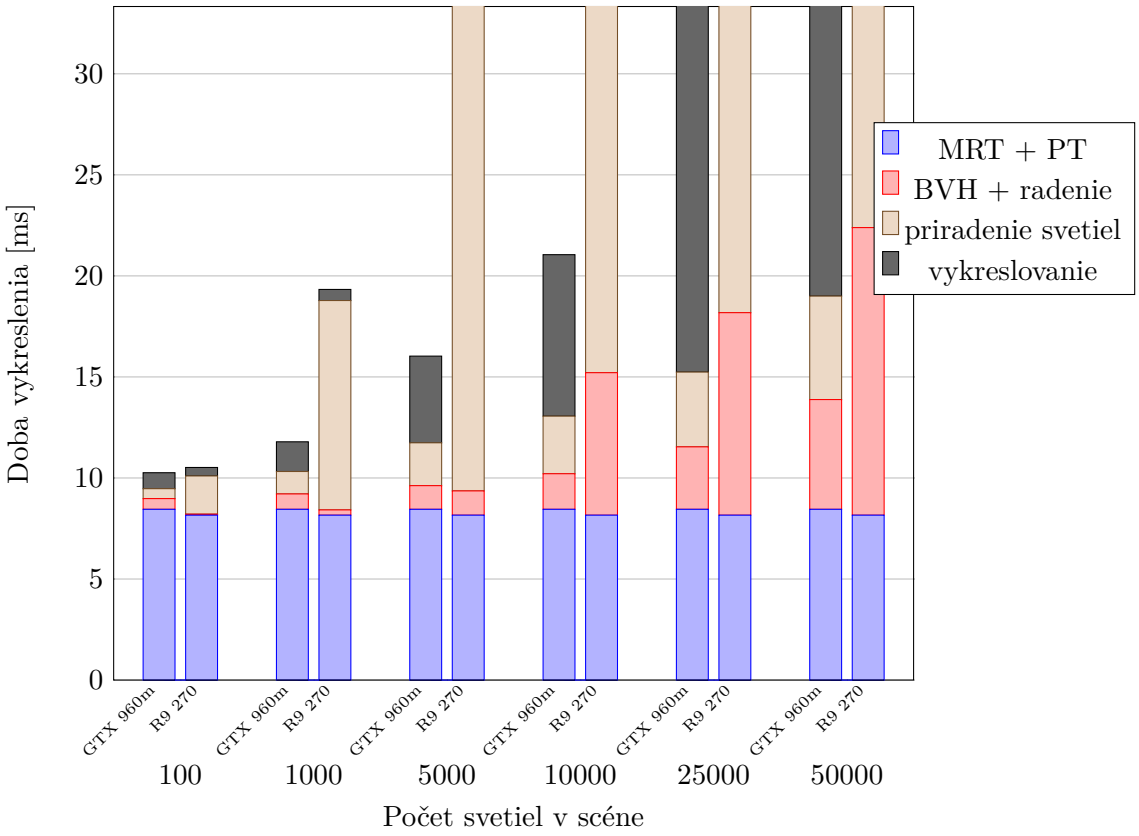
Obr. 5.2: Graf nameraných hodnôt pre scénu Creek Trestle Bridge. Z grafu je vidieť, že pri vyššom počte svetiel sa prejavujú hĺbkové nesúvislosti. Pri lepšom radiacom algoritme, by sa táto hranica výrazne posunula v prospech Clustered metódy.

5.3 Rýchlosť clustered deferred shading fázy

Výkonnosť jednotlivých fáz algoritmu clustred deferred shading bola testovaná na scéne sponza s veľkosťou *tile* 32×32. Výsledky testov boli získané pomocou aplikácie *Nvidia Nsight Graphics* pre Nvidia kartu a *GPU PerfStudio* pre Amd kartu. Nástroj *RenderDoc* ukazoval skreslené a často nepresné výsledky[2].

Použitie kvalitnejšieho radiaceho algoritmu by viedlo k značnému urýchleniu a možnosti použitia ešte väčšieho počtu (až milióny) svetiel. Z nameraných hodnôt (graf 5.3) je vidieť, že implementácia tejto práce je menej efektívna pre grafickú kartu od Amd.

[2]Predpokladám, že nepresné výsledky v nástroji *RenderDoc* boli spôsobené použitím compute fronty



Obr. 5.3: Na obrázku sú zobrazené časové úseky jednotlivých krokov algoritmu clustered deferred shading na testovaných zostavách. Vykreslenie geometrie do G-Bufferov (MRT) je zlúčené s vytváraním tabuliek stránok (PT), pretože vždy trvajú rovnakú dobu a zároveň vytvorenie stránok je veľmi rýchla operácia (0,3ms) a v grafe by sa táto informácia stratila. Z grafu je možné pozorovať, že implementácia metódy v tejto práci je menej efektívna na architektúre od Amd.

Kapitola 6

Záver

Cieľom tejto bakalárskej práce bolo vytvorenie aplikácie na vykresľovanie väčšieho počtu svetiel v reálnom čase. Vo výslednej aplikácii bola implementovaná metóda clustered deferred shading v modernom grafickom aplikačnom rozhraní Vulkan. Pre porovnanie bola naviac implementovaná predchádzajúca metóda tiled deferred shading.

Vo vytvorenej aplikácii je jednoduchým spôsobom možné meniť implementované metódy a veľkosť *screen-space tiles* za behu. Taktiež na jednoduché ladenie bol implementovaný preklad shader programov za behu aplikácie.

Výsledná aplikácia splnila zadané požiadavky a dokázala vykresľovať tisíce svetiel v reálnom čase na grafických kartách od Nvidie. Taktiež sa preukázala robustnosť implementovanej metódy voči predchádzajúcej metóde a vyriešili sa jej nedostatky pri veľkej frekvencii zmeny hĺbky. Avšak, pri grafických kartách od spoločnosti Amd sa preukázali nedostatky implementácie a implementovaná metóda bola na nich neefektívna, konkrétne fáza priradenia svetiel clustrom.

Táto práca bola vybratá na konferenciu Excel@FIT 2019. Ponúka základ pre techniku vykresľovania väčšieho počtu svetiel. Na docielenie väčšej vizuálnej presnoti by bolo vhodné doimplementovať tiene podľa článku *More Efficient Virtual Shadow Maps for Many Lights* [20]. V súčasnosti sa dostáva do popredia *physically based rendering*, pretože výkon dnešných grafických kariet je už postačujúci.

V terajšej implementácii sa prenáša na grafickú kartu zoznam všetkých svetiel. Tu sa otvára mnoho optimalizačných možností. Jedna z nich by predstavovala rozsegmentovať zoznam svetiel na sektory a BVH strom vytvárať len zo sektorov, ktoré sú blízko kamery. Ďalšia optimalizácia by mohla byť zlučovanie svetiel, ktoré sú ďaleko od kamery [28].

Taktiež použitý radiaci algoritmus je zbytočne komplikovaný a nemapuje sa dobre na architektúru grafických kariet. Implementovaním kvalitného radix sortu by sa dosiahol značný nárast výkonu, resp. zrýchlenie fázy vytvárania BVH stromu [24, 2].

Literatúra

[1] Andersson, J.: Parallel graphics in frostbite - current future. *SIGGRAPH Course: Beyond Programmable Shading*, 2009.

[2] Arkhipov, D. I.; Wu, D.; Li, K.; aj.: Sorting with GPUs: A Survey. *CoRR*, ročník abs/1709.02520, 2017, 1709.02520.
URL http://arxiv.org/abs/1709.02520

[3] Balestra, C.; Engstad, P.-K.: The technology of uncharted: Drake's fortune. *Game Developer Conference*, 2008.

[4] Billeter, M.; Olsson, O.; Assarsson, U.: Efficient Stream Compaction on Wide SIMD Many-core Architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-603-8, s. 159–166, doi:10.1145/1572769.1572795.
URL http://doi.acm.org/10.1145/1572769.1572795

[5] Billeter, M.; Olsson, O.; Assarsson, U.: Efficient Stream Compaction on Wide SIMD Many-core Architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-603-8, s. 159–166, doi:10.1145/1572769.1572795.
URL http://doi.acm.org/10.1145/1572769.1572795

[6] Cigolle, Z. H.; Donow, S.; Evangelakos, D.; aj.: A Survey of Efficient Representations for Independent Unit Vectors. *Journal of Computer Graphics Techniques (JCGT)*, ročník 3, č. 2, April 2014: s. 1–30, ISSN 2331-7418.
URL http://jcgt.org/published/0003/02/01/

[7] Courrèges, A.: DOOM (2016) - Graphics Study. 2016, [Online; navštívené 16.4.2019].
URL http://www.adriancourreges.com/blog/2016/09/09/doom-2016-graphics-study/

[8] Fischer, G.: SPIR-V Legalization and Size Reduction with spirv-opt. may 2018.
URL https://www.lunarg.com/wp-content/uploads/2018/06/SPIR-V-Shader-Legalization-and-Size-Reduction-Using-spirv-opt_v1.2.pdf

[9] Fuchs, H.; Poulton, J.; Eyles, J.; aj.: Pixel-planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-enhanced Memories. *SIGGRAPH Comput. Graph.*, ročník 23, č. 3, Júl 1989: s. 79–88, ISSN 0097-8930, doi:10.1145/74334.74341.
URL http://doi.acm.org/10.1145/74334.74341

[10] Gao, H.: Basic Concepts in GPU Computing. 2017, [Online; navštívené 24.4.2019].
URL https://medium.com/@smallfishbigsea/basic-concepts-in-gpu-computing-3388710e9239

[11] Heckbert, P. S. (editor): *Graphics Gems IV*. San Diego, CA, USA: Academic Press Professional, Inc., 1994, ISBN 0-12-336155-9.

[12] Karras, T.: Thinking Parallel, Part III: Tree Construction on the GPU. December 2012, [Online; navštívené 27.4.2019].
URL https://devblogs.nvidia.com/thinking-parallel-part-iii-tree-construction-gpu/

[13] Lauritzen, A.: Deferred rendering for current and future rendering pipelines. *SIGGRAPH Course: Beyond Programmable Shading*, 2010: s. 1–34.

[14] Mayer, A. J.: *Virtual Texturing*. Diplomová práca, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, Október 2010.
URL
https://www.cg.tuwien.ac.at/research/publications/2010/Mayer-2010-VT/

[15] McClanahan, C.: History and Evolution of GPU Architecture: A Paper Survey. *Last accessed*, ročník 31, č. 01, 2010: str. 2015.

[16] Mokbel, M. F.; Aref, W. G.; Kamel, I.: Analysis of multi-dimensional space-filling curves. *GeoInformatica*, ročník 7, č. 3, 2003: s. 179–209.

[17] Olena: A Brief History of GPU. 2018.

[18] Olsson, O.; Assarsson, U.: Tiled shading. *Journal of Graphics, GPU, and Game Tools*, ročník 15, č. 4, 2011: s. 235–251.

[19] Olsson, O.; Billeter, M.; Assarsson, U.: Clustered Deferred and Forward Shading. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, Goslar Germany, Germany: Eurographics Association, 2012, ISBN 978-3-905674-41-5, s. 87–96, doi:10.2312/EGGH/HPG12/087-096.
URL https://doi.org/10.2312/EGGH/HPG12/087-096

[20] Olsson, O.; Billeter, M.; Sintorn, E.; aj.: More Efficient Virtual Shadow Maps for Many Lights. *Visualization and Computer Graphics, IEEE Transactions on*, ročník 21, č. 6, 2015: s. 701–713, ISSN 1077-2626, doi:10.1109/TVCG.2015.2418772.
URL http://dx.doi.org/10.1109/TVCG.2015.2418772http://efficientshading.com/wp-content/uploads/clustered_shadows_tvcg.pdf

[21] Owens, B.: Forward Rendering vs. Deferred Rendering. 2013, [Online; navštívené 16.4.2019].
URL https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342

[22] Persson, E.: Practical clustered shading. *SIGGRAPH Course: Advances in Real-Time Rendering in Games*, 2013.

[23] Quilez, I.: Fixing frustum culling. 2013, [Online; navštívené 25.4.2019].
URL https:
//www.iquilezles.org/www/articles/frustumcorrect/frustumcorrect.htm

[24] Satish, N.; Harris, M.; Garland, M.: Designing Efficient Sorting Algorithms for Manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, Washington, DC, USA: IEEE Computer Society, 2009, ISBN 978-1-4244-3751-1, s. 1–10, doi:10.1109/IPDPS.2009.5161005.
URL https://doi.org/10.1109/IPDPS.2009.5161005

[25] Seitz, C.: Evolution of gpus. *Nvidia Corporation*, 2004.

[26] Swoboda, M.: Deferred lighting and post processing on playstation 3. In *Game Developer Conference*, 2009.

[27] Vries, J. D.: Normal Mapping. 2014, [Online; navštívené 29.4.2019].
URL https://learnopengl.com/Advanced-Lighting/Normal-Mapping

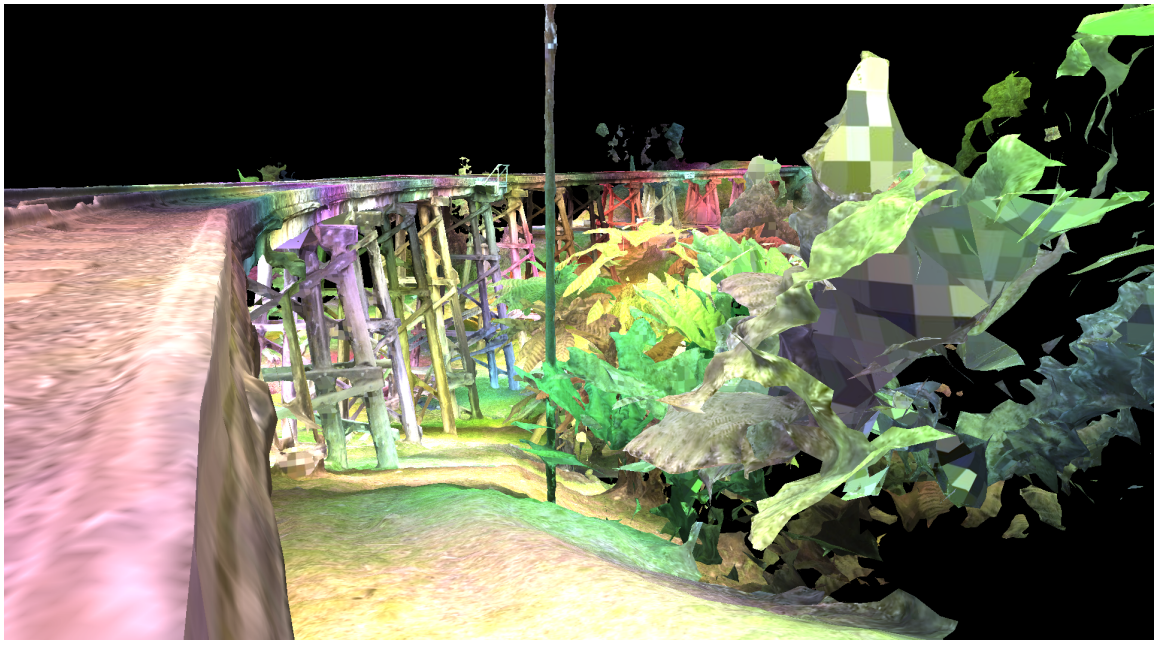
[28] Walter, B.; Fernandez, S.; Arbree, A.; aj.: Lightcuts: a scalable approach to illumination. In *ACM Transactions on graphics (TOG)*, ročník 24, ACM, 2005, s. 1098–1107.

[29] Ye, X.; Fan, D.; Lin, W.; aj.: High performance comparison-based sorting algorithm on many-core GPUs. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010, ISSN 1530-2075, s. 1–10, doi:10.1109/IPDPS.2010.5470445.

Príloha A

Snímky testovaných scén

Obr. A.1: Snímka scény *Sponza*, ktorá obsahuje 25k svetiel.



Obr. A.2: Snímka scény *Monbulk Creek Trestle bridge*, ktorá obsahuje 25k svetiel.