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

14. května 2014

Poděkování

Chtěl bych poděkovat vedoucímu této práce, panu Ing. Davidovi Sedláčkovi, Ph.D., za poskytnutí cenných rad a nápadů. Dále bych chtěl poděkovat Muzeu hlavního města Prahy za poskytnutí dat z Langweilova modelu Prahy.

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

This thesis deals with problem of texturation from photography, i.e. mapping of input photos to reconstructed 3D geometry. We analyze existing solutions for texture mapping and propose our algorithm. We present a real-time view-dependent solution that considers virtual camera position and orientation. By this approach, we try to minimize artifacts and create photo-realistic textures.

Abstrakt

Tato práce se zabývá problematikou texturování rekonstruovaných 3D modelů z fotografií. V práci jsou analyzována současná řešení mapování fotografií a na jejich základě je navržen a implementován algoritmus. Oproti tradičním metodám navržené řešení mapuje fotografie v reálném čase na základě pozice a orientace virtuální kamery. Tím se snaží omezit vznik artefaktů a zároveň vytvořit realisticky působící textury.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

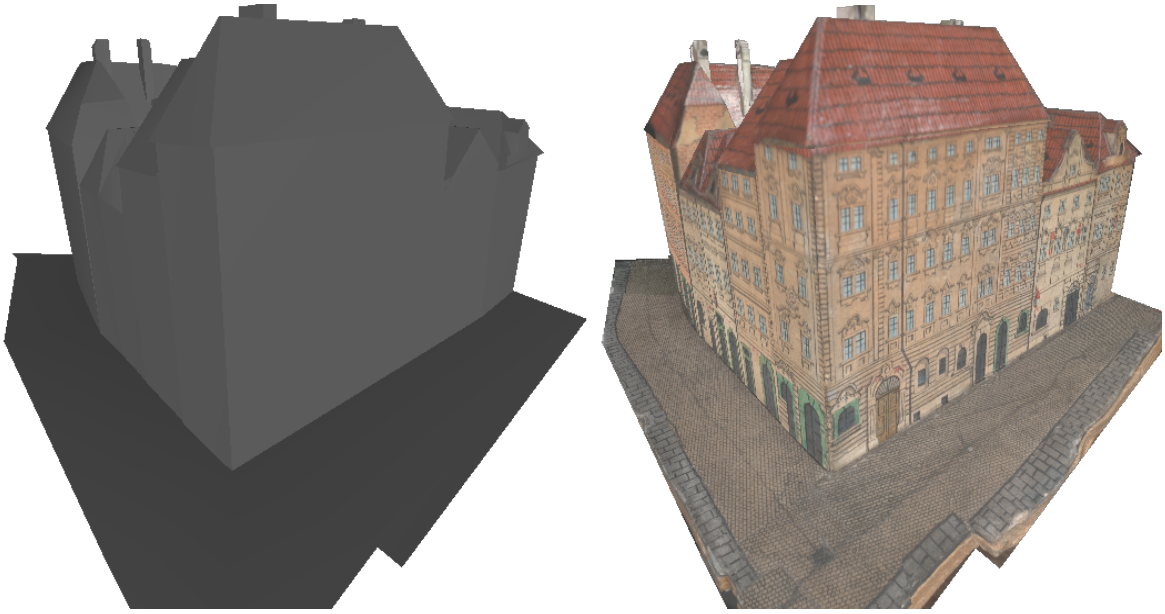
Kapitola 1

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 texturováním 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 [**?**], viz obr. **??**. 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.

Extrakce 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 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

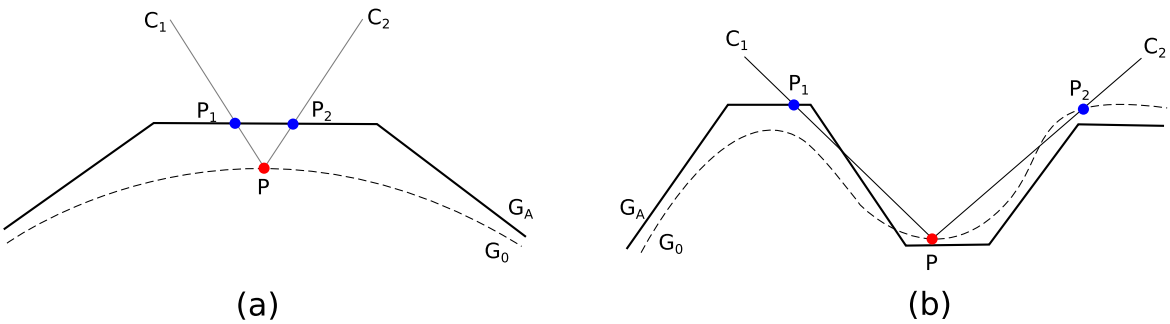
Práce v kapitole **??** rozebírá zadaný problém a popisuje související práce. Poté se věnuje popisu navrhované metody a řešení problémů. Kapitola **??** popisuje implementaci naší metody. Poté je v kapitole **??** provedeno testování aplikace na různých scénách. V závěru v kapitole **??** jsou diskutovány výsledky testování a jsou navržena možná 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 $\mathbf{L}(x, y, z, \theta, \phi)$ [**?**]. 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) [**?**]. 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. **??**.



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 **??**. 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 **??**.

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

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 **??** 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 **??** jsou zatížena chybou (obr. **??**). 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 [**?**]. 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 [**?**], kdy každá část modelu dostane svojí texturu z unikátního pohledu. Tato varianta je využita např. v [**?**]. Tento přístup má nevýhodu ve vzniku švů na okrajích jednotlivých částí textur, které je pak nutné odstraňovat [**?**]. 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 [**?**] 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 [**?**], 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 [**?**]. 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 [**?**]. 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_{Float}^v. 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 [**?**]. 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, váha 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 [**?**]. Oproti tomu naše navrhovaná

[1] v orig. optical flow

(a) (b)

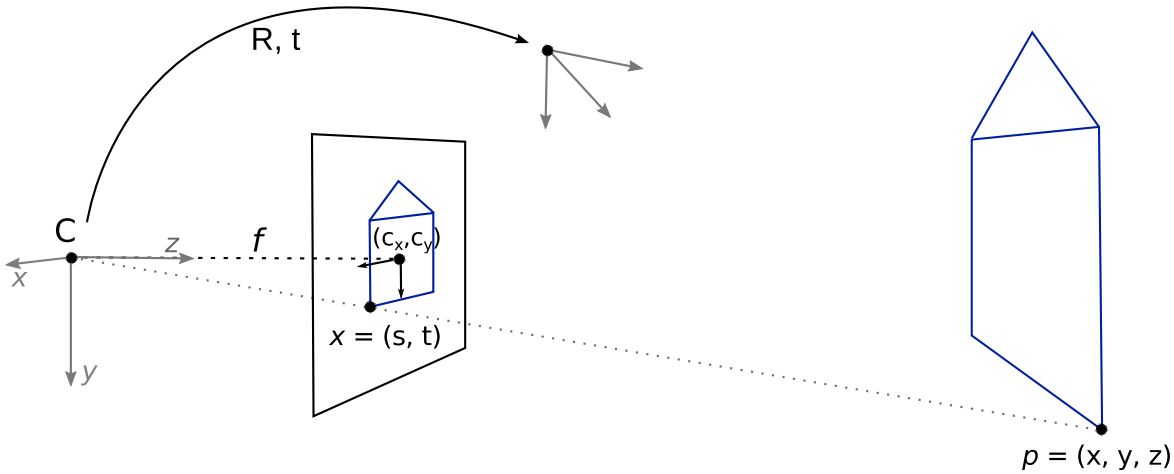
Obrázek 2.2: (a) Fotografie namapovaná na model vyrenderovaná z podobného pohledu jako původní fotografie. (b) Stejná fotografie vyrendrovaná z jiného pohledu.

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 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 **??** 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, která 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 souřadnic ve fotografii, viz obr. **??**. Nejběžněji se v počítačové grafice a vidění používá perspektivní projekce, která promítá 3D body p na 2D body x vydělením jejich z souřadnicí [**?**]. 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. **??**. 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:

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

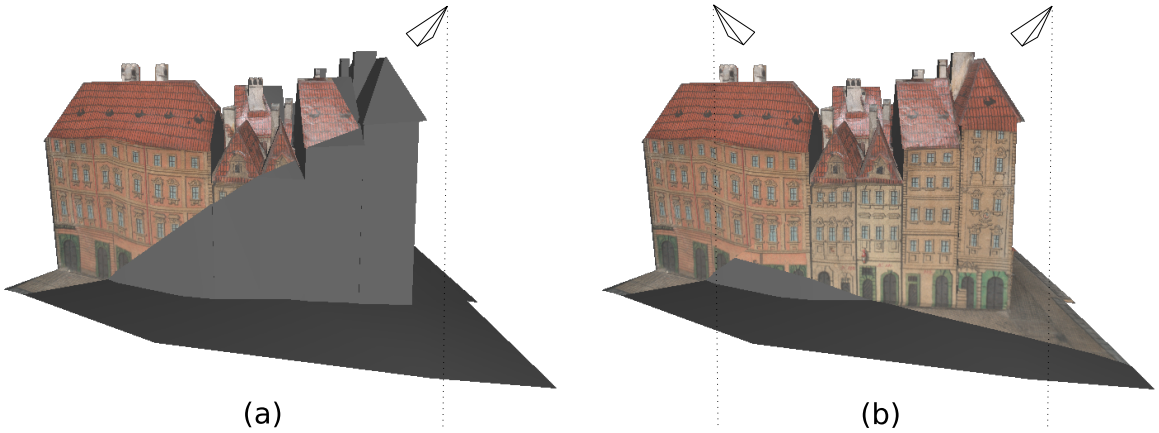
Existuje celá řada algoritmů pro nalezení matice kamery [**?**], to ale není součástí této práce, kalibrované kamery jsou poskytnuté na vstupu společně s rekonstruovaným modelem a fotografiemi.

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 [**?**], 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, je využití této metody nereálné. Předem vytvořit stínové mapy pro všechny kamery by bylo extrémně paměťově náročné. Výpočet stínových map pro desítky kamer v reálném čase je v současné době také nereálný.

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 maximální úhel mezi směrem kamery a normálou plochy, pro který může být daná plocha kamerou otexturována.

2.3.3 Výběr vhodných fotografií pro otexturování modelu

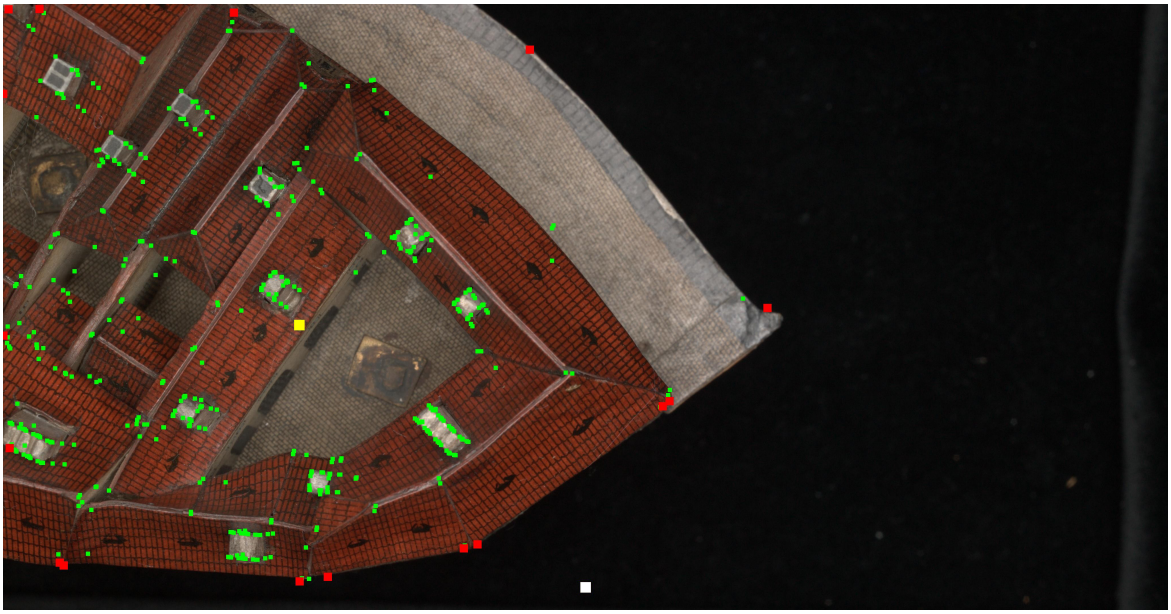
Téměř vždy nestačí k otexturování aktuálního pohledu pouze jedna fotografie. Proto je potřeba použít více fotografií pro vyrenderování modelu z nového pohledu. Na obrázku **??** je zobrazen model s jednou a se dvěma namapovanými fotografiemi. Při mapování více fotografií zároveň je nutné vyřešit dva 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 [**?**] promítnutých bodů, která přibližně ohraničuje oblast modelu zobrazenou na fotografii, viz obr. **??**. 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 **??**.

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,



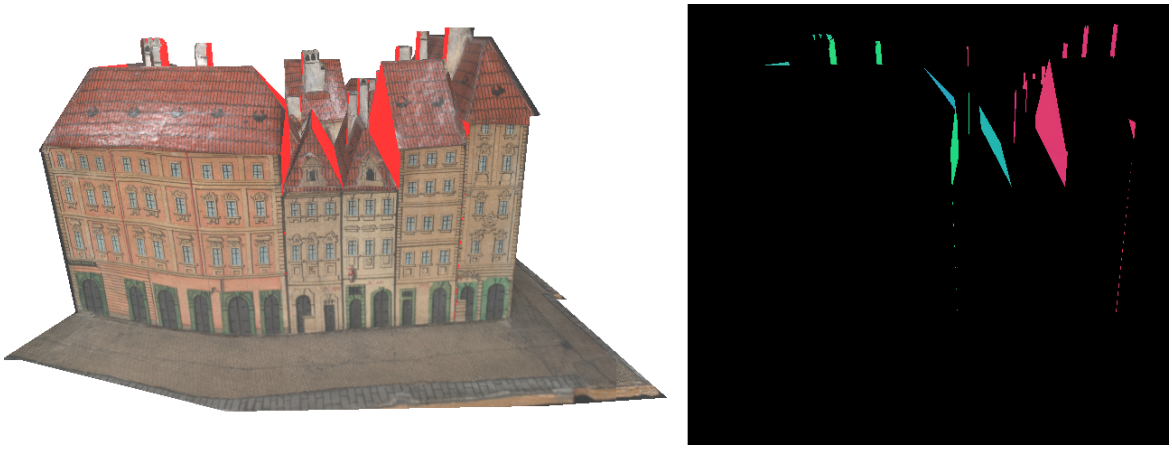
Obrázek 2.5: Výřez horní poloviny fotografie použité k texturování. Bílý bod označuje průnik optické osy s obrázkem, žlutý bod označuje vypočtený centroid. Ostatní body jsou vrcholy 3D modelu promítnuté do obrázku, červeně zvýrazněné vrcholy leží na konvexní obálce.

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. **??** (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. **??** (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



(a) (b)

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 pohledu jako (a).

pohledu je opačný k nalezeným normálám). V druhém průchodu se poté provede výsledné renderování.

2.3.4 Texturování z více fotografií

Během vykreslování 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 [**?**] vede k rozmazání textur a na testovacích datech vedlo 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 **??**. 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 renderování 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.

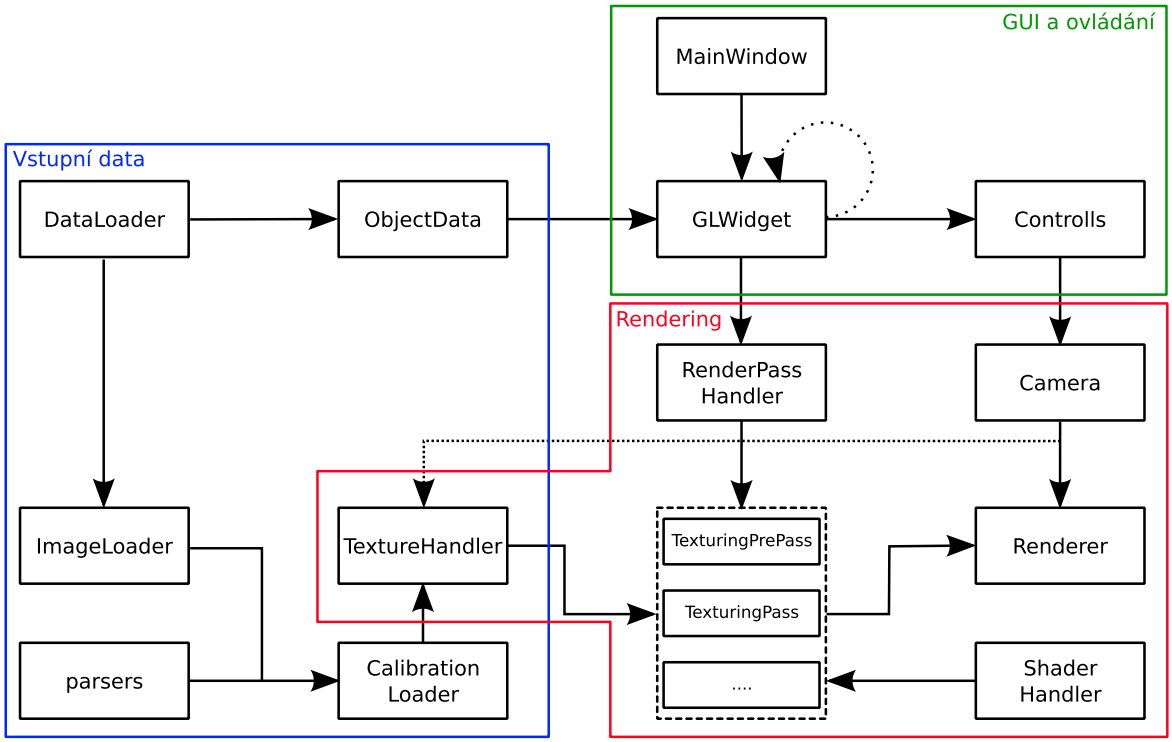
3.2 Použité technologie a knihovny

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 vykreslování. 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 **??**.



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 **??**.

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 [**?**]. 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 ležících 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 matice, kde n bodů na konvexní obálce je zapsáno do řádků matice a první bod se opakuje na konci, viz vztah **??**. Body musí být seřazené proti směru hodinových ručiček, což jde snadno zařídit již během Grahamova algoritmu.

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

Po načtení kalibrací kamer následuje načtení samotných fotografií. 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 4094×4096 px. 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 v rozlišení 16 MPx má v nekomprimovaném formátu velikost přibližně 45 MB. Moderní HDD jsou schopné číst data rychlostí přibližně 150 MB/s [**?**], to znamená, že načtení fotografie z HDD bude trvat minimálně 0.3 sec. I když budeme uvažovat moderní SSD disky s rychlostí čtení přes 500 MB/s [**?**], bude trvat načtení fotografie do RAM téměř 90 ms. To je samozřejmě naprosto nepoužitelné pro zobrazování v reálném čase, kdy při 30 fps je nutné vykreslit celý snímek za 33 ms.

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 512 px. 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 se zobrazí v plném rozlišení (konkrétní hodnoty doby načtení plných velikostí textur jsou uvedeny v kapitole testování v sekci **??**).

O načítání fotografií se stará třída `TextureHandler`. Tato třída má primárně za úkol vybírat správné fotografie pro texturování. Při výběru jsou fotografie seřazeny podle shody s aktuálním pohledem a nejvhodnější fotografie jsou vybrány pro texturování. 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 vlákno a fotografie 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 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 snímku, `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 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. Tento buffer slouží k rychlému nahrávání většího 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 **??**, 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 ale

```
1 struct TextureData {
2   mat4  u_Rt;            //matice R|t
3   vec3  u_cameraViewDir;//opraveny smer kamery
4   ivec2 u_textureSize;  //rozmery fotografie
5   float u_textureFL;    //ohniskova vzdalenost
6   float u_coveredArea;  //normalizovana plocha objektu ve fotce
7 };
8 layout(std140) uniform u_textureDataBlock {
9   TextureData ub_texData[MAX_TEXTURES];
10 };
11
12 void projectCoords(in int index, in vec4 pos, out vec2 coords) {
13   TextureData data = ub_texData[index];
14   vec3 c = (data.u_Rt * pos).xyz;
15   coords =  c.xy/c.z * data.u_textureFL + data.u_textureSize.xy * 0.5f;
16 }
17
18 bool inRange(in int index, in vec2 coords) {
19   ivec2 s = ub_texData[index].u_textureSize;
20   return coords.x >= 0 && coords.x < s.x && coords.y >= 0 && coords.y < s.y;
21 }
22
23 float computeWeight(in int index, in vec3 N, out vec2 coords, in float dl =
      dirLimit) {
24   TextureData data = ub_texData[index];
25   float weight = data.u_coveredArea;
26   weight *= dot(u_viewDir, data.u_cameraViewDir);
27
28   projectCoords(index, In.v_position, coords);
29   weight *= float(inRange(index, coords));
30
31   float dirDiff = dot(N, data.u_cameraViewDir);
32   weight *= -(dirDiff + 1.f) / (1.f - dl) + 1;
33   return weight;
34 }
```

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

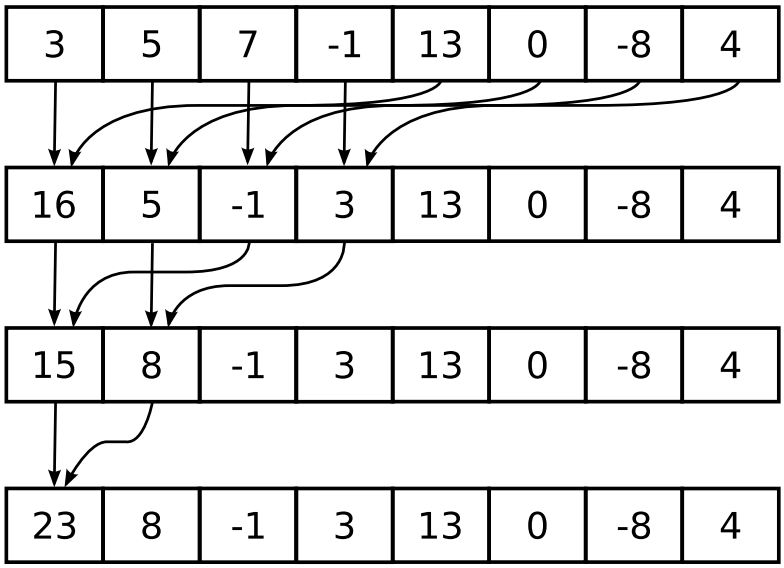
není úplně jednoduché, protože textura je uložena v paměti na GPU. Kopírovat texturu v každém snímku do RAM je velmi náročné a v real-time aplikaci nepoužitelné. Proto jsem se rozhodl provést klastrování normál na GPU pomocí výpočetních shaderů.

Výpočetní shadery jsou poměrně nová technologie, dostupná od OpenGL 4.3. Tyto shadery, na rozdíl od ostatních, nepatří do zobrazovacího řetězce a nepodílí se přímo na vykreslování. Naopak tyto shadery umožňují provádět na GPU obecné výpočty, které s grafikou vůbec nemusí souviset. 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 existující OpenGL aplikace, protože mapování bufferů či textur probíhá úplně stejně, jako u běžných shaderů. Zároveň výpočetní shadery také používají jazyk GLSL, který poskytuje řadu užitečných funkcí. Architektura GPU je masivně paralelní, proto je na GPU vhodné provádět pouze výpočty, které jdou ve velkém měřítku paralelizovat.

Pro klastrování normál je použit algoritmus k-means [**?**]. 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 klastru. 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 znázorněn na obrázku **??**.



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 klastry, 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 **??** 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 **??** 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. Proměnná `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 všechna vlákna nedosáhla 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.

```
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 iteruje přes všechny klastry, porovnává úhel mezi aktuální normálou a centroidem klastru a nastavuje index nejvhodnějšího klastru, viz ukázka **??**. 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
- HDD: SATA I 5400 ot/min
- OS: OpenSuse 13.1 64bit, linuxové jádro verze 3.11.10
- Překladač: gcc 4.8.1
- Rozlišení použité pro rendering: 1366 × 678px

Pro testování byly použity tři odlišné scény, viz tabulka **??**. První model je rekonstruovaný blok z Langweilova modelu Prahy [**?**], zapůjčený od Muzea hlavního města Prahy. Druhý model je rekonstrukce sochy a posledním modelem je částečná rekonstrukce sarkofágu. Rendery modelů jsou zobrazeny v sekci **??**.

	Trojúhelníků	**Vrcholů**	**Fotografií**	**Rozlišení**
Model 1	970	2910	321	4094 × 4096
Model 2	704940	2114820	37	2400 × 3200
Model 3	769333	2307999	9	3504 × 2336

Tabulka 4.1: Tabulka zobrazuje základní informace o testovacích scénách.

4.1 Testování výkonu

4.1.1 Testování doby vykreslování

Testování probíhalo pomocí měření doby vykreslení scény, kdy se kamera pohybovala kolem středu objektu po kružnici vždy s pohledem na střed objektu. Měření času je prováděno na GPU pomocí OpenGL časovačů. Ty používají asynchronní dotazy, které přesně změří dobu vykreslování, protože čas je 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 čekání na dokončení vykreslování, takže měření času se nijak neprojeví na výsledném výkonu.

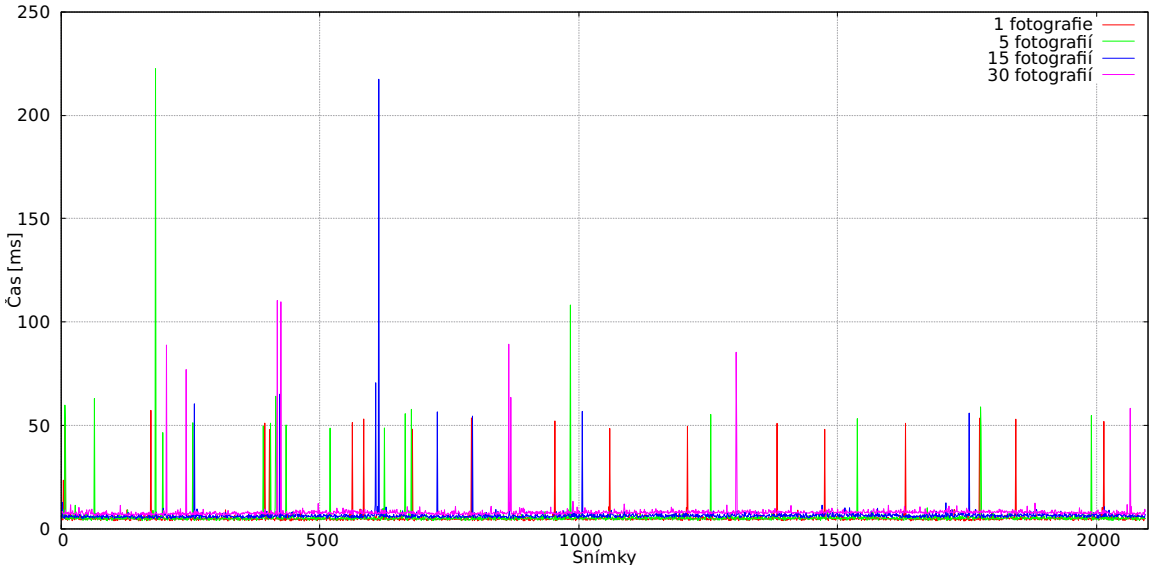
```
1  class GLTimer {
2    GLuint queryID[2];
3    uint queryBack, queryFront;
4
5    void swapQueries() {
6      if(queryBack) {
7        queryBack = 0, queryFront = 1;
8      }
9      else {
10       queryBack = 1, queryFront = 0;
11     }
12   }
13 public:
14   GLTimer() {
15     queryBack = 0, queryFront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

Zdrojový kód 4.1: Asynchroní měření času na GPU.

V ukázce **??** 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 asynchronní příkazy na GPU. Proto je odečten časovač na předním bufferu, který byl spuštěn a ukončen v minulém snímku a nyní již musí být dostupný. Poté se buffery přehodí. Časovač měří čas s udávanou přesností $10^{-9}s$, to je řádově více, než je pro běžné měření potřeba.

Model 1

První testování je provedeno pro jeden oběh virtuální kamery kolem modelu, to odpovídá 2094 měřeným snímkům. Při tomto měření nebyl zapnutý průchod pro doplňování neotexturovaných ploch.



Obrázek 4.1: Graf zobrazující čas vykreslování snímku při použití různého množství vstupních fotografií.

V grafu **??** jsou znázorněna měření při použití různého množství fotografií (1, 5, 15 a 30). V měření 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. V tabulce **??** jsou zobrazena vybraná data z grafu **??**.

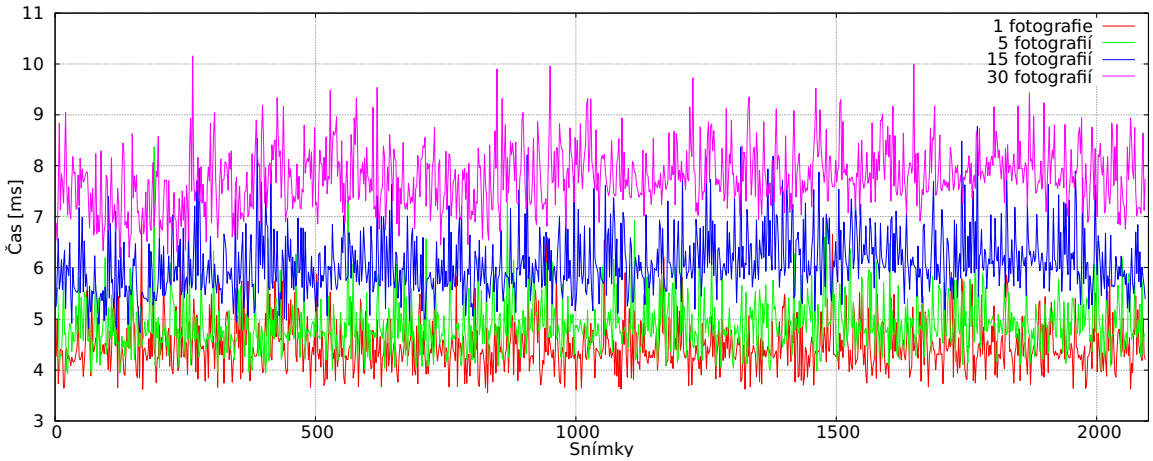
Počet fotografií:	**1**	**5**	**15**	**30**
Minimum [ms]	3,71	3,89	4,77	6,16
Maximum [ms]	57,28	222,44	217,37	110,41
Průměr [ms]	5,36	5,81	6,62	8,16
Průměr [fps]	186,48	172,17	151,00	122,48

Tabulka 4.2: Tabulka zobrazuje vybrané hodnoty z měření rychlosti vykreslování modelu 1.

Data ukazují, že počet použitých fotografií zpomaluje renderování, ale není to během vy-

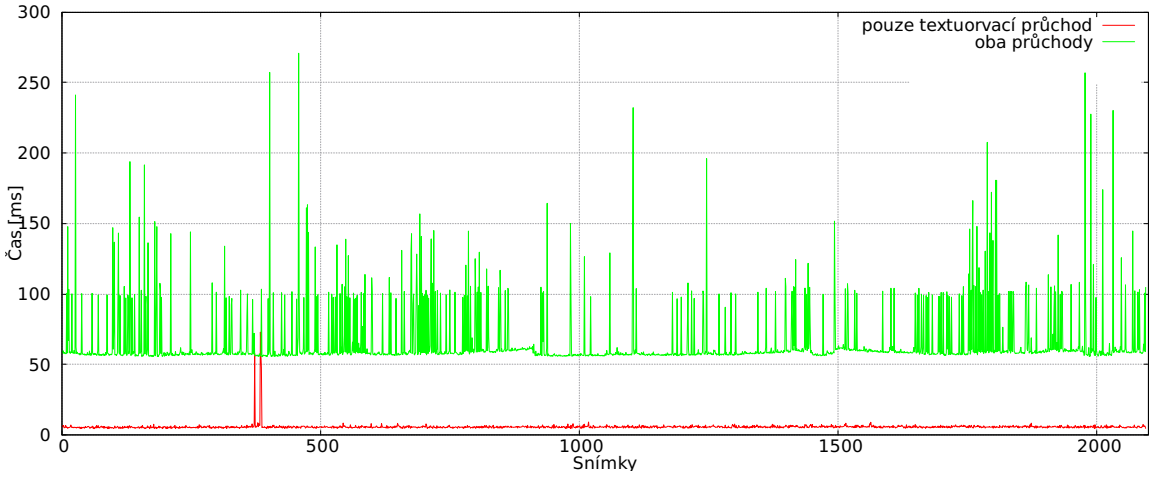
kreslování zásadní faktor. Při použití 30ti fotografií je vykreslování pomalejší o 30% než při použití 5ti fotografií.

Pro potvrzení předchozího tvrzení, že výrazné výchylky způsobuje načítání fotografií s velkým rozlišením do GPU, jsem provedl měření, při kterém byl přenos originálních velikostí do GPU vypnutý. V tomto měření se žádné výkyvy neprojevily, žádný snímek nepřekročil 11ms. Výsledky měření jsou zobrazeny v grafu **??**.



Obrázek 4.2: Graf zobrazující čas vykreslování snímků při použití různého množství vstupních fotografií pouze za použití náhledů.

V následujícím testu je porovnán běh aplikace při použití deseti vstupních fotografií se zapnutým průchodem pro doplňování neotexturovaných ploch a bez tohoto průchodu, výsledky jsou zobrazeny v grafu **??**.

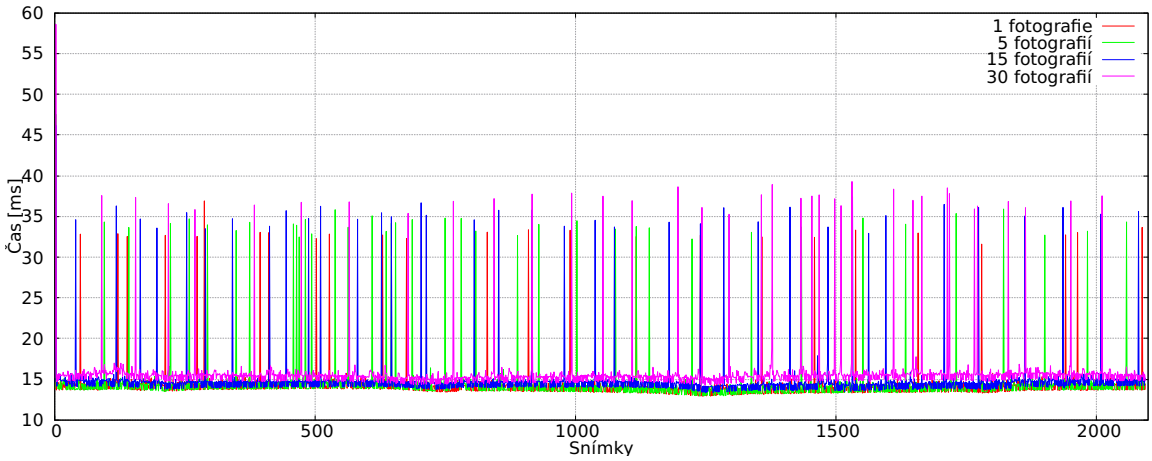


Obrázek 4.3: Graf zobrazující čas vykreslování snímků při využití průchodu pro doplňování neotexturovaných ploch.

V grafu je vidět, že při zapnutí obou průchodů je vykreslování výrazně pomalejší, průměrný čas pro vyrenderování snímku je 65,41 ms, to odpovídá 15,3 fps. V tomto případě již nemůžeme mluvit o real-time renderingu. Toto zpomalení je způsobeno zejména použitím atomických operací při paralelní redukci. Těm by se bylo možné vyhnout použitím další paralelní redukce pro součet výsledků ze všech pracovních skupin. Ta je implementačně složitější a bohužel jsem jí nestihl implementovat. Implementace toho průchodu se primárně soustředila na otestování, zda je myšlenka správná a průchod vylepšuje výsledné otexturování. Celkově je v tomto průchodu velký prostor pro případnou optimalizaci, pokud by bylo na této práci navázáno. Oproti tomu samotný texturovací průchod má průměrnou dobu vykreslení snímku 5,60ms (176 fps). Dále je při obou zapnutých průchodech vidět výrazný nárůst vrcholů způsobených načítáním textur do GPU. To je dané tím, že při pomalejším vykreslování snímků se stíhají nahrát plné velikosti všech fotografií, takže je jejich přenos do GPU mnohem častější.

Model 2

První testování proběhlo stejně jako v předchozím případě, bez zapnutého průchodu pro doplňování neotexturovaných ploch. Model má poměrně jednoduchou geometrii a tento průchod pro něj nepřináší vylepšené texturování, proto nebyl při testování vůbec použit.



Obrázek 4.4: Graf zobrazující čas vykreslování snímku při použití různého množství vstupních fotografií.

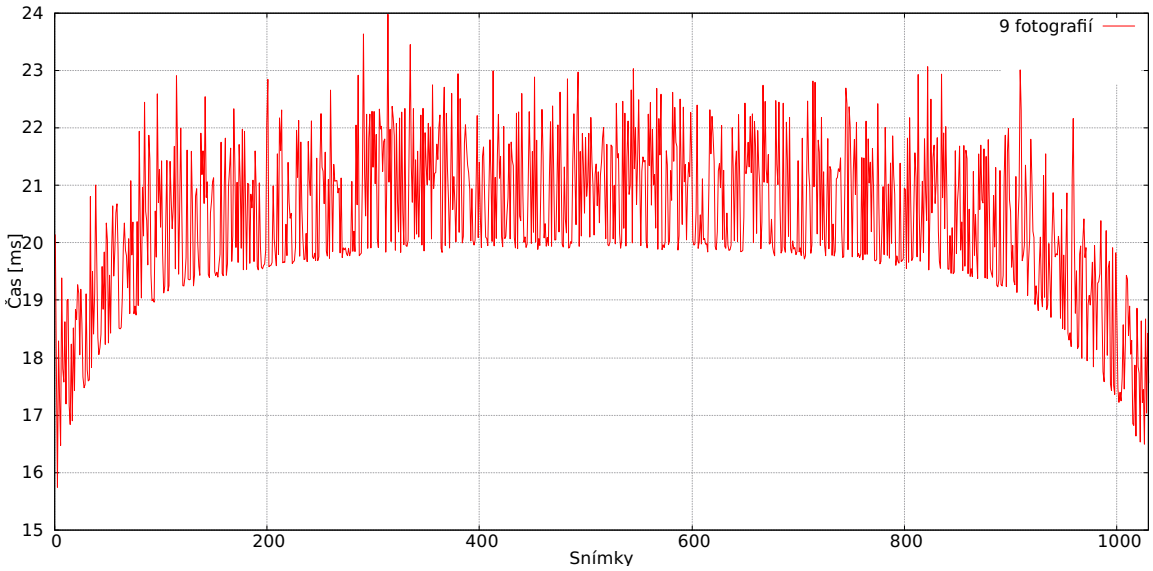
V grafu je zobrazeno podobné chování, jako u předchozího modelu. Při načítání fotografií je znatelné zpomalení vykreslování, ale protože tento model má nižší rozlišení fotografií, maximální časy nejsou tak výrazné. Celkově je vykreslování pomalejší než u předchozí scény, ale to je způsobeno zejména výrazně složitějším modelem, který má přes 700 tisíc trojúhelníků. Data jsou opět shrnuta v tabulce **??**.

Počet fotografií:	**1**	**5**	**15**	**30**
Minimum [ms]	12,93	12,94	13,34	14,27
Maximum [ms]	36,91	47,53	46,19	58,59
Průměr [ms]	14,28	14,45	14,86	15,80
Průměr [fps]	70,03	69,22	67,32	63,28

Tabulka 4.3: Tabulka zobrazuje hraniční a průměrné hodnoty z grafu **??**.

Model 3

Tento model má k dipozici pouze 9 fotografií, proto bylo měření provedeno pouze jednou se všemi fotografiemi. Průchod pro doplňování neotexturovaných ploch nebyl zapnutý, protože při použití všech dostupných fotografií nemá tento průchod smysl. Měření bylo provedeno pouze při rotaci objektu o 180°, protože se jedná pouze o jednu rekonstruovanou stěnu a vykreslování zezadu nemá smysl. Výsledky jsou zobrazeny na grafu **??**.



Obrázek 4.5: Graf zobrazující čas vykreslování snímku při devíti vstupních fotografiích.

Protože byly celou dobu použity všechny fotografie, na grafu nejsou žádné výkyvy způsobené přenosem textur na GPU. Průměrná doba vykreslení snímku byla 20,43ms, to odpovídá 49 fps. Nárůst výkonu na krajích měření je způsoben geometrií objektu, kdy z bočního pohledu je geometrie viditelná jen velmi málo.

4.1.2 Rychlost načítání fotografií z HDD

Fotografie v plné velikosti jsou z HDD načítány v nových vláknech. Testoval jsem, jestli je výhodnější načítat fotografie paralelně ve více vláknech nebo jestli je lepší využít pouze jedno vlákno. Měření bylo provedeno na modelu 1 rychlým otočením scény o 180°, kdy byly vyměněny všechny textury. Byl měřen čas od zastavení kamery do načtení poslední fotografie.

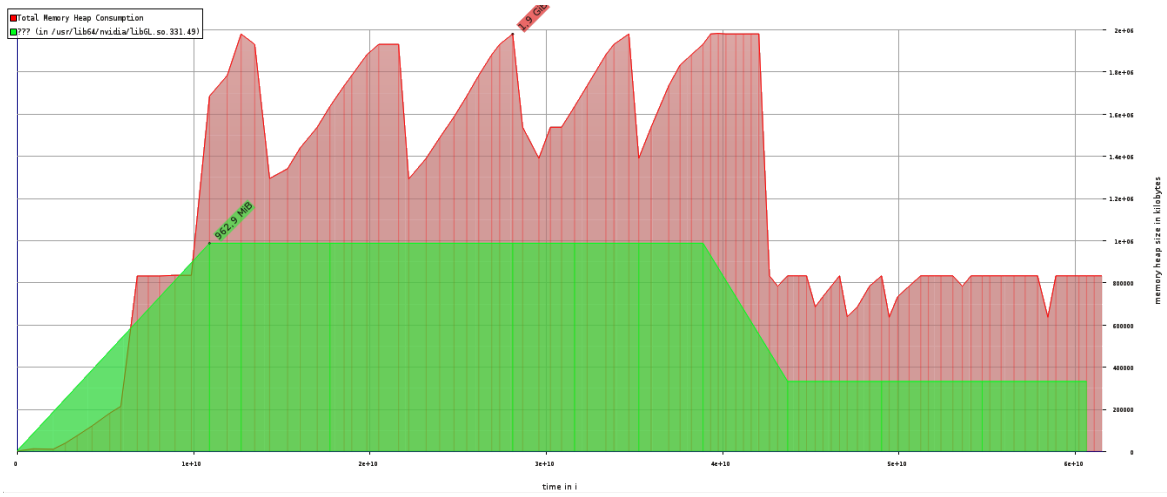
Počet fotografií:	**7**	**15**	**25**
3 vlákna [ms]	1951	10424	23765
1 vlákno [ms]	1155	8023	19094

Tabulka 4.4: Tabulka zobrazuje časy načítání různého množství fotografií z HDD. Zobrazené hodnoty jsou průměrem z deseti měření.

Výsledky ukazují, že je výhodnější načítat fotografie v jediném vláknu. To je způsobené tím, že přístup na HDD není možné paralelizovat a vlákna se musejí střídat. To způsobí, že přístupy na HDD jsou náhodné, oproti tomu při použití jediného vlákna jsou přístupy více sekvenční, tedy i rychlejší. Proto je ve výsledné aplikaci použito pouze jedno vlákno pro načítání fotografií. Naměřené hodnoty jsou poměrně vysoké, při 15 fotografiích o rozlišení 16 MPx je nutné čekat 8 s na kompletní načtení fotografií. Nicméně testování proběhlo na úsporném HDD v notebooku, na stolních počítačích se dají očekávat lepší výsledky.

4.1.3 Paměťová náročnost aplikace

Na grafu **??** je zobrazena paměťová náročnost aplikace při zobrazování modelu 1. K měření byl použit program Valgrind massif. Graf zobrazuje průběh alokované RAM v čase. První nárůst paměti odpovídá načtení náhledů při otevření scény, kterých bylo přibližně 320 MB. Poté následuje načtení 5 fotografií v plné velikosti, to odpovídá nárůstu 240 MB. V tu chvíli je velikost alokované RAM 814 MB. Za tento rozdíl je odpovědné OpenGL, které si z implementačních důvodů alokuje v RAM stejné množství paměti, jako je alokováno na GPU. Paměť alokovaná pomocí OpenGL je hrubě naznačena zelenou oblastí. Nejvýraznější nárůst paměti je způsobený zvýšením počtu používaných fotografií z 5ti na 15. Poté byla provedena rotace scény o 360°, to odpovídá velkým výkyvům uprostřed grafu, kdy byly vždy smazány nepoužívané fotografie a načteny nové. Poté proběhlo opět snížení počtu fotografií na 5, proto je uvolněna velká část paměti v RAM i na GPU. Nakonec je provedena další rotace scény, kdy jsou opět průběžně mazány a nahrávány nové fotografie.



Obrázek 4.6: Graf zobrazující množství paměti v RAM spotřebované při běhu aplikace. Osa x zobrazuje čas v procesorových cyklech a osa y spotřebovanou paměť v bytech.

Z grafu je patrné, že spotřebovaná paměť aplikace je přímo závislá na počtu aktuálně používaných fotografií. Ostatní alokovaná paměť aplikací je v porovnání s pamětí alokovanou pro fotografie zanedbatelná.

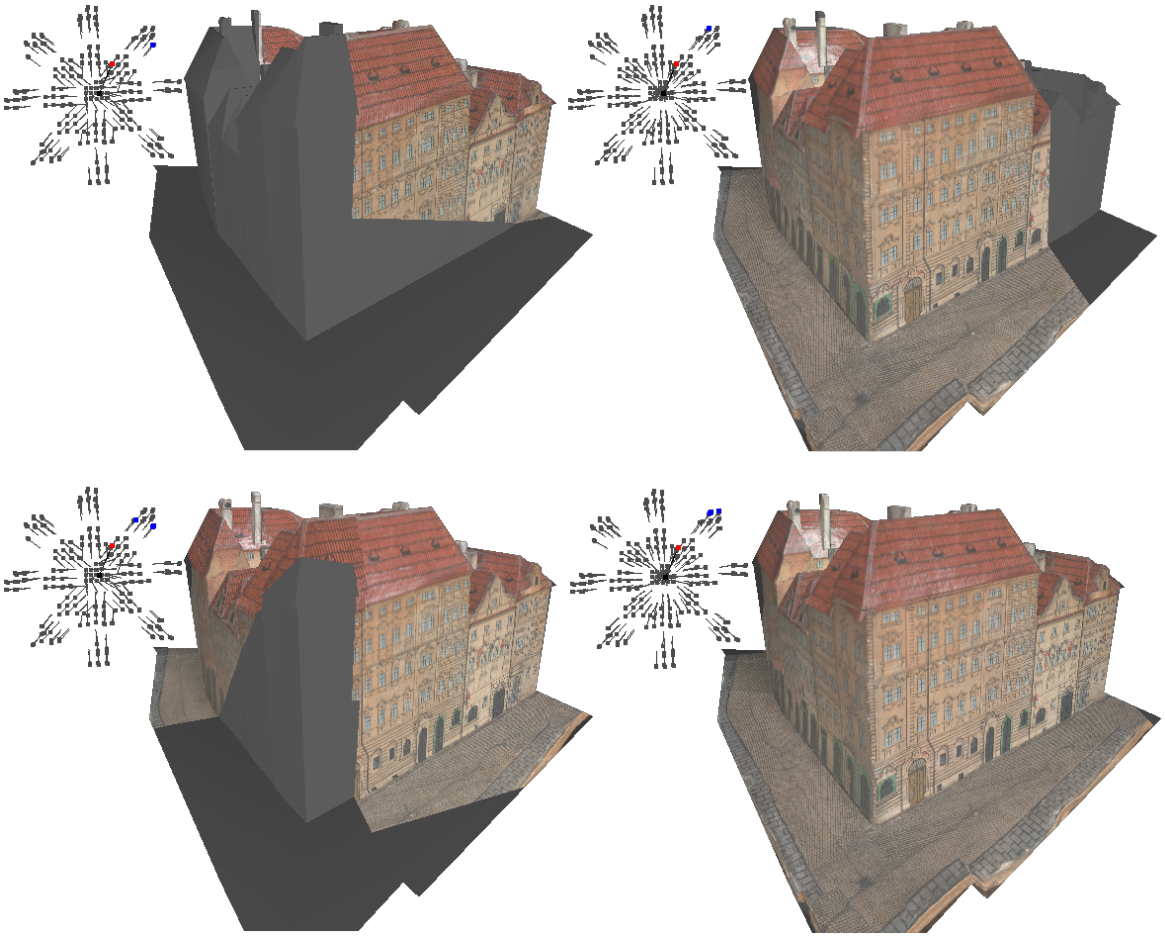
4.2 Testování kvality renderů

4.2.1 Texturování v závislosti na vstupních parametrech

V této sekci jsou ukázány rozdíly texturování v závislosti na nastavení volitelných parametrů. Těmi je počet použitých fotografií, využití průchodu pro doplňování neotexturovaných ploch a využití korigovaných směrů kamer. U každé varianty je zobrazen radar, který znázorňuje pozici a použité směry všech kamer, červeně zvýrazněnou virtuální kameru a modře zvýrazněné kamery použité pro texturování. Radar zobrazuje projekci kamer do 2D z pohledu shora, model je vždy uprostřed.

Model 1

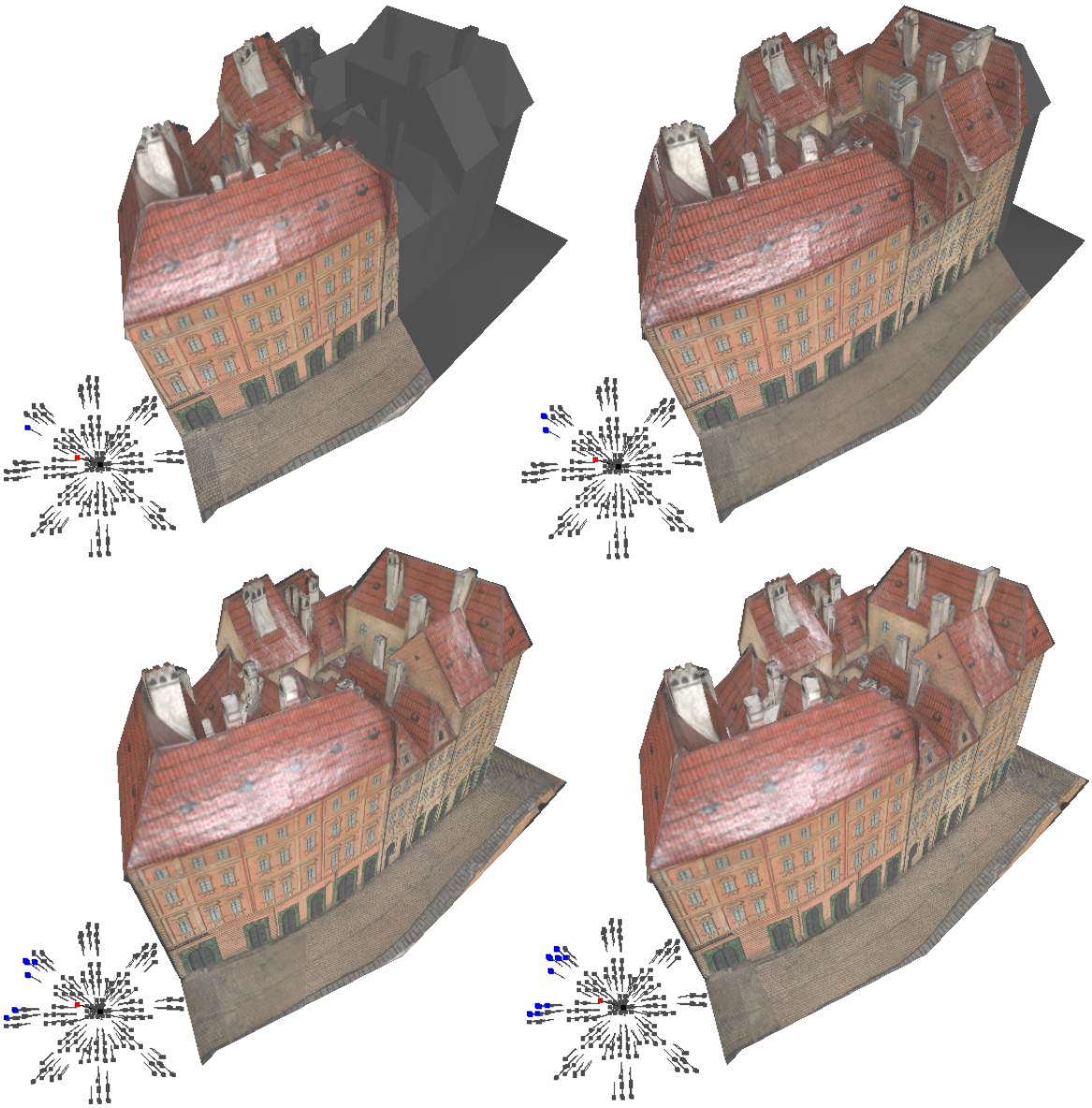
Korekce směrů kamer se nejvíce projeví při malém počtu použitých fotografií, kdy je důležité mít všechny fotografie co nejlépe vybrané, viz obr. **??**.



Obrázek 4.7: Obrázek znázorňuje texturování ze stejného pohledu bez korekce směru kamer (vlevo) a s korekcí směru kamer (vpravo) s použitím jedné (nahoře) a tří fotografií (dole).

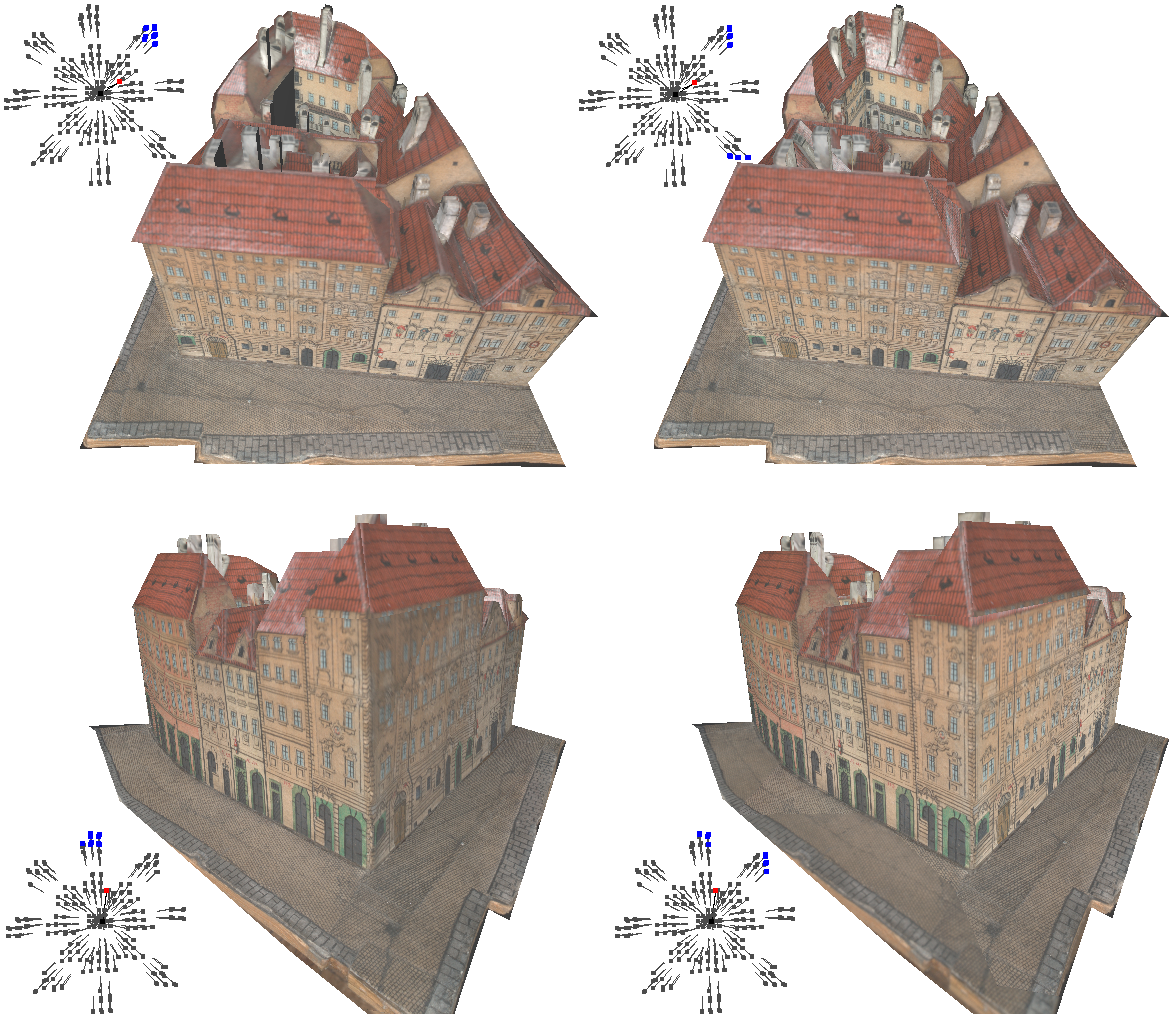
U tohoto modelu korekce způsobí velký rozdíl, protože fotografie často obsahovaly pouze části modelu a kamera nemířila na střed modelu. U ostatních modelů se korekce obvykle výrazně neprojeví. V následujících ukázkách tohoto modelu bude korekce vždy zapnutá.

Na obrázku **??** je zobrazen model s různým počtem použitých fotografií. Pro testovací scény často stačí k otexturování 10 - 15 fotografií, větší počet se již výrazně neprojeví. Zbytečně velký počet fotografií může v některých případech vést ke vzniku švů, proto je lepší nepoužívat maximální počet fotografií, pokud to není nutné.



Obrázek 4.8: Obrázek znázorňuje texturování ze stejného pohledu v závislosti na počtu použitých fotografií. Pro horní modely je použita 1 fotografie (vlevo) a 4 fotografie (vpravo). Spodní modely používají 10 fotografií (vlevo) a 20 fotografií (vpravo).

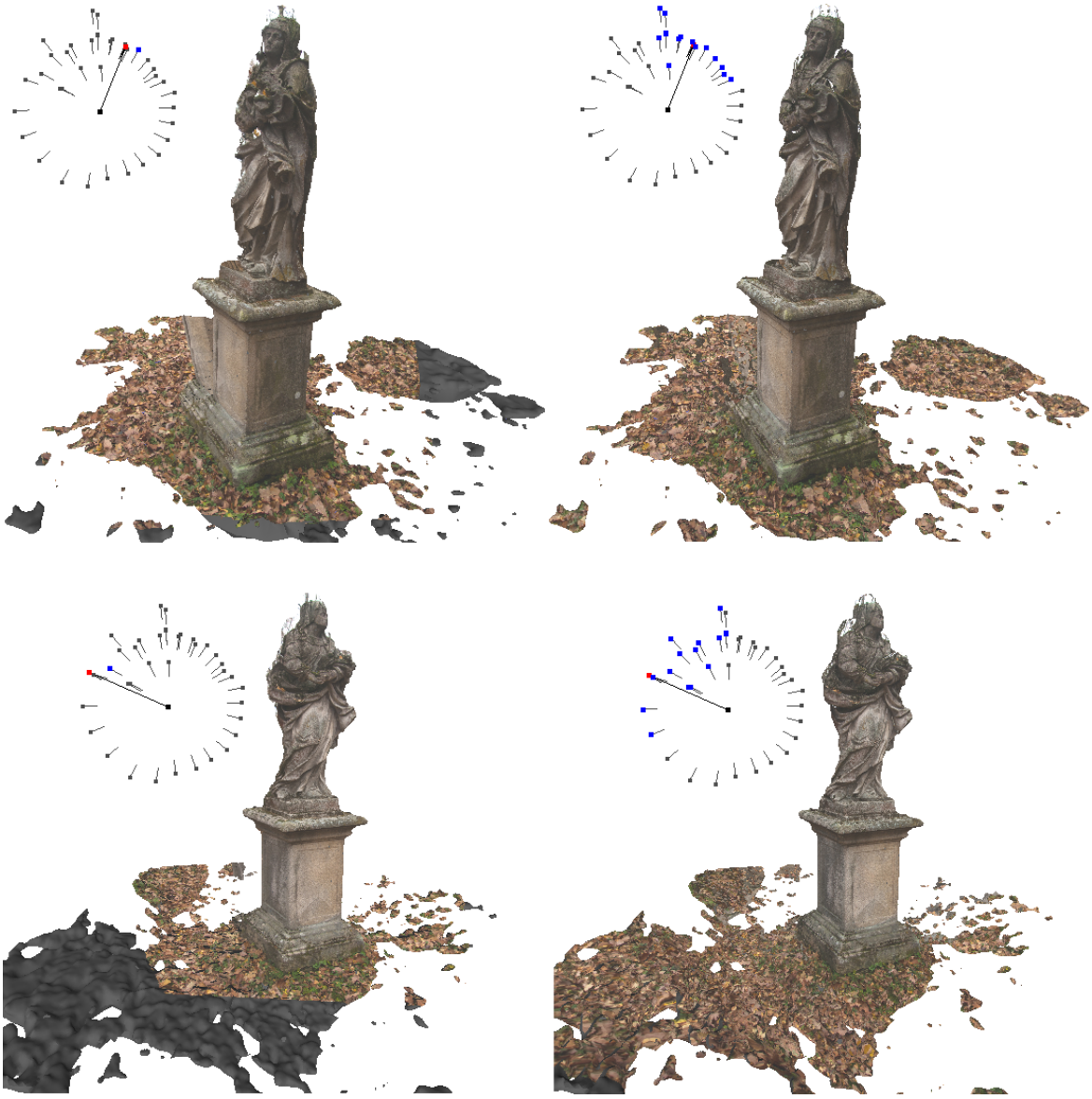
Na obrázku **??** je zobrazeno použití průchodu pro doplňování neotexturovaných ploch při použití 8mi a 12ti fotografií. Podobného výsledku jde často dosáhnout i zvýšením počtu použitých fotografií, obecně je to ale závislé na rozložení pozic a směrů kamer.



Obrázek 4.9: Obrázek znázorňuje texturování pouze s texturovacím průchodem (vlevo) a s průchodem pro pokrytí neotexturovaných ploch (vpravo). Pro horní render je použito 12 fotografií, pro spodní 8.

Model 2

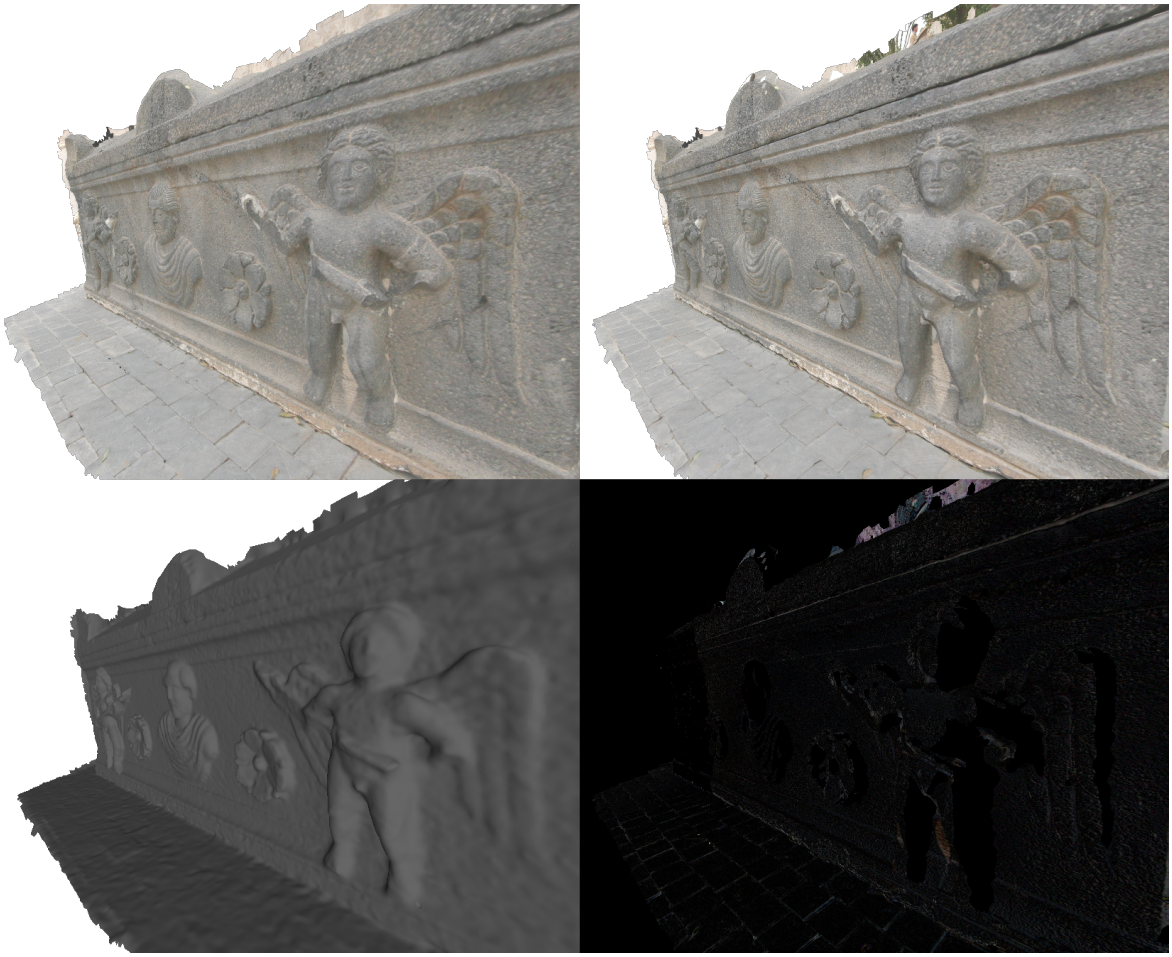
Druhý model má oproti předchozímu výrazně jednodušší geometrii a každá fotografie ze svého pohledu zabírá celý objekt. Proto nejsou rozdíly mezi počtem použitých fotografií tak výrazné. Korekce směrů v tomto případě nemá smysl, průchod pro doplňování neotexturovaných ploch také ne. Na obrázku **??** je zobrazený rozdíl mezi použitím jedné a patnácti fotografií ze dvou různých pohledů.



Obrázek 4.10: Porovnání renderu s jednou použitou fotografií (vlevo) a 15ti použitými fotografiemi (vpravo).

Model 3

Třetí model je typově podobný druhému, jedna fotografie pokrývá většinu modelu a fotografií je velmi malý počet. Na obrázku **??** je zobrazen detail při použití jedné a všech fotografií.



Obrázek 4.11: Detail modelu otexturovaný jednou fotografií (vlevo nahoře) a všemi devíti fotografiemi (vpravo nahoře). Dole je zobrazena neotexturovaná geometrie (vlevo) a rozdíl[2]mezi horními dvěma obrázky (vpravo).

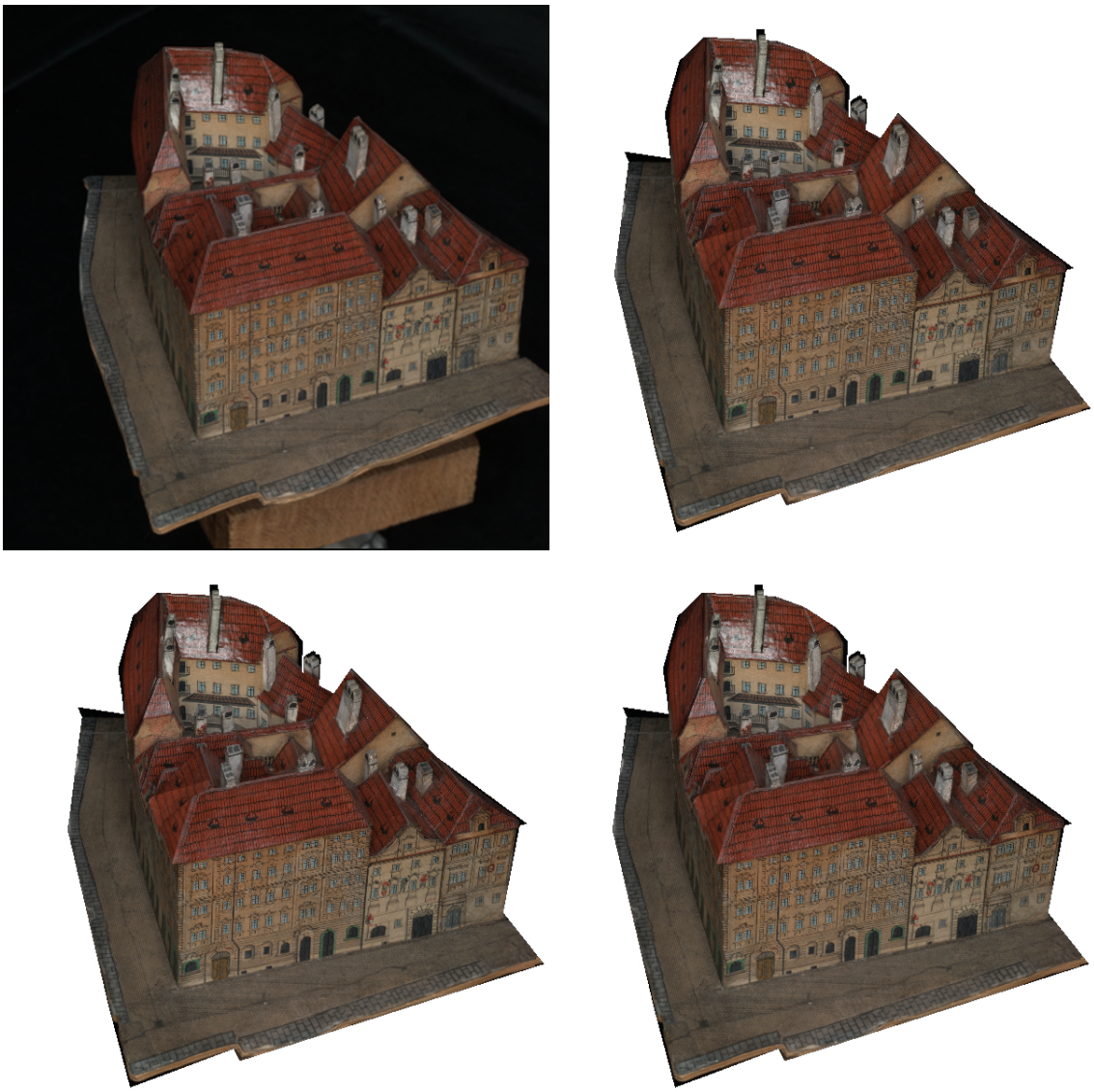
4.2.2 Porovnání s původními fotografiemi

V dalším testu jsem porovnával rendery modelů s originálními fotografiemi. Model byl vždy vykreslen ze stejné pozice a směru, jako byla původní kamera. Při testování jsem chtěl provést rozdíl obou obrázků, ale nepodařilo se mi nastavit projekční matici matici v aplikaci,

[2]Rozdíl obrázků je počítán jako absolutní rozdíl barev. Rozdíl dvou stejných pixelů je černá barva, rozdíl dvou odlišných pixelů má vždy přiřazenou absolutní hodnotu rozdílu jednotlivých barevných složek.

tak, aby přesně na pixel odpovídala původní projekční matici kamery. Rozdíly obrázků proto nebyly příliš vypovídající.

Model 1



Obrázek 4.12: Nahoře je zobrazena původní zdrojová fotografie a render s jednou použitou fotografií pro texturování. Dole jsou rendery s šesti (vlevo) a dvanácti fotografiemi (vpravo).

Při testování byla matice vybrané kamery nastavena jako projekční matice virtuální kamery. Na obrázku **??** je patrné, že původní fotografie pokryje celý model. Při použití většího množství fotografií pro texturování nenastanou velké rozdíly, protože původní fotografie bude mít váhu velmi blízkou 1 a bude použita pro většinu ploch. Výsledné rendery jsou velmi po-

dobné jako při použití jedné fotografie, obsahují jen malé rozdíly.

Model 2

Druhý model má velmi nepravidelnou geometrii, proto se při větším počtu textur výsledný render od původní fotografie liší více, než u předchozího modelu, viz obr **??**. Otexturování tohoto modelu není tak kvalitní jako u modelu 1, ale textura nemá žádné výrazné prvky, vizuálně to proto není příliš znatelné. Rekonstruovaný objekt má velká nepřesnosti v okolí hlavy a ramen, kde není dostatek obrazových dat (všechny fotografie jsou snímané spíše zespoda). S takto výraznými rozdíly si aplikace neporadí, proto se v okolí hlavy objevují části pozadí.

Obrázek 4.13: Porovnání originální fotografie a renderu ze stejného místa s deseti použitými fotografiemi.

Model 3

U třetího modelu se mi podařilo přesně nastavit projekci virtuální kamery shodnou spůvodní kamerou. Na obrázku **??** jsou proto zobrazeny i rozdíly mezi původní fotografií a vyrenderovanými snímky. Pro jednu fotografii je rozdíl nepatrný, při použití všech devíti fotografií je rozdíl výraznější. Je nutné poznamenat, že snímky byly vyrenderovány v nižším

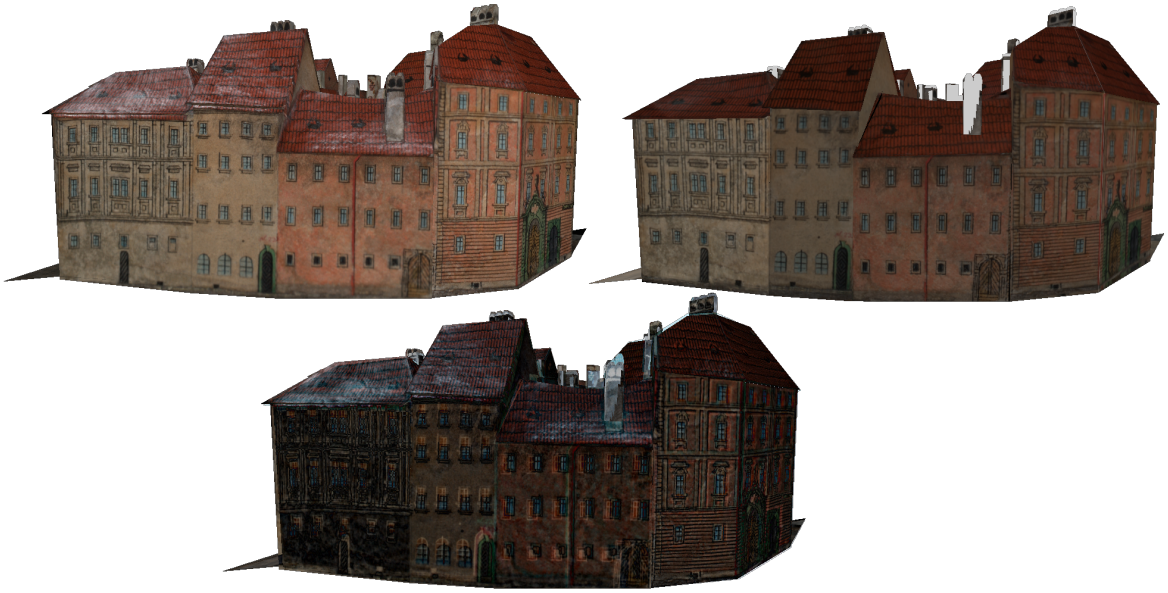
rozlišení než měla původní fotografie, do výsledku se tedy projeví i rozdíly vzniklé resamplingem textur i originální fotografie.

Obrázek 4.14: Obrázek zobrazuje v horní řadě původní fotografii a geometrii objektu. V prostřední řadě je zobrazen render s použitím jedné fotografie a rozdíl tohoto renderu vůči původní fotografii. Ve spodní řadě je zobrazen render z devíti fotografií a jeho rozdíl vůči původní fotografii.

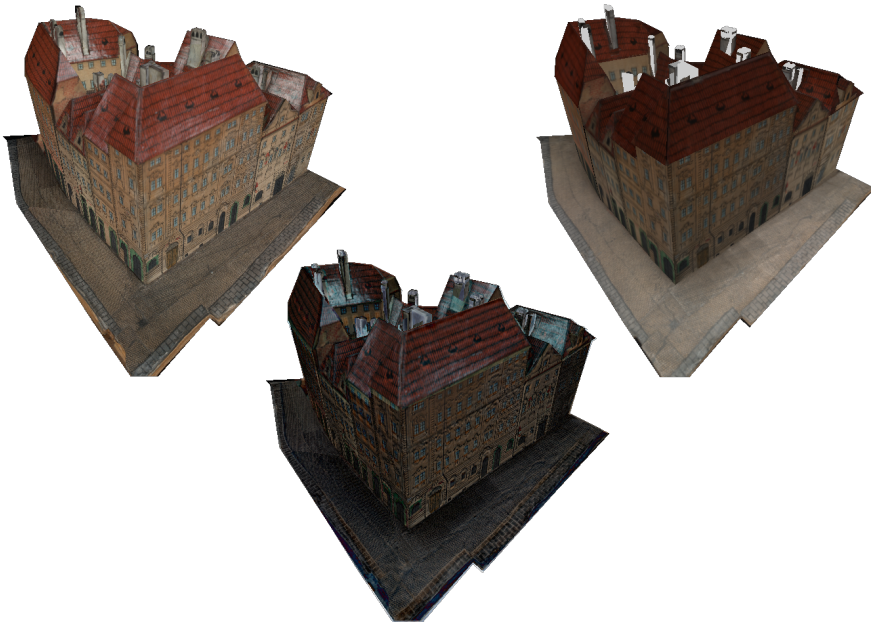
4.2.3 Porovnání s referenčními texturami

U modelu 1 jsou k dispozici textury, které byly vytvořeny ze stejných zdrojových fotografií pomocí metody popsané v [**?**]. Na obrázku **??** je porovnán render z aplikace s referenčními texturami včetně jejich rozdílu. Referenční textury jsou na modelu přesněji namapované a prošly korekcí jasu, proto je výsledný rozdíl poměrně výrazný. Na obrázku jsou chybně otexturované komíny, to je způsobené chybou ve 3D geometrii.

Na obrázku **??** je zobrazen stejný test z jiného pohledu.



Obrázek 4.15: Porovnání renderu z aplikace (vlevo) s referenčními texturami (vpravo). Uprostřed je zobrazen rozdíl obou renderů.

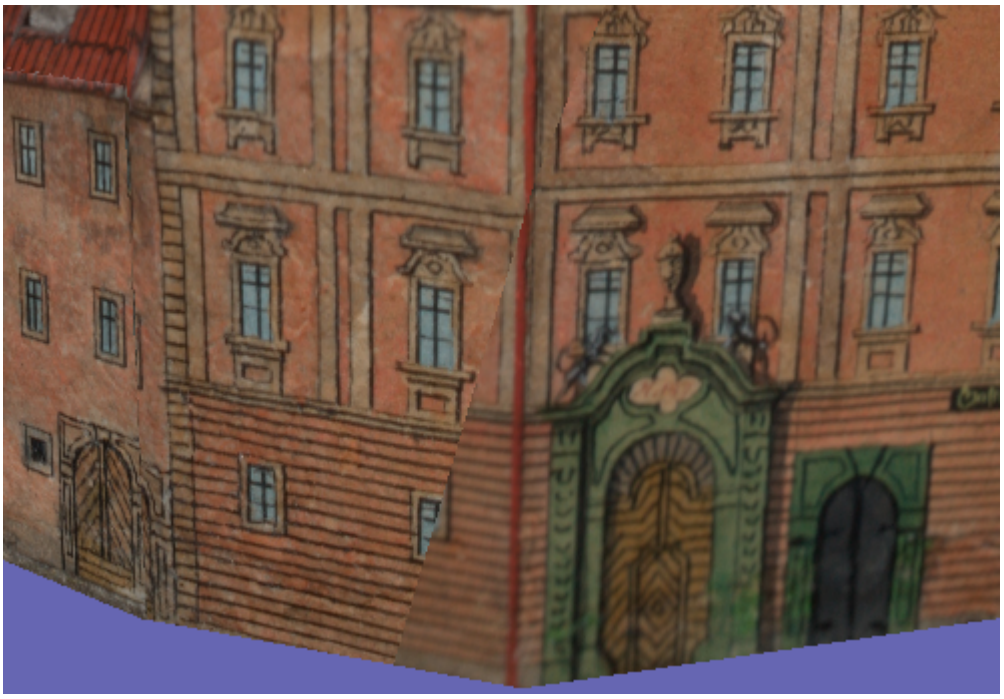


Obrázek 4.16: Porovnání renderu z aplikace (vlevo) s referenčními texturami (vpravo). Uprostřed je zobrazen rozdíl obou renderů.

4.3 Zhodnocení výsledků testování a náměty na vylepšení

Testování ukázalo, že aplikace je schopná dobře otexturovat testovací scény, ale odhalila i řadu nedostatků. Co se týče výkonu, největší problémy způsobuje kopírování velkých textur na GPU. OpenGL aktuálně nenabízí žádnou možnost, jak tento problém řešit. Existuje možnost využít DMA pomocí pixel buffer objektů, které umožňují asynchronní kopírování textur na GPU. Tato funkce je bohužel pouze optimalizací a je závislá na implementaci. Tato metoda navíc neumožňuje zjistit, zda byl asynchronní přenos textury na GPU dokončen, což přináší další velkou komplikaci pro použití v real-time aplikaci. Existují také metody mega textur [**?**] založené na metodě virtuálního texturování. Implementace této technologie je zatím pravděpodobně pouze uzavřená. Jako rozumné řešení se nabízí nahrávat textury do GPU po částech, které by se za jeden snímek nahrály bez výrazného zpomalení. Znamenalo by to ale složitější implementaci, protože by bylo nutné určit, jak velké části textury je možné zkopírovat během jednoho snímku. Zároveň by tyto části musely být co největší, aby nebyl přenos příliš dlouhý.

Související je problém načítání velkých textur z HDD, které je sice prováděno asynchronně a aplikaci neblokuje, ale v některých případech je nutné čekat na načtení všech náhledů až desítky vteřin. Dnes jsou již běžné SSD disky, které tento problém částečně omezí. Zde by bylo možné využít nějakou datovou strukturu, která by inteligentně nahrávala fotografie předem. Zároveň by ale nesměla neustálým nahráváním zpomalovat aplikaci, což byl problém, na který sem narazil při pokusu o implementaci pomocí kd-stromu.



Obrázek 4.17: Švy v texturách na hraně mezi namapovanými fotografiemi.

Posledním výraznějším problémem, co se týče výkonu, je průchod pro doplňování neotex-

turovaných ploch, který výrazně zpomaluje renderování. Tento průchod byl implementován zejména pro ověření funkčnosti, bohužel jsem neměl příliš času věnovat se jeho optimalizaci, pro kterou je rozhodně prostor. Průchod nicméně fungoval na modelu 1 poměrně dobře a vylepšoval kvalitu textur. Na ostatních scénách se neprojevil, protože scény nebyly tak rozsáhlé a pro kvalitní otexturování stačilo i relativně malé množství fotografií.

Z vizuálních chyb se nejčastěji objevují švy mezi jednotlivými texturami, viz obr. **??**. Nicméně pomocí použitého váženého schématu při texturování se podařilo výskyt švů omezit, objevují se zejména u krajů modelu a na hranách, kde nejsou výrazně viditelné.

Další chybou, která se v texturách vyskytuje, je nepřesné zarovnání textur vůči modelu. V tomto případě je textura namapována na plochu, pro kterou teoreticky existuje vhodnější textura, obr. **??**. Výskyt těchto chyb je zejména podél hran ve 3D geometrii při pohledech, které nejsou na plochu příliš kolmé. Tyto artefakty se často odstraní při zvětšení počtu používaných fotografií.

Obrázek 4.18: Chybně otexturovaná část plochy přesahující texturou.

Mezi další chyby patří zejména otexturování ploch, které jsou z dané kamery zastíněné, viz obr. **??**. Tento problém je výrazně omezen preferovaným mapováním fotografií s nejlepší shodou směru pohledu, ale není úplně eliminován. Zejména při detailních pohledech mohou být tyto chyby výrazné. Chyby je v některých případech možné řešit zvětšením počtu použitých fotografií, kdy je pro zastíněnou plochu použita lepší fotografie. Nicméně univerzální řešení toho problému se nám nalézt nepodařilo, zejména kvůli velkému počtu vstupních fotografií, který neumožňuje efektivní implementaci algoritmů založených na stínových mapách či vrhání paprsků.

Obrázek 4.19: Chybné otexturování části plochy, která ze zdrojové kamery není viditelná.

Na renderch scén je také patrné zobrazení spekulárních odlesků od blesku fotoaparátu, které snižují kvalitu textur. Řešení tohoto problému nebylo v práci zahrnuto, nicméně bylo by možné použít metodu navrženou v [**?**], která na stejných datech odlesky efektivně odstraňuje.

Další vizuální chyby vznikají zejména při použití malého počtu fotografií, kdy není dostatek dat pro kvalitní otexturování všech ploch. Obecně se ale na testovacích scénách ukázalo, že není potřeba co největší počet používaných fotografií, obvykle stačilo 8 - 15 fotografií pro kvalitní otexturování scény a větší počet fotografií se již výrazně neprojevil.

Čistě implementačním problémem je vznik aliasingu při zmenšování textur. Tento problém je tradičně řešení pomocí mipmapingu, bohužel jsem na začátku implementace pro jednodušší práci s fotografiemi zvolil použití obdélníkových textur, které mipmapy nepodporují. Tato úprava by nicméně neznamenala výrazné zásahy do aplikace.

Kapitola 5

Závěr

V práci byl navrhnut a implementován algoritmus pro pohledově závislé mapování textur v reálném čase. Výsledná aplikace pracuje v reálném čase a je schopná zároveň mapovat 32 i více fotografií. I když chod aplikace může být při použití velkých zdrojových fotografií krátkodobě zpomalen, jak je detailně popsáno v sekci **??**, průměrné hodnoty fps jsou pro zobrazování v reálném čase naprosto dostačující.

Hlavním cílem bylo kvalitní otexturování modelu z fotografií bez nutnosti náročného procesu tvorby textur. I když výsledky texturování nejsou tak kvalitní, jak bylo před implementací metody zamýšleno, podařilo se omezit vznik a výskyt běžných artefaktů a výsledné textury mají přijatelnou kvalitu. Pro dostatečné otexturování testovacích scén obvykle postačuje 5 - 15 používaných fotografií.

Během implementace se jako jeden z největších problémů ukázala práce s velkými fotografiemi, které bylo nutné načítat za běhu z HDD a poté je bylo nutné načítat průběžně do GPU. Tento problém byl nakonec vyřešen pomocí malých náhledů, které se použijí jako textura, dokud není nahraná fotografie v plném rozlišení. Dále se jako problematické ukázalo vytvořit robustní návrh OpenGL aplikace. I když real-time rendering je základem moderní počítačové grafiky, většina pozornosti je věnována rychlým algoritmům a akceleračním datovým strukturám a neexistují zažité obecné architektury, jako je např. MVC v desktopovém a webovém prostředí.

Příloha A

Seznam použitých zkratek

API Application Programming Interface, aplikační rozhraní

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, počet snímků za vteřinu

DMA Direct memory access

MVC Model view controller

OpenGL Open Graphics Library

CUDA Compute Unified Device Architecture

GLEW OpenGL Extension Wrangler Library

OpenCL Open Computing Language

GLM OpenGL Mathematics

Příloha B

Instalační a uživatelská příručka

B.1 Překlad aplikace

Program využívá pro překlad qmake. Nejprve je nutné vytvořit soubor Makefile spuštěním příkazu ve složce src:

```
qmake -o Makefile qt-Release.pro
```

Poté se provede překlad aplikace v závislosti na použitém kompileru, obvykle pomocí:

```
make -f Makefile
```

Tím se vytvoří spustitelný soubor ve složce `dist/Release`. Pro spuštění aplikace je dále nutné zkopírovat složku `shaders` do složky, kde se nachází spustitelný soubor.

Pro přeložení programu je nutné poskytnout následující knihovny:

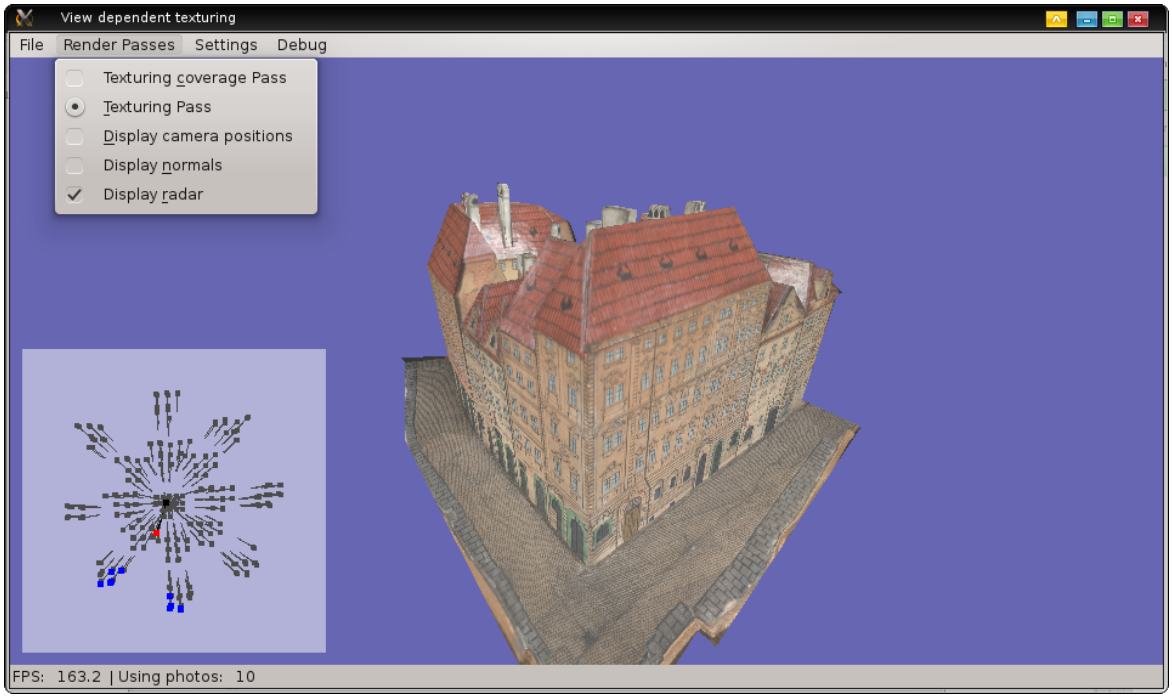
Knihovna	**Minimální verze**
Qt	4.8
OpenGL	4.0
GLEW	1.8
Assimp	3.0
libjpeg	8

Tabulka B.1: Knihovny nezbytné pro překlad a spuštění aplikace.

B.2 Ovládání aplikace

Aplikace obsahuje grafické rozhraní, které umožňuje standardní ovládání. Po spuštění je nutné načíst scénu pomocí volby File > Open v horním menu. Poté se otevře dialog, kde je možné zvolit typ kalibračních dat a je nutné nastavit příslušné cesty.

Kamera se ovládá klávesami `W,A,S,D`, rotace scény je prováděna myší se stisknutým levým tlačítkem. Pomocí volby Render Passes v horním menu je možné zapínat a vypínat jednotlivé průchody, viz obr. **??**. V položce Settings je možné otevřít okno pro rotaci modelu a okno pro nastavení parametrů. V něm je možné nastavit počet používaných fotografií,



Obrázek B.1: Ukázka aplikace.

zapnout korekci pohledových směrů a zapnout bilineární filtrování textur. Na spodní liště zobrazuje aplikace aktuální fps a počet používaných fotografií.

Příloha C

Obsah přiloženého CD

```
.
├── bin
├── data
│   ├── dum8
│   │   └── photos
│   ├── sarkofag
│   │   └── photos
│   └── socha
│       └── photos
├── doc
├── src
│   ├── lib
│   ├── shaders
│   └── src
└── text
    └── figures
```