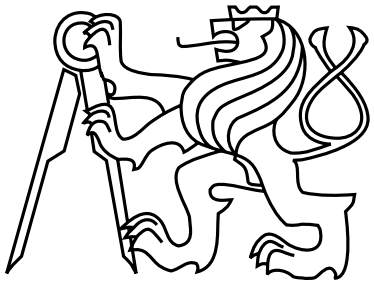
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Pohledově závislá aplikace textur v reálném čase

Bc. Daniel Princ

Vedoucí práce: Ing. David Sedláček, Ph.D.

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

10. května 2014

Poděkování

Chtěl bych poděkovat vedoucímu této práce, panu Ing. Davidovi Sedláčkovi, Ph.D., za poskytnutí cenných rad a nápadů.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze, 12. 5. 2014 .....

Abstract

TBD

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

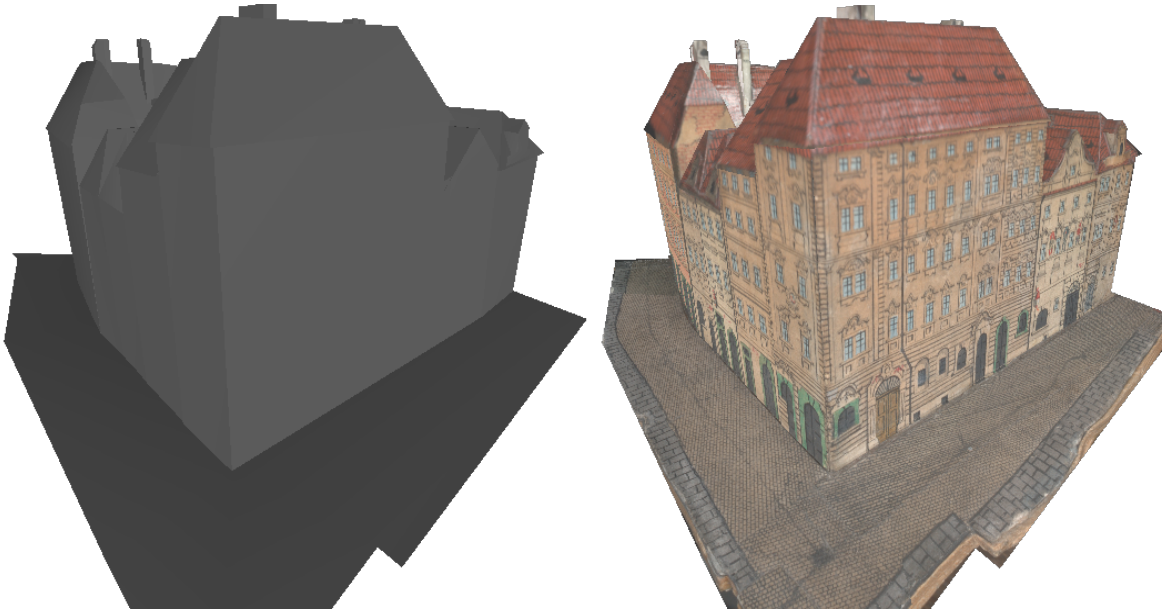
Kapitola 1

Úvod

V dnešní době umožňuje počítačová grafika renderovat realistické obrazy 3D světa. Nicméně aby toto bylo možné, musí existovat 3D modely objektů, které chceme zobrazovat. Tradiční způsob, jak získat tyto modely, je jejich ruční vytváření v modelovacích programech. To je velmi pracný a zdlouhavý proces a jeho výsledek není dostatečně realistický. Proto se v současnosti vyvíjí postupy, jak přímo digitalizovat reálné objekty. Existují dva základní přístupy, které se o toto pokoušejí. Nejpřesnější metoda je pravděpodobně laserové skenování objektů. Druhou, mnohem levnější a dostupnější variantou, je rekonstrukce objektů z fotografií. Těmito problémy se zabývá počítačové vidění a částečně také počítačová grafika, do které spadá zobrazování rekonstruovaných modelů. Cílem 3D počítačové rekonstrukce je tedy co nejvěrněji převést reálné objekty do digitální podoby. Aplikaci najdeme ve virtuální realitě, online prohlídkách, počítačových animacích nebo v herním či filmovém průmyslu.

Tato práce se zabývá jednou částí 3D rekonstrukce a to texturování objektů z fotografií. Na vstupu očekáváme již zrekonstruovaný 3D model (síť trojúhelníků) a větší množství (desítky až stovky) kalibrovaných fotografií z různých úhlů. Textura společně s materiálem slouží k popisu povrchu a je důležitá pro vnímání barvy a detailní struktury povrchu [ŽBSF04]. Aplikace textury vede z výraznému zvýšení vizuální kvality objektu za relativně malou cenu. Často je vizuálně i výkonově jednodušší použít detailní texturu a jednoduchý model s menším počtem vrcholů, viz obr. 1.1.

Syntéza textur z fotografií je poměrně složitá, protože na výsledný model se díváme i z jiných pohledů, než ze kterých byly pořízeny fotografie. Problémy, které při tomto procesu vznikají jsou popsány zejména v následující kapitole. Cílem této práce je vytvořit aplikaci, která v reálném čase mapuje fotografie na model. Protože aplikace funguje v reálném čase, může zohlednit aktuální pohled virtuální kamery a na základě pozice kamery vybrat nejvhodnější fotografie pro vytvoření textury. K tomu využívá velkého výkonu současných GPU.



Obrázek 1.1: Ukázka 3D modelu bez textur (vlevo) a s texturami (vpravo).

1.1 Struktura práce

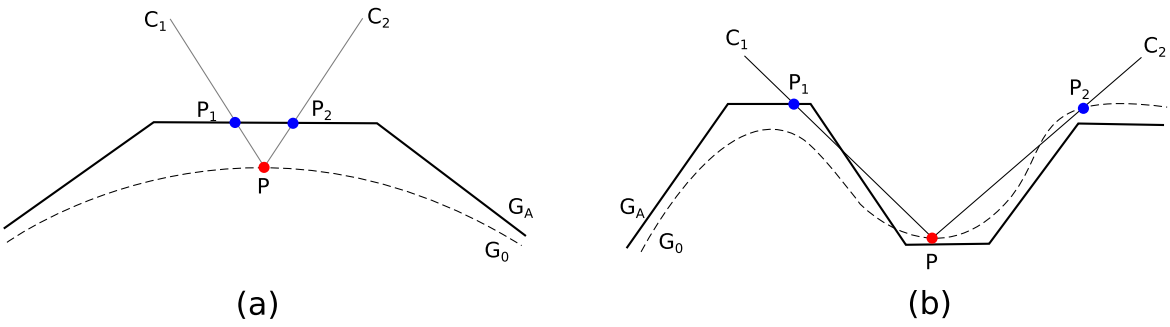
Práce v kapitole 2 rozebírá zadaný problém a popisuje související práce. Poté se věnuje popisu navrhované metody a řešení problémů. Kapitola 3 popisuje implementaci naší metody. Poté je v kapitole 4 provedeno testování aplikace na různých scénách. V závěru v kapitole 5 jsou diskutovány výsledky testování a jsou navrženy možné vylepšení metody a implementace.

Kapitola 2

Analýza problému a návrh řešení

2.1 Popis problému

Předpokládejme, že se díváme na statickou scénu s konstantním osvětlením, kde se pohybuje pouze pozorovatel. Za těchto podmínek můžeme popsat libovolnou fotografii pomocí pozice a orientace kamery. Pokud bychom vyfotili sférickou fotografii v každé možné pozici kamery, mohli bychom vyrenderovat kompletní scénu z libovolného pohledu. Kombinací všech 3D pozic kamery (x, y, z) a směrů (θ, ϕ) získáme plenoptickou funkci $\mathbf{L}(x, y, z, \theta, \phi)$ [AB91]. Snahou je aproximovat funkci $\mathbf{L}$ pomocí konečného množství diskrétních vzorků (x, y, z, θ, ϕ) a z této reprezentace efektivně renderovat nové pohledy (s pomocí 3D geometrie) [EDDM+08]. Povrch geometrie můžeme popsat jako funkci $\mathbf{G} : (x, y, z, \theta, \phi) \rightarrow (x_0, y_0, z_0)$, tedy jako mapování pohledových paprsků na 3D souřadnice povrchu. Jako $\mathbf{G}_0$ si označíme funkci skutečného povrchu, $\mathbf{G}_A$ reprezentuje aproximovaný povrch (rekonstruovaný 3D model), viz obr. 2.1.



Obrázek 2.1: (a) Chyby v geometrii způsobují chybnou projekci, bod P na původní geometrii je promítnut na body P_1 a P_2 na aproximované geometrii. (b) Chyby ve viditelnosti, bod P je chybně viditelný z kamery C_2 a naopak není viditelný z kamery C_1.

Důležitým prvkem je projekční matice kamery, která nám určuje projekční mapování 3D bodu (x, y, z) do 2D souřadnic (s, t) v i-té fotografii $\mathbf{P}_i : (x, y, z) \rightarrow (s, t)$, viz sekce 2.3.2. Z promítnuté pozice v obrázku pak můžeme přiřadit 3D bodu barvu $\mathbf{I}_i : (s, t) \rightarrow (r, g, b)$. Poté libovolný nový pohled $\mathbf{I}^V$ z virtuální kamery V můžeme vyjádřit pomocí rovnice 2.1.

$$\mathbf{I}^V(x, y, z, \theta, \phi) = \sum_i \mathbf{I}_i^V(x, y, z, \theta, \phi)\ \omega_i \tag{2.1}$$

kde

$$\mathbf{I}_i^V(x, y, z, \theta, \phi) = \mathbf{I}_i(\mathbf{P}_i(\mathbf{G}_A(x, y, z, \theta, \phi))) \tag{2.2}$$

$$\omega_i = \delta_i(\mathbf{G}_A(x, y, z, \theta, \phi))w_i(x, y, z, \theta, \phi) \tag{2.3}$$

a $\sum_i \omega_i = 1$. Notace $\mathbf{I}_i^V$ značí vyrenderovaný obrázek z pohledu V promítnutím vstupní fotografie $\mathbf{I}_i$ jako textury na povrch $\mathbf{G}_A$. δ_i určuje viditelnost a je 1, pokud je bod na povrchu $\mathbf{G}_A$ viditelný v kameře i a 0 pokud není. w_i je funkce, která určuje váhu kamery i pro každý paprsek. Rovnice 2.1 se snaží reprezentovat plenoptickou funkci jako lineární kombinaci promítnutých fotografií.

Zásadní problém spočívá v tom, že $\mathbf{G}_A \neq \mathbf{G}_0$, tedy již vstupní data vztahu 2.2 jsou zatížena chybou (obr. 2.1). Problémy přináší také nepřesná kalibrace kamery $\mathbf{P}_i$, což způsobuje další nepřesnosti při mapování.

2.2 Související práce

Existuje několik základních metod, jak mapovat fotografie z více pohledů na model, většina metod ale funguje na stejném principu:

1. Extrakce textur z fotografií pomocí vrhání paprsků či jiné podobné metody.
2. Vzájemná registrace textur
3. Sloučení textur do výsledné textury

Jednou z možností je projekce všech fotografií a jejich následné sloučení pomocí nějakého váženého průměru, jako např. v [BMR01]. Nevýhodou takové metody je zejména vznik artefaktů ve výsledné textuře, často se projevuje tzv. ghosting, kdy se v textuře objeví několik kopií jednoho obrazce. Další variantou je vytvoření atlasu textur [APK08], kdy každá část modelu dostane svojí texturu z unikátního pohledu. Tato varianta je využita např. v [AMK10]. Tento přístup má nevýhodu ve vzniku švů na okrajích jednotlivých částí textur, které je pak nutné odstraňovat [LI07]. Kromě artefaktů je u těchto přístupů častým problémem rozostření některých částí textur. Většina zmiňovaných problémů vzniká kvůli nepřesné kalibraci fotografií (špatnému odhadu radiální distorze či ohniskové vzdálenosti) nebo nepřesně rekonstruovaným modelům. Tyto nepřesnosti jsou obvyklé, i když jsou dnes již algoritmy pro 3D rekonstrukci velmi kvalitní. Získat velmi přesně kalibrované fotografie je časově náročný proces a někdy i téměř nerealizovatelný, např. ve venkovních scénách.

V článku [AMK10] jsou uvažovány nepřesně vytvořené modely, které je nutné otexturovat z původních fotografií, i když na model přesně nepasují. Navrhují podle modelu upravit původní fotografie a z nich následně vytvořit atlas textur. Úpravu provádí tak, že ve

fotografiích identifikují významné prvky, které naleznou na vytvořeném modelu a zpětně je promítají do původních pohledů. Poté deformují všechny fotografie, aby co nejlépe odpovídaly nepřesným modelům. Touto metodou se nezbaví všech artefaktů, ale omezí jejich výskyt.

Podobný přístup je použitý v [TM09], kde je popsán způsob, jak dynamicky deformovat několik textur najednou podle 3D modelu a poté je pomocí vážených faktorů sloučit dohromady na základě pozice virtuální kamery.

Odlišný přístup je použitý v článku [CCCS08]. Zde řeší texturování hustých modelů s jednotkami až desítkami milionů trojúhelníků. Pro takto husté modely nepoužívají textury, ale barvu přiřazují pouze vrcholům, což vzhledem k počtu vrcholů v modelu poskytuje dostatečné detaily. Základní princip algoritmu je takový, že model je vykreslen z pohledu jednotlivých kamer a poté jsou viditelné vrcholy promítnuty zpět do původních fotografií. Pokud je jeden vrchol vidět na více fotografiích, je použita funkce, která vybere nejvhodnější barvu pro daný vrchol. Tato funkce používá vážené masky, které jsou vygenerovány pro každou fotografii a udávají kvalitu jednotlivých pixelů. Pro určení kvality pixelu je použito několik různých metrik, pro každou fotografii je tedy vytvořeno více masek (každá maska má rozlišení stejné jako fotografie). Použité metriky jsou následující:

- Úhlová metrika - nejjednodušší metrika, která porovnává směr ke kameře s normálou plochy. Největší váha je, pokud jsou oba směry stejné.
- Hloubková metrika - váha pixelu je větší, pokud je povrch blíže ke kameře.
- Hraniční metrika - tato maska udává, jak daleko je pixel od okrajů fotografie a siluety v hloubkové mapě. Čím dále je pixel od okrajů, tím je jeho kvalita lepší.

Výsledná váha pixelu je získána vynásobením hodnot v jednotlivých maskách. Tím je zaručeno zachování lokálních minim v každé masce, což pomáhá odstraňovat pixely, které jsou v libovolné masce považovány za velmi špatné. Výsledná barva pro každý vrchol se získá porovnáním masek u všech fotografií, ze kterých je daný vrchol viditelný, a následným vybráním nejlepšího pixelu.

Výhoda tohoto způsobu je, že se nejedná o výpočetně složité operace, většina výpočtů je prováděna nad fotografiemi a není závislá na složitosti modelu. Další výhodou je možnost určit kvalitu jednotlivých fotografií podle maximální či průměrné kvality masky. Tím je možné některé nevhodné fotografie automaticky eliminovat a zrychlit celý proces. Nevýhodou této metody je nutnost hustých modelů, u modelů s nižším počtem vrcholů by výsledky této metody nebyly příliš kvalitní. Další problém nastává, pokud je rozlišení fotografií vyšší než rozlišení modelu (jeden vrchol se mapuje na více pixelů ve fotografiích), to vyžaduje další zpracování dat a zvyšuje složitost problému.

Všechny tyto algoritmy mají společné to, že z původních pohledů předem vytvoří texturu či atlas textur spojením všech fotografií, přičemž se snaží minimalizovat vznik artefaktů nebo případně vzniklé artefakty odstraňovat. Metoda navržená v této práci se zásadně liší tím, že žádnou takovou texturu nevytváří, ale pro každý aktuální pohled virtuální kamery vybírá množinu fotografií z nejvhodnějších pohledů a z této množiny vybírá nejvhodnější texely. Zásadní nevýhodou tohoto přístupu je velký výpočetní výkon, který je potřeba při zobrazování modelu. Tuto nevýhodu se snažíme minimalizovat efektivním využitím GPU.

Tato myšlenka není úplně nová, na podobném principu je založena např. metoda plovoucích textur [EDDM+08]. Tento algoritmus používá adaptivní nelineární metodu, která opravuje lokální nezarovnání textur vůči 3D modelu. K tomu určuje optický tok [1] mezi promítanými fotografiemi a příslušné textury kombinuje.

Navrhovaný postup využívá kombinaci lineární interpolace a odhadu optického toku. Nejprve je provedena projekce fotografií I_i na model z původních pohledů a scéna je vyrenderována z aktuálního pohledu virtuální kamery V. Tím vzniknou dočasné textury I_i^V. Poté je na jednotlivé páry textur I_i^V aplikován odhad optického toku, čímž vzniknou pole $W_{I_i^V \to I_j^V}$. Pro více než dvě vstupní fotografie je nutné provést lineární kombinaci vytvořených polí a sloučit je do výsledné textury I^v_{Float}. To je poměrně náročná operace, pro n vstupních fotografií je nutné vytvořit $O(n^2)$ polí. S tím se vyrovnávají v článku tím, že používají pro každý pohled jen 3 nejbližší fotografie. Plovoucí textury porušují epipolární geometrii, což umožňuje texturám kompenzovat nekvalitní kalibraci kamery a nepřesně zrekonstruované 3D modely.

Dále je nutné vypořádat se s vlastním zastíněním částí modelu, což je velmi běžná situace. K tomu je využita jemná mapa viditelnosti, která pro každý pixel určuje, zda je z dané kamery viditelný nebo ne. Oproti tradičním postupům, které obvykle využívají pouze hodnoty 1 a 0 (je nebo není vidět), je zde použita metoda, která nastavuje hodnotu z intervalu $(0, 1)$ pixelům, které jsou blízko hranic zastínění. Tím jsou odstraněny ostré hrany podél zastíněných částí, které jsou velmi často nepřesné a snižují výslednou vizuální kvalitu.

Na principu výběru nejlepší fotografie na základě aktuálního pohledu je založená metoda v článku [DTM96]. Při mapování jedné fotografie navrhují použití image-space stínových map pro řešení viditelnosti, protože to umožňuje efektivní implementaci pomocí z-bufferu. Při mapování více fotografií na model vybírají pro každý pixel vždy takovou fotografii, která se na daný povrch dívá pod nejlepším úhlem. To samozřejmě přináší viditelné švy, protože sousední pixely mohou pocházet z různých fotografií. Toto řeší zjemněním přechodů pomocí váženého průměru, váhu je určena podle rozdílu úhlu aktuálního pohledu a pohledu původní kamery. Dále pro lepší výsledky mají pixely na okraji fotografie menší váhu, čímž se snaží ještě více eliminovat vznik švů.

Dále navrhují jednoduchý algoritmus pro odstranění nežádoucích objektů, které se mohou vyskytovat na zdrojových fotografiích - např. auto či chodci před budovou, kterou chceme otexturovat. Uživatel může ručně vymaskovat tyto objekty předem zvolenou barvou a tyto pixely dostanou při texturování nulovou váhu a budou použita data z jiné fotografie. Pokud nejsou dostupná žádná data, vyplňují vzniklé mezery pomocí syntézy obrazu.

2.3 Návrh řešení

2.3.1 Princip navrhované metody

Jak již bylo zmíněno v předchozí sekci, tradiční řešení tohoto problému spočívá ve vytvoření jediné textury pomocí sloučení všech vstupních fotografií dohromady. Tento proces je často velmi náročný a zdlouhavý, na stejných testovacích datech, která jsou použita v této práci, trvá proces extrakce textur řádově hodiny [Kir08]. Oproti tomu naše navrhované

[1] v orig. optical flow

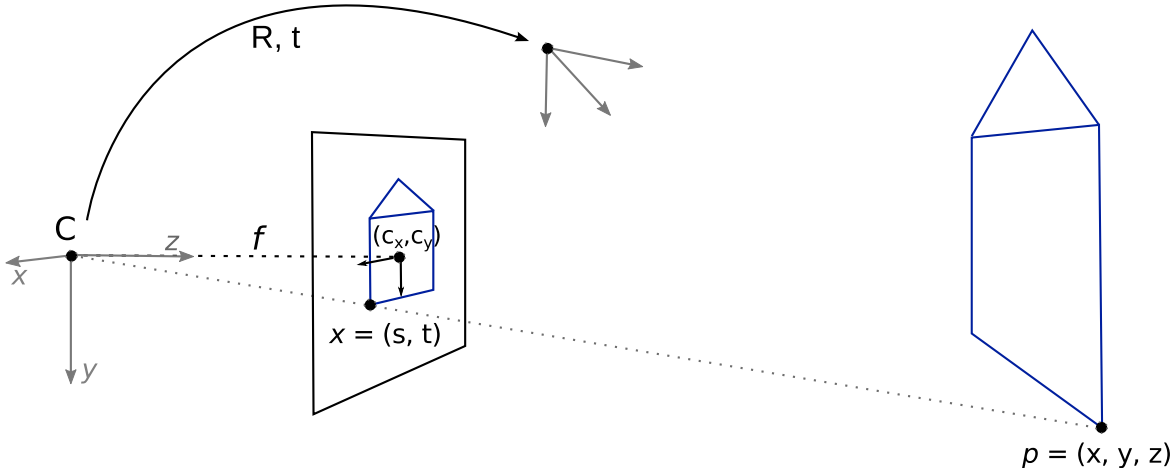
Obrázek 2.2: (a) Fotografie namapovaná na model vyrenderovaná z podobného pohledu jako původní fotografie. (b) Stejná fotografie vyrendrovaná z jiného pohledu.

metoda funguje v reálném čase, kdy provádí projekci vybraných fotografií na model. Načtení scény a předzpracování dat pro naší metodu trvá maximálně několik minut. Druhou výhodou našeho řešení je zahrnutí aktuálního pohledu virtuální kamery, které ostatní metody z principu uvažovat nemůžou. Tím se omezí vznik velkého počtu artefaktů v texturách a to jak vznik švů, tak vznik chyb způsobených vlastním zastíněním. Tyto problémy však kompletně nezmizí a je nutné s nimi počítat.

Na obrázku 2.2 je zobrazeno namapování fotografie na model bez výrazných geometrických detailů. Pokud je fotografie namapována z podobného úhlu jako je vyfocena původní fotografie, textura působí přirozeně. I když model neobsahuje geometrii okna ani dekorace nad oknem, namapovaná textura vyvolává dojem detailnější 3D geometrie. Oproti tomu stejná textura vyrenderovaná z jiného směru, který je výrazně odlišný od původní fotografie, působí nerealistickým dojmem.

Algoritmus se skládá ze tří základních kroků - projekce fotografií na model, výběr nejlepších fotografií pro pokrytí celého modelu a výběr nejvhodnějšího pixelu z více fotografií pro otexturování fragmentu[2].

[2]Fragmentem je v textu vždy myšlena plocha na 3D modelu, který odpovídá jednomu výslednému pixelu na obrazovce. Stejně, jako je termín fragment používán v OpenGL.



Obrázek 2.3: Obrázek znázorňuje situaci při projekci 3D bodu na 2D souřadnice.

2.3.2 Projekce fotografie na model

Mapování jedné fotografie na model můžeme popsat jako projekci $P : (x, y, z) \rightarrow (s, t)$, tedy jako projekci 3D bodů modelu do 2D souřadnice ve fotografii, viz obr. 2.3. Nejběžněji se v počítačové grafice a vidění používá perspektivní projekce, která promítá 3D body p na 2D body x vydělením jejich z souřadnicí [Sze10]. V homogeních souřadnicích má kanonická perspektivní matice $\mathbf{P}_0$ jednoduchou formu:

$$x = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\mathbf{P}_0} p \tag{2.4}$$

Po promítnutí 3D bodu projekční kamerou je nutné transformovat souřadnice na základě vlastností senzoru a orientace kamery vzhledem k počátku souřadnicového systému. *Kalibrační matice* $\mathbf{K}$ transformuje kanonickou perspektivní kameru na standardní projekční kameru $\mathbf{P}$:

$$\mathbf{P} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \mathbf{K}\mathbf{P}_0 \tag{2.5}$$

kde f je ohnisková vzdálenost v pixelech a bod (c_x, c_y) je optický střed vyjádřený v pixelech, obr. 2.3. Toto je zjednodušená matice kamery $\mathbf{K}$, která uvažuje senzor kolmý vůči optické ose a stejnou ohniskovou vzdálenost v osách x a y (což je v praxi nejběžnější varianta). Orientaci kamery vůči počátku souřadnicového systému definujeme pomocí 3×3 rotační kamery $\mathbf{R}$ a vektoru $\mathbf{t}$, čímž získáme výslednou 3×4 *matici kamery*:

$$\mathbf{P} = \mathbf{K} \, [\, \mathbf{R} \mid \mathbf{t} \,] \tag{2.6}$$

která provádí mapování bodu p_w v 3D světových souřadnicích do 2D souřadnic x ve fotografii:

$$x = \mathbf{P} \cdot p_w \tag{2.7}$$

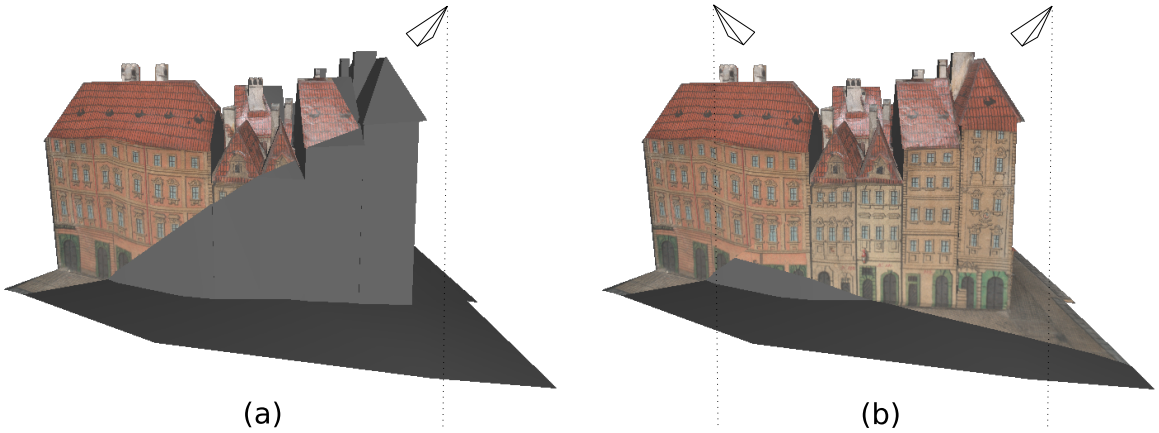
Existuje celá řada algoritmů pro nalezení matice kamery [HZ04], to ale není součástí této práce, kalibrované kamery jsou poskytnuté na vstupu společně s rekonstruovaným modelem a fotografiemi.

Při mapování fotografie na model je nutné brát v úvahu, že některé části modelu mohou být vzhledem ke kameře zastíněné a nemohou být z dané kamery správně otexturovány. Toto se nejčastěji řeší pomocí stínových map [SD02], které se předem vytvoří pro každou kameru. Aby byla tato metoda kvalitní, musí mít stínové mapy dostatečné rozlišení. Vzhledem k tomu, že aplikace bude využívat až stovky různých kamer, bylo by použití této metody paměťově velmi náročné.

Další možností je použít algoritmus vrhání paprsků, kdy se z kamery vrhají na scénu paprsky a podle zásahů s modelem se určí, které fragmenty jsou z kamery viditelné. Tato metoda také není příliš vhodná, protože aplikace bude v reálném čase využívat až desítky různých kamer zároveň a vrhání dostatečného množství paprsků z každé kamery by bylo výpočetně příliš náročné. Proto se jako nejlepší varianta se jeví využít jednodušší algoritmus, který spoléhá na seřazení kamer podle podobnosti se směrem virtuální kamery. Kvůli velkému množství vstupních kamer se dá očekávat, že pro většinu možných pohledů virtuální kamery bude existovat kamera s velmi podobným pohledem. Pro takovou kameru bude existovat jen velmi malé množství oblastí, které budou z kamery zastíněné, ale z virtuální kamery budou viditelné. Při texturování se kamery seřadí od “nejlepší” a každá kamera bude využita k otexturování oblasti, která ještě není pokrytá předchozí kamerou. Tím se výrazně omezí oblasti, které jsou otexturovány zastíněnou kamerou, ale zároveň jsou viditelné z virtuální kamery. K tomuto pravidlu je navíc zaveden práh, který určuje minimální úhel mezi směrem kamery a normálou plochy, pro který může být daná plocha kamerou otexturována.

2.3.3 Výběr vhodných fotografií pro otexturování modelu

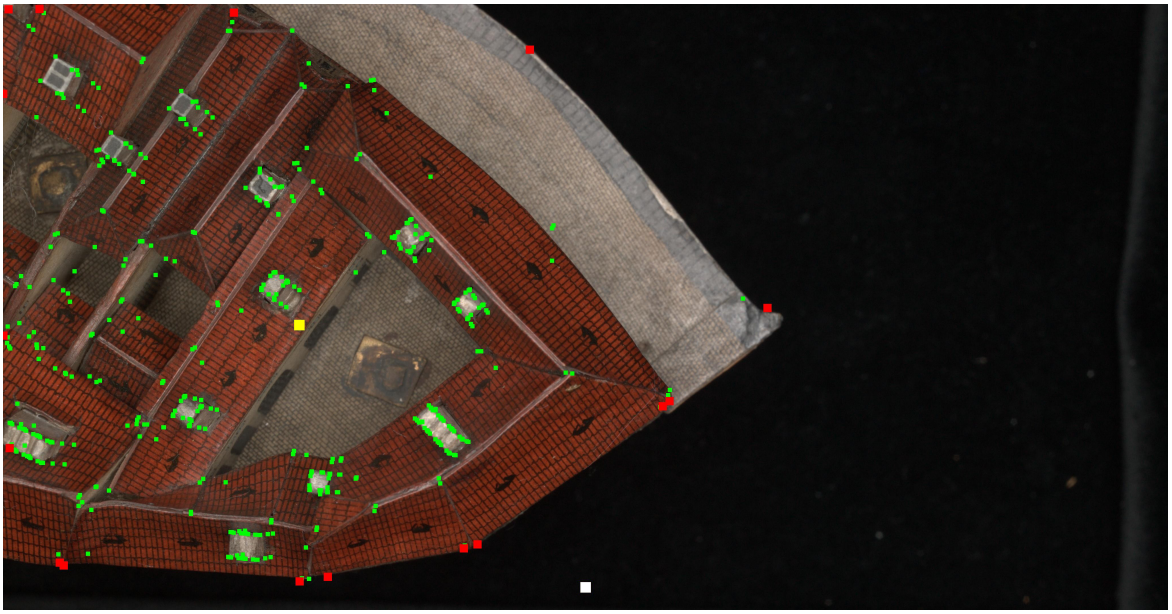
Téměř vždy nestačí k otexturování aktuálního pohledu pouze jedna fotografie. Proto je potřeba použít více fotografií pro vyrenderování modelu z nového pohledu. Na obrázku 2.4 je zobrazen model s jednou a se dvěma namapovanými fotografiemi. Při mapování více fotografií zároveň je nutné vyřešit dva základní problémy - které fotografie budou vybrány a jak bude fragment otexturován, pokud je jeho polohu možné promítnout do více kamer.



Obrázek 2.4: Projekce jedné (a) a dvou (b) fotografií na 3D model. Pozice kamer jsou pouze ilustrační.

Výběr fotografií je je prováděn na základě shody aktuálního pohledu virtuální kamery s pohledy vstupních kamer. Pro porovnání je nutné určit, kterým směrem jsou kamery natočeny. U vstupních kamer jde tento směr zjistit přímo z matice kamery, ale nemusí to být vždy výhodné. Pokud se kamera nedívá přímo na objekt (optická osa neprochází objektem), může být směr kamery poměrně zavádějící. Proto navrhujeme algoritmus, který tento směr koriguje na základě viditelné části modelu na fotografii. Tato metoda funguje tak, že se vrcholy rekonstruovaného modelu promítnou do fotografie, body ležící mimo fotografii se zahodí. Poté je nalezena konvexní obálka [And79] promítnutých bodů, která přibližně ohraničuje oblast modelu zobrazenou na fotografii, viz obr. 2.5. Z bodů ležících na konvexní obálce se spočítá centroid a nalezne se směrový vektor ze středu kamery procházející centroidem. Nalezený vektor se použije jako výsledný směr pohledu kamery. Algoritmus není příliš náročný, promítnutí n vrcholů do obrázku je provedeno se složitostí $\Theta(n)$, konvexní obálku ve 2D lze nalézt se složitostí $0(n\ log(n))$. Během tohoto algoritmu je zároveň vypočtena plocha konvexní obálky (plocha konvexního polygonu s vrcholy v bodech konvexní obálky). Tato plocha je později při texturování fragmentů zahrnuta do výpočtu váhy fotografie, viz sekce 2.3.4.

Podobná situace nastává při určování směru virtuální kamery. Pokud se díváme zvenku na model a otáčíme virtuální kamerou, vidíme model stále ze stejného směru a chceme, aby byl otexturován stejně. Proto není vhodné použít přímo směr, kterým se dívá virtuální kamera, ale spíše spojnici mezi pozicí kamery a modelem. Pro určení pozice modelu je použit centroid všech vrcholů. Odlišná situace nastává, pokud se virtuální kamera nachází uvnitř modelu. V tomto případě naopak musíme brát v úvahu přímo směr pohledu virtuální kamery,



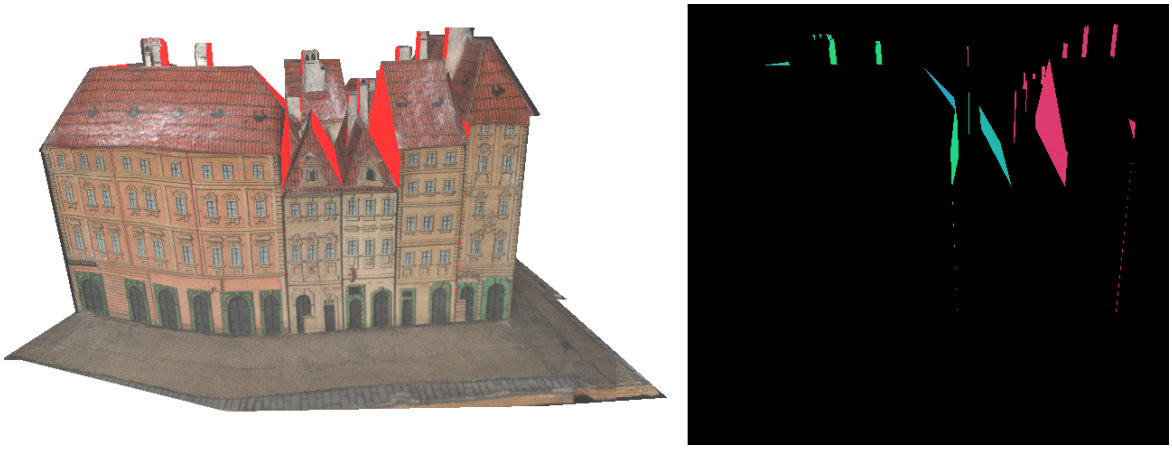
Obrázek 2.5: Výřez horní poloviny fotografie použité k texturování. Bílý bod označuje průnik optické osy s obrázkem, žlutý bod označuje vypočtený centroid. Ostatní body jsou vrcholy 3D modelu promítnuté do obrázku, červeně zvýrazněné vrcholy leží na konvexní obálce.

protože při otočení kamery vidíme jinou část modelu. K jednoduchému určení, zda je kamera uvnitř nebo vně modelu, je možné použít osově zarovnaný ohraničující kvádr (AABB).

Po vypočtení směru pohledu vybereme nejlepší fotografie porovnáním se směry všech vstupních kamer. Počet fotografií potřebných k otexturování je závislý na složitosti scény a i na samotných fotografiích, proto se tato hodnota nenastavuje automaticky, ale uživatel jí musí zadat na vstupu.

Pro otexturování modelu často nestačí vybrat pouze fotografie, které nejlépe odpovídají aktuálnímu pohledu. Takové fotografie dobře pokryjí plochy, které jsou na aktuální pohled přibližně kolmé. Oproti tomu velmi špatně pokryjí plochy, které jsou téměř paralelní se směrem aktuálního pohledu, ale zároveň jsou ještě dobře viditelné, obr. 2.6 (a). K určení takových ploch je nutné zavést práh, který definuje maximální úhel mezi normálou plochy a směrem kamery, kdy je ještě možné plochu kvalitně otexturovat. Během testování se ukázalo vhodné použít práh přibližně 70°.

K efektivnímu nalezení těchto ploch navrhujeme dvou průchodový algoritmus. Nejprve se vybere daný počet fotografií, které nejlépe odpovídají aktuálnímu pohledu. Poté se v prvním průchodu model vykreslí a zjistí se, které plochy je s daným prahem možné z vybraných fotografií otexturovat. Pokud fragment není možné pokrýt žádnou fotografií, vykreslí se do textury aktuální normála. V prvním průchodu tedy vznikne textura, která obsahuje normály ploch, které je nutné pokrýt, obr. 2.6 (b). Poté je provedeno klastrování těchto normál a jsou určeny nejvýznamnější směry. Tento přístup je vhodný, protože klastrování jde efektivně provést na GPU bez nutnosti přenosu velkého množství dat mezi GPU a CPU. Poté se vyberou nové fotografie, které nejlépe odpovídají nalezeným směrům (kamery, jejichž směr



Obrázek 2.6: (a) Model otexturovaný šesti fotografiemi z kamer, které nejlépe odpovídají pohledu virtuální kamery. Červeně jsou zvýrazněny plochy, které nejde z kamer kvalitně otexturovat. (b) Vyrenderovaná textura s uloženými normálami v prvním průchodu algoritmu ze stejného pohleduj jaklo (a).

pohledu je opačný k nalezeným normálám). V druhém průchodu se poté provede výsledný rendering.

2.3.4 Texturování z více fotografií

Během renderingu je nutné vyřešit, jak budou otexturovány fragmenty, pro které je možné použít data z více fotografií. Princip je velmi podobný, jako při výběru fotografií, ale snahou při texturování je také omezení vzniku švů a omezení chybného otexturování zastíněných oblastí. Během texturování se již používají pouze fotografie, které byly vybrány v předchozím kroku.

Pro každou fotografii je vypočtena váha, fotografie s nejvyšší váhou je použita pro otexturování. Jeden fragment je otexturován pouze z jedné fotografie, není použito žádné vážené míchání barev z více fotografií. To vede k tomu, že dva sousední pixely mohou být vybrány z různých fotografií a tím mohou vznikat viditelné švy. Vážené míchání barev, jako např. v [DTM96] vede k rozmazání textur a na testovacích datech k vizuálně horšímu výsledku.

Váha w_i i-té fotografie je složena z několika faktorů. V úvahu je brána shoda směru kamery d_c se směrem virtuální kamery d_v, stejně jako při výběru fotografií. Dále se uvažuje úhel mezi směrem kamery a normálou plochy N_f. Tím se preferují kamery s co nejvíce kolmým pohledem na danou plochu a zároveň je pravděpodobnější, že celá plocha bude otexturována z jedné fotografie. Nakonec je do výpočtu zahrnuta velikost plochy s_i, kterou na fotografii zabírá model, viz výpočet konvexní obálky v sekci 2.3.3. Celkový výpočet váhy shrnuje následující rovnice:

$$w_i = dot(d_v, d_c) \times \left(\frac{1 + dot(-N_f, d_c)}{1 - cos(\alpha)} + 1 \right) \times s_i \times \delta(i) \tag{2.8}$$

$$\delta(i) = \begin{cases} 1 & \text{pokud } \exists P_i : (x, y, z) \rightarrow (s, t) \\ 0 & \text{v opačném případě} \end{cases} \tag{2.9}$$

kde *dot* značí skalární součin dvou vektorů a α je limitní úhel, který určuje, zda je plochu možné z dané kamery otexturovat.

Musíme brát v úvahu, že máme dvě rozdílné množiny fotografií. Nejprve jsme vybrali fotografie, které nejlépe odpovídají aktuálnímu pohledu. Poté jsme nalezli doplňující fotografie, které pokryjí zbývající plochy. Proto musí rendering probíhat ve dvou částech. Nejprve se aplikuje původní množina fotografií. Až poté se použijí doplňující fotografie, které mohou otexturovat pouze fragmenty, které se nepodařilo otexturovat v první části. Tímto se jednak preferují fotografie blízké aktuálnímu pohledu a zároveň se omezí situace, kdy by doplňující fotografie mohly chybně otexturovat plochy, které jsou z jejich pohledu zastíněné. K tomu i tak může dojít, ale pouze ve velmi omezených částech výsledného renderu, vizuálně to tedy nebude příliš výrazné.

Kapitola 3

Implementace

3.1 Požadavky na implementaci

Aplikace by měla provádět texturování v reálném čase, proto je základním požadavkem rychlost zobrazování. Pro programy zobrazující 3D grafiku se jako dostatečná frekvence zobrazování snímků uvažuje 30fps. Dalším důležitým požadavkem je kvalita výsledných textur, která by měla být co nejlepší. To znamená zejména minimální výskyt švů mezi texturami a omezení chybného otexturování zastíněných oblastí. V optimálním případě by se render modelu z místa shodného s pozicí některé vstupní kamery neměl lišit od pořízené fotografie. Mezi další požadavky patří přenositelnost programu na více platforem a jednoduché ovládání pomocí grafického rozhraní.

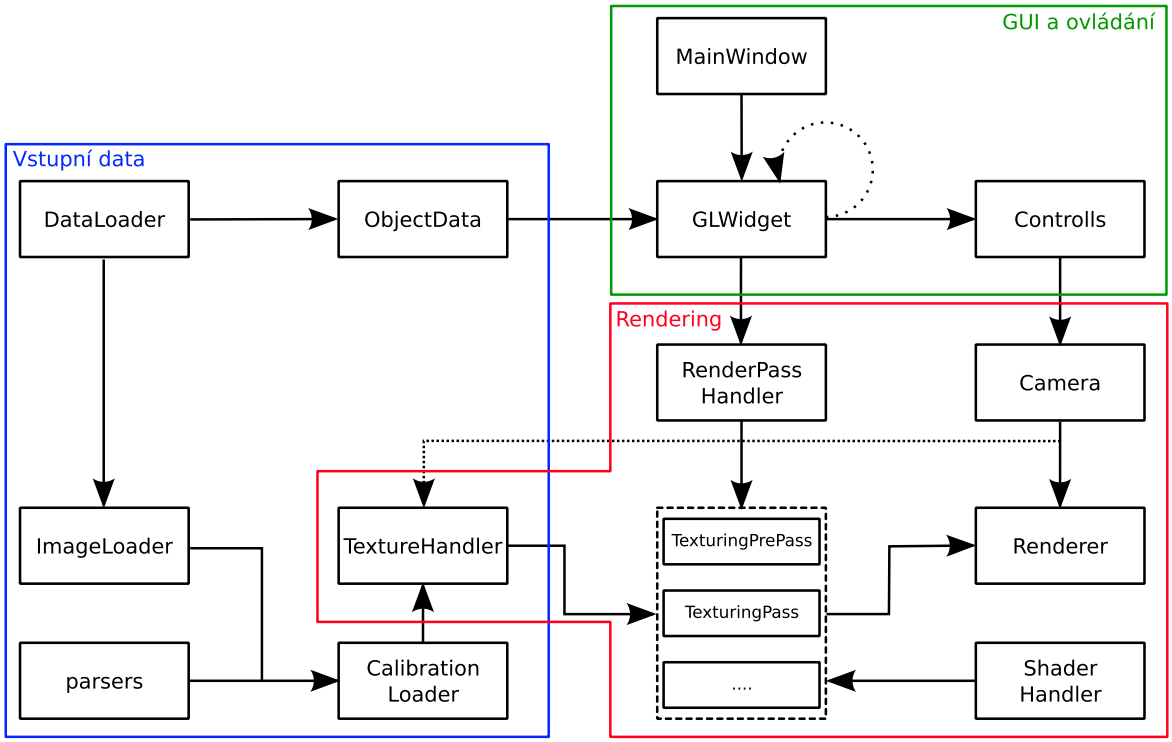
3.2 Použité technologie a knihovny

Aplikace je implementována v ANSI C++11 a je možné ji přeložit v linuxových operačních systémech i v MS Windows. Základ aplikace je postaven na multiplatformní knihovně Qt 4.8. Qt je jedna z nejrozšířenějších knihoven pro tvorbu aplikací s grafickým rozhraním (GUI), ale poskytuje i další podpůrné moduly, které s grafickým rozhraním přímo nesouvisí. Kromě tvorby GUI je Qt využito pro načítání vstupů z periférií a pro správu vláken. Pro efektivní zobrazování 3D grafiky je využito OpenGL, které poskytuje API pro rendering a výpočty na grafickém procesoru (GPU). Společně s OpenGL je použit jazyk GLSL, který slouží k tvorbě tzv. shaderů. To jsou samostatné programy, které se spouští na grafické kartě a řídí jednotlivé části programovatelného zobrazovacího řetězce. Pro běh aplikace je nutná grafická karta podporující OpenGL 4.0, což je jakákoliv novější grafická karta. Část programu využívá OpenGL 4.3, ale pro základní funkčnost aplikace není tato verze vyžadována.

Společně s OpenGL je využita knihovna GLEW 1.9, která slouží k načítání OpenGL rozšíření. S OpenGL je také úzce spjatá knihovna GLM, která poskytuje matematické operace běžně používané v počítačové grafice. Knihovna poskytuje základní datové struktury jako jsou vektory a matice a různé algebraické operace s těmito strukturami. Další použité knihovny jsou libjpeg 8 pro efektivní načítání obrázku ve formátu JPG a Assimp 3 pro načítání 3D modelů ve velkém množství formátů. Všechny využité knihovny jsou dostupné pod otevřenou licencí.

3.3 Architektura aplikace

Architektura aplikace zobrazující 3D grafiku v reálném čase se liší od běžných desktopových aplikací. Základem je efektivně využít zobrazovací řetězec a zajistit distribuci dat mezi aplikací a grafickou kartou tak, aby byly omezeny přenosy dat mezi CPU a GPU, které jsou velmi drahé. Zároveň je důležité, aby všechny zobrazovací průchody měly přístup k aktuálním datům potřebným pro rendering. Samotné OpenGL je možné popsat jako data-driven architekturu, nicméně aplikace využívá tradičního objektového návrhu. Základním prvkem OpenGL aplikace je hlavní smyčka, která neustále provádí aktualizaci scény bez ohledu na uživatelské vstupy. Ty se obvykle ukládají do fronty a jsou zpracovány vždy na začátku každé iterace smyčky. Zjednodušený návrh architektury aplikace je zobrazený na obrázku 3.1.



Obrázek 3.1: Diagram ilustrující komunikaci základních komponent v aplikaci.

Základem aplikace je třída `MainWindow`, která se stará o všechny komponenty GUI. Obsahuje aplikační menu a stará se o otevírání dialogových oken s nastavením programu. Jako centrální widget obsahuje třídu `GLWidget`, která má na starost zobrazování dat vykreslených pomocí OpenGL. Třída obsahuje metodu `paintGL`, která slouží jako hlavní smyčka, ve které se překresluje scéna. Zároveň se také třída stará o vytvoření nové scény a inicializuje načtení vstupních dat. Design a používání GUI je detailněji popsáno v příloze B.

3.4 Načítání a zpracování vstupních dat

Aplikace na vstupu očekává 3D model a kalibrované fotografie. O načítání 3D modelu se stará třída `DataLoader` za pomoci knihovny Assimp. O načtení kalibrovaných fotografií se stará třída `CalibrationLoader`. Ta na základě zvoleného formátu vybírá parser pro načtení konfiguračních souborů s daty a stará se o přiřazení správných fotografií k načteným kamerám. Aplikace v současnosti podporuje dva formáty kalibrovaných dat:

- Bundler - výstupní soubor *.out z rekonstrukčního programu Bundler.
- REALVIZ Ascii Camera - soubor *.rz3 s kalibračními daty a textový soubor, který k datům z rz3 souboru přiřazuje názvy fotografií.

Po načtení kalibrovaných fotografií dochází ke korekci pohledových směrů kamer. Vrcholy modelu jsou projekční maticí promítnuty do souřadnic ve fotografii. Z bodů, které leží uvnitř fotografie se vytvoří konvexní obálka pomocí Grahamova algoritmu [And79]. Tento algoritmus má optimální asymptotickou složitost $O(n\ log(n))$. Základem algoritmu je setřídění bodů podle souřadnice x, po setřídění je potřeba již pouze jeden průchod bodů se složitostí $O(n)$. Algoritmus začne s dvěma body s nejnižší x-ovou souřadnicí. Poté postupně prochází seřazené body a zjišťuje se, zda je vybraný bod nalevo nebo napravo od přímky, která prochází předchozí dvojicí vrcholů. Pokud je nalevo, druhý bod z dvojice nemůže ležet na konvexní obálce a je nahrazen nově nalezeným bodem. Tento proces pokračuje, dokud existuje bod nalevo od posledních dvou bodů na dočasné konvexní obálce. Algoritmus je možné snadno implementovat pomocí zásobníku. Body na dočasně nalezené konvexní obálce se vkládají na vrchol zásobníku. Pokud je nalezen bod vlevo od posledních dvou na zásobníku, odebírají se body ze zásobníku, dokud toto pravidlo opět neplatí. Poté se do zásobníku vloží nově nalezený bod a algoritmus pokračuje.

Po nalezení konvexní obálky je vypočítán centroid bodů na konvexní obálce. Pohledový vektor kamery je poté určen jako vektor procházející středem kamery a nalezeným centroidem. Zároveň je vypočtena plocha konvexní obálky pro pozdější použití při výpočtu váhy textury. Plochu lze snadno spočítat jako jednu polovinu determinantu matic, kde n bodů na konvexní obálce je zapsáno do řádků matice a první bod se opakuje na konci, viz vztah 3.1. Body musí být seřazené proti směru hodinových ručiček, což jde snadno zařídit již během Grahamova algoritmu. Po načtení kalibrací kamer následuje načtení samotných fotografií.

$$s_i = \frac{1}{2}\begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_n & y_n \\ x_1 & y_1 \end{vmatrix} = \frac{1}{2}[(x_1y_2 + x_2y_3 + \ldots + x_ny_1) - (y_1x_2 + y_2x_3 + \ldots + y_nx_1)] \tag{3.1}$$

Vstupní fotografie jsou dodány v nejběžnějším formátu JPEG. Tento formát je však nevhodný pro textury, protože je komprimovaný, jeho načítání je netriviální a příliš pomalé. To by ani nebyl takový problém, pokud bychom fotografie načítali pouze jednou při spuštění aplikace. Problém je v tom, že fotografie mohou mít velké rozlišení a může jich být

velké množství. Během testování byly použity scény s několika stovkami fotografií v rozlišení 4094×4096px. Takové množství fotografií v nekomprimovaném formátu zabere až desítky GB paměti. Nemůžeme proto očekávat, že je možné nahrát všechny fotografie do RAM a bude je nutné za běhu načítat z pevného disku. Z toho důvodu jsou při prvním načtení scény všechny fotografie načteny z původních JPG souborů a na disk se uloží každá fotografie také v nekomprimovaném formátu RAW. Při každém dalším spuštění se používají již nekomprimovaná data. Toto má nevýhodou v tom, že je nutné mít na disku dostatečné množství místa pro nekomprimovaná data. Nicméně pro běh aplikace v reálném čase je toto nezbytné.

Bohužel ani vytvoření RAW souborů není dostatečné pro načítání fotografií v reálném čase. Fotografie o v rozlišení 16MPx má v nekomprimovaném formátu velikost přibližně 45MB. Moderní HDD jsou schopné číst data rychlostí přibližně 150MB/s [toma], to znamená, že načtení fotografie z HDD bude trvat minimálně 0.3 sec. I když budeme uvažovat moderní SSD disky s rychlostí čtení přes 500MB/s [tomb], bude trvat načtení fotografie do RAM téměř 90ms. To je samozřejmě naprosto nepoužitelné pro zobrazování v reálném čase, kdy při 30 fps je nutné vyrenderovat celý snímek za 33ms.

Jako řešení se nabízí možnost předem načítat omezené množství fotografií, které se vejde do RAM, aby fotografie byla vždy již načtená ve chvíli, kdy je jí třeba použít. Toto jsme implementovali za pomoci kd-stromu, kdy se načítaly fotografie nejbližší k aktuální pozici. Toto řešení se ukázalo velmi neefektivní, protože nebylo možné načítat dostatečné množství fotografií pro pokrytí všech možných směrů pohybu kamery. Ve výsledku byl procesor vytížený neustálým načítáním fotografií a aplikace byla nepoužitelná. Proto jsme zvolili jiný přístup. Pro každou fotografii jsou vytvořeny náhledy o velikosti šířce 512px. Tyto náhledy je již možné nahrát do RAM pro všechny vstupní fotografie. Při texturování se poté využívají náhledy a originální fotografie je načtena pouze v případě, kdy je opravdu použita. To vede ke snížení kvality textur při změně pohledu virtuální kamery, ale pokud se pohled virtuální kamery příliš nemění, textury v plném rozlišení se načtou poměrně rychle (konkrétní hodnoty jsou uvedeny v kapitole testování v sekci TODO).

O načítání fotografií se stará třída `TextureHandler`. Tato třída má primárně za úkol vybírat správné fotografie pro texturování. Při výběru jsou nejprve seřazeny fotografie podle shody s aktuálním pohledem a poté jsou vybrány fotografie s největší shodou. Shoda úhlů je počítána jednoduše pomocí skalárního součinu vektorů, který pro normalizované vektory vrací cosinus úhlu mezi vektory. Pokud je zapnutý průchod pro doplňování neotexturovaných ploch, polovina ze zadaného počtu fotografií pro texturování se vybere podle aktuálního směru a druhá polovina se rozdělí mezi neotexturované plochy. Pokud je potřeba dotexturovat plochy z více směrů, počet fotografií se rozdělí váženě podle velikostí daných ploch. Poté, co je fotografie určena k použití, se inicializuje načítání originální fotografie v plné velikosti. Toto je realizováno vícevláknově za použití třídy `QThreadPool` z Qt, která si interně udržuje frontu fotografií čekajících na načtení. Při rychlém pohybu virtuální kamery se může stát, že fotografie již není aktuální dříve, než se uvolní volné vlákno a začne se načítat. Taková situace je ošetřena booleovskou hodnotou u každé fotografie, která indikuje zda se aktuálně fotografie používá. Pokud by nebylo toto ošetřeno, mohlo by dojít k načítání fotografií, které již nejsou vůbec potřeba, na úkor aktuálních. Zároveň je také nutné hlídat, které fotografie se přestaly používat. Tyto fotografie je nutné z paměti odstranit, jinak by postupně došlo k zaplnění paměti.

Třída se zároveň spravuje texturovací objekty, které slouží k připojení textur do OpenGL. Protože v OpenGL je omezené množství texturovacích jednotek, implementace má omezené maximální množství fotografií, které mohou být současně použity pro texturování. OpenGL garantuje minimálně 16 texturovacích jednotek[1], ale běžně GPU podporují 32 nebo více texturovacích jednotek, což je pro potřeby aplikace dostatečné. Problém ale nastává v tom, že do paměti na GPU není možné nahrát všechny náhledy. Pokaždé, když je fotografie vybrána, je nutné nahrát do GPU náhled a později i plnou velikost. Toto se samozřejmě neděje v každém framu, `TextureHandler` si udržuje přehled o tom, které fotografie jsou v GPU nahrány. Přenosy na GPU jsou prováděny pouze pokud byla vybrána nějaká nová fotografie. V takovém případě některá z textur na GPU přestala být aktuální. `TextureHandler` se postará o to, aby se negenerovala nová textura, ale použije se existující texturovací objekt, pouze se přepíšou obrazová data.

[1] Ve skutečnosti OpenGL garantuje 16 texturovacíh jednotek pro každou fázi zobrazovacího řetězce, ale protože texturování provádíme pouze na fragmentovém shaderu, zajímá nás pouze množství texturovacích jednotek dostupné v této fázi.

3.5 Rendering

V aplikaci je implementováno několik renderovacích průchodů. O správu texturovacích průchodů se stará třída `RenderPassHandler`. Třída umožňuje přidávat a odebírat průchody a deleguje vykreslení objektu jednotlivým průchodům ve správném pořadí. Jednotlivé průchody dědí z třídy `RenderPass`, která obsahuje společné prvky průchodů. O renderování dat se stará třída `Renderer`, která zapouzdřuje přenos dat na GPU a poskytuje jednotlivým průchodům jednotné rozhraní. O správu shaderů se stará třída `ShaderHandler`. Třída se stará o načítání, kompilaci a linkování shaderů a ukládá si jejich identifikátory, které poté poskytuje renderovacím průchodům.

3.5.1 Texturovací průchod

Základním průchodem je `TextuingRenderPass`, který se stará o texturování modelu. Tento průchod zavolá aktualizaci třídy `TextureHandler`, od které obdrží aktuální fotografie společně s jejich kalibracemi. Protože vstupní fotografie mohou mít libovolné rozměry, jsou použity textury typu `GL_TEXTURE_RECTANGLE`, které nemusí být čtvercové a jejich rozměry nemusí být zarovnané na mocniny 2. Jejich další výhodou je, že na shaderech se do těchto textur nepřistupuje pomocí normalizovaných souřadnic, ale přímo pomocí souřadnic v pixelech. To je výhodné, protože po promítnutí pozice fragmentu do obrázku pomocí matice kamery získáme souřadnice také v pixelech a nemusíme je normalizovat. Nevýhodou těchto textur je absence mipmap.

Kromě textur je do shaderu nutné připojit informace o kameře. To je realizováno pomocí uniformního bufferu. Tyto buffer slouží k rychlému nahrávání vetšího množství uniformních dat do shaderů. Uniformní buffer je umístěn v lokální paměti, proto je čtení dat z bufferu rychlé, což je na shaderu velmi důležité. Navíc je optimalizovaný pro sekvenční přístup, který je při iteraci kamer na shaderu vhodný. Buffer používá `std140` layout, který definuje přesné zarovnání datových typů, to umožňuje snadnou správu paměti v bufferu.

3.5.2 Průchod pro pokrytí scény

Pokrytí celé scény pomocí vyhledávání neotexturovaných ploch je řešené jako samostatný průchod `TexturingPrePass`, který je vždy spuštěn před texturovacím průchodem. Texturovací průchod ale není závislý na prvním průchodu. To má velkou výhodu v tom, že první průchod může být volitelný, a je možné ho za běhu aplikace zapínat a vypínat. K tomu jsou dva hlavní důvody - první průchod je výpočetně poměrně náročný a výrazně zpomaluje běh aplikace, což může být problém na méně výkoném hardwaru. Dále tento průchod vyžaduje OpenGL 4.3 s rozšířením `GL_NV_shader_atomic_float`, které je v současnosti dostupné pouze na kartách od výrobce NVIDIA. Aplikaci je tedy možné používat i bez tohoto průchodu, ale v některých scénách budou výsledky vizuálně horší.

Jak již bylo popsáno v sekci 2.3.3, texturovací průchod nejprve zjistí, které plochy není možné otexturovat. To zjistí výpočtem váhy, která probíhá stejně jako v texturovacím průchodu. Pokud je váha 0, fragment nelze otexturovat. V tom případě se do textury vykreslí aktuální normála. Vykreslování do textury je v OpenGL standardní operace a jde realizovat jednoduše. Poté je potřeba texturu přečíst a zjistit, které směry jsou v textuře uloženy. To

```
struct TextureData {
  mat4  u_Rt;           //matice R|t
  vec3  u_cameraViewDir;//opraveny smer kamery
  ivec2 u_textureSize;  //rozmery fotografie
  float u_textureFL;    //ohniskova vzdalenost
  float u_coveredArea;  //normalizovana plocha objektu ve fotce
};
layout(std140) uniform u_textureDataBlock {
  TextureData ub_texData[MAX_TEXTURES];
};

void projectCoords(in int index, in vec4 pos, out vec2 coords) {
  TextureData data = ub_texData[index];
  vec3 c = (data.u_Rt * pos).xyz;
  coords =  c.xy/c.z * data.u_textureFL + data.u_textureSize.xy * 0.5f;
}

bool inRange(in int index, in vec2 coords) {
  ivec2 s = ub_texData[index].u_textureSize;
  return coords.x >= 0 && coords.x < s.x && coords.y >= 0 && coords.y < s.y;
}

float computeWeight(in int index, in vec3 N, out vec2 coords, in float dl =
    dirLimit) {
  TextureData data = ub_texData[index];
  float weight = data.u_coveredArea;
  weight *= dot(u_viewDir, data.u_cameraViewDir);

  projectCoords(index, In.v_position, coords);
  weight *= float(inRange(index, coords));

  float dirDiff = dot(N, data.u_cameraViewDir);
  weight *= -(dirDiff + 1.f) / (1.f - dl) + 1;
  return weight;
}
```

Zdrojový kód 3.1: Výpočet váhy a projekce souřadnice na fragmentovém shaderu.

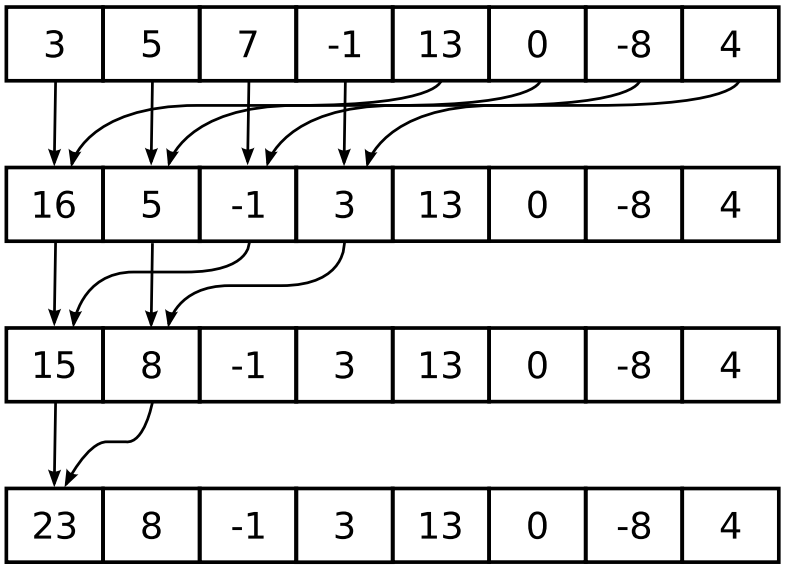
ale není úplně jednoduché, protože textura je uložena v paměti na GPU. Kopírovat texturu v každém framu do RAM je velmi náročné a real-time aplikaci nepoužitelné. Proto jsem se rozhodl provést klastrování normál na GPU pomocí výpočetních shaderů.

Výpočetní shadery jsou poměrně nová technologie, dostupná od OpenGL 4.3. Tyto shadery narozdíl od ostatních nepatří do zobrazovacího řetězce a nepodílí se přímo na renderingu. Naopak tyto shadery umožňují provádět na GPU obecné výpočty, které s grafikou vůbec nemusí souviset. Architektura GPU je masivně paralelní, proto je na GPU vhodné provádět pouze výpočty, které jdou ve velkém měřítku paralelizovat. Výpočetní shadery jsou podobné technologiím CUDA nebo OpenCL, které také slouží k provádění obecných výpočtů na GPU. Výhoda výpočetních shaderů spočívá v tom, že je velmi snadné je zapojit do aplikace, které již OpenGL používá, protože mapování bufferů či textur probíhá úplně stejně, jako u běžných shaderů. Zároveň výpočetní shadery také používají jazyk GLSL, který poskytuje řadu užitečných funkcí.

Pro klastrování normál je použit algoritmus k-means [Llo82]. Tento algoritmus na začátku rozdělí data náhodně do k skupin. Poté spočítá centroid každého klastru a porovná prvky se všemi centroidy. Pokud je prvek blíže k některému centroidu z jiného klastru, je do tohoto klastru přesunut. Takto algoritmus iterativně pokračuje, dokud existuje prvek, který se v dané iteraci přesunuje do jiného clusteru. Případně je možné algoritmus ukončit po daném počtu iterací. Algoritmus je primárně určen pro klastrování bodů v prostoru, ale je možné ho použít i pro klastrování vektorů, kdy vzdálenost směrů určuje úhel mezi vektory. Pro algoritmus je předem nutné zadat počet klastrů, ale některé klastry mohou zůstat na konci algoritmu prázdné. Během testování se ukázalo, že dostatečný počet klastrů pro normály je 4. V běžných scénách není pravděpodobné, že by nebyly otexturovány plochy z více jak čtyř význačných směrů.

Algoritmus je vhodný pro paralelní implementaci na GPU. Pro každý pixel vstupní normálové textury se spustí jedno vlákno, které porovná úhel se všemi klastry a určí, zda se normála přesune do jiného klastru. Poté je nutné v rámci všech klastrů spočítat jejich centroid, tedy sečíst všechny normály v klastru a výsledek vydělit počtem normál. Tento krok jde také paralelizovat pomocí jednoho ze základních paralelních algoritmů - redukce.

Paralelní redukce je algoritmus, který umožňuje paralelně sečíst sekvenci čísel. Funguje tak, že v každé iteraci jedno vlákno sečte dvě unikátní a čísla a uloží částečný výsledek. Tím vznikne z n prvků původní množiny $\frac{n}{2}$ částečných výsledků. Takto se rekurzivně pokračuje, dokud algoritmus nenajde výsledek. Běh algoritmu je znázorněna na obrázku 3.2.



Obrázek 3.2: Obrázek znázorňuje paralelní redukci.

Při spuštění výpočetního shaderu musíme explicitně zadat, kolik vláken chceme spustit. Vlákna se spouští v tzv. *pracovních skupinách*, kdy vlákna v jedné pracovní skupině mohou vzájemně komunikovat pomocí sdílené paměti. Pracovní skupiny jsou organizovány ve 3D mřížce, která usnadňuje indexování dat v mřížce. V rámci skupiny se nastavuje *lokální velikost skupiny*, která určuje počet vláken spouštěných ve skupině. Tato lokální velikost je opět tří dimenzionální. Pokud je tedy lokální velikost skupiny $(128, 1, 1)$, kterou spustíme v

pracovní skupině velikosti $(4, 8, 16)$, spustíme 32768 vláken (instancí výpočetního shaderu). Každou instanci lze unikátně identifikovat pomocí systému vestavěných proměnných, které určují index vlákna v rámci lokální velikosti a index pracovní skupiny, ve které je vlákno spuštěno.

Pro zpracování normálové mapy je použita lokální velikost skupiny $(16, 16, 1)$ a počet pracovních skupin se dopočítává podle aktuálního rozlišení obrazovky. Textura je tedy zpracovávána po blocích 16×16 pixelů. Toho můžeme využít při inicializaci klastrů. Namísto toho, abychom v první iteraci normály náhodně rozdělili mezi klastry, můžeme vždy normály v jednom bloku vložit do jednoho zvoleného klastru. Protože se dá očekávat, že normály v jednom bloku budou velmi pravděpodobně patřit do jedné plochy, můžeme tímto klastrování urychlit. Po počátečním rozdělení je nutné spočítat centroidy klastrů, to provedeme pomocí výše popsané paralelní redukce.

Paralelní redukce je provedena v rámci každé pracovní skupiny ve sdílené paměti. Sdílená paměť je velmi rychlá, proto je pro tento typ operace vhodná. Protože máme 4 clustery, nejde provést redukce jednoduše v rámci jednoho pole, protože v částečných součtech během redukce by nešlo určit, která data patří do jakého klastru. Proto je nutné vytvořit pro každý klastr jedno pole s počtem prvků stejným jako je lokální velikost skupiny. V ukázce 3.2 jsou deklarace polí, které se během redukce využívají. Deklarace označené klíčovým slovem `buffer` jsou v hlavní paměti GPU a jsou přístupné ze všech vláken. Pole označená jako `shared` jsou ve sdílené paměti, do které mají přístup pouze vlákna v rámci jedné pracovní skupiny.

```
1  struct Cluster {
2    vec3 cntr;
3    uint size;
4  };
5  coherent restrict layout(std430, binding = 0) buffer destBuffer {
6    Cluster clusters[CLUSTERS];
7    bool moving;
8  };
9  coherent restrict layout(std430, binding = 1) buffer idxBuffer {
10   uint indices[];
11 };
12
13 shared vec3 cache[CLUSTERS][SIZE];
14 shared uint sizeCache[CLUSTERS][SIZE];
```

Zdrojový kód 3.2: Datové struktury potřebné pro sečtení normál pomocí paralelní redukce.

V ukázce 3.3 je zobrazen kompletní algoritmus paralelní redukce. Na řádcích `1-7` je provedena inicializace dat ve sdílené paměti. Všechna data jsou inicializována na 0, poté jsou inicializována data jednotlivých klastrů. Proměná `centroidIDX` je index klastru, do kterého přísluší aktuální vlákno. Tato proměnná je v prvním průchodu inicializovaná na základě pozice pracovní skupiny, v dalších iteracích je hodnota načtena z pole `indices`. Následuje cyklus, který provádí samotnou redukci odděleně pro každý klastr. Po každém cyklu je nutné provést paměťovou synchronizaci vláken pomocí funkce `memoryBarrierShared`, která zajistí viditelnost zapsaných výsledků pro všechny vlákna. Zároveň je nutné provést synchronizaci pomocí `barrier`, která zajistí, že žádné vlákno v rámci skupiny nebude po-

kračovat, dokud všechny vlákna nedosáhly bariéry. Podmínku na řádce 21 vždy splní pouze jedno vlákno v pracovní skupině. Toto vlákno přidá pomocí atomické operace výsledek z redukce v dané skupině do bufferu v hlavní paměti. Tím získáme finální výsledek redukce. Atomické operace jsou velmi náročné a je vhodné se jim vyhýbat. V tomto případě je ale atomická operace nad proměnnou provedena pouze jednou pro každou pracovní skupinu, takže nezpůsobí výraznou ztrátu výkonu.

```
1  uint localID = gl_LocalInvocationIndex; //unikatni index v ramci skupiny
2  for(int i = 0; i < CLUSTERS; ++i) {
3    sizeCache[i][localID] = 0;
4    cache[i][localID] = vec3(0, 0, 0);
5  }
6  sizeCache[centroidIDX][localID] = int(isNormal);
7  cache[centroidIDX][localID] = int(isNormal) * N;
8
9  int stepv = (SIZE >> 1);
10 while(stepv > 0) { //redukce v ramci skupiny
11   if (localID < stepv) {
12     for(int i = 0; i < CLUSTERS; ++i) {
13       sizeCache[i][localID] += sizeCache[i][localID + stepv];
14       cache[i][localID] += cache[i][localID + stepv];
15     }
16   }
17   memoryBarrierShared(); //synchronizace vlaken
18   barrier();
19   stepv = (stepv >> 1);
20 }
21 if (localID == 0) { //secteni vysledku ze vsech skupin
22   for(int i = 0; i < CLUSTERS; ++i) {
23     if(sizeCache[i][0] != 0) atomicAdd(clusters[i].size, sizeCache[i][0]);
24     if(cache[i][0].x != 0) atomicAdd(clusters[i].cntr.x, cache[i][0].x);
25     if(cache[i][0].y != 0) atomicAdd(clusters[i].cntr.y, cache[i][0].y);
26     if(cache[i][0].z != 0) atomicAdd(clusters[i].cntr.z, cache[i][0].z);
27   }
28 }
```

Zdrojový kód 3.3: Paralelní redukce na výpočetních shaderech.

Protože na výpočetních shaderech není možné synchronizovat všechna vlákna napříč pracovními skupinami, není možné po provedení redukce vydělit získaný centroid počtem vláken a pokračovat v klastrování. Jediný způsob, jak zajistit globální synchronizaci, je spustit nový shader. Proto jsou data z paralelní redukce zkopírována na CPU, kde proběhne výpočet a normalizace centroidu a poté je spuštěn nový shader, na kterém je provedeno klastrování. Přenos klastrů z GPU na CPU není problém, protože každý klastr obsahuje pouze jeden vektor a počet normál v klastru. Po výpočtu centroidu je spuštěn další shader, který provádí samotné klastrování. Shader má připojené stejné buffery, jako ten, který provádí redukci. Tento shader je již poměrně jednoduchý, pouze pro iteruje přes všechny klastry, porovnává úhel mezi aktuální normálou a centroidem klastru a nastavuje index nejvhodnějšího klastru, viz ukázka 3.4. Poté se opět provede redukce a pokračuje se další iterací.

Klastrování je omezeno na 5 iterací. To se ukázalo jako dostatečné, protože klastry jsou v datech obvykle výrazné a algoritmus konverguje ke správnému rozdělení velmi rychle.

```
float myDot = -1;

if(isNormal) {
  for(int k = 0; k < CLUSTERS; ++k) {
    memoryBarrier();
    float d = dot(N, clusters[k].cntr);
    if(d > myDot) {
      myDot = d;
      indices[id] = k;
      moving = true;
    }
  }
}
```

Zdrojový kód 3.4: K-means klastrování na výpočetních shaderech.

3.5.3 Další průchody

V aplikaci jsou implementovány ještě některé další jednoduché průchody. Pro vizualizaci vstupních kamer a lepší orientaci ve scéně slouží `RadarRenderPass`, který v rohu okna zobrazuje 2D pohled na scénu shora, kde jsou znázorněny pozice a směry všech vstupních kamer a také pozice virtuální kamery. Dále jsou zde zvýrazněné aktuálně využívané kamery pro texturování. `CameraPointsRenderPass` do scény umožňuje vykreslit body na pozice vstupních kamer, to je vhodné zejména pro kontrolu správného načtení vstupních dat a testování.

Kapitola 4

Testování

Jsou provedeny 2 základní druhy testování. Jednak jde o testování výkonu aplikace a dále potom testování vizuální kvality výsledných rendererů. Testování probíhalo na několika odlišných scénách.

Veškeré testování probíhalo v následující konfiguraci:

- CPU: Intel Core i5 3210M, 2,5 GHz, 3MB L2 cache
- GPU: NVIDIA GeForce GT630M, 2GB paměti, proprietární ovladač verze 331.49
- RAM: 8GB DDR3, 1.6GHz
- OS: OpenSuse 13.1 64bit, linuxové jádro verze 3.11.10
- Překladač: gcc 4.8.1
- Rozlišení použité pro rendering: 1366×678px

4.1 Popis testovacích scén

4.1.1 Model 1

Jedná se o rekonstruovaný blok z Langweilova modelu Prahy [TODO]. Data jsou zapůjčena od Muzea hlavního města Prahy. Data obsahují:

- 3D model: 970 trojúhelníků a 2910 vrcholů
- Fotografie: 321 fotografií v rozlišení 4094×4096

4.1.2 Model 2

Jde o rekonstrukci sochy.

- 3D model: TODO
- Fotografie: TODO

4.2 Testování výkonu

4.2.1 Testování doby renderingu

Testování probíhalo pomocí měření doby renderingu scény, kdy se kamera pohybovala kolem středu objektu po kružnici a vždy se dívala na střed objektu. Měření času je prováděno na GPU pomocí OpenGL časovačů. Ty používají asynchroní dotazy, které přesně změří dobu renderingu, protože čas může být zaznamenán až po dokončení všech předchozích vykonávaných OpenGL funkcí. Zároveň za pomoci techniky dvou bufferů se vždy odečítá čas o jeden snímek později. To umožňuje odečítat čas bez nutnosti čekat na dokončení renderingu, takže měření času se nijak neprojeví na výsledném výkonu.

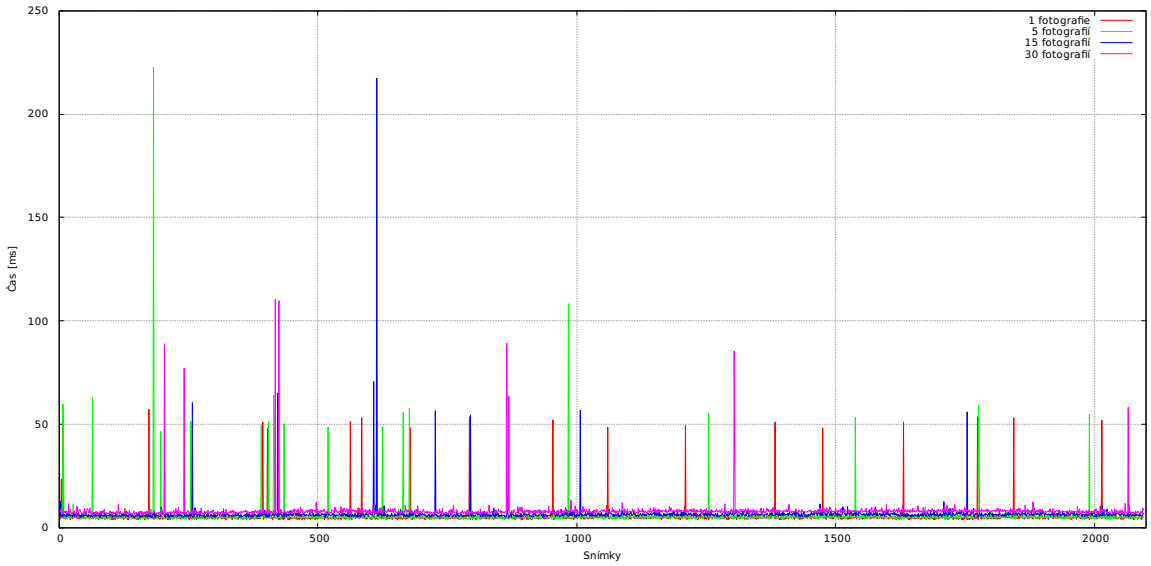
```
1  class GLTimer {
2    GLuint queryID[2];
3    uint queryBack, queryFront;
4
5    void swapQueries() {
6      if(queryBack) {
7        queryBack = 0, queryFront = 1;
8      }
9      else {
10       queryBack = 1, queryFront = 0;
11     }
12   }
13 public:
14   GLTimer() {
15     queryBack = 0, queryFront = 1;
16     glGenQueries(2, &queryID[0]);
17     //odstraneni chyb pri prvnim mereni
18     glBeginQuery(GL_TIME_ELAPSED, queryID[queryFront]);
19   }
20
21   void start() {
22     glBeginQuery(GL_TIME_ELAPSED, queryID[queryBack]);
23   }
24
25   float end() {
26     glEndQuery(GL_TIME_ELAPSED);
27     GLuint64 time;
28     glGetQueryObjectui64v(queryID[queryFront], GL_QUERY_RESULT, &time);
29     swapQueries();
30     return time / 1000000.0f; //vraci uplynuly cas v ms
31   }
32 }
```

Zdrojový kód 4.1: Asynchroní měření času na GPU.

V ukázce 4.1 je zobrazena implementace měření času. V metodě `start` se spustí časovač na zadním bufferu a v metodě `end` se tento časovač ukončí. V tuto chvíli není možné časovač odečíst, protože nemusely být dokončeny všechny asynchroní příkazy na GPU. Proto je odečten časovač na předním bufferu, který byl spuštěn a ukončen v minulém framu a nyní již musí být dostupný. Poté se buffery přehodí. Časovač měří čas s udávanou přesností $10^{-9}s$.

Model 1

První testování je provedeno pro jeden oběh virtuální kamery kolem modelu, to odpovídá 2094 měřeným snímků. Při tomto měření nebyl zapnutý průchod pro kompletní otexturování scény.



Obrázek 4.1: Obrázek znázorňuje paralelní redukci.

Na grafu 4.1 jsou znázorněna měření při použití různého množství fotografií (1, 5, 15, 30). Na grafu jsou očividné výrazné výchylky v některých snímcích. Tyto výchylky jsou způsobené nahráváním plné velikosti textury z CPU na GPU. Ve chvíli, kdy je fotografie načtena z HDD, proběhne její nahrání z RAM do GPU a to způsobí výrazné zpomalení v daném snímku v závislosti na velikosti nahrávané textury. Pokud se sejde více fotografií v jednom snímku, je toto prodlení znásobené. Tento problém se mi bohužel nepodařilo odstranit. Při běžném prohlížení modelu je toto znatelné, ale není to tak výrazný problém. Při rychlejším pohybu kamery se snímky z HDD nestíhají načítat a obvykle se načtou až při zastavení kamery. V tu chvíli uživatel krátkodobý pokles fps nezaznamená.

	1	**5**	**15**	**30**
Minimum [ms]	3,71	3,89	4,77	6,16
Maximum [ms]	57,28	222,44	217,37	110,41
Průměr [ms]	5,36	5,81	6,62	8,16
Průměr [fps]	186,48	172,17	151,00	122,48

Tabulka 4.1: capt

4.3 Testování kvality renderů

Kapitola 5

Závěr

Literatura

[AB91] Edward H. Adelson and James R. Bergen. The plenoptic function and the elements of early vision. In *Computational Models of Visual Processing*, pages 3–20. MIT Press, 1991.

[AMK10] Ehsan Aganj, Pascal Monasse, and Renaud Keriven. Multi-view texturing of imprecise mesh. In *Computer Vision – ACCV 2009*, volume 5995 of *Lecture Notes in Computer Science*, pages 468–476. Springer Berlin Heidelberg, 2010.

[And79] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inf. Process. Lett.*, 9(5):216–219, 1979.

[APK08] C. Allene, J.-P. Pons, and R. Keriven. Seamless image-based texture atlases using multi-band blending. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1–4, 2008.

[BMR01] Fausto Bernardini, Ioana M. Martin, and Holly Rushmeier. High-quality texture reconstruction from multiple scans. *IEEE Transactions on Visualization and Computer Graphics*, 7:318–332, 2001.

[CCCS08] M. Callieri, P. Cignoni, M. Corsini, and R. Scopigno. Masked photo blending: Mapping dense photographic data set on high-resolution sampled 3d models. *Comput. Graph.*, 32(4):464–473, August 2008.

[DTM96] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. SIGGRAPH '96, pages 11–20, New York, NY, USA, 1996. ACM.

[EDDM+08] Martin Eisemann, Bert De Decker, Marcus Magnor, Philippe Bekaert, Edilson de Aguiar, Naveed Ahmed, Christian Theobalt, and Anita Sellent. Floating textures. *Computer Graphics Forum (Proc. of Eurographics)*, 27(2):409–418, April 2008.

[HZ04] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[Kir08] Jan Kirschner. 3d model reconstruction from photographs. Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Computer Graphics and Interaction, 2008.

[LI07] V. Lempitsky and D. Ivanov. Seamless mosaicing of image-based texture maps. In *Computer Vision and Pattern Recognition*, pages 1–6, 2007.

[Llo82] S. Lloyd. Least squares quantization in pcm. *IEEE Trans. Inf. Theor.*, 28(2):129–137, March 1982.

[SD02] Marc Stamminger and George Drettakis. Perspective shadow maps. *ACM Trans. Graph.*, 21(3):557–562, July 2002.

[Sze10] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[TM09] Takeshi TAKAI and Takashi MATSUYAMA. Harmonized texture mapping. *The Journal of The Institute of Image Information and Television Engineers*, 63(4):488–499, apr 2009.

[toma] tomshardware.com. Charts, benchmarks HDD Charts 2013, [01] Read Throughput Average. ¡http://www.tomshardware.com/charts/hdd-charts-2013/-01-Read-Throughput-Average-h2benchw-3.16,2901.html¿. Datum: 2014-06-06.

[tomb] tomshardware.com. Charts, benchmarks SSD Charts 2013, AS-SSD Sequential Read. ¡http://www.tomshardware.com/charts/ssd-charts-2013/AS-SSD-Sequential-Read,2782.html¿. Datum: 2014-06-06.

[ŽBSF04] J. Žára, B. Beneš, J. Sochor, and P. Felkel. *Moderní počítačová grafika*. Computer Press, 2004.

Příloha A

Seznam použitých zkratek

API

CPU Central processing unit, procesor

GPU Graphic processing unit, grafický procesor

HDD Hard disk drive, pevný disk

SSD Solid-state drive

GUI Graphics user interface, grafické uživatelské rozhraní

JPEG Joint photographic experts group

fps frames per second, obrázků za vteřinu

Příloha B

Instalační a uživatelská příručka

B.1 Překlad aplikace

B.2 Používání aplikace

Příloha C

Obsah přiloženého CD