VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

Fakulta informačních technologií
Faculty of Information Technology

Ústav počítačové grafiky a multimédií
Department of Computer Graphics and Multimedia

LOD pro GPUEngine

LOD for GPUEngine

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR
AUTHOR

Bc. Jan Staněk

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Tomáš Starka

Brno, 2018

Abstrakt

Při vykreslování 3D polygonálních modelů se objevují problémy s reprezentací na různých úrovních detailu. Vysoce detailní modely umístěné ve větší vzdálenosti od kamery trpí nežádoucím aliasingem plynoucím z hrubosti vzorkování jejich povrchu a vzhledem k zobrazené velikosti spotřebuje jejich vykreslení neúměrně velké množství času. Méně detailní modely naopak snižují vizuální kvalitu scény, pokud se vyskytují dostatečně blízko ke kameře.

Tato práce se zabývá teorií a praktickými technikami pro řešení těchto problémů. Jsou zde rozebrány různá publikovaná řešení a jejich principy, a navržena a provedena implementace vybraných technik pro knihovnu GPUEngine.

Klíčová slova

Úroveň detailu, LoD, GPUEngine, automatizace, zpracování 3D modelů.

Abstract

The representation of 3D polygonal model on several levels of available detail is a problem inherent in the process of rendering a scene. Highly-detailed models, if placed far from the camera, suffer from spatial aliasing that results from inadequate sampling of their surface, and require disproportionately large amount of time to render. Low-detailed models on the other hand reduce the visual quality of the scene when placed too near to the camera.

This report delves in both the theory and the practical techniques used for solving these problems. It describes various published solutions and the principles behind them, and presents a design and an implementation of selected techniques for the GPUEngine library.

Keywords

Level of Detail, LoD, GPUEngine, automation, processing of 3D models.

STANĚK, J., Bc. *LOD pro GPUEngine*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Starka.

Prohlášení

Prohlašuji, že jsem tuto práci vypracoval samostatně pod vedením Ing. Tomáše Starky. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Bc. Jan Staněk
v Brně 22. května 2018

Poděkování

Děkuji svému vedoucímu Ing. Starkovi za trpělivost a obětavost při konzultacích této práce. Také děkuji všem, kteří mě při tvorbě práce podporovali.

Obsah

1 Úvod **1**

2 Teorie **2**
2.1 Pojem Level of Detail . . . 2
2.1.1 Statický LoD . . . 2
2.1.2 Dynamický LoD . . . 3
2.2 Zjednodušování modelu . . . 4
2.2.1 Sjednocení hrany . . . 4
2.2.2 Sjednocení trojúhelníka . . . 6
2.2.3 Odstranění vrcholu . . . 6
2.3 Chybové metriky . . . 7
2.3.1 Geometrické metriky . . . 7
2.3.2 Metriky založené na atributech modelu . . . 8
2.3.3 Quadric Error Metric . . . 8
2.4 Výběr prvků modelu pro zjednodušení . . . 10
2.4.1 Neoptimalizovaný výběr . . . 10
2.4.2 Hladový a líný výběr . . . 10

3 Návrh rozšíření **12**
3.1 GPUEngine . . . 12
3.2 Struktura rozšíření . . . 12
3.3 Grafová abstrakce polygonové sítě . . . 13
3.4 Prvky zjednodušujícího procesu . . . 14

4 Implementace **16**
4.1 Abstrakce grafu . . . 16
4.2 Prvky simplifikace . . . 18
4.2.1 Předávání informací mezi komponentami . . . 18
4.2.2 Metriky . . . 19
4.2.3 Operátory . . . 20
4.2.4 Řídící algoritmy . . . 20
4.2.5 Veřejné rozhraní knihovny . . . 21
4.3 Demonstrační aplikace . . . 21
4.4 Další vývoj a možná rozšíření . . . 22

5 Závěr **24**

1 Úvod

Při počítačovém vykreslování 3D modelů popsaných pomocí sítě polygonů (*mesh*) záhy narazíme na problémy spojené s převodem 3D dat na 2D plochu obrazovky. Pokud je kamera ve scéně umístěna blízko modelu, je většinou žádoucí, aby model obsahoval větší množství detailních informací. Problém nastane, pokud se stejný model pokusíme vykreslit naopak ve velké vzdálenosti od kamery – a to třeba takové, že výsledná 2D reprezentace bude na obrazovce zabírat pouhých pár jednotek pixelů. Nejenže strávíme nezanedbatelné množství času renderováním detailů, které na takto malé ploše nemohou být rozumně zobrazeny, díky hrubosti vzorkování při převodu z 2D do 3D bude také docházet k aliasingu a jiným nežádoucím jevům.

Řešením tohoto a jemu podobných problémů se zabývá oblast počítačové grafiky souhrnně nazývána jako *Level of Detail* – Úroveň detailu. Techniky z tohoto odvětví se snaží upravovat vykreslování 3D modelů tak, aby bylo možné používat detailní modely při pohledu zblízka a zároveň nedocházelo k plýtvání výkonem a nežádoucím jevům při pohledu z větší vzdálenosti.

Cílem této práce je poskytnout čtenáři teoretický přehled technik a jejich principů z oblasti automatického zpracování 3D polygonálních modelů za účelem generování zjednodušených úrovní detailu. Dále si práce klade za cíl rozšířit existující knihovnu GPUEngine o implementaci vybraných technik z této oblasti.

V následující kapitole je obsažen teoretický přehled základních principů, technik a úhlů pohledu používaných v současné době pro práci s úrovněmi detailu. Cílem této kapitoly je uvést čtenáře do problematiky a položit základy pro návrh implementace.

Třetí kapitola se zabývá návrhem samotného rozšíření knihovny GPUEngine. Popisuje výchozí stav této knihovny společně se souvisejícími možnostmi a omezeními. Dále se zabývá konceptuálním popisem vhodných datových struktur, návrhem jednotlivých komponent rozšíření a popisem vztahů mezi těmito komponentami.

Následující kapitola se věnuje samotné implementaci. Tato kapitola popisuje jednotlivé prostředky a prvky poskytnuté implementovaným rozšířením. Důraz je kladen zejména na popis zamýšleného použití jednotlivých komponent. V závěru kapitoly je pak stručně popsána demonstrační aplikace a navrženy možné směry budoucího vývoje.

Poslední, závěrečná kapitola popisuje dosažené výsledky. Je zde shrnut stav výsledné knihovny a způsob získání jejích zdrojových kódů. Také jsou zde zhodnoceny dosažené výsledky a nastíněno možné budoucí pokračování práce.

2 Teorie

Tato kapitola se zabývá přehledem přístupů k problematice reprezentace a zpracování úrovní detailu. V první části jsou rozebrány a osvětleny obecné pojmy související s touto problematikou. Další části tvoří přehled různých přístupů k automatickému vytváření zjednodušených úrovní detailu existujících modelů.

2.1 Pojem Level of Detail

Level of Detail (LoD) je zároveň moderní i velmi stará oblast počítačové grafiky. Její jádro spočívá v hledání kompromisů mezi kvalitou zobrazení a rychlostí vykreslování.

Původní myšlenka této oblasti byla popsána v článku *Hierarchical Geometric Models for Visible Surface Algorithms*, který publikoval Clark v roce 1976 [1]. Autor si uvědomil, že pokud bude například koule pozorována vždy z dostatečně velké vzdálenosti, lze ji nahradit dvanáctistěnem. Problém nastává ve chvíli, kdy se pozorovaný objekt přiblíží k pozorovateli, a tato záměna začne být zřetelná. Jedním z možných řešení by bylo používat pro vykreslování pouze plně detailní modely, důsledkem čehož by se ale ve scéně objevovalo mnohem více polygonů, než je možné v přijatelném čase zpracovat. Jako řešení tohoto dilematu autor navrhuje použití jednoho plně detailního modelu společně s méně detailními náhradami. Vykreslující algoritmus pak vybere jeden z těchto modelů na základě počtu pixelů, které bude reprezentace modelu po vykreslení zabírat na obrazovce.

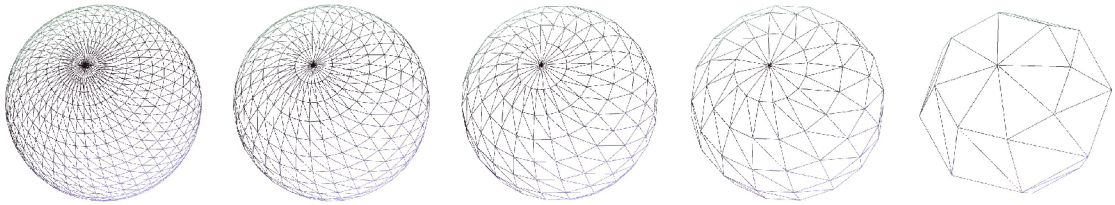
Tento základ byl postupem času upravován a rozvíjen. Vznikala různá alternativní řešení pro druhy modelů, pro které byl výše nastíněný způsob nevyhovující. S měnícím se grafickým hardwarem přibývaly úpravy snažící se jej využívat. Výsledkem je poměrně široký záběr jak této problematiky, tak publikovaných řešení. Abychom byli schopni začít mezi jednotlivými technikami rozlišovat, klasifikujeme si je do obecných skupin podle přístupu ke zjednodušovaní modelu: *statický* a *dynamický* LoD.

2.1.1 Statický LoD

Pod označením *Statický Level of Detail* (v angličtině *Static* nebo také *Discrete Level of Detail* [2]) se skrývá výše nastíněný tradiční způsob zjednodušování modelů. Jedná se pravděpodobně o v praxi nejvíce využívanou skupinu algoritmů.

Algoritmy z této kategorie v přípravné fázi (*preprocessing*) vytvoří několik diskrétních variant zjednodušovaného modelu. Během samotného vykreslování je poté z těchto variant zvolena ta nejvhodnější pro daný snímek a pozici ve scéně. Příklad diskrétních variant modelu lze nalézt na obrázku 2.1.

Mezi hlavní výhody tohoto přístupu patří jednoduchost implementace. Protože zjednodušení modelu probíhá v přípravné fázi před vykreslováním, není zapotřebí zjednodušovací algoritmus přehnaně optimalizovat na rychlost. Naopak lze brát ohled na použitý grafický hardware a vytvářet varianty modelu optimalizované pro rychlé vykreslení. Rychlostně optimalizovaný pak musí být pouze způsob výběru správné varianty během vykreslování.



(a) ± 5500 polygonů (b) ± 2880 polygonů (c) ± 1580 polygonů (d) ± 670 polygonů (e) ± 140 polygonů

ZDROJ: MaxDZ8 (Higly tassellated wireframe sphere) [Public domain], via Wikimedia Commons

Obrázek 2.1: Příklad diskrétních úrovní detailu. Vysoce detailní model koule (a) byl postupně zjednodušen až na hrubou a „hranatou“ aproximaci (e).

Protože ale předzpracování modelu probíhá ve zvláštní fázi, nemohou techniky spadající do této skupiny brát ohled na polohu kamery ve scéně (odtud také pochází další možné označení této skupiny: *view-independent* – „na pohledu nezávislé“). Jak bude popsáno dále, toto omezení je pro některé druhy modelů činí téměř nepoužitelnými.

2.1.2 Dynamický LoD

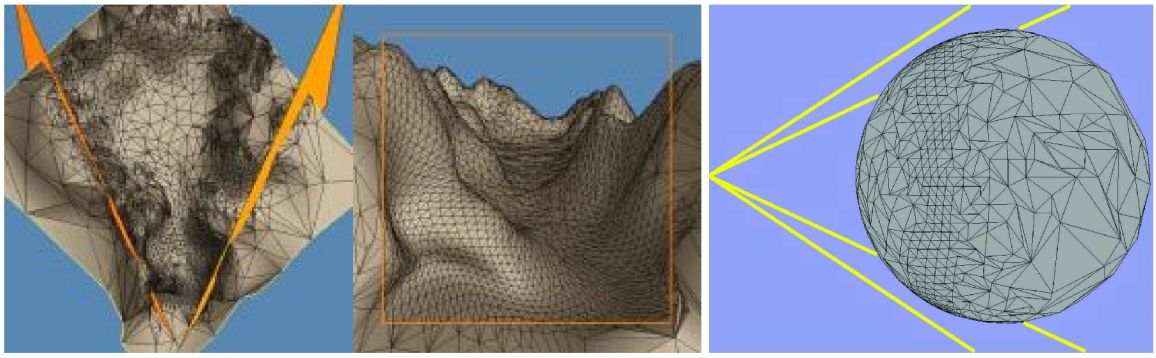
Pojem *Dynamický* nebo také *Progresivní Level of Detail* (podle článku *Progressive Meshes* [3]) popisuje skupinu technik, která k problému konstrukce zjednodušeného modelu přistupuje téměř z opačné strany. Namísto několika diskrétních úrovní zjednodušení je model popsán jako (v rámci možností) kontinuální spektrum detailů. Toto spektrum je v paměti uloženo ve speciální datové struktuře, která umožňuje rychlé extrahování požadované úrovně detailu přímo během vykreslování.

Hlavní výhodou tohoto přístupu je velká granularita jednotlivých úrovní. Díky jemnějšímu spektru detailů, než jaké nabízí Statický LoD, lze pro požadovanou situaci vybrat (nebo lépe řečeno zkonstruovat) vhodnou úroveň detailu s přesností na jednotlivé polygony modelu.

Další nespornou výhodu tento přístup přináší u větších modelů načítaných z disku nebo skrze síťové spojení. Vzhledem k formátu popisu modelu lze začít vykreslovat ještě před načtením celého modelu. Také je tento formát relativně odolný na přerušení spojení – pokud se nepodaří načíst všechna data, stále je možné model vykreslit, i když pouze ve zjednodušené podobě [2].

V rámci popisu dynamických technik je nutné vzpomenout i tzv. adaptivní nebo také „na pohledu závislý“ (*view-dependent*) LoD. Jeho podstatou je úprava progresivních metod takovým způsobem, aby bylo možné na jednom snímku použít více úrovní detailu v rámci jednoho modelu. Tento přístup je v podstatě nutností u krajin a podobně rozsáhlých modelů. V opačném případě by pro zachování vizuální kvality bylo nutné používat plně detailní variantu, i když většina modelu splňuje kritéria pro použití variant méně detailních [4].

Příklad adaptivního výběru úrovně detailu na modelu krajiny lze nalézt na obrázku 2.2a. Na obrázku 2.2b je pak zobrazeno další možné použití této techniky: použití detailního popisu modelu na viditelných okrajích koule pro zachování obrysu a méně detailního popisu na zbývajících částech modelu.



(a) Adaptivní model krajiny.
Zdroj: Hoppe 1996 [3]

(b) Adaptivní model koule.
Zdroj: Luebke 2001 [5]

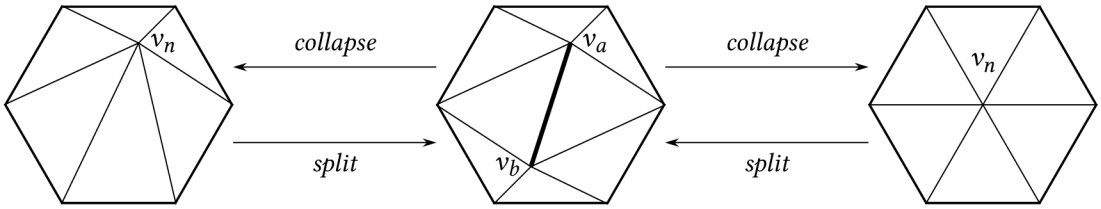
Obrázek 2.2: Příklady adaptivních úrovní detailu. Oranžové roviny a žluté linie vyznačují zorné pole pozorovatele. Za povšimnutí stojí rozložení detailů zejména u (b), kde jsou detailně vykresleny pozorované obrysy, zatímco zbývající oblasti jsou zjednodušeny.

2.2 Zjednodušování modelu

Generování zjednodušeného modelu je realizováno aplikací tzv. *operátorů*. Pojem *operátor* v tomto kontextu označuje operaci modifikující daný model. Tato práce se zabývá *lokálními operátory*, které zjednodušují geometrii modelu v malém (lokáním) okolí nějaké složky modelu (vrcholu, hrany, trojúhelníka, ...). Tyto operátory typicky s každou aplikací odstraní malé množství polygonů modelu, ideálně bez modifikace celkové topologie. Existují i *globální operátory*, které pracují s modelem jako s celkem a typicky zjednodušují jeho topologii [2]. Následující text popisuje principy vybraných lokálních operátorů.

2.2.1 Sjednocení hrany

Operátor sjednocení hrany (*edge collapse*) poprvé publikovali Hoppe et al. [6] v roce 1993. Principem je odstranění jedné hrany z modelu a její nahrazení jedním vrcholem. Tímto zároveň dojde k odstranění dvou sousedních trojúhelníků. Inverzním operátorem je *rozdělení vrcholu* (*vertex split*).



(a) *Half-edge collapse* (b) Původní síť polygonů (c) *Full-edge collapse*

Obrázek 2.3: Demostrace operátorů sjednocení hrany. V obou případech je sjednocena zvýrazněná hrana z (b). V případě (a) je nový vrchol v_n ekvivalentní s jedním ze sjednocených vrcholů, v případě (c) jde o vrchol na zcela nové pozici.

Algoritmus

Vybraná hrana (v_a, v_b) je sjednocena do jediného vrcholu v_n. Podle pozice v_n rozeznáváme dvě varianty tohoto algoritmu:

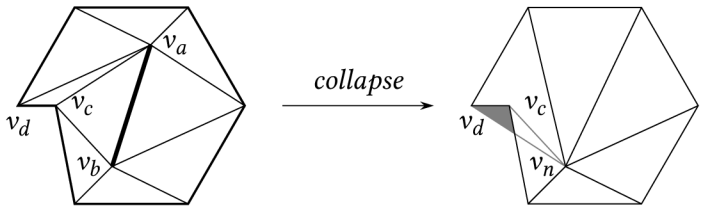
Half-edge collapse – Nový vrchol v_n je identický s jedním z původních vrcholů hrany ($v_n = v_a$ nebo $v_n = v_b$). Výsledek aplikace této varianty demonstruje obrázek 2.3a.

Full-edge collapse – Vrchol v_n je nově vytvořený, nejčastěji někde na původní hraně (typicky uprostřed mezi v_a a v_b). Tato varianta se také někdy označuje zkráceným názvem *edge collapse*. Demonstraci aplikace této varianty lze nalézt na obrázku 2.3c.

Hlavní výhodou tohoto operátoru je jednoduchost jeho implementace a rychlost provádění. Při *half-edge* sjednocení navíc není potřeba implementovat výpočet nových atributů vrcholu – použijí se atributy příslušného původního vrcholu. Při *full-edge* sjednocení je nutné pro nový vrchol atributy dopočítat, což může být složité (zvláště pokud se nový vrchol nenachází na spojnici vrcholů původních).

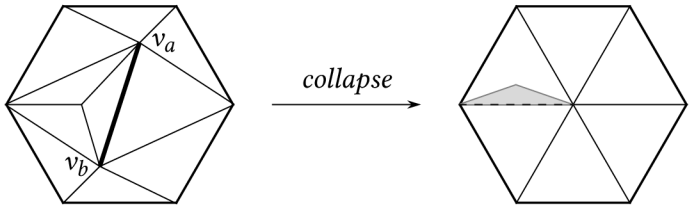
Problematické aplikace

Ačkoliv je aplikace tohoto operátoru poměrně přímočará, existují situace, ve kterých vede k poškození modelu či nežádoucím vedlejším efektům. Výskyt těchto situací je nutné ověřit před aplikací operátoru, a v případě jejich detekování tuto aplikaci přeskočit.



Obrázek 2.4: Příklad překrytí polygonů při nevhodném sjednocení hrany. Šedý trojúhelník byl touto operací otočen původně vnější stranou dovnitř.

Překrytí polygonů je nežádoucím vedlejším efektem některých sjednocení hrany [7]. Při sjednocení hrany (v_a, v_b) na obrázku 2.4 je trojúhelník (v_a, v_c, v_d) transformován na nový trojúhelník (v_n, v_c, v_d), čímž v síti polygonů vznikne „sklad“. Tento jev lze detekovat sledováním změn směru normálových vektorů ovlivněných trojúhelníků. Pokud je úhel mezi normálou trojúhelníku před a po sjednocení hrany větší než 90°, došlo by aplikací tohoto operátoru ke „skladu“ a není tedy vhodné tuto aplikaci provádět.

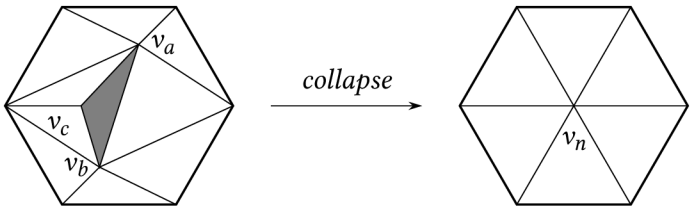


Obrázek 2.5: Příklad vzniku *non-manifold* sítě. Následkem sjednocení tučně zvýrazněné hrany je čárkovaná hrana sdílena více než dvěma trojúhelníky.

Pokud mají vrcholy sjednocované hrany alespoň 3 sousední *společné* vrcholy, dojde po jejich sjednocení ke vzniku *non-manifold polygonu* (alespoň jednoho). Příklad takového sjednocení lze nalézt na obrázku 2.5. Toto porušení topologie pak může působit problémy jak při další simplifikaci, tak při obecné práci s modelem.

2.2.2 Sjednocení trojúhelníka

Operátor sjednocení trojúhelníka (*triangle collapse*) lze považovat za zrychlení operátoru sjednocení hrany. Jeho aplikací jsou najednou odstraněny až 4 trojúhelníky, takže postup sjednocení je jak rychlejší, tak paměťově méně náročný než ekvivalentní postup realizovaný pomocí sjednocení hrany. Na druhou stranu jde o hrubší operaci, která nemusí být aplikovatelná na všechny druhy polygonových sítí. Metodu poprvé publikoval Hamann [8] v roce 1994.



Obrázek 2.6: Příklad aplikace operátoru sjednocení trojúhelníka. Šedý trojúhelník je nahrazen jediným vrcholem.

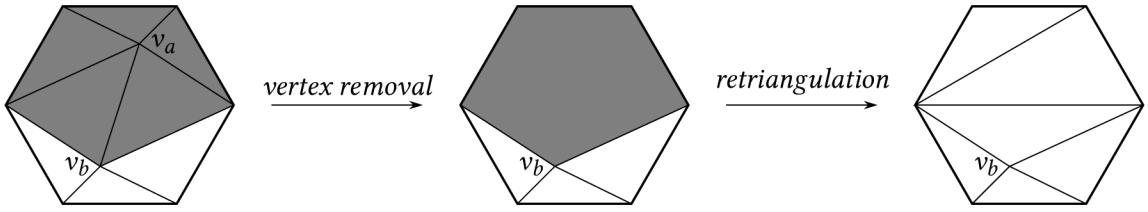
Algoritmus

Jak je ilustrováno na obrázku 2.6, vybraný trojúhelník $\Delta_x = (v_a, v_b, v_c)$ je sjednocen do jediného vrcholu v_n. Tímto jsou odstraněny až 4 trojúhelníky: jak Δ_x samotný, tak jeho přímí sousedé (trojúhelníky sdílející hranu). Množina okolních hran nového vrcholu v_n je sjednocením množin hran obsahujících vrcholy sjednoceného trojúhelníka. Nový vrchol v_n lze umístit jak na pozici některého z vrcholů v_a, v_b nebo v_c, tak na pozici zcela novou.

Za povšimnutí stojí, že tento operátor je ekvivalentní dvojitému aplikování operátoru sjednocení hrany. Z této vlastnosti plyne výše uvedené zrychlení a šetření paměti – pro ekvivalentní zjednodušení je potřeba poloviční počet aplikací operátoru.

2.2.3 Odstranění vrcholu

Tuto metodu publikovali v roce 1992 Schroeder et al. [9]. Operátor odstranění vrcholu (*vertex removal*) odstraní z modelu právě jeden vrchol. Každou aplikací jsou také odstraněny dva polygony.



Obrázek 2.7: Příklad odstranění vrcholu. Odstraněním vrcholu v_a vznikne v šedě vyznačené oblasti díra, kterou je následně potřeba zacelit.

Algoritmus

Vybraný vrchol v_a je odstraněn z polygonové sítě společně se všemi hranami, které jej obsahují. Tímto odstraněním vznikne díra, kterou je potřeba zacelit novými trojúhelníky (*retriangulation*). Celý proces ilustruje obrázek 2.7.

Zacelení této díry lze dosáhnout různými způsoby, přičemž jeden z nich je ekvivalentní s *half-edge* odstraněním hrany obsahující vrchol v_a [2]. Celkový počet možných způsobů zaplnění díry po odstranění vrcholu je dán následující rovnicí, kde $C(i)$ značí počet způsobů zaplnění pro díru s $(i+2)$ hranami:

$$C(i) = \frac{1}{i+1} \cdot \binom{2i}{i} = \frac{1}{i+1} \cdot \frac{(2i)!}{(i! \cdot (2i-i)!)} = \frac{1}{i+1} \cdot \frac{(2i)!}{i!i!} = \frac{(2i)!}{(i+1)!i!} \tag{2.1}$$

Výběr ideálního zacelení ze všech těchto možností lze považovat za problém diskrétní optimalizace, který může být poměrně náročný.

2.3 Chybové metriky

Po definici různých možností zjednodušení modelu v předchozí sekci je nyní nutné položit si další otázku: Jakým způsobem určit prvky modelu, na které bude zvolený operátor aplikován, aby byl výsledek dostatečně podobný originálu? Tento problém je typicky řešen použitím *chybové metriky* – hodnoty udávající, jak moc se upravený model liší od originálu.

V následujícím textu jsou nastíněny možné principy konstrukce takové metriky. Ve druhé části jsou pak uvedeny příklady v praxi publikovaných a používaných metrik, většinou specializovaných pro konkrétní operátory.

2.3.1 Geometrické metriky

Jednou z nejzákladnějších možných metrik je *geometrická vzdálenost* mezi body povrchu originálního a upraveného modelu. Protože zjednodušování polygonálního povrchu spočívá principiálně v redukci počtu vrcholů, mění se tím i obrysy daného modelu. Minimalizace geometrické chyby zároveň minimalizuje rozdíly v obrysech jednotlivých úrovní detailu [2].

Pro měření rozdílu mezi vrcholy postačí základní definice vzdálenosti ve 3D eukleidovském prostoru:

$$|(v_a, v_b)| = \sqrt{(v_{a_1} - v_{b_1})^2 + (v_{a_2} - v_{b_2})^2 + (v_{a_3} - v_{b_3})^2} \tag{2.2}$$

Pokud je třeba zohledňovat vzdálenost *povrchů* (například posun polygonů před a po vykonání operace), lze použít *Hausdorffovu vzdálenost*. Tato metrika přiřadí každému bodu z povrchu (množiny) A geometricky nejbližší bod povrchu B, spočítá jejich vzdálenosti, a použije tu největší z nich:

$$H(A,B) = max(h(A,B), h(B,A)) \tag{2.3a}$$

$$h(A,B) = \max_{a \in A} \min_{b \in B} |a - b| \tag{2.3b}$$

Nevýhodou této metriky může být její asymetrie: protože může platit $h(A,B) \neq h(B,A)$, je třeba zvlášť počítat obě varianty vzdálenosti. Navíc se mohou vyskytnout oblasti, ve kterých není „mapování“ bodů spojité, případně oblasti s asymetrickou asociací bodů (více bodů z A je spárováno

s jediným bodem z B, a naopak některé body z B nemají asociovaný bod z A). Díky tomu může být problematické vypočítávat hodnoty atributů pro vygenerované povrchy.

Možnou alternativou je definice vlastního bijektivního zobrazení mezi zkoumanými povrchy a použití maximální vzdálenosti mezi takto asociovanými body:

$$D(F) = \max_{a \in A} |a - F(a)| \qquad F : A \longrightarrow B \tag{2.4}$$

2.3.2 Metriky založené na atributech modelu

Kromě geometrických souřadnic lze u modelů zkoumat i další atributy, které jsou buď součástí definice modelu jako přídavná informace, nebo odvozené z jeho geometrie. Typickými atributy mohou být barva, normálový vektor či texturovací souřadnice. Na tyto atributy lze také nahlížet jako na geometrické prostory a využít jejich rozdílů pro měření chyby.

Barva

Atribut pro barvu bývá reprezentován trojicí hodnot (r, g, b), které reprezentují intenzitu červené, zelené a modré složky viditelného světla. Jedním z možných způsobů měření rozdílu mezi barvami je použít tyto složky jako souřadnice ve 3D prostoru a aplikovat na ně geometrickou vzdálenost:

$$d_{RGB} = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2} \tag{2.5}$$

Problém s tímto přístupem spočívá v tom, že lidské oko nevnímá vzdálenosti v RGB prostoru lineárně, a naměřená chyba tedy nemusí odpovídat pozorované chybě. Toto je ovšem možné řešit konverzí do jiného barevného prostoru, který je vnímán lineárně – například do CIE-LUV [10].

Normálové vektory

Ze souřadnic vrcholů každého trojúhelníka v modelu lze vypočítat normálový vektor $\vec{n}$, který udává jeho orientaci v prostoru:[1]

$$\vec{s_1} = \vec{v_1} - \vec{v_0} \qquad \vec{s_2} = \vec{v_2} - \vec{v_0} \qquad \vec{n} = \vec{s_1} \times \vec{s_2} \tag{2.6}$$

Za velikost chyby pak lze považovat úhel φ mezi normálovým vektorem trojúhelníka před a po úpravě modelu:

$$\varphi = arccos\left(\vec{n_x} \cdot \vec{n_y}\right) \tag{2.7}$$

Tato metrika je důležitá také pro detekci přeložení polygonu (obrázek 2.4 na straně 5). Pokud je při výpočtu normálových vektorů zachováno pořadí vrcholů, bude v případě přeložení velikost φ typicky větší než 90° [2].

2.3.3 Quadric Error Metric

Tuto metriku, často označovanou zkráceně jako QEM, původně publikoval Garland et al. pod názvem *Surface Simplification Using Quadric Error Metrics* v roce 1997 [11].

QEM pracuje nad operátorem sjednocení hrany[2]. Základem hodnotící funkce je vzdálenost nového (zjednodušeného) vrcholu od rovin okolních trojúhelníků (*supporting plane distance*).

[1] Jedná se o skutečné normálové vektory jednotlivých ploch, bez úprav případnými normálovými mapami.

[2] V původním článku [11] se počítá i se spojením vrcholů, které nesdílí žádnou hranu (*virtual edge collapse*), ale touto variantou se tato sekce nezaobírá.

Na jednotlivé trojúhelníky modelu lze nahlížet jako na výřezy rovin ve 3D prostoru. Takovou rovinu lze vyjádřit parametrickým vektorem $p = (a, b, c, d)$. Parametry a, b a c jsou v tomto případě složky normálového vektoru dané roviny, d značí vzdálenost od počátku (bodu $[0, 0, 0]$). Pro jejich hodnoty platí:

$$ax + by + cz = 0 \tag{2.8a}$$

$$a^2 + b^2 + c^2 = 1 \tag{2.8b}$$

Chybová funkce $\Delta(v)$ je pak definována jako součet čtverců vzdáleností vrcholu k trojúhelníkům, které jej obklopují ($planes(v)$):

$$\Delta(v) = \Delta\left(\left[v_x, v_y, v_z, 1\right]^T\right) = \sum_{p \in planes(v)} \left(p^T v\right)^2 \tag{2.9}$$

Pokud bychom používali chybovou rovnici v tomto tvaru, bylo by nutné pro každý nový vrchol zkonstruovat novou množinu okolních trojúhelníků podle vztahu $planes(v_n) = planes(v_a) \cup planes(v_b)$, což může vyžadovat větší množství paměti.

S použitím několika algebraických úprav lze rovnici (2.9) přepsat do tvaru, který umožňuje vyjádřit velikost chyby ve vrcholu v pomocí jediné matice Q_v:

$$Q_p = pp^T = \begin{bmatrix} a^2 & ab & ac & ad \\ ba & b^2 & bc & bd \\ ca & cb & c^2 & cd \\ da & db & dc & d^2 \end{bmatrix} \tag{2.10a}$$

$$Q_v = \sum_{p \in planes(v)} Q_p \tag{2.10b}$$

$$\Delta(v) = \sum_{p \in planes(v)} (v^T p)(p^T v) = \sum_{p \in planes(v)} v^T (pp^T) v$$

$$\Delta(v) = v^T Q_v v \tag{2.10c}$$

Místo ukládání sady trojúhelníků je pro každý vrchol uchovávána pouze jediná symetrická matice 4×4 (10 unikátních desetinných čísel), která je pro sjednocený vrchol vypočítána jediným maticovým součtem:

$$Q_{v_n} = Q_{v_a} + Q_{v_b} \tag{2.11}$$

Kromě výpočtu samotné velikosti chyby dokáže metrika QEM nalézt i ideální pozici vrcholu, do kterého by měla být hrana sjednocena, tak, aby byla naměřená chyba minimální. Protože je chybová funkce $\Delta(v)$ kvadratická, nalezení jejího minima je lineární problém, a lze jej uskutečnit výpočtem rovnice $\frac{\delta\Delta}{\delta x} = \frac{\delta\Delta}{\delta y} = \frac{\delta\Delta}{\delta z} = 0$. Řešení této rovnice je ekvivalentní [11] s řešením následující rovnice, kde hodnoty q_{ij} odpovídají příslušným hodnotám matice Q_v:

$$v = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.12}$$

2.4 Výběr prvků modelu pro zjednodušení

Dalším stavebním kamenem zjednodušovacího algoritmu je způsob výběru prvků, které mají být z originálního modelu odstraněny. V ideálním případě by měly být prvky vybrány takovým způsobem, aby chyba naměřená mezi původním a zjednodušeným modelem byla minimální. Tato sekce ilustruje některé z používaných přístupů k řešení tohoto problému.

2.4.1 Neoptimalizovaný výběr

Pravděpodobně nejjednodušším způsobem výběru prvků k odstranění je výběr v podstatě náhodný. Například algoritmus, který publikoval Rossignac et al. v roce 1993 [12] rozdělí zjednodušovaný model uniformní 3D mřížkou a poté sjednotí všechny vrcholy v rámci buňky do jednoho. Pokud je buňka dostatečně malá, libovolné pořadí operací v rámci této buňky vyprodukuje zhruba stejně velkou chybu oproti původnímu modelu. Díky tomu může být postup výběru, ilustrovaný algoritmem 2.1, poměrně jednoduchý.

```
1: for každou úroveň zjednodušení do
2:     for každou možnou nezávislou operaci op do
3:         Apply(op)
```

Algoritmus 2.1: Zjednodušování modelu bez optimalizací

Název *neoptimalizovaný* se v tomto případě vztahuje pouze na způsob výběru prvků. Samotná aplikace operátoru typicky velikost chyby optimalizuje: vybranou chybovou metriku lze například použít pro výběr nejlepší pozice nového vrcholu u *full-edge* sjednocení hrany nebo způsob retriangulace u odstranění vrcholu.

2.4.2 Hladový a líný výběr

Hladový výběr pracuje ve dvou fázích. Nejprve pomocí vybrané metriky ohodnotí všechny potenciálně zjednodušitelné prvky modelu a na základě tohoto ohodnocení je vloží do prioritní fronty. Ve druhé fázi pak vždy z této fronty vyjme prvek s nejmenší chybou a aplikuje na něj vybraný operátor. Aplikace operátoru ovšem může změnit ohodnocení sousedních prvků, takže je nutné po každé aplikaci všechny sousedící prvky vyjmout z fronty, přepočítat jejich ohodnocení a znovu vložit do fronty na správnou pozici.

Nevýhodou tohoto přístupu může být několikanásobné přepočítání ceny mnohých prvků dlouho před jejich případnou aplikací na model. Tento problém řeší *líný výběr* (algoritmus 2.2), který publikoval Cohen [13] v roce 1998. Místo přepočítání ceny je po aplikaci operace sousedním prvkům pouze nastaven příznak neaktuální ceny (*dirty flag*, řádek 10). Při vyjmutí z fronty jsou pak zpracovávány pouze prvky s aktuální cenou (řádky 8-10). Prvky s neaktuální cenou jsou oceněny a vloženy zpět do fronty (řádky 12-14).

Require: Prioritní fronta Q, cenová funkce CalculateCost.

```
1:  for každou možnou operaci op do
2:      CalculateCost(op)
3:      op.dirty ← False
4:      Q.insert(op)
5:  while Q není prázdná do
6:      op ← Q.extractMin()
7:      if op.dirty == False then
8:          Apply(op)
9:          for každou sousední operaci op_n do
10:             op_n.dirty ← True
11:     else
12:         CalculateCost(op)
13:         op.dirty ← False
14:         Q.insert(op)
```

Algoritmus 2.2: Výběr zjednodušujících operací s líným přepočítáním ceny

3 Návrh rozšíření

Tato kapitola se zabývá návrhem rozšíření pro knihovnu GPUEngine poskytujícím prostředky pro generování zjednodušených úrovní detailu. Nejprve je stručně popsán výchozí stav této knihovny společně se souvisejícími prostředky a omezeními. Následuje popis návrhu datových struktur pro reprezentaci 3D grafu, která je pro implementaci příslušných algoritmů téměř nezbytná. V další části je pak navržena podoba jednotlivých prvků zjednodušujícího procesu a jejich vzájemné komunikace.

3.1 GPUEngine

GPUEngine[1] je souhrnný název pro sadu nástrojů pro zpracování a vykreslování 3D scén. Jde o C++ knihovnu poskytující mimo jiné objektové rozhraní pro práci s OpenGL, implementaci grafu scény, nástroje pro práci s kamerou a podobně. Jako volitelné rozšíření pak poskytuje rozhraní k populárním externím knihovnám použitelným pro vykreslování scén – SDL2 a Qt pro grafická rozhraní, Assimp[2] pro načítání a ukládání modelů v rozličných formátech a další. Projekt je stále v aktivním vývoji a ještě neobsahuje všechnu plánovanou funkcionalitu.

Systém volitelných rozšíření je jednou ze základních vlastností tohoto návrhu. Aby nebyl uživatel GPUEngine nucen do svého projektu zahrnout práci s úrovněmi detailu, i když ji nijak nevyužije, je tento návrh koncipován jako nové rozšíření. Díky tomu bude výstupem překladu zdrojových kódů samostatný objektový soubor, který lze zahrnout či vynechat z projektu nezávisle na jiných (nevyžadovaných) částech GPUEngine.

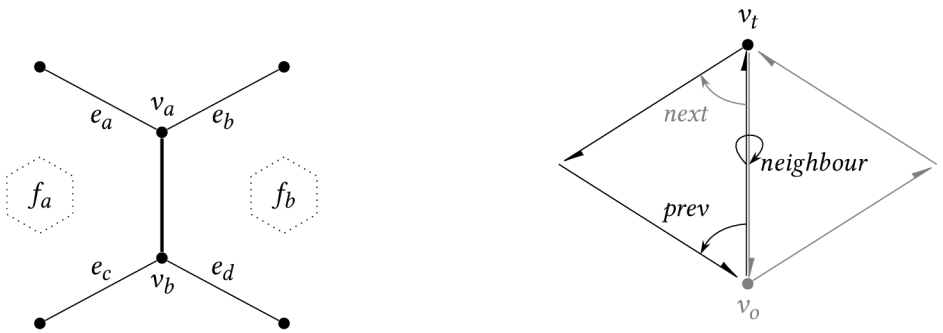
Důležitým omezením návrhu je minimalizace externích závislostí. V současné době je jedinou povinnou závislostí GPUEngine knihovna glm[3]. Přáním autorů je zachovat tento stav, aby uživatel nebyl nucen instalovat množství dalších knihoven jen pro základní použití GPUEngine. V případě volitelného rozšíření lze uvažovat o přidání nepovinné závislosti, ale je upřednostňováno řešení bez nich.

3.2 Struktura rozšíření

Navržená struktura rozšíření je inspirována existující implementací simplifikace polygonů v knihovně CGAL [14]. Hlavní ideou je implementovat jednotlivé části zjednodušovacího procesu jako samostatné jednotky. Kompletní řešení pak vznikne vhodným zkombinováním těchto jednotek. Tento druh návrhu (*policy-based design* [15]) umožňuje vytvářet velké množství kombinačních řešení, aniž by docházelo k nežádoucímu komplikování kódu.

Podstatnou překážkou implementace těchto prvků v GPUEngine je způsob reprezentace modelu v rámci grafu scény. Jednotlivé modely a jejich atributy jsou interně uchovávány v podobě velmi

[1] *Rendering-FIT/GPUEngine*. Dostupné z: `https://github.com/Rendering-FIT/GPUEngine` [cit. 2018-01-11].
[2] *Assimp: Open Asset Import Library*. Dostupné z: `http://assimp.org/` [cit. 2018-01-11].
[3] *OpenGL Mathematics*. Dostupné z: `https://glm.g-truc.net` [cit. 2018-01-12].



(a) *Winged-edge data structure* (b) *Half-edge data structure*

Obrázek 3.1: Srovnání *winged-edge* a *half-edge* datových struktur. Zatímco u *winged-edge* ukládá každá hrana referenci na všechny vyobrazené prvky (vrcholy v_x, hrany e_x a plochy f_x), *half-edge* hrana uchovává pouze reference na cílový vrchol v_t, předchozí hranu *prev* a sousední hranu *neighbour* (na obrázku černě). Reference na šedě vyobrazené prvky (hranu *next* a vrchol v_o) lze získat výpočtem s konstantní složitostí.

podobné bufferům v knihovně OpenGL: data pro každý druh atributu jsou uchovávána v nestrukturovaném poli, a vnitřní struktura je popsána pomocí přídavných parametrů tohoto pole[4]. Data v tomto formátu lze jednoduše předat grafické kartě k vykreslení, ale pro nelineární zpracování a modifikaci nejsou příliš vhodná.

Kromě algoritmů samotných je tedy nutno implementovat i takovou abstrakci nad daty modelu, která umožní nahlížet na tyto data jako na popis 3D grafu. Tato abstrakce pak musí umožnit jednoduchou oboustrannou konverzi mezi původní a vlastní reprezentací.

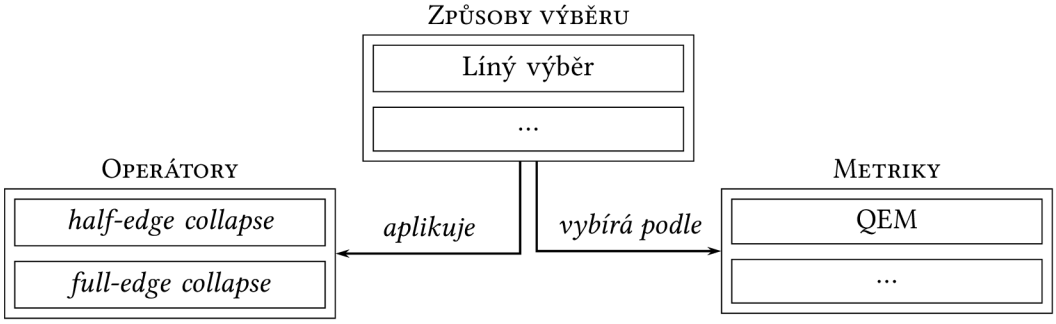
3.3 Grafová abstrakce polygonové sítě

Způsobů reprezentace dat modelu v paměti je celá řada [16]. Základním požadavkem na datovou strukturu vhodnou pro LoD algoritmy je snadný přístup k okolí libovolného prvku grafu. Tento požadavek lze splnit například použitím *winged-edge data structure*, která je ilustrována na obrázku 3.1a.

Abstrakce grafu postavená nad touto strukturou uchovává celkem 3 druhy prvků: uzly (*nodes*) představující vrcholy, hrany (*edges*) spojující jednotlivé vrcholy a plochy (*faces*) ohraničené těmito hranami. Každý uzel uchovává (kromě dalších atributů) referenci na jednu ze svých hran. Podobně každá plocha uchovává referenci na jednu z hran, které ji obklopují. Naproti tomu každá hrana uchovává dvě reference na své hraniční vrcholy, dvě reference na sousedící plochy, a 4 reference na „křídla“ — hrany přímo sousedící v rámci některé ze sousedících ploch.

Optimalizací této datové struktury je *half-edge data structure*, jejíž ilustraci lze nalézt na obrázku 3.1b. Tato struktura je specializována na reprezentaci pouze trojúhelníkových sítí. Uzly jsou reprezentovány stejným způsobem jako u *winged-edge*. Naproti tomu hrany jsou reprezentovány takzvanými „půlhranami“. Každá „půlhrana“ uchovává referenci na cílový uzel, předchozí hranu v rámci trojúhelníka a referenci na sousední „půlhranu“ s opačnou orientací. Zbývající reference, ja-

[4] *Stride, offset* a podobné.



Obrázek 3.2: Diagram vztahů mezi navrženými prvky. Kompletní algoritmus pro generování LoD je uživatelem sestaven z předpřipravených částí, zejména způsobu výběru prvků, metriky pro jejich výběr a operátoru.

ko je například výchozí uzel nebo následující hrana v trojúhelníku lze získat výpočtem s konstantní složitostí[5] [17].

Výhodou *half-edge* struktury je její schopnost reprezentovat i *non-manifold* síť. V takovém případě je reference na sousední hranu nahrazena vhodným indikátorem porušení sítě. Tento indikátor je vhodné implementovat takovým způsobem, aby bylo možné rozlišit různé druhy porušení sítě a případně na ně reagovat.

3.4 Prvky zjednodušujícího procesu

Pro modulární návrh, zmíněný v první sekci této kapitoly, je důležitá možnost libovolně vyměňovat části implementace za jiné části stejného druhu. Je tedy nutné definovat jak rozhraní jednotlivých komponent, tak způsob předávání informací mezi těmito komponentami. Analýzou přístupů popsaných v kapitole 2 lze vypozorovat tyto skutečnosti (ilustrované na diagramu 3.2):

1. Celý proces simplifikace je řízen výběrem prvků k odstranění.
2. Tento výběr je ovlivňován zvolenou metrikou, která pracuje se stejným druhem prvků, jako řídící algoritmus.
3. Jednotlivé prvky jsou pak odstraňovány pomocí aplikace zvoleného operátoru.

Algoritmus výběru prvků jako jediná část pracuje s celou sítí, ale pouze nepřímo. Jeho základními vstupy jsou tedy síť samotná, hodnotící metrika a operátor. Je ovšem vhodné poskytnou možnost řídit způsob zastavení procesu zjednodušení, typicky pomocí horní hranice ceny aplikace operátoru. Výstupem této komponenty je zjednodušená varianta modelu.

Vstupem jednotlivých metrik jsou prvky modelu, které budou následně odstraňovány. Pokud každý prvek nese dostatečné množství informací o svém okolí, nepotřebuje metrika typicky přístup k celé síti. Minimálním výstupem metriky je cena odstranění hodnoceného prvku ze sítě. Protože však některé metriky mohou poskytovat i další informace (například optimální pozici nového uzlu při použití *full-edge collapse*), měla by je implementace předávání informací mezi metrikou a operátorem zachovat.

[5] Například výchozí uzel je ekvivalentní cílovému uzlu předcházející hrany, následující hrana je přecházející hranou předešlé hrany.

Různé metriky pracují nad různými druhy prvků modelu. Při jejich implementaci je vhodné signalizovat, které druhy prvků jsou podporovány, a upozornit uživatele, pokud se pokusí použít nedefinovanou kombinaci.

Operátory přijímají jako svůj vstup prvek modelu, který mají odstranit. Protože jde typicky o destruktivní operaci, která nemusí být omezena na okolí jednoduše dosažitelné z odstraňovaného prvku, je vhodné operátoru poskytnout i přístup k celé síti. Co se týče přídavných informací volitelně poskytovaných metrikou, v ideálním případě si implementace operátoru vystačí pouze s určením prvku ke zjednodušení, a přídavné informace využije, pokud jsou k dispozici.

Výstupem operátoru je označení prvků, které byly aplikací ovlivněny, ale nikoliv odstraněny. Tyto prvky je typicky nutné znovu ohodnotit, protože aplikace operátoru změnila jejich okolí, a tedy potenciálně i cenu.

Libovolná implementace operátoru by také měla zajistit, že libovolná aplikace nepoškodí stávající síť (viz Problematické aplikace na straně 5). V případě detekce možného poškození musí implementace tuto operaci přeskočit bez aplikace jakýchkoli změn.

Při návrhu programové knihovny je také vhodné uvážit předpokládané způsoby jejího použití a učinit jej co uživatelsky nejpříjemnější. Na tuto přívětivost lze nahlížet ze dvou možných pohledů: pohled zběžného uživatele a pohled experta. Pro zběžného uživatele je důležitá možnost dosáhnout požadovaných výsledků bez hlubší znalosti nebo nutnosti specifikace detailů. Uživatel-expert by naopak neměl být návrhem omezován a měl by být schopen všechny požadované detaily specifikovat. Z těchto důvodů počítá návrh s implementací pomocných funkcí, které by měly poskytnout co nejjednodušší přístup k funkcionalitě celé knihovny, nejlépe ve formě jediné funkce. Vhodným použitím výchozích hodnot „expertních“ parametrů by pak měl být tuto funkci schopen použít jak zběžný uživatel, tak expert se specifickými požadavky.

4 Implementace

Implementační kapitola se zabývá popisem implementovaných struktur a algoritmů se zaměřením na jejich předpokládané použití. Jsou zde rozebrány jak prvky použité pro abstrakci grafu, tak jednotlivé komponenty zjednodušujícího procesu. U komponent zjednodušujícího procesu je zároveň uveden popis požadovaného rozhraní, který nelze rozumně vyjádřit přímo v rámci zdrojového kódu, ale je klíčový pro případná budoucí rozšíření této implementace.

V další části kapitoly je popsána demonstrační aplikace spojená s implementovanou knihovnou včetně stručného popisu jejího ovládání. V závěru je pak navrženo několik možných směrů dalšího vývoje a rozšiřování této knihovny.

4.1 Abstrakce grafu

Pro realizaci grafové abstrakce polygonové sítě byla zvolena *half-edge* struktura. Hlavními důvody byly nižší paměťové nároky a potenciální podpora *non-manifold* těles.

Dalším nutným rozhodnutím, které bylo třeba učinit, byla volba podoby referencí mezi prvky grafu. V původním článku [17] je počítáno s jejich realizací pomocí buď obecných ukazatelů do paměti, nebo indexů v rámci pole všech prvků daného typu. Obě varianty mají své pro a proti — při použití indexů lze některé operace nahradit aritmetickými operacemi[1] a lze vytvořit hlubokou kopii grafu, aniž by bylo třeba upravovat vnitřní reference. Na druhou stranu například odstranění hrany z grafu zneplatní reference jak na hranu samotnou, tak na všechny, které se nacházejí v poli za ní. Protože odstraňování různých prvků z grafu je těžištěm této práce, byly pro reprezentaci referencí nakonec zvoleny ukazatele.

Třída `lod::graph::Mesh` představuje celý graf. Její implementace se velmi podobá matematické definici grafu — jde o množinu vrcholů a hran mezi nimi. Pro reprezentaci množiny byla použita implementace ze standardní knihovny jazyka C++ `std::unordered_set`. Tato reprezentace přináší dvě klíčové vlastnosti: automaticky odstraňuje duplikáty (prvek již v množině obsazený není vložen znovu) a umožňuje vyhledat či otestovat přítomnost prvku v konstantním čase[2]. Časová složitost těchto kontejnerů je činí ideálními pro zjednodušující algoritmy, které potřebují přistupovat a modifikovat jednotlivé prvky v předem neurčeném pořadí.

S použitím `std::unordered_set` se ovšem pojí požadavek na uchovávané datové typy — musí pro ně být definována *hash* funkce. Tato implementace řeší tento problém dvěma způsoby. Pokud je instance dané třídy jasně identifikovatelná nějakou kombinací svých dat, je pro ni nadefinována i specializace standardního funktoru `std::hash`. V opačném případě je využito existující definice tohoto funktoru pro ukazatele, a v množině jsou pak obsaženy vlastnící (*owning*) ukazatele na daný prvek[3].

Kromě ukládání dat grafu poskytuje třída `lod::graph::Mesh` také prostředky pro konvertování těchto dat mezi grafovou reprezentací a reprezentací v rámci grafu scény (`ge::sg::Mesh`). Konverzi

[1] Zdrojový článek uvádí jako příklad výpočet indexů předcházející a následující hrany z indexu hrany stávající.

[2] Respektive v průměru v konstantním, v nejhorším případě v lineárním čase [18].

[3] Prvek je tedy jednoznačně určen svojí pozicí v paměti.

z grafu scény zajišťuje příslušný konstruktor, konverze nazpět je pak implementována v konverzním operátoru (`operator ge::sg::Mesh()`).

Pro reprezentaci uzlů grafu slouží třída `lod::graph::Node`. Její implementace odpovídá popisu uzlu v předchozí kapitole — jde o pozici v prostoru s přidaným ukazatelem na nějakou hranu z tohoto uzlu vycházející. Protože je uzel jednoznačně určen svojí pozicí, je tato pozice použita i pro definici *hash* funkce[4], a uzly jsou tedy v množině grafu ukládány přímo.

Druhou datovou strukturou používanou v reprezentaci grafu je `lod::graph::DirectedEdge`. Jejím základem je v předchozí kapitole navržená podoba reprezentace „půlhrany“. Zde se ale objevuje problém ukazatele na hranu v sousedním trojúhelníku. Tento ukazatel totiž musí reprezentovat několik vzájemně výlučných stavů:

1. Pokud se hrana nenachází na okraji sítě, tento ukazatel uchovává referenci na existující sousední hranu.
2. Pokud je hrana na okraji sítě, sousední hrana neexistuje a ukazatel je tudíž prázdný.
3. Při porušení *manifold* vlastností indikuje tento ukazatel druh chyby.

První dva body lze reprezentovat běžným ukazatelem, který je nulový na okraji sítě a nenulový na jiných místech. Reprezentaci chyby už ale pouze ukazatelem implementovat nelze.

Z těchto důvodů byla do projektu zahrnuta i jedno-hlavičková knihovna VARIANT-LITE[5]. Tato knihovna poskytuje typově bezpečnou alternativu pro *union* typy. To v praxi znamená zejména vyvolání výjimky, pokud se algoritmus pokusí přistoupit k nevalidní variantě obsahu. S její pomocí je tedy ukazatel na sousední hranu implementován jako „*union*“ mezi ukazatelem samotným a výčtovým typem indikujícím *non-manifold* porušení sítě.

Specifikem typu `DirectedEdge` je způsob jejího vytváření. Nejvíce ukazatelů v rámci grafové abstrakce je právě na instance této třídy, a v mnohých případech se ukázalo být výhodné mít tyto reference „samo-zneplatňující“ — tedy pokud je odstraněna hrana, všechny existující reference na tuto hranu jsou automaticky zneplatněny, aniž by bylo nutné mít ke všem těmto referencím přístup a zneplatnit je explicitně. Tuto vlastnost má typ `std::weak_ptr` ze standardní knihovny, který je v konečné implementaci používán pro všechny nevlastnící ukazatele na `DirectedEdge`.

S jeho použitím se ovšem pojí nutnost vytvářet hranu vždy na haldě s pomocí typu `std::shared_ptr`. Aby nedocházelo k omylům programátora při používání (či opomenutí) tohoto typu, je konstruktor třídy `DirectedEdge` definován jako *protected*. Následkem této definice je programátor nucen použít tovární funkci (*factory function*) `DirectedEdge::make()`, která hranu vytváří vždy na haldě. Ukazatele jsou pak ukládány i ve třídě `Mesh`, protože konstrukce *hash* funkce pro instance třídy `DirectedEdge` je problematická[6].

Existují ale situace, kde by se definice *hash* funkce pro hranu velmi hodila — například při konverzích polygonové sítě mezi různými reprezentacemi. Vzhledem ke způsobu uložení dat je při konverzi z `ge::sg::Mesh` nutné vkládat do grafu samostatné trojúhelníky, s odděleným napojením na již existující sousedy. Za tímto účelem byl nadefinován pomocný typ `lod::graph::UndirectedEdge`. Tento typ je obalovým typem (*wrapper type*) pro `DirectedEdge`, který je jednoznačně určen pomocí

[4] S využitím existujících definic *hash* funkcí v rozšíření knihovny GLM.

[5] *martinmoene/variant-lite*. Dostupné z: `https://github.com/martinmoene/variant-lite` [cit. 2018-05-12].

[6] Jedním z požadavků na *hash* funkci vhodnou pro `std::unordered_set` je neměnnost její hodnoty po dobu života prvku v množině. U reference na cílový uzel a sousední hranu se předpokládají změny během zjednodušování sítě, což by vyžadovalo vyjmutí a vložení prvku při každé změně. Navíc hodnota `std::weak_ptr` referencí se může měnit i bez zásahu do hrany samotné.

obou vrcholů hrany[7]. Při konverzi sítě mezi reprezentacemi, kdy je implicitně předpokládáno, že nedochází ke změnám či odebírání existujících hran, pak množina instancí `UndirectedEdge` slouží jako *cache* pro rychlé vyhledání sousední „půlhrany“ (stejný *hash*).

4.2 Prvky simplifikace

Implementace jednotlivých prvků je založena na již zmíněném *policy-based* návrhu. Jednotlivé komponenty jsou představované šablonovými typy (*template types*). S těmito typy se v jazyce C++ pojí dvě vlastnosti podstatné pro tuto implementaci:

1. Pokud není určitá část šablony nikde v kódu použita, není odpovídající kód kompilátorem generován. V takovém případě ani nemusí být k dispozici definice této varianty.
2. Pro speciální případy lze kromě obecné definice šablony implementovat i upravenou specializaci.

Ze spojení těchto dvou faktů vyplývá, že pokud jsou v kódu využívané pouze specializace šablony, není nutné vůbec definovat obecnou podobu. Díky tomu je možné deklarovat například šablonu pro metriku nad obecným prvkem, a implementovat ji pouze pro podporované prvky. Pokud se pak uživatel pokusí použít tuto metriku nad nepodporovaným prvkem, program nepůjde sestavit.

Takto použité šablony v podstatě vytváří novou úroveň programového rozhraní (API). Toto API bohužel není možné „definovat“ žádnou syntaktickou konstrukcí, jakou jsou například abstraktní bázové třídy pro definici rozhraní odvozených tříd. Je tedy nutné vycházet z podoby již existujících šablon, způsobu jejich použití a z dokumentace. Z tohoto důvodu jsou v následujících podsekcích popsány požadavky a očekávaná podoba šablon pro jednotlivé prvky simplifikace.

S definicí jednotlivých šablon souvisí i způsob selekce správných variant. Protože každá komponenta může mít obecně 1 až n variant (podle podporovaných prvků a způsobu práce s nimi), je potřeba poskytnout možnost explicitního výběru druhu prvku. Tato selekce je implementována pomocí takzvaných *tag typů*.

Pojmem *tag typ* se označuje prázdný[8] typ (typicky `struct`), jehož účelem je umožnit explicitní řízení výběru správné varianty algoritmu (přetížené funkce, šablonové specializace a podobně). Ačkoliv tento typ neuchovává žádná data, lze v jeho rámci definovat například typové aliasy.

V současné podobě knihovny jsou definovány tři takové typy (všechny ve jmenném prostoru `lod::operation`): `NodeTag`, `HalfEdgeTag` a `FullEdgeTag`. `NodeTag` indikuje práci s uzly, `HalfEdgeTag` a `FullEdgeTag` pak práci nad hranami. Poslední dva zmíněné také napovídají, který druh sjednocení hrany (*half-* nebo *full-edge collapse*) bude používán[9].

4.2.1 Předávání informací mezi komponentami

Kromě rozhraní na úrovni šablon je třeba definovat i obecnou podobu informací předávaných mezi jednotlivými komponentami. Pro tento účel slouží v této implementaci typy `lod::operation::Simple` a jeho rozšíření `lod::operation::VertexPlacement`.

Typ `operation::Simple` je nejobecnějším typem pro předávání dat mezi metrikou, operátorem a řídícím algoritmem. Instance tohoto typu nesou ukazatel na zpracovávaný (hodnocený, zjednodušovaný) prvek, a cenu provedení této operace. Nad tímto typem jsou také definované relační

[7] *Hash* funkce je konstruována jako kombinace *hash* hodnot obou vrcholů.
[8] Prázdný ve smyslu neuchovávající žádná data.
[9] Samozřejmě jen v případě výběru tohoto operátoru uživatelem.

operátory, řízené primárně cenou operace, které umožňují použít jeho instance v prioritní frontě, vybírat minimum či maximum a podobně.

Zajímavým implementačním detailem je ukazatel na zpracovávaný prvek, realizovaný dedikovaným typem `lod::operation::ElementPointer`. Účelem tohoto typu je skrýt rozdíly mezi prvky uchovávanými v grafu přímo a prvky spravovanými pomocí `std::shared_ptr`. Jde opět o šablonu s vhodnou specializací, která ukládá buď obyčejný ukazatel, nebo `std::weak_ptr`. Jeho rozhraní je ale v obou případech syntakticky a sémanticky stejné. Díky tomu lze v jednotlivých komponentách zacházet s tímto ukazatelem bez nutnosti řešit syntaktické rozdíly mezi běžnými ukazateli a `std::shared_ptr`.

Typ `VertexPlacement` je odvozen od typu `Simple` a představuje příklad rozšíření komunikace o další informace. Teto prvek je generován metrikou QEM, která dokáže pro sjednocení hrany vypočítat optimální pozici nového vrcholu. Při použití pouze typu `Simple` by byla tato bonusová informace zahozena a operátor by musel používat nějakou heuristiku pro určení nové pozice. Pro podobné případy je tedy vhodné definovat dedikované komunikační typy obohacené o další informace. Důležité je, aby byly děděním odvozeny od `Simple` — díky typovému polymorfismu je pak lze použít i s implementacemi, které pro přidané informace nemají použití.

4.2.2 Metriky

Požadavky kladené na šablonu metriky jsou dva: šablona musí být parametrizovatelná právě jedním tag typem a výsledkem této parametrizace musí být třída popisující funkční objekt (funktor). V případě, že pro konkrétní tag typ neexistuje implementace, výsledek parametrizace by neměl být definován, což vede k selhání selekce šablony a chybě při kompilaci.

Poměrně přirozenou formou implementace metriky je funkce. Implementace konkrétní metriky však může těžit i z informací uchovávaných mezi jednotlivými hodnoceními — například u metriky QEM je potřeba uchovávat existující hodnocení uzlů. Z tohoto důvodu je požadována implementace ve formě funkčního objektu (funktoru) — u jednoduchých metrik jde jen o drobnou změnu podoby implementace[10] a u komplexnějších přináší možnost perzistentních informací.

S perzistentními informacemi souvisí životnost jednotlivých instancí metriky. Předpokládá se, že tyto instance bude pro svoji potřebu vytvářet řídící algoritmus, a nebudou uživatelem používány přímo. Je tedy možné předpokládat životnost instance (a tedy i perzistentních informací) v rámci kompletního procesu zjednodušení jednoho modelu.

Od implementace metody `operator()` libovolné metriky je pak vyžadováno, aby se chovala podobně jako výše zmíněná funkce. Tato metoda musí přijímat jako svůj jediný parametr konstantní referenci na příslušný prvek grafu, a vracet ohodnocení daného prvku společně s případnými dalšími informacemi ve formě některého z „komunikačních“ typů (viz výše).

QEM

Za referenční implementaci metriky v knihovně lze považovat šablonu `lod::metric::QEM`, která implementuje stejnojmennou metriku. Jak již bylo popsáno v kapitole Teorie, tato metrika ohodnocuje jednotlivé hrany. V implementaci je využíváno výhod funktoru — hodnotící matice pro libovolný uzel je vypočítána pouze jednou a poté uložena do perzistentní cache.

Za povšimnutí stojí rozdíl mezi implementacemi této metriky při použití `HalfEdgeTag` a `FullEdgeTag`. Při použití `FullEdgeTag` je metrika implementována přesně tak, jak je popsáno v kapitole Teorie —

[10] Implementace formou metody `operator()` místo samostatné funkce.

ze stávajících kvadrik pro vrcholy hrany se vypočte hodnotící matice nového vrcholu, a na jejím základě pak ideální pozice tohoto vrcholu a velikost chyby. Při použití `HalfEdgeTag` se počítá s operátorem *half-edge* sjednocení hrany, kde výpočet ideální pozice nového vrcholu nemá smysl. V tomto případě je pouze vypočtena velikost chyby při sjednocení hrany do jejího cílového vrcholu.

4.2.3 Operátory

Požadavky kladené na šablony operátorů jsou podobné jako u metrik. Opět musí jít o šablonu funkčního objektu parametrizovatelnou právě jedním tag typem. Stejně jako u metrik neexistující specializace pro konkrétní tag typ značí, že daný druh operace je operátorem nepodporován.

Od metrik se definice operátorů liší v požadované podobě metody `operator()`. Tato metoda musí přijímat referenci na grafovou reprezentaci a referenci na komunikační objekt obsahující informace pro danou operaci. Implementace této metody je pak zodpovědná za aplikaci příslušné operace na předaný graf a jeho uvedení do konzistentního stavu.

Pokud je před aplikací operátoru potřeba provést kontrolu validity dané operace, je za ni také zodpovědná metoda `operator()`. Tyto kontroly by měly být nemodifikující — pokud nelze operátor validně aplikovat, graf nesmí být nijak změněn.

Požadovanou návratovou hodnotou aplikace operátorů je kontejner obsahující ukazatele na prvky, které byly touto aplikací změněny, ale nikoliv odstraněny ze sítě — tedy hrany se změněnými koncovými uzly, v prostoru posunuté uzly a podobně[11]. Tyto ukazatele signalizují řídícímu algoritmu potenciální nepřesnosti ve stávajícím hodnocení. Pokud nebyl graf změněn vůbec, je očekávanou návratovou hodnotou prázdný kontejner.

Sjednocení hrany

V přidružené knihovně jsou implementovány obě varianty operátoru sjednocení hrany. Obě varianty jsou realizovány šablonou `lod::oper::EdgeCollapse`, požadována varianta je pak zvolena na základě použitého tag typu.

Obě varianty implementují kontroly aplikace detekující potenciální přeložení polygonů a vznik *non-manifold* hran ve stávající síti, které byly popsány v kapitole 2. Kromě nich implementují také ověřování aplikace na okrajích sítě. Protože modely se v praxi mohou skládat z několika samostatných, ale navazujících polygonových sítí, změna vrcholů nebo hran na okrajích sítě může vést k porušení návaznosti mezi sítěmi. Kontrola aplikace na okrajích detekuje právě potenciální změnu (posun) okrajových hran a způsobí přeskočení těch aplikací, které by takovou změnu způsobily.

4.2.4 Řídící algoritmy

Z pohledu modulárního návrhu jsou řídící algoritmy spojovacím prvkem celé knihovny. Díky tomu jsou nároky kladené na podobu jejich implementace podstatně volnější než u metrik či operátorů. Konkrétní parametry a vstupy jsou ale zcela závislé na vnitřní podobě algoritmu a uvážení programátora.

Konceptuálně je předpokládáno, že algoritmus bude opět realizován šablonou funkce či funkčního objektu. Tato šablona by pak měla jako parametry přijímat nějakou kombinaci šablon pro metriky a operátory společně s jejich parametry (typicky tedy tag typ). Výsledkem vytvoření instance této šablony by pak měla být funkce přijímající originál polygonové sítě a produkující zjednodušenou variantu či varianty této sítě.

[11] Je vyžadováno hlášení změny pouze u prvků, které zpracovává daný operátor — tedy například u *edge collapse* operátoru postačí indikovat změněné hrany.

Líný výběr

Konkrétním příkladem implementace algoritmu v přidružené knihovně je líný výběr. Tento algoritmus je opět implementován formou šablony funkčního objektu. Jeho parametry jsou tag typ pro výběr operace, šablona metriky a šablona operátoru. Instance těchto šablon jsou pak algoritmem využívány pro hodnocení prvků a modifikaci grafu.

Základním způsobem použití tohoto algoritmu je vygenerování jedné zjednodušené varianty z originální polygonové sítě. V tomto případě přijímá algoritmus kromě vstupní sítě i číselnou hodnotu reprezentující maximální přijatelnou cenu aplikace operátoru. Je vhodné poznamenat, že konkrétní hodnotu tohoto prahu je nutno přizpůsobit použité metrice — algoritmus samotný ji nijak neinterpretuje, pouze ji srovnává s cenou aktuálně zpracovávaného prvku.

V případě generování více zjednodušených variant tímto algoritmem by bylo poměrně neefektivní začínat pro každou variantu od originálního modelu. Pro tento případ jsou definovány přetížené varianty metody `operator()`, které přebírají seřazený seznam prahových hodnot a postupně generují příslušné zjednodušené varianty. Pokaždé, když je přesažena hodnota aktuálního prahu, je exportována aktuální podoba modelu, a poté se pokračuje ve zjednodušování s následující hodnotou prahu.

Dalším možným způsobem použití je generování zjednodušených variant nikoliv na základě maximální ceny, ale na základě počtu zbývajících prvků v grafu. V tomto případě uživatel specifikuje požadovaný počet variant a algoritmus generuje nové varianty v pravidelných intervalech vždy po zpracování odpovídajícího počtu prvků.

4.2.5 Veřejné rozhraní knihovny

Pro snazší použití implementované knihovny novým uživatelem byly jednotlivé algoritmy zpřístupněny v jediném hlavičkovém souboru `LoDGenerator.h`. Prvky, které jsou určeny k přímému použití mimo knihovnu (tedy zejména jednotlivé prvky zjednodušujícího procesu) jsou při použití tohoto souboru také dostupné v rámci kořenového prostoru jmen `lod` — lze tedy používat například `lod::LazySelection` místo delšího `lod::algorithm::LazySelection`.

Kromě těchto zjednodušení definuje tento soubor také předpřipravenou funkci pro generování úrovní detailu nazvanou `simplify`. Tato funkce je tenkým obalem (*wrapper*) nad kombinací líného výběru, metriky QEM a operátoru *full-edge* sjednocení hrany. Účelem této funkce je poskytnout rychlé řešení těm uživatelům, kteří se nechtějí příliš zaobírat detaily implementace a potřebují „pouze“ rychle vygenerovat zjednodušené varianty modelu.

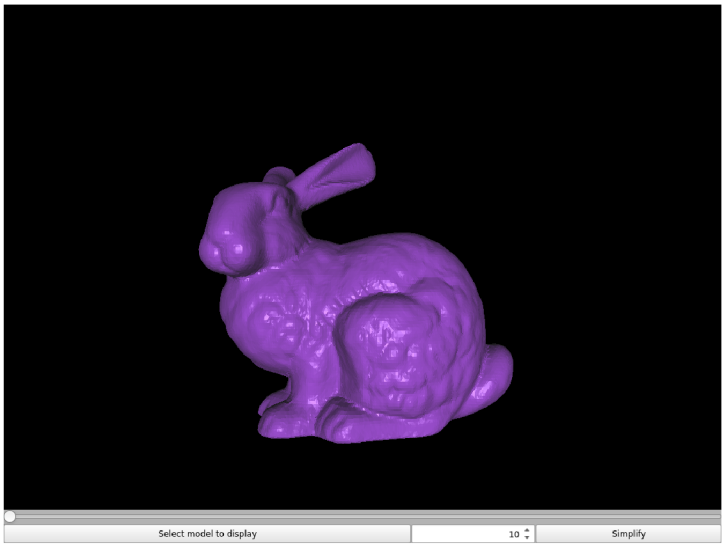
4.3 Demonstrační aplikace

Kromě knihovny samotné je součástí implementace i demonstrační aplikace. Účelem této aplikace je prakticky předvést možnosti knihovny a prezentovat výsledky implementovaných algoritmů.

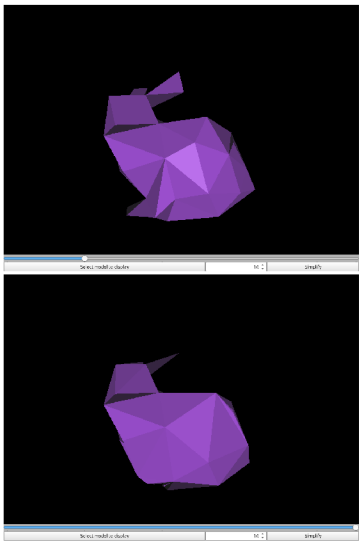
Demonstrační aplikace je koncipována jako interaktivní prohlížeč 3D modelů spojený s automatickým generátorem úrovní detailu. Jde o aplikaci s grafickým uživatelským rozhraním ovládanou převážně s pomocí myši. Ukázku tohoto rozhraní lze nalézt na obrázku 4.1a.

Většinu okna aplikace zabírá zobrazení aktuálně zvolené úrovně detailu vybraného modelu. Model lze pomocí stisknutí a tažení levého tlačítka myši kolem jeho středu[12]. S pomocí kolečka myši pak lze zobrazený model přibližovat a oddalovat od kamery.

[12] Přesně řečeno kolem středu ohraničujícího kvádru (*bounding box*).



(a) Demonstrační aplikace zobrazující originální variantu modelu.



(b) Generované úrovně detailu.

Obrázek 4.1: Snímky demonstrační aplikace. Zobrazený model lze tažením myši rotovat kolem středu a přiblížit či oddálit kolečkem myši. Tlačítka ve spodní části slouží k načtení originálního modelu a generování úrovní detailu. Posuvníkem pak lze volit zobrazenou úroveň detailu.

K výběru originálního modelu a generování úrovní detailu slouží tlačítka a číselník na spodním okraji okna aplikace. Pomocí tlačítka *Select model to display*, které se nachází vlevo dole, lze načíst originální variantu modelu ze souboru na disku. Po jeho stisknutí je uživatel pomocí standardního dialogu pro výběr souboru vyzván k výběru souboru s definicí modelu. Podporovány by měly být všechny souborové typy, které je schopna načíst knihovna Assimp[13].

Číselník a tlačítko *Generate*, nacházející se vpravo od tlačítka pro výběr modelu, slouží ke generování zjednodušených variant zvoleného modelu. Po stisknutí tlačítka *Generate* je z originálního modelu vygenerován zvolený počet variant. Ke generování je použita výše popsaná předpřipravená kombinace algoritmů — tedy *full-edge* sjednocení hrany, metrika QEM a algoritmus líného výběru.

Po vygenerování jednotlivých úrovní detailu lze použít posuvník umístěný nad ostatními ovládacími prvky pro výběr zobrazované úrovně. Tento posuvník simuluje přepínání mezi úrovněmi detailu v reálné aplikaci a umožňuje vizuální studium výstupů popsané implementace.

4.4 Další vývoj a možná rozšíření

Popsaná knihovna se nachází ve stavu funkčního prototypu. Prostor pro možná rozšíření je tak poměrně široký.

První důležitou oblastí vhodnou k rozšíření je práce s dalšími atributy modelu. V současném stavu jsou z původního modelu extrahovány pouze pozice jednotlivých vrcholů, ostatní atributy jsou zahozeny. U v praxi používaných modelů, které často používají texturovací souřadnice a další doplňující informace pro každý vrchol, toto omezení představuje vážný problém.

Přidání podpory dalších atributů lze realizovat v několika krocích. Prvním z nich je přidání vhodné reprezentace těchto atributů do implementace grafu (konkrétně struktury `Node`). Po tomto kro-

[13] Demonstrační aplikace využívá pro načítání modelu existující rozšíření knihovny GPUEngine: AssimpModelLoader. Podpora souborových typů v aplikaci je tedy plně závislá na podpoře typů v tomto rozšíření.

ku již začne být možné generovat zjednodušené varianty se zachováním atributů pomocí operátoru *half-edge* sjednocení hrany. Protože množina vrcholů v libovolné úrovni detailu generované s pomocí tohoto operátoru je podmnožinou vrcholů původního modelu, atributy jednotlivých vrcholů jsou automaticky „recyklovány“ i v generovaných úrovních.

Dalším krokem je implementace metrik a operátorů podporujících práci s dalšími atributy. Jako příklad lze uvést rozšíření metriky QEM o interpolaci libovolného množství dalších atributů, kterou publikoval Hoppe v článku *New quadric metric for simplifying meshes with appearance attributes* [19]. Konkrétně tato metrika by si pravděpodobně vyžádala rozsáhlejší změny ve stávající implementaci, zejména změnu implementace maticových operací[14]. Lze předpokládat, že podobné problémy mohou mít i jiné metriky či operátory. Jejich podpora tedy bude vyžadovat ještě netriviální množství práce.

Podobným druhem rozšíření je podpora *non-manifold* modelů. Ačkoliv implementovaná grafová abstrakce tyto modely v principu podporuje, stávající algoritmy jednotlivých operátorů předpokládají *manifold* síť a při porušení tohoto předpokladu skončí s chybou. Možné rozšíření grafové reprezentace je navrženo v původním článku [17] a mělo by postačit jej upravit pro tuto implementaci. Rozšíření operátorů o tuto podporu však již není triviální a bude vyžadovat další studium.

Jinou možnou oblastí pro rozšíření stávající implementace je přidání dalších operátorů, metrik a algoritmů. Díky kombinační charakteristice návrhu by implementace těchto rozšíření měla být celkem přímočará — postačí dodržet požadované rozhraní daného typu komponenty.

V neposlední řadě je také třeba konfrontovat stávající implementaci s reálnými modely a opravit příslušné nedostatky. Ačkoliv knihovna samotná obsahuje sadu testů, tato sada testuje pouze základní funkčnost knihovny a zdaleka není vyčerpávající. Lze tedy očekávat nalezení chyb nad reálnými daty. Rozšíření testovací sady a postupné odlaďování lze tedy také považovat za smysluplný směr další práce.

[14] Stávající řešení pomocí knihovny GLM podporuje matice do velikosti 4 × 4. Uvedená metrika vyžaduje výpočty s maticemi obecně $n \times n$, kde n je počet dimenzí všech atributů — tedy například podpora pozice (3 rozměry) a texturovacích souřadnic (2 rozměry) vyžaduje podporu matic 5 × 5.

5 Závěr

Tato práce se zabývala návrhem a implementací rozšíření pro knihovnu GPUEngine, které by k této knihovně přidalo prostředky pro vytváření úrovní detailů z existujících modelů. V rámci práce bylo navrženo jak požadované rozšíření, tak podpůrné datové struktury pro reprezentaci 3D grafu nezbytné pro implementaci rozšíření samotného. Navržené komponenty pak byly implementovány v rámci knihovny GPUEngine bez přidání externích závislostí.

Zdrojové kódy rozšíření lze nalézt na přiloženém DVD. Protože implementace probíhala v rámci existujících zdrojových kódů knihovny GPUEngine, nachází se na přiloženém DVD kompletní kód této knihovny včetně částí popsaných v této práci. Zdrojové kódy vlastního rozšíření lze nalézt ve složce `source/geAd/LevelOfDetail`, zdrojové kódy demonstrační aplikace se pak nacházejí ve složce `source/examples/LevelOfDetail`. Obě části lze sestavit pomocí přiložených instrukcí pro systém CMake. Díky integraci do stávajícího systému je pří sestavení nutné použít správné přepínače — rozšíření vyžaduje `GPUENGINE_BUILD_geAd=ON` a `GPUENGINE_BUILD_GESG=ON`, demonstrační aplikace pak ještě navíc `GPUENGINE_BUILD_EXAMPLES=ON`. Všechny části také vyžadují kompilátor s plnou podporou standardu C++14.

Cílem práce bylo seznámit se s algoritmy pro LoD a knihovnou GPUEngine a navrhnout rozšíření této knihovny pro statický LoD. Přehledem různých přístupů k LoD se zabývá kapitola Teorie. Stručný popis knihovny GPUEngine a návrh příslušného rozšíření lze nalézt v kapitole Návrh rozšíření.

Další částí zadání byla implementace alespoň dvou metod LoD pro knihovnu GPUEngine. Dále bylo cílem implementovat *morphing* a *blending* jednotlivých úrovní detailu. Popisem implementace těchto metod se zabývá kapitola Implementace. Díky nutnosti implementace nejen těchto metod, ale i podpůrné reprezentace 3D grafu již nezbyly časové prostředky na implementaci *morphing* a *blending*, a tyto operace tedy v práci popsány nejsou.

Dalšími požadovanými body této práce byly tvorba demonstrační aplikace a doprovodného plakátu či videa. Demonstrační aplikace byla zmíněna výše a lze ji nalézt na přiloženém DVD. Doprovodné video pak lze nalézt tamtéž.

Hodnocením práce se zabývají následující odstavce. Obecně lze práci hodnotit jako převážně úspěšnou. Hlavní cíl — poskytnutí implementace metod pro LoD – se split podařilo. Také byly položeny základy pro další práci v této oblasti, jak ve formě obecné grafové abstrakce, tak formou relativně snadno rozšiřitelné knihovny pro implementaci různých metod LoD. Nicméně výsledná knihovna je spíše funkčním prototypem než kompletním řešením, a pro praktické použití je třeba ji vhodně doplnit. Návrhy možných rozšíření a doplňků jsou popsány v závěru kapitoly Implementace.

Bibliografie

[1] CLARK, J. H. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*. 1976-10, roč. 19, č. 10, s. 547–554. ISSN 0001-0782. Dostupné z DOI: 10.1145/360349.360354.

[2] LUEBKE, D. et al. *Level of Detail for 3D Graphics*. New York, NY, USA: Elsevier Science Inc., 2002. ISBN 1-55860-838-9.

[3] HOPPE, H. Progressive Meshes. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 1996, s. 99–108. SIGGRAPH '96. ISBN 0-89791-746-4. Dostupné z DOI: 10.1145/237170.237216.

[4] LINDSTROM, P. et al. Real-time, Continuous Level of Detail Rendering of Height Fields. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 1996, s. 109–118. SIGGRAPH '96. ISBN 0-89791-746-4. Dostupné z DOI: 10.1145/237170.237217.

[5] LUEBKE, D. P. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications*. 2001-05, roč. 21, č. 3, s. 24–35. ISSN 0272-1716. Dostupné z DOI: 10.1109/38.920624.

[6] HOPPE, H. et al. Mesh Optimization. In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. Anaheim, CA: ACM, 1993, s. 19–26. SIGGRAPH '93. ISBN 0-89791-601-8. Dostupné z DOI: 10.1145/166117.166119.

[7] XIA, J. C.; EL-SANA, J.; VARSHNEY, A. Adaptive real-time level-of-detail based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*. 1997-04, roč. 3, č. 2, s. 171–183. ISSN 1077-2626. Dostupné z DOI: 10.1109/2945.597799.

[8] HAMANN, B. A data reduction scheme for triangulated surfaces. *Computer Aided Geometric Design*. 1994, roč. 11, č. 2, s. 197–214. ISSN 0167-8396. Dostupné z DOI: 10.1016/0167-8396(94)90032-9.

[9] SCHROEDER, W. J.; ZARGE, J. A.; LORENSEN, W. E. Decimation of Triangle Meshes. In: *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 1992, s. 65–70. SIGGRAPH '92. ISBN 0-89791-479-1. Dostupné z DOI: 10.1145/133994.134010.

[10] RIGIROLI, P. et al. Mesh refinement with color attributes. *Computers & Graphics*. 2001, roč. 25, č. 3, s. 449–461. ISSN 0097-8493. Dostupné z DOI: 10.1016/S0097-8493(01)00068-1.

[11] GARLAND, M.; HECKBERT, P. S. Surface Simplification Using Quadric Error Metrics. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, s. 209–216. SIGGRAPH '97. ISBN 0-89791-896-7. Dostupné z DOI: 10.1145/258734.258849.

[12] ROSSIGNAC, J.; BORREL, P. Multi-resolution 3D approximations for rendering complex scenes. In: *Modeling in Computer Graphics: Methods and Applications*. Ed. FALCIDIENO, B.; KUNII, T. L. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, s. 455–465. ISBN 978-3-642-78114-8. Dostupné z DOI: 10.1007/978-3-642-78114-8_29.

[13] COHEN, J. D. *Appearance-preserving Simplification of Polygonal Models* [online]. Chapel Hill, 1998 [cit. 2018-01-06]. Dostupné z: http://www.cs.unc.edu/xcms/wpfiles/dissertations/cohen.pdf. Disertační práce. The University of North Carolina at Chapel Hill. Vedoucí práce D. MANOCHA.

[14] CACCIOLA, F. Triangulated Surface Mesh Simplification. In: *CGAL User and Reference Manual* [online]. 4.11. CGAL Editorial Board, 2017 [cit. 2018-01-11]. Dostupné z: http://doc.cgal.org/4.11/Manual/packages.html#PkgSurfaceMeshSimplificationSummary.

[15] ALEXANDRESCU, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001. C++ in-depth series. ISBN 9780201704310. Dostupné také z: https://books.google.cz/books?id=aJ1av7UFBPwC.

[16] DE FLORIANI, L.; KOBBELT, L.; PUPPO, E. A Survey on Data Structures for Level-of-Detail Models. In: DODGSON, N. A.; FLOATER, M. S.; SABIN, M. A. (ed.). *Advances in Multiresolution for Geometric Modelling*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, s. 49–74. ISBN 978-3-540-26808-6. Dostupné z DOI: 10.1007/3-540-26808-1_3.

[17] CAMPAGNA, S.; KOBBELT, L.; SEIDEL, H.-P. Directed Edges — A Scalable Representation for Triangle Meshes. *Journal of Graphics Tools*. 1998-12, roč. 3, č. 4, s. 1–11. ISSN 1086-7651. Dostupné z DOI: 10.1080/10867651.1998.10487494.

[18] SMITH, R. (ed.). *Standard for Programming Language C++* [Final Working Draft] [online]. 2014-10-07 [cit. 2018-05-11]. Dostupné z: https://github.com/cplusplus/draft/blob/master/papers/n4140.pdf?raw=true.

[19] HOPPE, H. New quadric metric for simplifying meshes with appearance attributes. In: *Visualization '99. Proceedings*. 1999-10, s. 59–510. ISSN 1070-2385. Dostupné z DOI: 10.1109/VISUAL.1999.809869.