Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Obálka tělesa reprezentovaného trojúhelníkovou sítí

Plzeň, 2011 David Cholt

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 28. dubna 2011 .....

David Cholt

Poděkování

Tímto bych chtěl poděkovat panu Ing. Josefu Kohoutovi, Ph.D. za podklady, rady a usměrnění mých myšlenek při vzniku této práce po celý akademický rok 2010/2011. Dále bych chtěl poděkovat rodině a přátelům za jejich morální a finanční podporu v průběhu celého mého dosavadního studia, bez níž bych neměl možnost se k této práci dostat.

Abstract

Progressive hull of triangular meshes

Mesh deformation algorithms often use coarse hulls for a redution of the computation complexity, general optimisation and problem simplification. This thesis investigates possibilities of coarse mesh hull creation. It gives a survey of existing approaches and, in detail, it describes "Progressive Hull" algorithm which can provide outer coarse hulls while preserving shape details of the original mesh. A few optimisation techniques and quality enhancements of this algorithm are proposed. The implementation part of this thesis deals with various specifics of this algorithm, implementation-level optimisations, compatibility with the used libraries and with the integration of the produced implementation into the mesh deformation method by P. Kellnhofer. The impact on the stability of this deformation method and on the quality of its results is evaluated.

Obsah

Seznam obrázků

Upřesnění pojmů a zkratek

Pro porozumění následujícímu textu je zapotřebí definovat několik pojmů:

Primitivum - element triangulované sítě - vrchol, hrana či trojúhelník
Decimační vrchol - vrchol, kterým je nahrazena decimovaná hrana
Decimační filtr - mnou naprogramovaný VTK modul pro vytváření hrubých sítí
Deformační filtr - programové vybavení pro deformaci triangulovaných sítí vyvinuté Petrem Kellnhoferem jako součást jeho bakalářské práce [10]
STL - Standard Template Library - soubor základních datových struktur jazyka C++
VTK - Visualisation toolkit, knihovna pro operace s daty a jejich vizualizaci
Wrapper - Třída napsaná v požadovaném jazyce, která zpřístupňuje metody knihovny napsané v jiném jazyce
DDR2 - Double Data Rate 2 - Druhá verze dvojitého způsobu přístupu do paměti (zápis a čtení na náběžné i sestupné hraně řídicího signálu)
Dual Channel - Paralelní uspořádání 2 paměťových modulů pro zvýšení šířky pásma pro přístup do operační paměti
x64 - 64bitová architektura operačního systému

1 Úvod

Při vizualizaci trojrozměrných dat je často nutné operovat s velmi objemnými daty. Při těchto operacích se často používají metody pro zjednodušení sítí, kdy je požadovaná a časově náročná operace aplikována pouze na tuto jednodušší řídicí síť a její data jsou poté přenesena na původní těleso. Takovéto zjednodušené sítě lze vytvářet pomocí decimace původních dat, tedy snížením počtu primitiv (trojúhelníků, hran či vrcholů) původní sítě. Těchto metod existuje celá řada, jejich aplikace však není vždy vhodná.

Jednou z těchto aplikací je deformace tělesa podle změny zadané kostry. Zadaná kostra deformuje zjednodušenou řídící síť a data této sítě jsou poté přenesena na původní těleso. Tento postup však vyžaduje kvalitní řídicí síť splňující určité parametry, kterých nelze běžnými algoritmy ve významném počtu případů dosáhnout. Takovým parametrem je například požadavek obálky, kdy řídící síť musí zcela obalovat řízenou jemnou síť. Běžné decimační algoritmy tuto podmínku často nesplňují.

Tato práce se zabývá vytvářením obálek zadaného triangulovaného tělesa, které zachovávají detaily, zejména metodou *Progressive Hulls* - metodou vytváření obálek iteračním způsobem z původních obalovaných dat. Důraz je kladen na kvalitu obálek, zachování tvaru původního tělesa a na optimalizaci výpočtu. Tyto obálky mohou mít mnohá využití - zjednodušování detailních modelů a jejich ořez siluetou pro vykreslování zjednodušeného modelu s vizuálními parametry jemné sítě [19], pro rychlou detekci kolizí a výpočet průsečíků tělesa a přímky (například v raytracingu) [15] a samozřejmě pro použití jako řídící sítě.

Práce by měla být mimo jiné řešením problémů, na které narazil P. Kellnhofer během své práce [10] na deformacích triangulovaných těles. Jeho práce postrádá dobrý algoritmus pro vytváření tzv. hrubých sítí, které využívá pro řízení svých deformací. Využívá decimačního algoritmu pro vytváření zjednodušených sítí, které jsou následně upraveny na obálky. Tato metoda není vždy spolehlivá, decimací totiž může dojít ke ztrátě důležitých dat (dojde například ke kolapsu části sítě do jediného bodu) a takto vytvořená obálka poté nezaručuje, že bude celá jemná síť uvnitř sítě řídící. Při využití metody *Progressive Hulls*, která tento problém řeší, se předpokládá zlepšení výsledků jeho algoritmů.

V kapitole 2 se zabývám teorií obálek a hrubých sítí, jejich použitím, návrhy algoritmů pro jejich získání a jejich výhodami a nevýhodami a hledám návrh algoritmu, který bude výhody využívat a nevýhody pokud možno eliminovat. Kapitola 3 se zabývá algoritmem *Progressive Hulls* pro výpočet obálek se zachováním detailů a detaily z hlediska jeho myšlenky a postupu při decimaci sítě. Úskalími tohoto algoritmu a jejich vylepšením na základě pozorování výsledků se zabývám v kapitole 4 a dále jeho teoretickou optimalizací tak, aby bylo co nejlépe splněno zadání práce.

Ve druhé části práce se v kapitole 5 zabývám využívanými knihovnami; implementačními detaily, datovými strukturami a jednotlivými metodami modulu

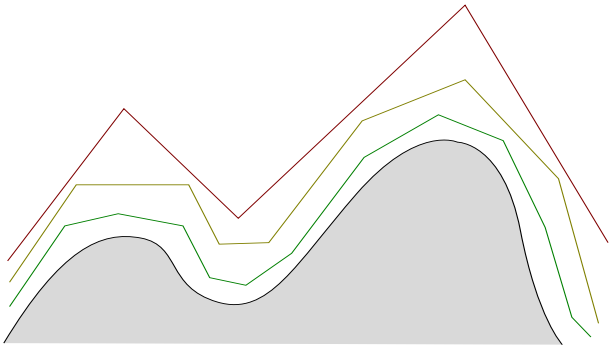
v kapitole 6 a předkládám dosažené výsledky, jak z pohledu mojí práce tak z pohledu jejího začlenění do práce P. Kellnhofera (kapitola 7). V závěru práce naznačuji další možné úpravy algoritmu pro lepší výsledky při jeho použití pro vytváření hrubých řídicích sítí.

2 Hrubé sítě a obálky těles

Tato práce je především určena pro generování tzv. hrubé sítě [11], proto se podíváme, co tento pojem znamená. Hrubé sítě jsou zjednodušené trojúhelníkové sítě použité pro výpočet časově a paměťově náročných operací a jejich výsledky jsou poté ze použití deformačních podmínek z původní detailní sítě na tuto síť aplikovány. Tím je zachována stabilita výpočtu za významného urychlení celého algoritmu. Takováto zjednodušená síť však musí mít několik vlastností:

- Musí být složena z méně trojúhelníků, než síť původní
- Musí být ve všech bodech vně sítě původní, tedy musí být obálkou původního tělesa
- Měla by zachovávat původní tvar původního tělesa
- Její trojúhelníky by se neměly navzájem protínat
- Měla by být hladká a spojitá

Tyto vlastnosti jsou důležité pro zachování stability barycentrických souřadnic [12] při výpočtu deformace původní sítě. Na obrázku 2.1 jsou zobrazeny příklady obálek (ve dvojrozměrném prostoru pro lepší představu). Takovéto zjednodušené sítě lze vytvářet několika způsoby, které lze rozdělit do skupin z pohledu vlastnosti, ze které chceme vycházet.



Obrázek 2.1: Příklady obálek. Zelená obálka má více hran, ale lépe vystihuje tvar původního tělesa. Červená obálka má málo hran, ale její tvar již původnímu tělesu odpovídá jen vzdáleně.

2.1 Globální přístup

Chceme-li vycházet z vlastnosti obalování původního tělesa, můžeme například použít globální přístup. Nejjednodušší obálkou tělesa je *ohraničující kvádr* (anglicky *Bounding Box*). Takový kvádr lze nalézt velmi jednoduše hledáním globálních maximálních a minimálních vzdáleností vrcholů původního tělesa od počátku souřadnic ve všech třech osách. Taková obálka však nezachovává původní tvar tělesa.

Mohli bychom použít některý z algoritmů dělení prostoru, jako například *Octree* [20], tedy rozdělit obálku na 4 menší kvádry a eliminovat ty, ve kterých nejsou žádná primitiva původní sítě. Rekurzivně bychom mohli získat jakousi „kostrbatou“ obálku. Vylepšení této obálky by poté spočívalo v přizpůsobení koncových kvádrů obálky částem sítě, které jednotlivé kvádry obsahují (přibližovat vrcholy obálky z vnějšku blíže k tělesu). Takovou změnou už by pak ale nebylo zaručeno, že bude ve všech částech vně původního tělesa. Tomu bychom mohli zabránit hledáním průsečíků tělesa a obálky, což je však velmi výpočetně náročné a proto se mu budeme chtít vyhnout.

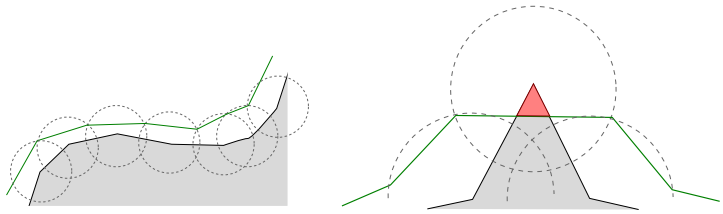
Podobným přístupem by mohl být opačný přístup, tedy vytvořit *ohraničující kvádr* pro každé primitivum prostoru a tyto kvádry spojovat spojovat pomocí stromové struktury. Tohoto postupu využívá algoritmus *Automatic Bounding Box Tree* [6], původně navržen pro aplikaci v raytracingu. I tímto přístupem však dojdeme ke stejnému závěru jako výše. I když tedy těmito postupy lze vytvořit obálku tělesa, nezískáme hladké obálky či obálky výrazně zachovávající tvar tělesa.

Hladkou globálně tvořenou obálkou je *konvexní obálka* (*Convex hull*). Takovou obálku bychom si mohli představit jako vyfouknutý pouťový balónek, ve kterém je umístěno původní těleso. Nejpoužívanější algoritmy pro tvorbu těchto obálek pro trojrozměrný prostor jsou dobře, jednoduše a s ukázkami vysvětleny v [14]. Tyto obálky ale v konkávních částech původní sítě nezachovávají tvar a tím se vzdálenost zjednodušené sítě od původní v různých místech tělesa mění. To také není to, co hledáme, naše obálka by měla být od původního tělesa pokud možno vzdálena všude stejně.

Globální metody tedy nebudou to, co potřebujeme, jelikož nezachovávají tvar původní sítě.

2.2 Lokální přístup

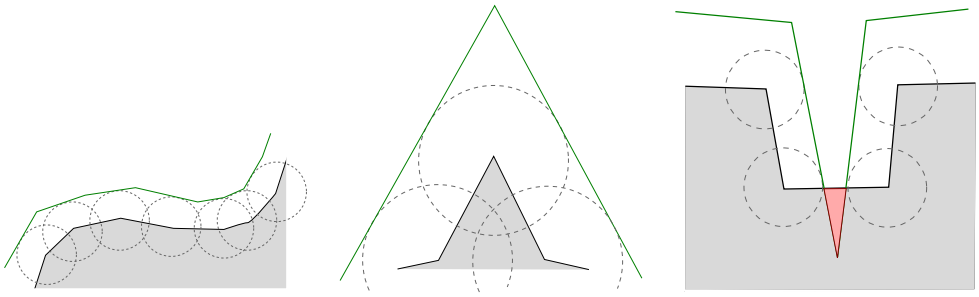
Pokud vycházíme z požadavku dosáhnutí podobnosti původní sítě a obálky, jedním z návrhů je lokální řešení, s využitím tzv. „bublinové“ metody neboli metody *sjednocených koulí* (*Union of balls*) [4]. Tato metoda je původně myšlena pro rekonstrukci povrchových sítí nad zadanou množinou bodů, což nepotřebujeme, jelikož již máme povrchovou sít zadanou, podíváme se ale, zda bychom ji nemohli využít.



(a) (b)

Obrázek 2.2: (a) Obálka pomocí průniků bublin. (b) Problém této metody.

Do každého vrcholu původní sítě by bylo možné umístit myšlenou kouli. Abychom docílili vnější obálky, její vrcholy by byly v průnicích těchto koulí (obrázek 2.2a). Již zde je však problém s nalezením průniku koulí vně tělesa[1], tomu jsme se chtěli vyhnout už v kapitole 2.1. Počet vrcholů obálky by byl sice stejný jako počet trojúhelníků původní sítě [2], ale obálka by byla velmi pravidelná, i když její vzdálenost od původní sítě by také byla v různých místech rozličná. Problém této metody je, že selhává v situacích, kde je úhel mezi původními trojúhelníky velmi ostrý (obrázek 2.2b). Obrázky jsou kresleny pro dvojrozměrné případy pro lepší představu.



(a) (b) (c)

Obrázek 2.3: (a) Obálka pomocí tečen bublin. (b) Vyřešení problému 2.2a. (c) Problém této metody.

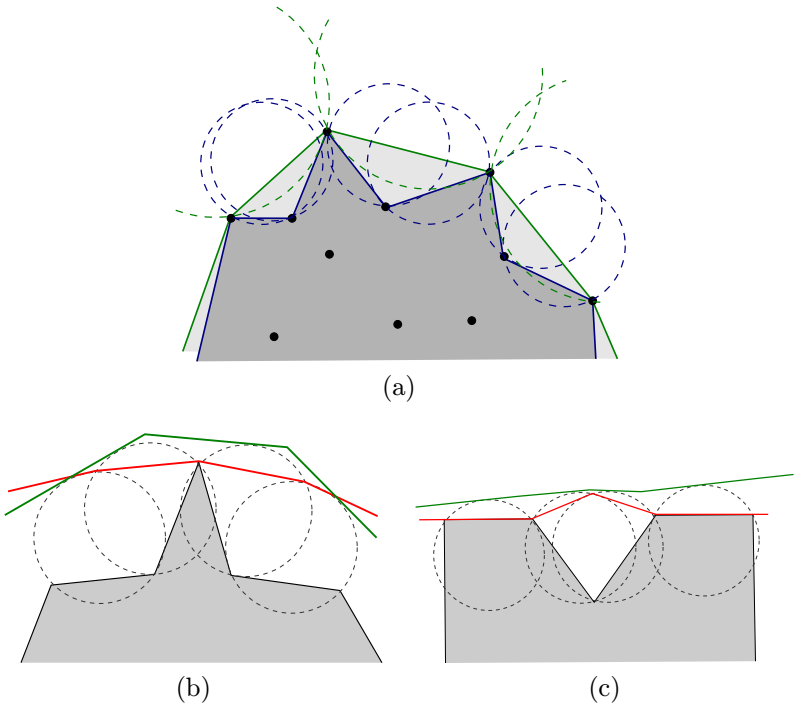
Tento problém by se dal vyřešit tím, že by nové trojúhelníky byly definovány pomocí tečen - nový trojúhelník by byl umístěn na ploše, která je tečnou plochou tří sousedních koulí (obrázek 2.3a). Tento způsob řeší problém s ostrým vnějším úhlem (2.3b), generuje sítě pravidelněji vzdálené od sítě původní, ale selhává na ostrých vnitřních úhlech (2.3c). Opět také ale nastává hledání vnějších tečných bodů. Každé tři koule mají v ideálním případě dvojici trojic tečných bodů. Po-

[1] Každá trojice koulí generuje dva průniky

[2] Každé 3 vrcholy generují jeden vnější průsečík stejně jako generují jeden trojúhelník sítě. Pokud průsečík neexistuje, algoritmus by tvořil v obálce díry

kud koule nemají vzájemné průniky, může být tečných ploch více, nebo naopak nemusí vůbec existovat. Velikost koulí by tedy musela být alespoň taková, aby k tomuto problému nedocházelo. Bublinová metoda tedy pravděpodobně spolehlivě fungovat nebude a navíc je u ní velmi obtížné regulovat potřebné zjednodušení sítě.

Další metodou hledání obálek související s metodou *sjednocených koulí* je metoda *alfa tvarů* (*alpha shapes*) [5]. Tato metoda je podobná předchozí metodě, používá také koulí pro hledání triangulovaného povrchu, ale koule jsou umísťovány tak, aby na jejich povrchu byly 3 vrcholy původní sítě (2 vrcholy v dvojrozměrném případě). Z těchto koulí vybereme pouze ty koule, které ve svém objemu nemají jiné vrcholy (vizte obrázek 2.4a). Tato metoda má oproti bublinové metodě výhodu snadnější regulace počtu výsledných trojúhelníků pro velmi členité okolí sítě. Na obrázku 2.4a jsou dvě velikosti umísťovaných koulí a vidíme, že u zelené obálky dochází k zjednodušení sítě. Na konvexních částech sítě si ale se zjednodušením příliš nepomůžeme. Dalším problémem je, že obálka využívá původní body tělesa, není tedy vždy vnější.



Obrázek 2.4: Metoda *alpha shapes* (a) Tvorba obálky z množiny bodů (b) a (c) Situace u problémů 2.2b a 2.3c. Červená obálka je pomocí průsečíků koulí, zelená pomocí pomocí jejich tečných ploch.

Můžeme se pokusit aplikovat způsob tvorby obálky z *bublinové metody* pomocí průsečíků nebo tečných ploch koulí. Na obrázcích 2.4b a 2.4c vidíme, že problémy bublinové metody nenastávají. Problém použití průsečíků je však zjevný, jeden

průsečík je vždy v původním vrcholu a druhý může být vně, nebo uvnitř tělesa. To bychom museli opět rozhodnout, což je časově náročné a chceme se tomu vyhnout. Přístup pomocí tečen má stejný problém s jejich nejednoznačností jako u *bublinové metody*. Dále má metoda tyto nevýhody:

- Opět nevelké snížení počtu vrcholů obálky, zvláště na konvexních částech sítě
- Příliš malé koule mohou způsobit, že bude síť roztržena na více komponent (jak naznačuje i článek [5]) a příliš velké koule by způsobily nedostatečné zachování detailů. U sítí s velmi různými velikostmi trojúhelníků tak tato metoda selhává[3]
- Výpočetní složitost. Jen nalezení všech odpovídajících koulí je dle [5] složitosti $O(n^2) + O(nlogn)$, v nejhorším případě $\theta(n^2) + \theta(n^2 log n^2)$. Následovalo by hledání sousedních koulí a výpočet tečen/průniků a jejich umístění vzhledem k původnímu tělesu.

Lokální přístup je tedy lepší než globální co se týče výsledků a má výhodu v podobnosti původní sítě a její obálky, ale redukce počtu vrcholů není příliš velká a algoritmy jsou výpočetně složité.

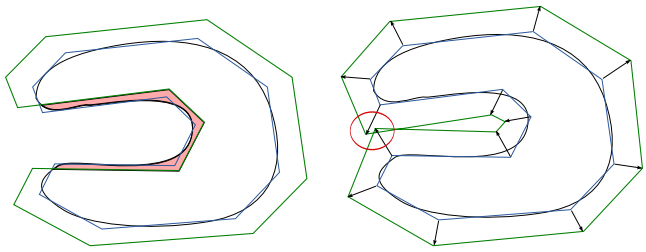
2.3 Přístup ze strany decimace

Lze také přemýšlet ze třetího pohledu - ze strany snížení počtu vrcholů, tedy decimací. Existují decimační algoritmy (například *decimace s využitím kvadratické metriky* [9]), které fungují na principu nahrazování trojúhelníků nebo hran vrcholy podle určité metriky. Dostatečným zvětšením (*scale*) takto decimované sítě získáme v určitých případech obálku.

Tato metoda ovšem opět selhává v mnoha případech, například situace, kde je síť nekonvexní (obrázek 2.5a). Tento postup se však zdá být dobrý, je pouze nutné jej upravit. Jednou z možností (použitou v [10]) je použít normály vrcholů a všechny vrcholy posunout ve směru těchto normál o konstantní vzdálenost (obrázek 2.5b). Toto řešení má však dvě nevýhody:

- Sebeprotínání (na obrázku 2.5b je tento problém naznačen v levé části). To lze odstranit druhou decimací sítě po posunu vrcholů (opět vizte [10]).
- Velká závislost na decimačním algoritmu. To je největším problémem této metody. Většina decimačních algoritmů nemá možnost pracovat s parametry decimací jednotlivých primitiv, proto mohou vznikat nepříjemné artefakty, které způsobují nestabilitu jakékoliv další aplikace zdecimované sítě pro další výpočty.

[3] Velikost koulí musí být konstantní, jinak tato metoda ztrácí smysl.



(a) (b)

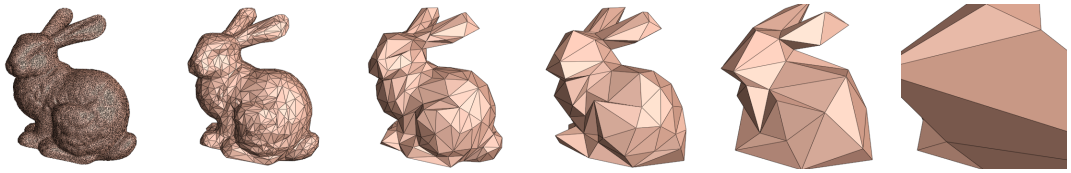
Obrázek 2.5: Obálka pomocí decimace. (a) Modrá zdecimovaná síť je zvětšena na zelenou. V nekonvexní části dochází díky zvětšení k chybě, původní síť je vně obálky. (b) Lepší přístup pomocí normál.

Lepším řešením je použití algoritmu, který bude kombinovat decimaci a lokální posun od původní sítě najednou a provázaně. Jedním takovým je algoritmus s názvem *Progressive Hulls*, je naznačen v článku *Silhouette clipping* [19] a jeho aplikací se tato práce zabývá.

3 Progressive Hulls

3.1 Motivace

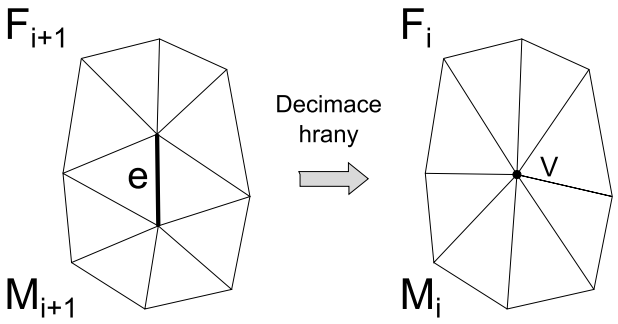
Progressive Hull je typ obálky, která je vytvářena z původní sítě s ohledem na její tvar. Příklad aplikace této metody je na obrázku 3.1. Je vidět, že je obálka se snižujícím se počtem trojúhelníků stále větší a zachovává tvar.



(a) 69674 △ (b) 2000 △ (c) 500 △ (d) 200 △ (e) 100 △ (f) 38 △

Obrázek 3.1: Příklad decimace. Převzato z [19]

Myšlenka tohoto algoritmu popsaného v článku [19] je jednoduchá - nahrazovat hrany původní sítě pouhými vrcholy tak, aby původní těleso leželo vždy v objemu nového tělesa, tedy pokud je povrch decimován, zůstane buď stejný, nebo se lokálně posune směrem ven. Decimace celé sítě je potom sekvencí těchto nahrazení (příklad jednoho nahrazení je zobrazen na obrázku 3.2).



Obrázek 3.2: Decimace jedné hrany.

3.2 Elementární decimace

Zvýrazněná hrana e (na obrázku 3.2) v okolí F_{i+1} (okolí hrany před decimací) je nahrazena vrcholem V v okolí F_i (po decimaci). Dle původního článku se musí vrchol V nacházet v průniku poloprostorů, které jsou nad plochami definovanými všemi trojúhelníky f v okolí F_{i+1} . Směr „nad“ je definován, jelikož díky orientovanosti sítě[4] lze dopočítat normály trojúhelníků a hledané poloprostory poté budou v jejich směru, nebo ve směru opačném (záleží na orientaci trojúhelníků). Průnik může být prázdný, v tom případě není decimace pro tuto hranu povolena. Díky tomuto způsobu decimace je tedy zaručeno, že objem sítě M_i bude větší, než objem sítě M_{i+1} nebo stejný (v případě, že je celé okolí F_{i+1} v rovině).

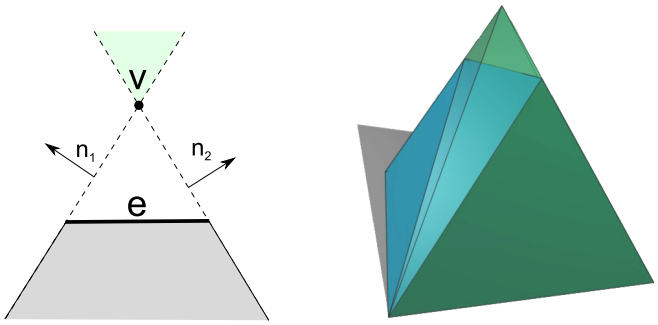
Poloha vrcholu V je dopočítána pomocí lineárního programování. Omezujícími podmínkami jsou rovnice výše uvedených poloprostorů a cílovou funkcí je minimalizace zvětšení objemu decimované sítě. Toto navýšení lze počítat jako změnu lokálního objemu, kterou lze definovat jako rozdíl objemových příspěvků[5] jednotlivých trojúhelníků f v okolích F_i a F_{i+1}, tedy $V(F_i) - V(F_{i+1})$.

Na obrázku 3.3a vidíme náčrt takové decimace pro dvojrozměrný případ. Máme decimovanou hranu e a hledáme vrchol V který ji může nahradit. Sousední hrany hrany e určují vnější poloprostory a jejich průnik (zeleně) určuje prostor, kde lze hledat polohu vrcholu V. Jelikož minimalizujeme objem, bude jeho poloha právě v bodu průniku přímek, na kterých hrany leží. Obrázek 3.3b ukazuje vizualizaci takové elementární decimace v trojrozměrném prostoru. Původní okolí F_{i+1} je zobrazeno modře (uvnitř) a nové okolí F_i zeleně.

Jednotlivé hrany sítě vstupují do prioritní fronty podle vhodné metriky (článek [19] navrhuje použít nárůst objemu) a postupně zpracovávány - decimovány.

[4]Orientovaností rozumíme pořadí vrcholů jednotlivých trojúhelníků, levotočivé či pravotočivé. Aby síť byla orientovaná, musí všechny její trojúhelníky mít toto pořadí stejné

[5]Každý trojúhelník sítě se podílí na celkovém objemu tělesa. Změna trojúhelníku se projeví na jeho příspěvku k celkovému objemu



(a) (b)

Obrázek 3.3: Elementární decimace hrany. (a) Dvojrozměrný náčrt (b) Trojrozměrná vizualizace

3.3 Algoritmus

Při decimaci celé sítě je nutné brát v úvahu větší množství operací, které původní článek nutně nepopisuje. Algoritmus popsaný v článku [19] lze tedy detailněji popsat takto:

- Příprava
  1. Vyhledat vztahy[6] mezi primitivy
  2. Pro každý trojúhelník spočítat jeho přírůstek k objemu a jeho normálu
  3. Pro každou hranu původní sítě:
     - I Najít okolní trojúhelníky (nalezení F_{i+1})
     - II Pro všechny trojúhelníky f z okolí F_{i+1} vyhledat poloprostory ve směrech jejich normál a jejich rovnice použít jako podmínky lineárního programování
     - III Sestavit cílovou funkci úlohy lineárního programování - minimalizace změny lokálního objemu
     - IV Vyřešit úlohu lineárního programování. Má-li úloha řešení, zařadit hranu do prioritní fronty s prioritou odpovídající možné změně lokálního objemu a přiřadit k hraně vypočtené souřadnice vrcholu nahrazujícího tuto hranu

[6] Informace o tom, které primitivum sousedí se kterým

- Decimace
  1. Vyzvednout hranu z prioritní fronty a nahradit ji přiřazeným vrcholem
  2. Odstranit ze sítě primitiva, která se v F_i nevyskytují, tedy trojúhelníky, které s hranou sousedí a duplicity v jejich hranách[7].
  3. Obnovit vztahy mezi primitivy v celém okolí F_i
  4. Pro všechny trojúhelníky v F_i aktualizovat normály a lokální objemy
  5. Pro všechny hrany v F_i znovu přepočítat vrcholy decimace a priority a upravit jejich pozice v prioritní frontě
  6. Odstraněné hrany a hrany z okolí F_i, jejichž decimace již není možná v důsledku jejich úprav, odstranit z prioritní fronty
  7. Opakovat body 1 až 6 pro všechny položky prioritní fronty nebo pro požadovaný počet decimovaných hran

3.4 Formulace úlohy lineárního programování

Jak je z algoritmu vidět, jediným opravdovým výpočtem je řešení úlohy lineárního programování. Lineární programování je typem optimalizace, kdy hledáme minimum (respektive maximum) zadané funkce o n proměnných vzhledem k omezením daným omezujícími nerovnostmi - podmínkami.

Naší cílovou funkcí úlohy lineárního programování je minimalizace nárůstu objemu. Jelikož je ale objem před decimací konstantní, stačí pouze minimalizovat nový objem po decimaci. Jelikož se však mění jen okolí F_{i+1} na F_i, stačí minimalizovat jen lokální objem okolí F_i (objem okolí F_{i+1} je konstantní). Objem okolí lze počítat jako sumu přírůstků objemu všech trojúhelníků f. Tento přírůstek můžeme počítat jako orientovaný objem tetrahedronu (čtyřstěnu) vzniklého přidáním jednoho libovolného vrcholu ke třem vrcholům tohoto trojúhelníka. Je zřejmé, že pro všechny trojúhelníky sítě použijeme stejný přidaný vrchol, pro jednoduchost výpočtu počátek souřadnic, tedy $[0, 0, 0]$. Takto zadaný tetrahedron má objem

$$V = \frac{\vec{V_1} \cdot (\vec{V_2} \times \vec{V_3})}{6}$$

kde $\vec{V_1}$, $\vec{V_2}$ a $\vec{V_3}$ jsou vektory souřadnic vrcholů trojúhelníka. Dělení konstantou můžeme vypustit, jelikož pro diferenci objemů nehraje roli. Objem okolí F_i lze tedy psát jako

$$\sum_{F_i} \vec{X} \cdot (\vec{V_2} \times \vec{V_3})$$

[7] Jde o hrany trojúhelníků přilehlých k decimované hraně. Odstraněním (smrštěním) jedné hrany trojúhelníka dojde k překrytí zbylých dvou, což je nežádoucí

kde $\vec{X}$ vektor hledaného vrcholu a $\vec{V_2}$ a $\vec{V_3}$ jsou vektory právě těch vrcholů trojúhelníka, které se s decimací nemění (tyto vrcholy nenáleží decimované hraně)[8]. Vidíme, že suma vektorových součinů je z pohledu výpočtu objemu neměnná, vyjádříme si ji tedy jako vektor $\vec{A}$

$$\vec{A} = \sum_{F_i} (V_2 \times V_3)$$

a cílovou funkci lze tedy psát ve tvaru

$$\vec{A} \cdot \vec{X} = a_1 x_1 + a_2 x_2 + a_3 x_3$$

Vidíme, že tato funkce *je* lineární a tedy může být předmětem k řešení. Najdeme-li minimum této funkce pro zadané omezující podmínky, nalezené souřadnice minima jsou zároveň souřadnicemi vrcholu, do kterého je možné hranu uprostřed okolí zdecimovat.

Zbývá tedy ještě dodefinovat omezující podmínky. Víme, že hledaný vrchol musí být nad rovinami trojúhelníků v okolí F_{i+1}. Napíšeme si tedy normálovou rovnici této roviny

$$\vec{V_f} \cdot \vec{n_f},$$

kde $\vec{V_f}$ je libovolný vrchol trojúhelníka a $\vec{n_f}$ je jeho normála[9]. Hledaný vrchol $\vec{X}$ musí být nad touto rovinou nebo na ní, tedy ve směru její normály [17], a tedy

$$\vec{n_f} \cdot \vec{X} \geq \vec{n_f} \cdot \vec{V_f}$$

Omezující podmínky mají na pravé straně konstantu, přepíšeme tedy tuto rovnici na

$$\vec{n_f} \cdot \vec{X} - \vec{n_f} \cdot \vec{V_f} \geq 0$$

a máme omezující podmínku pro zadaný trojúhelník hotovu. Podmínky takto sestavíme pro všechny trojúhelníky v okolí F_{i+1}.

Dodatečnou podmínkou algoritmu je zabránění sebeprotnutí sítě, tedy že vrchol $\vec{X}$ nesmí být uvnitř tělesa. Článek [19] naznačuje, že tato podmínka není nutná a testováním jsem došel k závěru, že opravdu není. K sebeprotnutí dochází v místech, kde se k sobě blíží dvě tělesa, tento jev se však našich datech nevyskytuje a pokud ano, lze sebeprotínání zamezit omezením parametru maximální členitosti decimované sítě (vizte kapitolu 4.5).

Pro řešení této úlohy lze použít mnoho algoritmů, my se však jejich teorií a implementací zabývat nebudeme (použijeme knihovnu *lp_solve*, vizte kapitolu 5.2). Pokud však čtenáře tento problém zajímá, dobrým a používaným algoritmem pro řešení problémů lineárního programování je *Revidovaný simplexový algoritmus* [3].

[8] Je nutné zachovat orientaci trojúhelníka a tedy pořadí vrcholů v tomto výpočtu

[9] Normálu trojúhelníka lze vypočítat jako vektorový součin vektorů $(V_2 - V_1)$ a $(V_3 - V_1)$ [16]

4 Vylepšení algoritmu

Cílem této práce je produkovat pokud možno kvalitní obálky v co nejlepším čase. Při testování výše zmíněného algoritmu jsem však přišel na několik nedostatků, které jsem se pokusil vylepšit.

4.1 Výpočet priorit

Článek [19] nejasně mluví o způsobu výpočtu priorit, zmiňuje pouze, že jako prioritu lze použít nárůst objemu, nespecifikuje však jakého. Čtenář by jistě pochopil, že jde o **změnu objemu v okolí** F_{i+1}, jelikož je to právě tato změna, která se při výpočtu nového bodu minimalizuje.

Takto implementovaný algoritmus vždy decimuje nejprve hrany s nejmenším lokálním nárůstem objemu. Jsou-li dvě takové hrany „u sebe“, dojde přirozeně k jejich okamžité decimaci a tedy ke spojení těchto malých lokálních objemů. Výsledný objem těchto spojených lokálních objemů však již může být větší, než nárůst objemu po decimaci jiné, „vzdálené“ hrany. To ale algoritmus ignoruje. Nabaluje na tyto spojené objemy další a další malé objemy z okolí, jelikož jejich priorita (lokální nárůst objemu) je nižší, než priorita (opět pouze lokální nárůst objemu) decimace jiné „vzdálené“ hrany. Není totiž do priority započítán nárůst objemu z předchozích decimací.

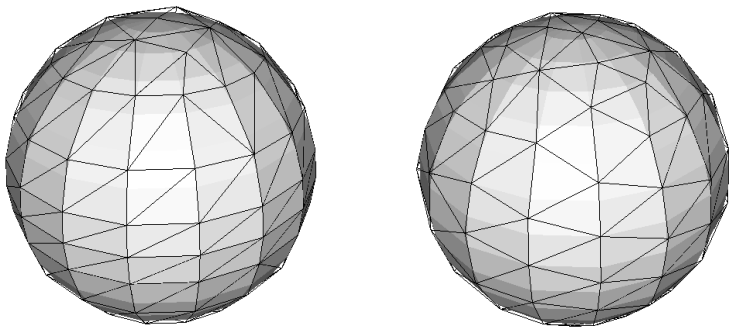
Výsledkem tohoto postupu je však nerovnoměrná decimace. Lokální nárůst objemu a priorita jsou totiž dva rozdílné parametry. Priorita musí odrážet nárůst objemu vzhledem k objemu **původní sítě**, nikoliv k objemu výsledku předchozí iterace. Je tedy nutné měřit rozdíl mezi objemem původní sítě a objemem decimované sítě po poslední iteraci a součet tohoto rozdílu a nárůstu lokálního objemu použít jako prioritu.

Nerovnoměrnost decimace způsobenou tímto zdánlivým detailem ukazuje obrázek 4.1.

4.2 Rychlost algoritmu

Autoři článku [19] se optimalizaci algoritmu příliš nevěnovali, jelikož obálky předpočítávají a následně je používají již vypočítané. My ale potřebujeme obálky počítat v rozumném čase, jelikož je jejich výpočet součástí složitějšího výpočtu deformací. Algoritmus je časově velmi náročný díky častému řešení úlohy lineárního programování (v bodě 3IV a zejména pak v bodě 5). I původní článek však říká, že prioritu lze počítat jiným, vhodným způsobem. Použijeme tedy tuto možnost a oddělíme výpočet priority a výpočet decimačního bodu. Tím budeme řešit úlohu lineárního programování jen pokud je to nutné, ne jen pro výpočet priority.

Potřebujeme tedy pouze nový algoritmus pro výpočet priority. Velmi rychlý algoritmus popisuje článek [15]. Ten říká, že je pro výpočet priority postačující



(a) (b)

Obrázek 4.1: Kulička po decimaci z 480 na 250 trojúhelníků. (a) Špatné počítání priorit. Algoritmus střední část kuličky ignoruje. (b) Správné počítání priorit. Celá kulička je decimována rovnoměrně.

vypočítat algebraický průměr souřadnic vrcholů v okolí F_{i+1} (bez vrcholů náležících decimované hraně) a tento vrchol použít pro výpočet nárůstu objemu a tento v absolutní hodnotě použít jako prioritu. Samozřejmě je nutné také aplikovat odchylku od objemu původní sítě popsanou v kapitole 4.1.

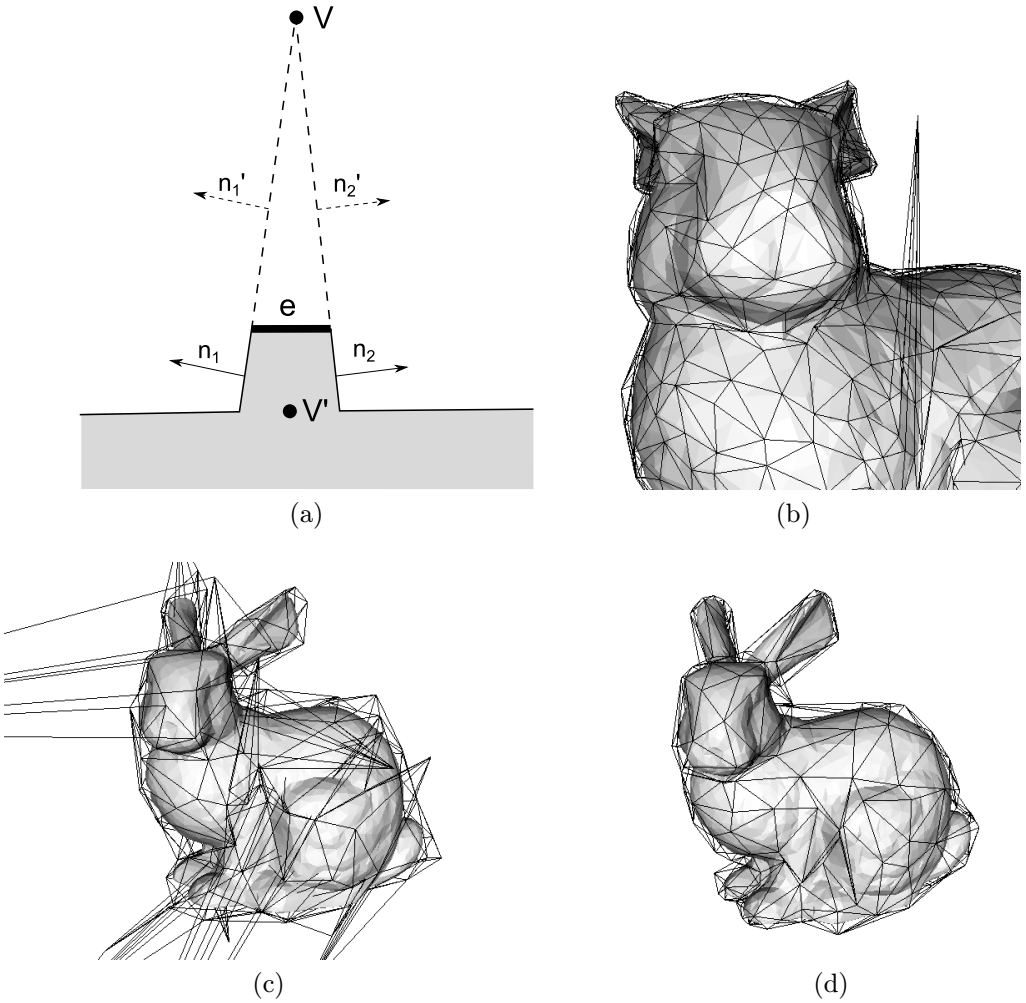
Takto vypočtený bod má podobné vlastnosti jako výpočet pomocí lineárního programování. Pro malá okolí F_{i+1} generuje malé objemy a tedy je jejich priorita nižší. Je-li okolí rovné, objem je také malý. Předpokládáme-li, že je počet vrcholů v okolí F_{i+1} pro složitost výpočtu zanedbatelný, je výpočet složitosti $O(1)$, což je oproti $O(n^x)$ složitosti *simplexové metody* řešení úlohy lineárního programování velmi dobré. Na druhou stranu je takto vypočtená priorita velmi nepřesná pro členitá okolí F_{i+1}. Správný bod decimace může být úplně jinde, než algebraický průměr okolních vrcholů. To může způsobovat problémy.

4.3 Stabilita algoritmu

Nepřesný výpočet priorit a ne příliš pravidelná vstupní data mohou způsobit nestabilitu algoritmu, který poté vytváří nečekané artefakty, které pojmenovávám „jehly“. Obrázek 4.2a naznačuje jejich vznik. Priorita decimace hrany e je vypočítána pro „vrchol“ V' a je tedy velmi malá. Algoritmus ji tedy vybere z fronty a vypočte vrchol, do kterého se hrana musí zdecimovat, aby byl zachován objem, tedy vrchol V, a decimaci provede. Vidíme však, že nárůst objemu je velký a výsledná decimace je nepěkná a pro další výpočty nevhodná. Na obrázku 4.2b vidíme, jak se tento problém projevuje v praxi.

Při hlubší decimaci[10] se tento problém akumuluje a decimovaná síť se hroutí

[10]Při decimaci velkého počtu hran



(a) (b) (c) (d)

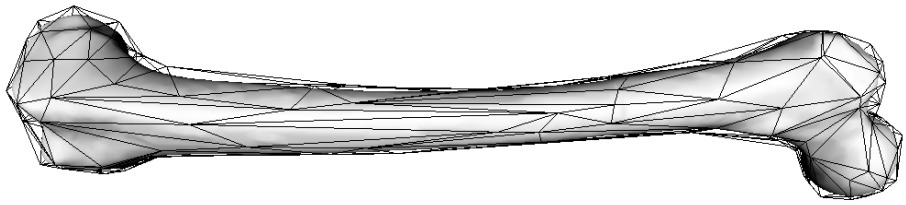
Obrázek 4.2: Vznik jehel. (a) Náčrt jejich vzniku (b) Ukázka v praxi (c) Zhoršení problému při hlubší decimaci (d) Stejná decimace po aplikování pravidla pro eliminaci „jehel“

(obrázek 4.2c). Dochází k sebeprotínání a síť je pro další numerické výpočty nepoužitelná.

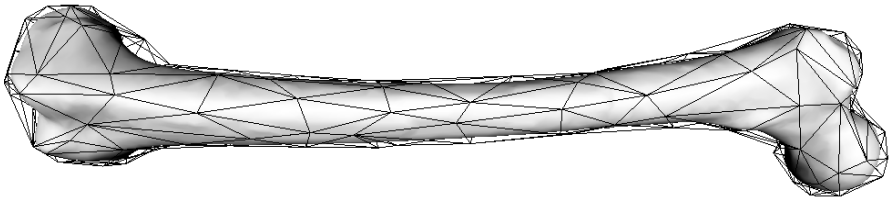
Potřebujeme tedy zavést pravidlo, kterým takovéto decimace bude možné zakázat. Článek [15] navrhuje použití kontroly odchylek normál trojúhelníku popsané v článku [7]. Jde o kontrolu rozdílu normál trojúhelníků před a po decimaci. Je-li tento rozdíl (úhel mezi normálami) mimo rozsah $[0, \pi/2]$, je decimace zamítnuta. Zjistíme však, že tato metoda na eliminování jehel příliš nefunguje, jelikož (jak je vidět na obrázku 4.2a) je normála trojúhelníků před a po decimaci stejná. Budeme tedy kontrolovat normály mezi trojúhelníky pouze v okolí F_i a budou-li příliš odkloněné (a tím okolí příliš špičaté), decimaci zakážeme (více vizte kapitolu 6.3.4). Jako bonus tento postup generuje pravidelnější sítě (vizte kapitolu 7.3). Ukázka aplikace tohoto pravidla je pro porovnání na obrázku 4.2d

4.4 Kvalita generovaných trojúhelníků

Nyní je již algoritmus téměř připraven, zbývá však dořešit poslední problém. Jelikož bude naše síť použita pro další výpočty, je nutné, aby trojúhelníky této sítě nebyly příliš úzké (z důvodu nestability numerických výpočtů na těchto tvarech trojúhelníků). Zavedeme tedy ještě jedno pravidlo decimace - kontrolu nejmenšího úhlu všech trojúhelníků v okolí F_i. Je-li trojúhelník úzký, je jeho nejmenší úhel malý. Povolíme tedy decimace takových hran, pro něž jsou tyto úhly pro všechny okolní trojúhelníky nad určitou hranicí.



(a)



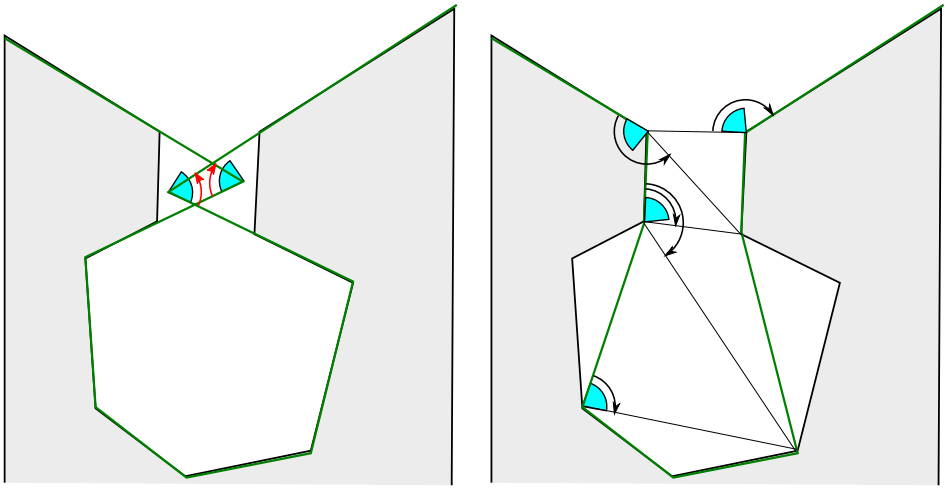
(b)

Obrázek 4.3: Výsledky pravidla pro eliminaci úzkých trojúhelníků. (a) Před zavedením pravidla (b) Po jeho zavedení

Menší problém tohoto omezení však nastává, pokud již vstupní data tenké trojúhelníky obsahují. Toto pravidlo tedy aplikujeme pouze pokud jsou trojúhelníky v okolí F_{i+1} široké, jinak decimaci provedeme vždy. Tato změna zaručí, že algoritmus alespoň nebude zhoršovat kvalitu sítě. Z testování však plyne, že decimací okolních trojúhelníků dojde k pohlcení tenkých trojúhelníků v původní síti a decimovaná síť je již neobsahuje. Tento dodatek má význam jen pokud původní sít obsahuje velký počet těchto tenkých trojúhelníků.

4.5 Sebeprotínání sítě

Jak již bylo řečeno, autoři článku [19] se domnívají, ze není nutné zabraňovat sebeprotínání při deformaci sítě. Při testování algoritmu jsem došel k závěru, že pro běžná data, tedy pro hladké modely jako jsou svaly či kosti, tento problém nenastává. Může však nastat extrémní případ, kdy k sebeprotnutí dojít může. Obrázek 4.4a tuto možnost naznačuje. Stejná situace platí, pokud jde o 2 komponenty modelu v jedné scéně.



(a) (b)

Obrázek 4.4: Možnost sebeprotnutí (a) Bez omezení (b) S omezením parametru *MeshRoughness*

Testy však ukázaly, že omezením parametru *MeshRougness* (v kapitole 6.3.4 se dozvíme, že je použit pro kontrolu okolí na „jehly“ kontrolou úhlu mezi trojúhelníky právě vznikajícího okolí F_i) dosáhneme zákazu této decimace a tím i vzniku sebeprotnutí. Obrázek 4.4b tuto možnost ukazuje, zdecimují se nejprve krajní hrany a poté dojde k „vyplnění“ celé „baňky“, jak naznačují světlé hrany. Tyto decimace jsou povoleny, protože úhly, které jsou v procesu kontrolovány, špičaté nejsou (na obou obrázcích je minimální úhel pro povolení decimace zobrazen světle modře). Možná se zdá, že je výběr decimací zvláštní či libovolný, je si třeba ale uvědomit, že postupným zamítnutím okolních decimací je možné právě k těmto vhodným decimacím dojít. Pokud parametr *MeshRougness* omezíme více, decimace vůbec nenastane a celé okolí bude zachováno v původní podobě. Je nutné také dodat, že je kontrolován vždy ten menší úhel, ať vnitřní či vnější. Vhodnou volbou parametru *MeshRougness* tedy lze sebeprotínání předejít a to i pro modely o více komponentách, jak ukazuje obrázek B.3.

Možností, jak zabránit sebeprotnutí, by bylo hranám, které by sebeprotnutí mohly způsobit, přidat další podmínky lineárního programování pro jejich protější trojúhelníky. Je ale zřejmé, že by došlo k decimaci a vzniku malé „jehly“ na levé straně náčrtu. Decimace druhé strany by neproběhla (jinak by došlo k sebeprotnutí), postupně by však proběhly všechny ostatní decimace a výsledek by byl velmi podobný obrázku 4.4b. Nevýhodou tohoto přístupu je, že bychom museli hledat protější trojúhelníky. To je výpočetně velmi složité, nabízí se zjednodušení pomocí Octree [20] - rozdělení prostoru na menší části a poté sestavení podmínek pro všechny trojúhelníky v té části, ve které se nachází decimovaná hrana, případně trojúhelníků v okolních částech. Tento postup však není nutný, pokud nám stačí pouze najít vhodný parametr *MeshRougness*. Pokud bychom

však potřebovali algoritmus více zobecnit, tuto kontrolu by bylo nutné zahrnout.

5 Knihovny použité pro implementaci

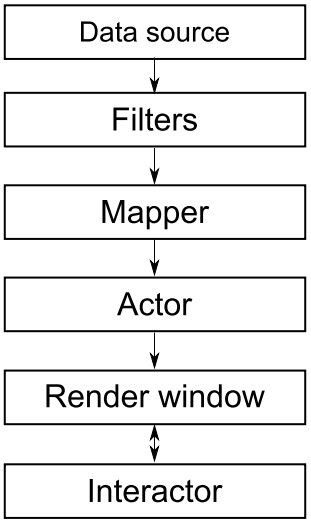
5.1 VTK - Visualisation Toolkit

Jedním z požadavků na tuto práci je kompatibilita s balíkem knihoven *Visualisation Toolkit* neboli *VTK*. Tento systém zprostředkovává mimo jiné knihovny pro zpracování a vizualizaci dat a pro práci s 3D grafikou většinou prostřednictvím OpenGL. Systém je vyvinut v jazyce C++, je však přizpůsoben pro použití i v jiných jazycích (Java, Python, ...). Velmi dobrými zdroji pro pochopení práce s *VTK* jsou tutoriál [1] a dokumentace [8].

Obrázek 5.1: Logo *VTK*. Převzato z [8]

VTK používá systém bloků kaskádovitě seřazených do *potrubí* - *pipeline*. Tuto *pipeline* zobrazuje obrázek 5.2. Zdroj dat (*Data source*) poskytuje data ke zpracování. Může jít o generovanou aproximaci nějakého geometrického tělesa, výsledek matematické funkce, nebo o data načtená ze souboru. Zdrojová data jsou předána `Filtrům` ke zpracování (různé matematické operace) a na jejich výstupu jsou hotová data připravená k zobrazení. Dále je využit `Mapper`, který z dat vytvoří jejich vizuální podobu, která je upravena blokem `Actor`. Ten je schopen měnit vizuální parametry, jako je barva, způsob stínování, drátěná reprezentace apod. Posledním blokem *pipeliny* je *Renderer*. Ten má za úkol vykreslení dat na obrazovku (do vlastního okna `RenderWindow`, nebo je použit `Viewport` v okně již vytvořeném). Ovládání je poskytnuto pomocí `Interactoru`, kterých je opět několik typů (myš, klávesnice...) v závislosti na požadovaném způsobu interakce s uživatelem. Celá *pipeline* funguje na principu požadování dat od předchozího bloku, bloky jsou tedy spouštěny jen když je potřeba a tím jsou eliminovány nadbytečné výpočty a zatížení systému.

Tento balík knihoven nám nejen usnadní zobrazení výsledků naší práce, ale také definuje abstraktní třídu `vtkPolyDataToPolyDataFilter`, kterou použijeme pro implementaci našeho kódu. Jakákoliv třída dědící od této abstraktní třídy totiž může být zařazena do *pipeliny* a poskytnout změny v datech, což je přesně to, co pro implementaci decimačního filtru potřebujeme. Hlavní metodou této



Obrázek 5.2: Pipeline systému VTK. Převzato a upraveno z [1].

třídy je metoda `Execute()` která je spuštěna kdykoliv jsou potřebná data vyžadována následujícím blokem *pipeliny*. Implementací této metody se budu věnovat v kapitole 6.3.

Knihovna je distribuována ve zdrojových kódech, je tedy ji nutné nejprve přeložit pomocí nástroje CMake.

5.2 lp_solve

Knihovnu lp_solve [2], Mixed Integer Linear Programming solver, aktuálně ve verzi 5.5.2.0, použijeme pro výpočet úlohy lineárního programování. Tato knihovna implementuje *Revidovaný simplexový algoritmus* [3] pro úlohy s reálnými čísly a metodu *větví a mezí* (*Branch-and-bound*) pro výpočet celočíselných úloh. Je schopna řešit čistě lineární, smíšené (celočíselné a binární), polospojité a speciální úlohy. Podporuje mnoho programovacích jazyků (C, C++, Pascal, Delphi, Java, VB, C#, VB.NET, Excel...), většinou pomocí *wrapperů*. My však využijeme nativně přeloženou knihovnu[11] kterou budeme volat přes lp_solve API. Knihovna slibuje vysokou stabilitu výpočtu a téměř neomezený počet proměnných a omezujících podmínek. Pro naše účely je tedy více než vyhovující (budeme mít pouze 3 proměnné a maximálně cca 20 podmínek podle počtu trojúhelníků v jednotlivých F_{i+1}). Distribuce obsahuje zdrojové kódy knihovny i její staticky či dynamicky použitelné binární podoby, pro OS Windows a Linux. Pro použití na jiném operačním systému si uživatel musí stáhnout zdrojové kódy knihovny a zkompilovat je pod tímto systémem; knihovna je vyvinuta v jazyce ANSI C.

[11] Používám dynamickou knihovnu, protože statická byla v konfliktu s *VTK*

6 Implementace

6.1 Způsob implementace

Implementaci jsem provedl v jazyce C++ s využitím vývojových prostředí *Microsoft Visual Studio 2008* (v rámci projektu 5) a *Microsoft Visual Studio 2010* (v rámci bakalářské práce) a za použití výše uvedených knihoven. Algoritmus samotný je implementován ve třídě `vtkProgressiveHull` která dědí od třídy `vtkPolyDataToPolyDataFilter` balíku *VTK*. Pro testování a demonstrační účely jsem vytvořil aplikaci `DemoApp`. Prioritní fronta je implementována v souboru `priorityQueue.h`. Po odladění mého algoritmu jsem přistoupil k jeho začlenění do třídy `MeshSurface`, která je součástí řešení P. Kellnhofera.

6.2 DempApp - demonstrační aplikace

Demonstrační aplikace je implementována v souboru `DemoApp.cxx`. Implementuje základní *pipeline* popsanou v kapitole 5.1. Jako zdroj dat je použit soubor. *VTK* podporuje více datových formátů, nás ale budou zajímat pouze dva - formát `.vtk` a formát `.obj`. Formát .vtk (popsaný v [13]) budeme využívat, protože jde o formát, ve kterém máme dostupná data. Formát .obj (popsaný v [18]) budeme využívat, protože do něj lze exportovat sítě vytvořené v programu Blender (pro testovací účely). Pro jejich čtení využijeme `vtkPolyDataReader` respektive `vtkOBJReader`. Obě tyto třídy předávají na výstup objekt typu `vtkPolyData`.

Po načtení dat je do *pipeline* zařazen můj filtr a následně je implementován zbytek *pipeline* podle obrázku 5.2. Tato je dvojitá, pro současné zobrazení původních dat a výstupu mého filtru.

6.3 Implementace samotného filtru

Aby `vtkProgressiveHull` splňoval parametry `vtkPolyDataToPolyDataFilter`, musí implementovat proměnné `InputMesh` a `OutputMesh` typu `vtkPolyData` a metodu `Execute()`. Odkaz na vstupní data je předán rodičovskou metodou `GetInput()` do proměnné `InputMesh`, která jsou poté zpracována metodou `Execute()` a předána na výstup `OutputMesh`. Metoda `Execute()` volá další metody, jejichž funkce je popsána v následujícím textu.

Filtr má pouze prázdný konstruktor, pro veškeré parametry jsou přidány `Get`ry a `Set`ry:

- `TargetReduction` - poměr počtu vrcholů původní a zdecimované sítě
- `MeshRoughness` - drsnost povrchu zdecimované sítě (-1: hladký povrch; 1: velmi hrubý povrch [kontrola „jehel“ je vypnuta])

- `MeshQuality` - kvalita trojúhelníků v zdecimované síti (0: nekontroluje se; 1: maximální dosažitelná kvalita)
- `EnlargeMeshAmount` - dodatečné zvětšení/zmenšení zdecimované sítě sítě (poměr)

6.3.1 InitMesh()

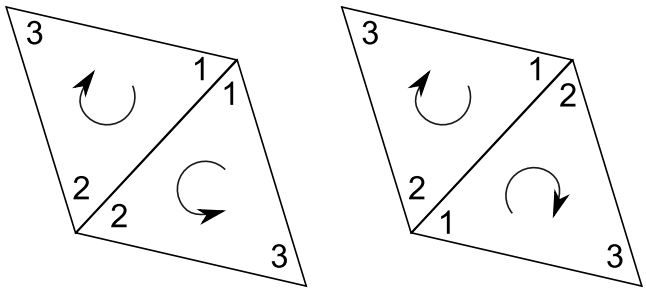
Použití dat přímo v reprezentaci pomocí `vtkPolyData` může být pomalé a nepohodlné. Metoda InitMesh() má tedy kromě inicializace filtru za úkol také převedení vstupních dat do vnitřních datových struktur (každé primitivum je reprezentováno instancí vnořené třídy `CVertex`, `CTriangle` nebo `CEdge`) a je využita struktura `vector` z knihovny *Standard Template Library (STL)* pro seznamy těchto primitiv. Převedení je provedeno postupně, nejprve jsou načteny vrcholy a poté trojúhelníky. Třída `vtkPolyData` reprezentuje trojrozměrná data, seznam vrcholů a tzv. *buněk*(*Cells*).

Není však pravidlem, že jedna buňka obsahuje jeden trojúhelník. Zjistíme tedy jakého typu buňka je a data z ní adekvátním způsobem převezmeme. Současná verze podporuje buňky s jednotlivými trojúhelníky, dvojicemi trojúhelníků v uspořádání *triangle strip* (použito ve VTK souborech) a dvojicemi trojúhelníků v uspořádání *quad* (použito v některých OBJ souborech). Jakákoliv jiná data jsou ignorována.

Pro všechny trojúhelníky jsou také spočteny normály a jejich objemové příspěvky. Jsou také nastaveny čítače (trojúhelníků, vrcholů) a proměnné pro výpočet úrovně decimace, celkový objem původního tělesa atd. Je také vytvořena instance třídy `lp_solve`.

6.3.2 BuildMesh()

Tato metoda vytváří vztahy mezi trojúhelníky a vrcholy (hledá tzv. *One Ring Primitives*, okolní primitiva pro každé primitivum, např. `OneRingVertex` - seznam okolních vrcholů daného primitiva) a z dat jednotlivých trojúhelníků hledá hrany. Víme-li, že dva trojúhelníky sdílejí dva vrcholy, víme, že tyto vrcholy spojuje hrana. Metoda `BuildMesh()` však kontroluje, zda jsou trojúhelníky správně orientovány (vizte obrázek 6.1). Jsou-li indexy obou vrcholů hrany pro oba trojúhelníky ve stejném pořadí, znamená to, že je jeden z trojúhelníků chybně orientován. Takováto hrana je označena jako chybná a není později decimována. Pravděpodobně by bylo možné pořadí vrcholů těchto trojúhelníků obrátit, to jsem však neimplementoval. Mohlo by to být předmětem další práce na filtru. Po průchodu touto metodou tedy máme informace o vrcholech, trojúhelnících, hranách a sousednostech mezi těmito primitivy.



(a) (b)

Obrázek 6.1: Orientace trojúhelníků podle pořadí jejich vrcholů. (a) Chybná orientace (b) Správná orientace

6.3.3 PrioritizeEdges()

Tato metoda volá metodu `double ComputePriorityFast(int edgeID)` pro výpočet priority všech hran sítě a předává tyto hrany spolu s prioritou do prioritní fronty. Zdrojový kód prioritní fronty byl dodán vedoucím práce, rozhodl jsem se jej ale přepsat s použitím *STL* pro konzistenci a rychlost. Tato fronta je implementována jako halda na poli `m_Data`, seřazená podle priority od nejnižší k nejvyšší a jednotlivé prvky jsou mapou `m_MapTable` namapovány k jejich objektům (v našem případě hranám) pro možnost úpravy jejich priority.

Metoda `double ComputePriorityFast(int edgeID)` je implementována podle kapitoly 4.2 s využitím seznamu `OneRingVertex` obou vrcholů zadané hrany. Výpočet těchto priorit je pro jednotlivé hrany disjunktní, bylo by tedy možné jej urychlit paralelizací. Jelikož je však tento výpočet krátký vzhledem k výpočtu decimací, není paralelizace implementována. Vzhledem k algoritmu 3.3 je tato metoda přípravnou fází algoritmu.

6.3.4 Decimate() a Substitute()

Tato metoda vybírá hrany z prioritní fronty a volá na nich metodu `Substitute-(CEdge* e)` dokud není fronta prázdná nebo dokud není dosažena požadovaná decimace sítě (řízena parametrem `TargetReduction`) . Metoda `Substitute(CEdge* e)` je nejdůležitější a nejsložitější metodou celého filtru. Nejprve je pro hranu `e` vypočten vrchol decimace metodou `ComputeDecimationPoint(int edgeID)`. Je-li hrana připravena k decimaci a není li již označena jako smazána, je volána metoda `bool CheckSpikeAndTriangles()`.

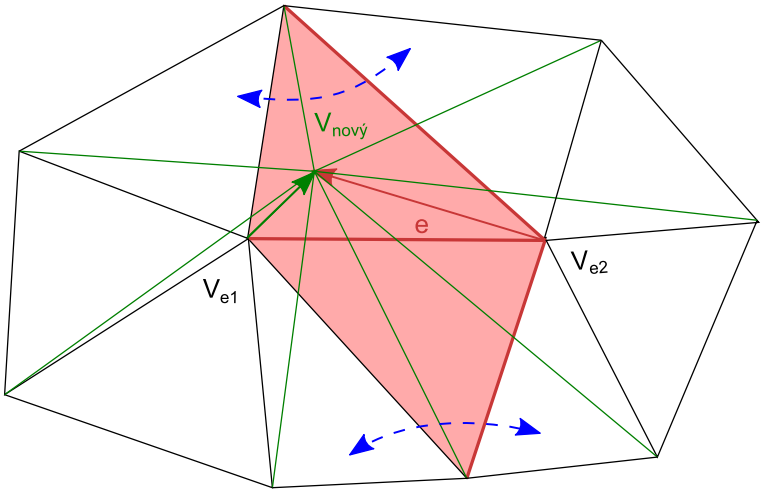
Tato metoda kontroluje možnost vzniku „jehly“ tak, že provede „virtuální decimaci“[12] a vytvoří pole normál všech trojúhelníků v jejím okolí. Porovnáním těchto normál skalárním součinem (každou s každou) získáme maximální

[12]Nahrazení vrcholů hrany decimovaným vrcholem ve všech okolních trojúhelnících, ale pouze dočasně, v původních strukturách se nic nemění

odklonění normál, na jehož základě decimaci hrany povolíme nebo zamítneme, podle parametru `MeshRoughness`. Tato metoda také kontroluje kvalitu trojúhelníků podle kapitoly 4.4 dle parametru `MeshQuality`.

Pokud navrhovaná decimace hrany projde všemi podmínkami, je na řadě její nahrazení vypočteným vrcholem. To provedeme tak, že jeden (první) vrchol hrany budeme považovat za nahrazující vrchol a okolí F_{i+1} upravíme na okolí F_i. Tento proces naznačuje obrázek 6.2.

- Vrchol V_{e1} posuneme na souřadnice nahrazujícího vrcholu
- Označíme primitiva, která se v F_i nevyskytují, jako smazaná (na obrázku 6.2 červeně)
- Okolní trojúhelníky, hrany a vrcholy[13] vrcholu V_{e1} přiřadíme vrcholu V_{e1}.
- Opravíme sousednosti trojúhelníků sousedících s mazanými trojúhelníky (na obrázku 6.2 modře)
- Přepočítáme normály a přírůstky objemů všech trojúhelníků v právě vytvořeném okolí F_i a aktualizujeme priority všech hran v tomto okolí



Obrázek 6.2: Nahrazení hrany a rekonstrukce okolí po decimaci

[13] Zde zabráníme duplicitním výskytům vrcholů v seznamu okolních vrcholů vrcholu V_{e1} kvůli pozdějšímu správnému výpočtu vrcholu pro výpočet priority. Duplicity ostatních primitiv budou ignorovány.

6.3.5 ComputeDecimationPoint() a SolveLP()

Tato metoda vyhledá data (trojúhelníky z okolí F_{i+1} pro omezující podmínky a sestaví vektor $\vec{A}$ pro cílovou funkci (vizte kapitolu 3.4) pro metodu SolveLP(). Ta projde tyto data a vytvoří pro ně omezující podmínky a nastaví lp_solve. Je nutné nejprve vymazat podmínky z předchozí iterace, přepnout *solver* do *řádkového módu* (rychlejší přidávání a eliminace zcela zjevně zcestných podmínek), postupně podmínky přidat, nastavit koeficienty cílové funkce (z vektoru $\vec{A}$) a nastavit cílovou operaci na minimalizaci. Je velice důležité odstranit přednastavené podmínky nezápornosti proměnných, jelikož námi vypočítaný vrchol pro decimaci může být kdekoliv v prostoru. Vypočtená data získáme metodami `get_objective()` (vypočtený nárůst objemu) a `get_variables()` (souřadnice vrcholu pro decimaci). Byl-li výpočet neúspěšný, je nutné hranu označit jako nepřipravenou k decimaci, aby nedošlo k chybám v síti.

6.3.6 EnlargeMesh()

Tato metoda dodatečně zvětší síť o požadovaný počet jednotek (nastavený parametrem `EnlargeMeshAmount`). Posune všechny vrcholy sítě ve směru jejich normály (průměr normál okolních trojúhelníků). Pozor, tato metoda může způsobit sebeprotínání sítě.

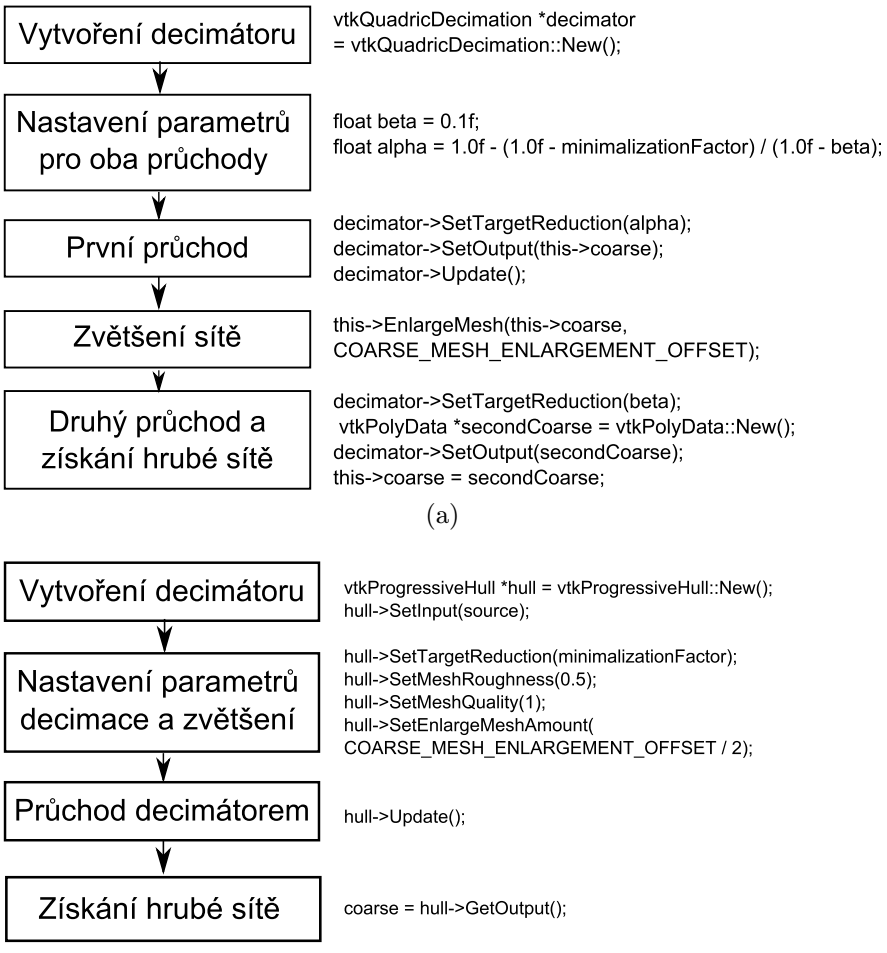
6.3.7 DoneMesh() a ClearMesh()

Metoda `DoneMesh()` zrekonstruuje výstupní `vtkPolyData` z dat v interních strukturách. Primitiva označená jako smazaná jsou vyřazena a identifikátory vrcholů jsou přemapovány, aby byly odstraněny „díry“ v jejich seznamu.

Metoda `ClearMesh()` uvolní veškerou alokovanou paměť, včetně *solveru*.

6.4 Začlenění filtru do třídy MeshSurface

`MeshSurface` je třída naprogramovaná Petrem Kellnoferem pro využití ve filtru `vtkMEDPolyDataDeformationPK`, jehož autorem je také on. Tato třída má za úkol vytvořit hrubou síť a vypočítat vztah bodů této sítě k původní trojúhelníkové síti tělesa. Nás zajímá hlavně metoda `CreateCoarseMesh(vtkPolyData* source, float minimalizationFactor)`, která tuto hrubou síť vytváří. Původní postup a mnou použitý postup naznačuje obrázek 6.3



Obrázek 6.3: Začlenění mého filtru. (a) Původní způsob vytváření hrubé sítě (převzato z [10]) (b) Použití vtkProgressiveHull

Původní postup používá 2 průchody sítě decimátorem, první pro získání sítě o menším počtu vrcholů a druhý pro zamezení sebeprotínání po zvětšení sítě. Pro každý průchod je vypočtena poměrná decimace tak, aby výslednou síť byla zredukována podle parametru `minimalizationFactor`. Síť je zvětšena, aby bylo zajištěno, že bude vně původního tělesa. Můj filtr tuto podmínku splňuje

již během decimace a decimace je provedena v jednom průchodu přímo na redukci `minimalizationFactor`. Je ale stále nutné mým filtrem vytvořenou síť zvětšit, aby byla filtrem P.Kellnhofera akceptována. Více o tomto problému vizte kapitolu 7.4.

7 Testy a dosažené výsledky

Všechny testy jsem provedl na této sestavě:

Procesor: AMD Athlon(tm) 64 X2 4200+
Počet jader: 2
Frekvence jader: 2,21 GHz

Paměti: Kingston KVR667D2N5K2/2G
Typ: DDR2
Uspořádání: 2x 1GB Dual Channel
Taktovací/pracovní frekvence: 333/667 MHz
Propustnost: 5,333 GB/s

Operační systém: Windows 7 Professional
Verze: x64

Testy jsem provedl na aplikaci přeložené s nastavením *Release*.

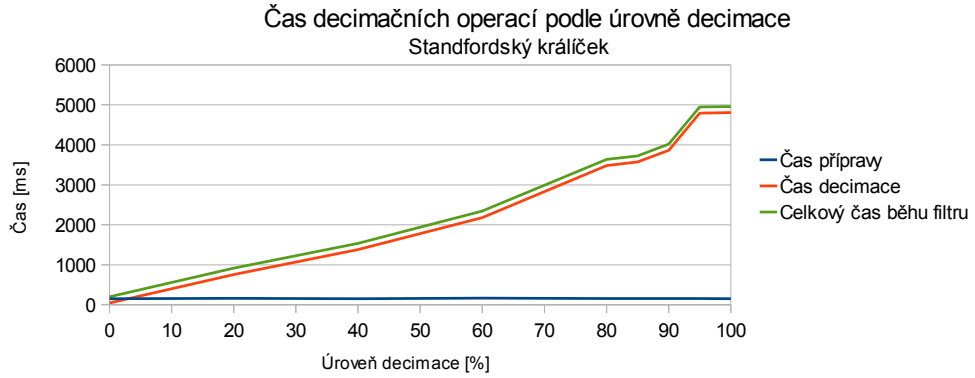
7.1 Časová náročnost

Provedl jsem zátěžové testy na třech vstupních souborech

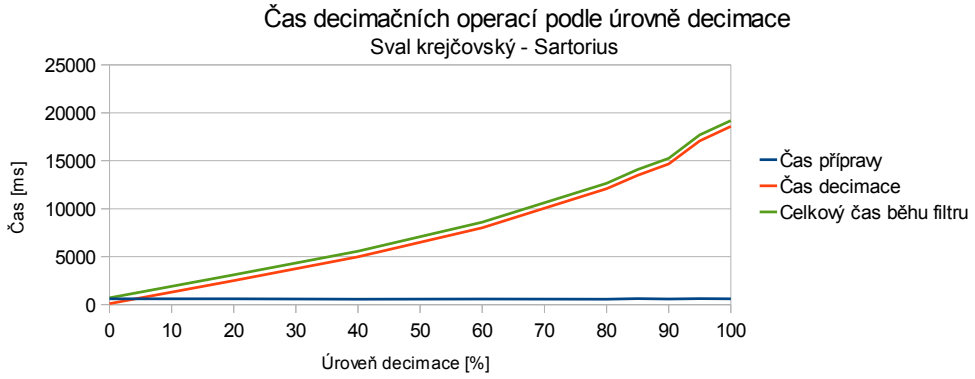
1. Redukovaný *Stanfordský králíček* - 2 503 vrcholů, 4 968 trojúhelníků, často používaný model pro testování algoritmů počítačové grafiky
2. *Sval krejčovský*(*Sartorius*) - 9 001 vrcholů, 17 982 trojúhelníků, reprezentuje běžně objemná data používaná filtrem P. Kellnhofera
3. *Pánev*(*Pelvis*) - 89 997 vrcholů, 179 994 trojúhelníků, objemná data pro velkou zátěž algoritmu

pro různé (cílové) úrovně decimace s nastavením *MeshRoughness = 0,3* a *MeshQuality = 1,0* pro všechny testy. Naměřená data jsou v tabulce B.1a a měření zobrazují grafy 7.1a, 7.1b a 7.1c.

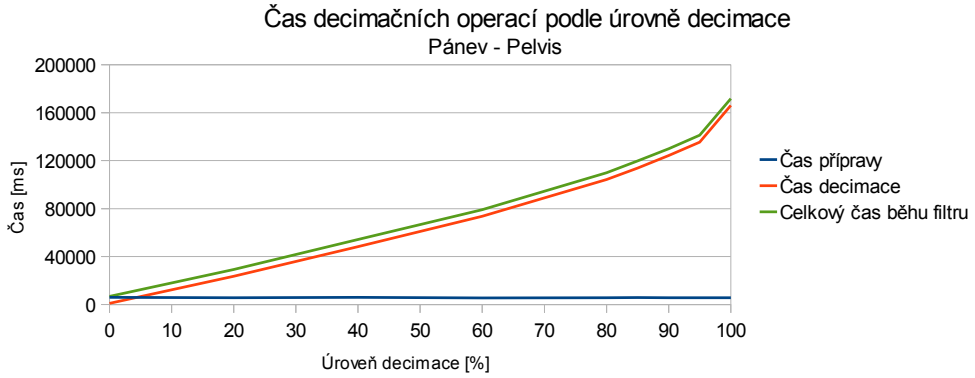
Z naměřených dat a z grafů je patrné, že časová náročnost algoritmu je téměř lineární, ale je závislá na vstupních datech. Je vidět, že počet operací a tedy i časová náročnost není přímo závislá na počtu decimovaných hran, jelikož je



(a)



(b)



(c)

Obrázek 7.1: Časy operací decimace vzhledem k úrovni decimace pro modely (a) Stanfordský králíček (b) Sval krejčovský - Sartorius (c) Pánev - Pelvis

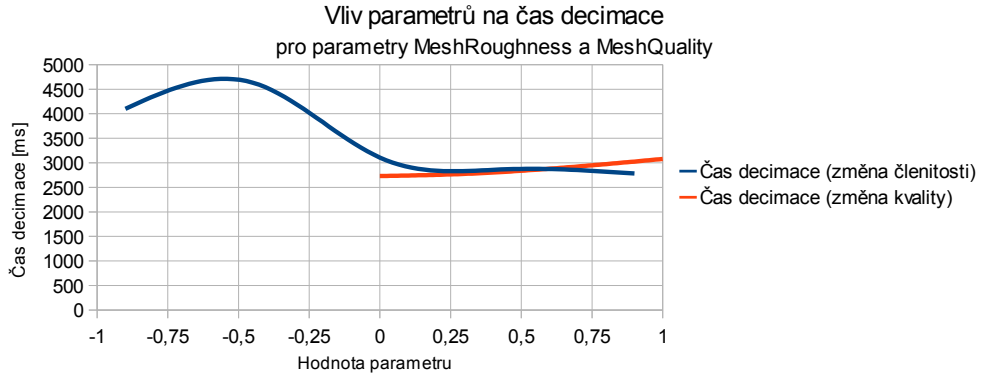
nemálo elementárních decimací v procesu zakázáno aplikací podmínek z kapitoly 4 a počet těchto zákazů v průběhu decimace roste. Pokud se decimace blíží k maximu, výpočetní náročnost stoupá nejstrměji, což je způsobeno tím, že je při velmi hluboké decimaci počet zamítnutých decimací podstatně vyšší. Dokud je

tedy síť hustá, podmínky nehrají příliš velkou roli, v malém počtu trojúhelníků jsou však nevhodné decimace pravděpodobnější a tím je jich více zamítnuto.

Z grafu 7.1a je také patrné, jak algoritmus podmínky dodržuje. Není-li žádná hrana ve frontě podle podmínek decimovatelná, decimace sítě je ukončena. Pro model *Stanfordského králíčka* je tedy nemožné pro parametry *MeshRoughness = 0,3* a *MeshQuality = 1,0* dosáhnout vyšší než 95%-ní decimace. Je také vidět, že čas potřebný pro přípravné výpočty algoritmu je konstantní a vzhledem k času potřebnému k hluboké decimaci zanedbatelný.

7.2 Vliv parametrů na rychlost decimace

Zjistili jsme, že podmínky z kapitoly 4 ovlivňují rychlost algoritmu. Provedl jsem tedy další měření na modelu *Stanfordského králíčka*, abych zjistil jakým způsobem. Cílem všech těchto měření je 80%-ní decimace.



Obrázek 7.2: Vliv parametrů na čas decimace

Z grafu 7.2 je vidět, že parametr *MeshRoughness* ovlivňuje rychlost decimace pro záporné hodnoty velmi výrazně. To je dáno tím, že pro záporné hodnoty parametru jsou povoleny pouze decimace, které jsou „hladké“ - úhel mezi trojúhelníky v okolí je po decimaci větší než $\pi/2$. Jelikož je model *Stanfordského králíčka* velmi oblý, decimace tuto podmínku často splňují, model ale obsahuje mnoho detailů (uši, tlapky, ocásek), kde tuto podmínku není možné splnit a decimace se zpomaluje. Pro hodnotu *MeshRoughness = -0.9* je také vidět, že nelze dosáhnout 80%-ní decimace (parametr je příliž omezující, pro náhled vizte kapitolu 7.3). Parametr *MeshQuality* na model *Stanfordského králíčka* příliš velký vliv nemá, jelikož je model pravidelný a neobsahuje úzké trojúhelníky. Tento algoritmus není příliš efektivní, bylo by lepší kvalitu trojúhelníků vylepšit až v postprocessingu (například posunutím některých bodů).

7.3 Vizuální porovnání

V kapitole 3.1 jsme měli možnost vidět výsledky autorů tohoto algoritmu. Předkládám tedy vizuální porovnání mého upraveného algoritmu pro porovnání. Obrázek B.1 zobrazuje obálky pro různé úrovně decimace. Nastavení parametrů je *MeshRoughness = 0* a *MeshQuality = 1*. Ač algoritmus pro tato nastavení není schopen vytvořit obálku o 38 trojúhelnících, je výpočet mnohem rychlejší, než v případě původního algoritmu (časy původního algoritmu byly pro tento model v řádech minut).

Dále přikládám vizuální porovnání vlivu parametru *MeshRoughness* na kvalitu sítě na obrázku B.2. Je vidět, že při velmi nízkém nastavení (-0,9) algoritmus decimuje pouze oblé plochy. Je ale vidět (zvláště na uších na B.2a) jak tento parametr nutí decimovanou síť lépe obklopovat původní těleso a generovat pravidelnější síť. Ideální nastavení tohoto parametru se tedy pohybuje od 0 do 0,5.

Tento parametr také může mít vliv na případný výskyt sebeprotínání. Obrázek B.3 ukazuje *Pánev*(*Pelvis*) pro *MeshRoughness* roven hodnotám 0 a -0,4. Je vidět, že dalším omezením parametru lze docílit přiblížení decimované sítě k původní síti a zamezit tak výskytu sebeprotínání.

7.4 Výsledky po začlenění filtru do třídy MeshSurface

Dle mého názoru můj filtr v současnosti produkuje kvalitní obálky. Po začlenění filtru do třídy `MeshSurface` jsem tedy očekával zlepšení výsledků celého deformačního filtru P. Kellnhofera. Hledal jsem vhodná nastavení, závěry však nejsou stoprocentní. Nejlepších výsledků jsem dosáhl s nastavením *MeshRoughness = 0,5*; *MeshQuality = 1* a zvětšení o 5 jednotek. Na obrázku B.4 je zobrazena *Stehenní kost* (*Femur*) před a po aplikaci mého filtru. Je vidět, že je zdeformovaná a „nafouklá“. V průběhu hledání jsem došel k závěru, že deformační filtr vyžaduje obálky velmi vzdálené od původní sítě. To je pro můj filtr problém, jelikož produkuje obálky, které jsou velmi blízké původní síti. Přišel jsem se třemi možnostmi řešení:

1. Upravit deformační filtr, aby byl schopen pracovat i s velmi blízkými sítěmi
2. Po vytvoření obálky mým filtrem ji zvětšit podobně jako byly zvětšovány obálky původní[14]
3. Upravit vytváření obálky tak, aby byla postupně zvětšována již při decimaci[15]

[14] Tuto možnost jsem vyzkoušel. Bohužel je ale deformační filtr velmi náchylný k chybám tam, kde v hrubé síti dochází k sebeprotínání (vizte tenké chyby na obrázku B.4a). K tomu však při zvětšení obálky často dochází.

[15] Tento přístup by eliminoval nutnost hledat sebeprotínání během decimace.

Vyzkoušel jsem tedy můj filtr pro deformaci *Svalu krejčovského* (*Sartoria*), se kterým byl problém v původní verzi deformačního filtru, kvůli kolapsu decimace na konci tohoto svalu při vytváření obálky. Je patrné zlepšení (můj filtr nekolabuje), ovšem výsledek stále není ideální, pravděpodobně kvůli sebeprotínání. Porovnání je na obrázku B.5.

8 Závěr

V průběhu práce jsem si vyzkoušel práci s knihovnami VTK a programování v jazyce C++. Naučil jsem se kombinovat více myšlenek do jednoho celku za účelem lepšího výsledku. Projekt 5 ukázal, že algoritmus *Progressive Hull* je funkční. V rámci bakalářské práce jsem jej vylepšil a optimalizoval tak, aby jeho výsledky (a výsledky implementovaného decimačního filtru) byly co nejlepší. Mnohdy šlo o metodu pokus-omyl, jelikož i malá změna v teorii či kódu má velký dopad na stabilitu/rychlost/výsledky algoritmu. Pro dodaná data se mi však podařilo odstranit většinu nedostatků algoritmu a ten nyní celkem rychle produkuje kvalitní obálky, které zachovávají tvar původního tělesa (někdy až příliš).

Jednou z mála slabin algoritmu je závislost na chybách ve vstupních datech. Největším problémem je existence lokálních míst v síti s nulovým objemem (neboli lokální místa „papírové“ tloušťky). Tento problém nyní algoritmus přeskakuje a tato lokální okolí jsou zachována v původním stavu, stejně jako u okrajů (místa, kde jsou hranice sítě, kde trojúhelníky nemají další sousedy).

V implementační části jsem vyzkoušel použití algoritmu v praxi, ukázalo se však, že současná verze vhodná pro požadované účely vhodná zcela není, i když slibuje lepší výsledky než původně použitý postup. Výsledky, kterých jsem dosáhl však naznačují, že bude nutná pouze malá úprava algoritmu, několik návrhů této úpravy jsem již naznačil v kapitole 7.4. Je třeba zmínit, že rychlost algoritmu je extrémně závislá na způsobu překladu (*Debug/Release* nastavení překladače).

Body zadání týkající se vytváření vnějších obálek těles reprezentovaných triangulovanými sítěmi se zachováním detailů se mi dle mého názoru podařilo splnit, výsledky po začlenění do programového vybavení P. Kellnhofera však nejsou tak kvalitní, jak bylo očekáváno.

Sazba tohoto dokumentu byla vytvořena pomocí nástroje LaTeX s použitím programu TeXnicCenter.

Reference

[1] Bell, J. T.: Visualization Toolkit (VTK) Tutorial. [Online], 2004, platné k 18.4.2011.
URL `http://www.cs.uic.edu/~jbell/CS526/Tutorial/Tutorial.html`

[2] Berkelaar, M.; Dirks, J.; Eikland, K.; aj.: lp_solve API reference. [Online], 2010.
URL `http://lpsolve.sourceforge.net/5.5/`

[3] Chvátal, V.: *Linear Programming*. W. H. Freeman and Company, New York, 1983, ISBN 0-71671-587-2, 478 s.

[4] Edelsbrunner, H.: The union of balls and its dual shape. *Discrete Computational Geometry*, ročník 13, 1995: s. 415–440.

[5] Edelsbrunner, H.; Mücke, E. P.: Three-dimensional alpha shape. *ACM Transactions on Graphics*, ročník 13, č. 1, 1994: s. 43–72.

[6] Goldsmith, J.; Salmon, J.: Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Comput. Graph. Appl.*, ročník 7, May 1987: s. 14–20, ISSN 0272-1716, doi:10.1109/MCG.1987.276983.
URL `http://portal.acm.org/citation.cfm?id=31468.31469`

[7] Guéziec, A.: Locally Toleranced Surface Simplification. *IEEE Transactions on Visualization and Computer Graphics*, ročník 5, Duben 1999: s. 168–189, ISSN 1077-2626, doi:10.1109/2945.773810, kapitola 5: Description of the Algorithm, strana 11, třetí test.
URL `http://portal.acm.org/citation.cfm?id=614274.614429`

[8] van Heesch, D.: VTK 4.0.2 Documentation. [Online], 2001, revize 1.1080.2.1, platná k 12.10.2010.
URL `http://www.vtk.org/doc/release/4.0/html/`

[9] Hoppe, H.: New quadric metric for simplifying meshes with appearance attributes. In *Proceedings of the 10th IEEE Visualization 1999 Conference (VIS '99)*, VISUALIZATION '99, Washington, DC, USA: IEEE Computer Society, 1999, ISBN 0-7803-5897-X, s. –.
URL `http://portal.acm.org/citation.cfm?id=832273.834119`

[10] Kellnhofer, P.: *Deformace povrchového modelu se zachováním objemu*. Bakalářská práce, Západočeská univerzita, Plzeň, Česká Republika, 2010.

[11] Kellnhofer, P.: *Deformace povrchového modelu se zachováním objemu*. Bakalářská práce, Západočeská univerzita, Plzeň, Česká Republika, 2010, kapitola 5.4: Použití hrubé sítě.

[12] Kellnhofer, P.: *Deformace povrchového modelu se zachováním objemu*. Bakalářská práce, Západočeská univerzita, Plzeň, Česká Republika, 2010, kapitola 5.2.1: Barycentrické souřadnice pro 3D sítě.

[13] Kolektiv autorů: VTK File Formats for VTK Version 4.2. [Online], Březen 2010, dostupné online.
URL http://www.vtk.org/VTK/img/file-formats.pdf

[14] Lambert, T.: Convex Hull Algorithms. [Online], 1998, platné k 27. 4. 2011.
URL http://www.cse.unsw.edu.au/~lambert/java/3d/hull.html

[15] Lindstrom, P.; Turk, G.: Fast and Memory Efficient Polygonal Simplification. *Visualization Conference, IEEE*, ročník 0, 1998: s. 279+, doi: 10.1109/VISUAL.1998.745314.
URL http://dx.doi.org/10.1109/VISUAL.1998.745314

[16] Žára, J.; Beneš, B.; Sochor, J.; aj.: *Moderní počítačová grafika*, kapitola 22.2.4. Brno: Computer Press, druhé, přepracované a rozšířené vydání, 2005, ISBN 80-251-0454-0, str. 558.

[17] Žára, J.; Beneš, B.; Sochor, J.; aj.: *Moderní počítačová grafika*, kapitola 22.2.3. Brno: Computer Press, druhé, přepracované a rozšířené vydání, 2005, ISBN 80-251-0454-0, str. 557.

[18] Reddy, M.: Wavefront Object Files. [Online], 1994, platné k 18.4.2011.
URL http://www.martinreddy.net/gfx/3d/OBJ.spec

[19] Sander, P. V.; Gu, X.; Gortler, S. J.; aj.: Silhouette clipping. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, ISBN 1-58113-208-5, s. 328–329, doi:http://dx.doi.org/10.1145/344779.344935.
URL http://dx.doi.org/10.1145/344779.344935

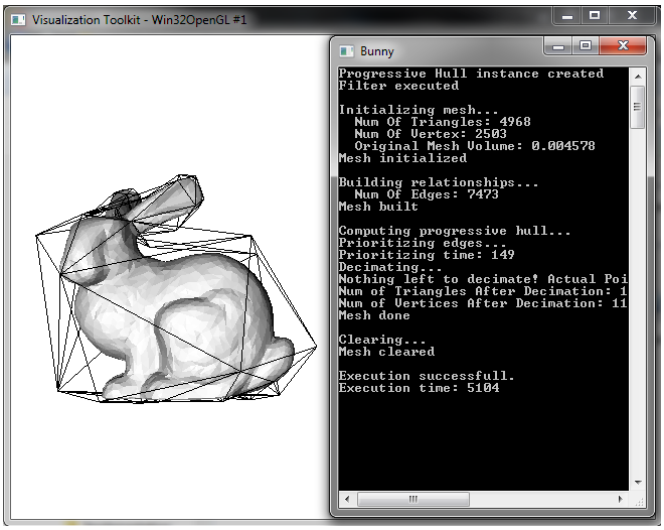
[20] Surhone, L.; Tennoe, M.; Henssonow, S.: *Octree*. VDM Verlag Dr. Mueller AG & Co. Kg, 2010, ISBN 9786131288753.
URL http://books.google.com/books?id=UfiicQAACAAJ

Přílohy

A Uživatelská příručka

Instalace programu není potřeba, jde o přeložené binární soubory. Veškeré knihovny jsou v přeložené podobě přiloženy. Pokud by to však bylo vyžadováno, je nutné stáhnout z webu `http://www.vtk.org/` zdrojové soubory a přeložit je za pomoci programu CMake. Aplikace je kompatibilní s VTK verze 4.2 a vyšší.

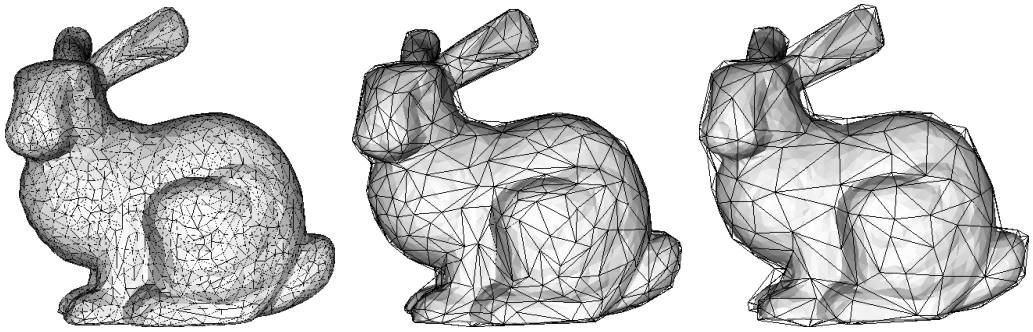
Program `DemoApp.exe` se spouští jako konzolová aplikace, vyžaduje parametry s názvem vstupního souboru, úroveň decimace, členitost sítě (-1,0 - 1,0) a kvalita sítě (0 - 1,0). Volitelně lze definovat název výstupního souboru, do kterého je výsledek uložen ve formátu VTK (pokud je specifikován soubor *.vtk) nebo jako obrázek do souboru PNG (pokud je specifikován soubor *.png). Při nastavení výstupního souboru je běh programu ukončen po decimaci. Statistiky běhu jsou ukládány do souboru `Log.txt` Běh programu je zachycen na obrázku A.1. Formát vstupních parametrů: *DemoApp.exe název_vstup úroveň_decimace členitost_sítě kvalita_sítě [název_výstup]*



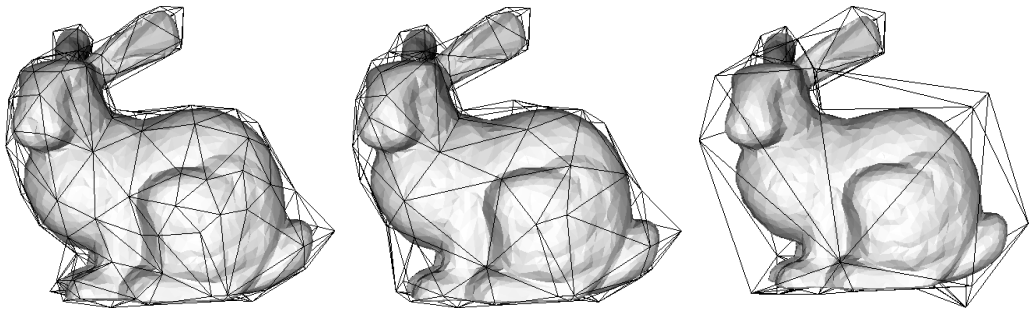
Obrázek A.1: Běh programu

Po zobrazení okna s vizualizací dat je možné modelem otáčet, přibližovat jej a posouvat myší. Ovládání je závislé na použité verzi VTK, ale je velmi intuitivní.

B Obrázky a tabulky

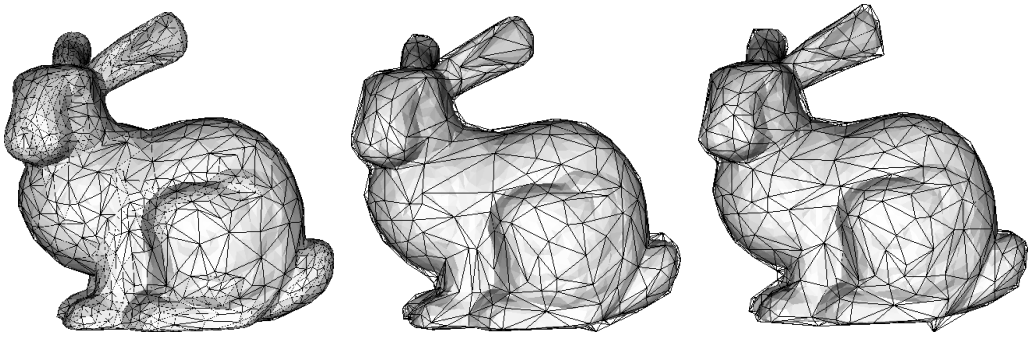


(a) 2503 (b) 599 (c) 300



(d) 199 (e) 149 (f) 93

Obrázek B.1: Výsledky algoritmu pro různé úrovně decimace. Číselný údaj udává počet vrcholů obálky.



(a) MR: -0,9 n: 1568 (b) MR:0 n: 500 (c) MR:0,9 n: 500

Obrázek B.2: Výsledky algoritmu pro různé úrovně parametru *MeshRoughness*. Číselný údaj udává nastavení tohoto parametru a počet vrcholů obálky. Cílová decimace byla pro všechny testy 80%-ní, kvůli velkému omezení v případě (a) však požadovaná decimace nebyla možná.

Model	Úroveň decimace	0,00%	20,00%	40,00%	60,00%	80,00%	85,00%	90,00%	95,00%	Maximální
Bunny	Vrcholů po decimaci	2503	2002	1501	1001	500	375	250	140	140
	Čas přípravy [ms]	153	160	153	165	156	154	155	156	153
	Čas decimace [ms]	45	757	1382	2181	3479	3571	3866	4792	4808
	Celkový čas [ms]	198	917	1535	2346	3635	3725	4021	4948	4961
	Čas/Počet decimací	-	1,51	1,38	1,45	1,74	1,68	1,72	2,03	2,03
Sartorius	Vrcholů po decimaci	9001	7200	5400	3600	1800	1350	900	450	336
	Čas přípravy [ms]	601	613	577	589	565	621	583	623	601
	Čas decimace [ms]	105	2508	4992	8014	12085	13467	14667	17064	18586
	Celkový čas [ms]	706	3121	5569	8603	12650	14088	15250	17687	19187
	Čas/Počet decimací	-	1,39	1,39	1,48	1,68	1,76	1,81	2,00	2,14
Pelvis	Vrcholů po decimaci	89997	71997	53998	35998	17999	13499	8999	4499	310
	Čas přípravy [ms]	5917	5651	5999	5536	5581	5892	5685	5754	5689
	Čas decimace [ms]	988	23548	48319	73550	104286	113801	124232	135456	166093
	Celkový čas [ms]	6905	29199	54318	79086	109867	119693	129917	141210	171782
	Čas/Počet decimací	-	1,31	1,34	1,36	1,45	1,49	1,53	1,58	1,85

(a)

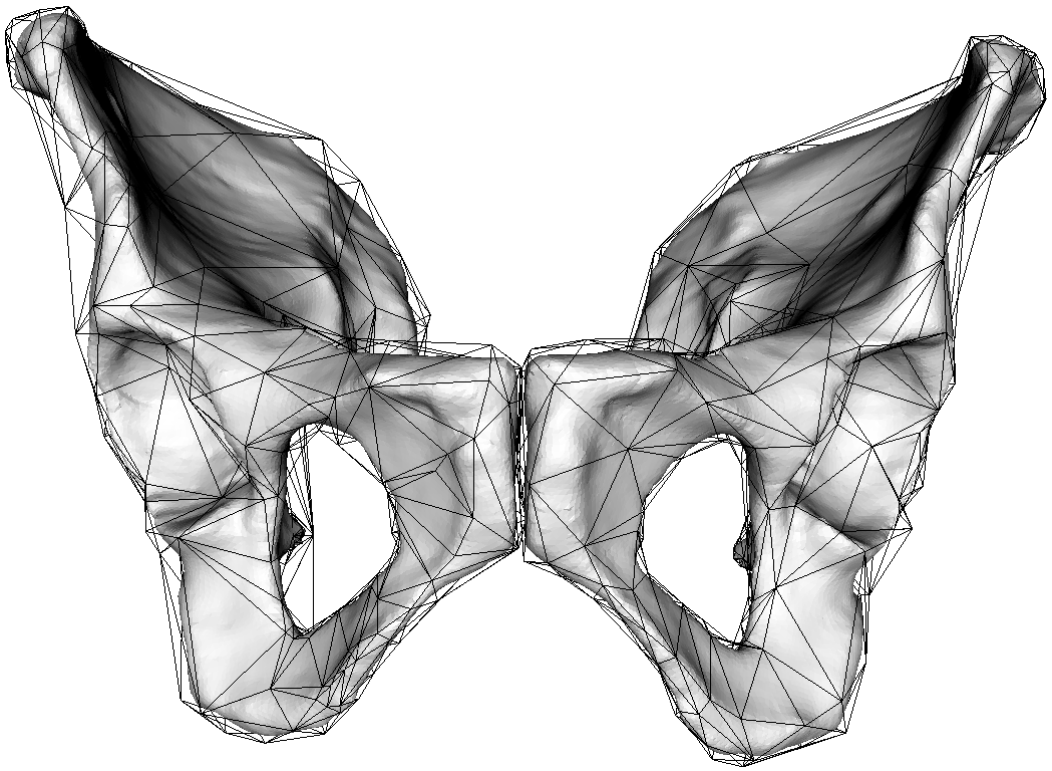
Členitost sítě	-0,9	-0,45	0	0,45	0,9
Zbylých vrcholů	1568	500	500	500	500
Čas decimace	4102	4634	3106	2869	2781

(b)

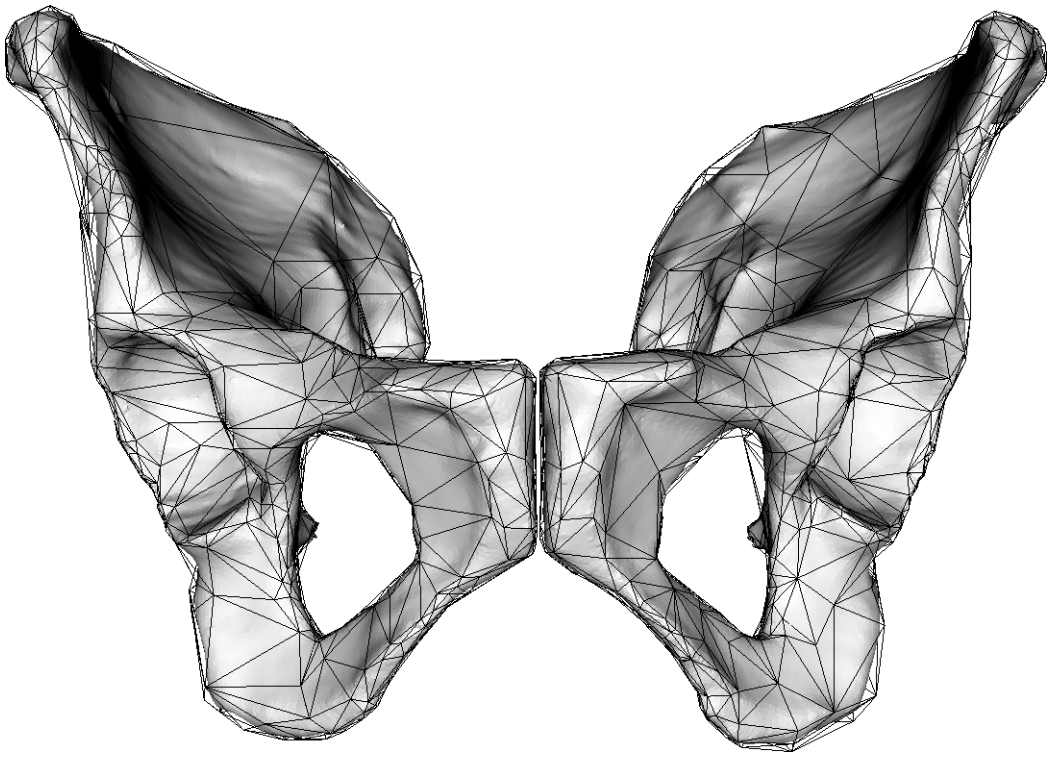
Kvalita sítě	0	0,25	0,5	0,75	1
Zbylých vrcholů	500	500	500	500	500
Čas decimace	2731	2763	2838	2949	3078

(c)

Tabulka B.1: (a) Rychlost algoritmu pro různé úrovně decimace; Vliv parametru *MeshRoughness* (b) a *MeshQuality* (c) na čas decimace

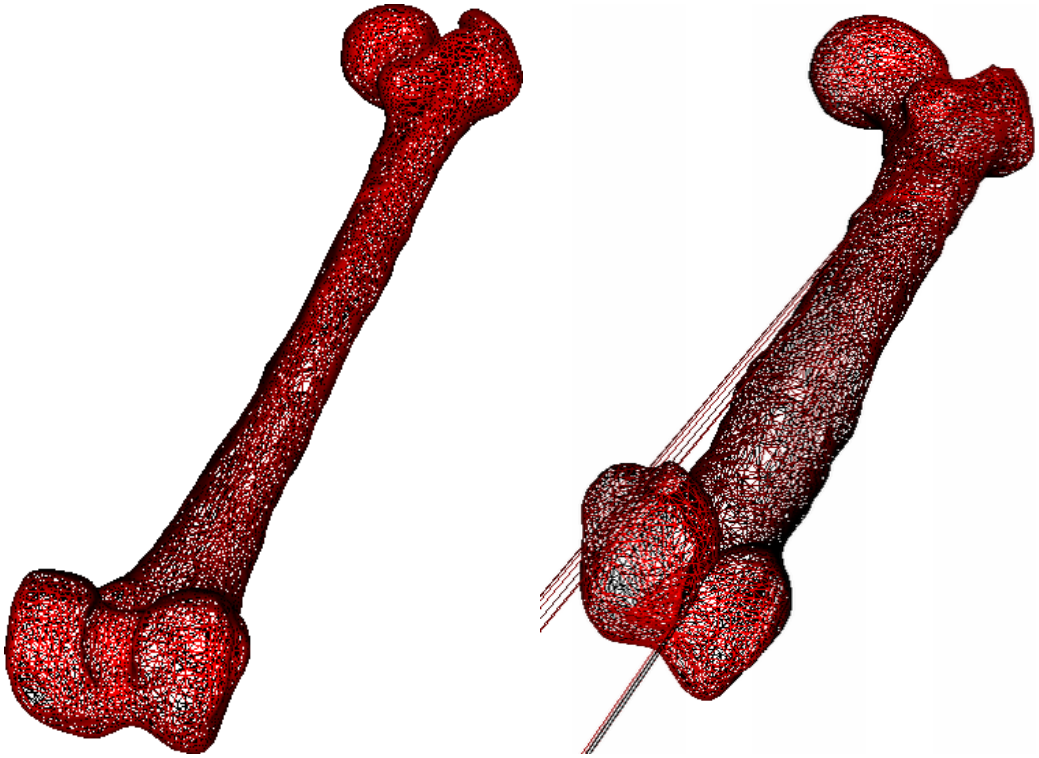


(a) 0



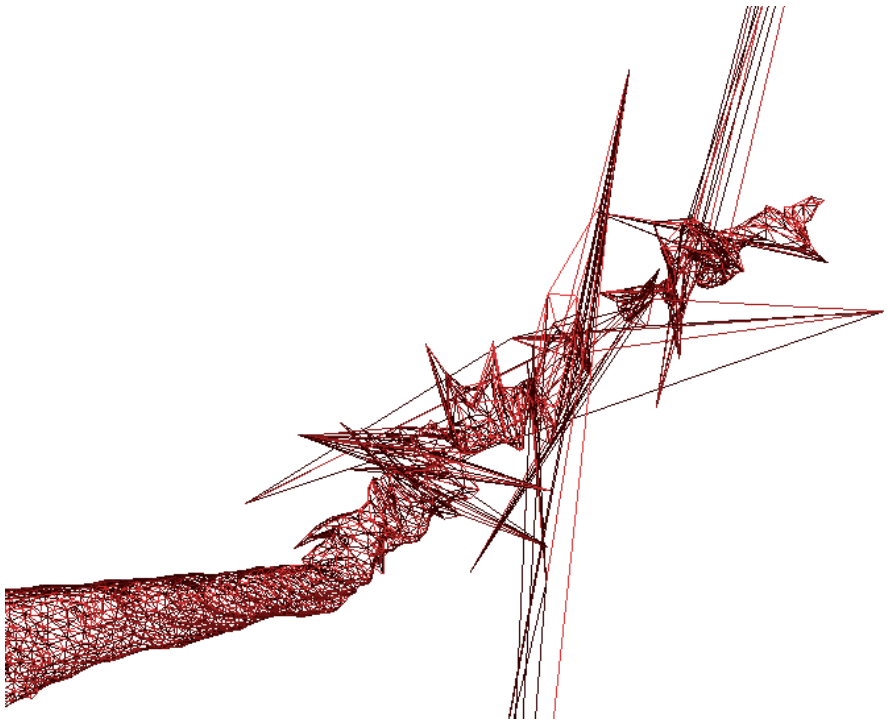
(b) -0,4

Obrázek B.3: Výsledky algoritmu pro různé úrovně parametru *MeshRoughness* pro zabránění sebeprotínání. Číselný údaj udává nastavení tohoto parametru.

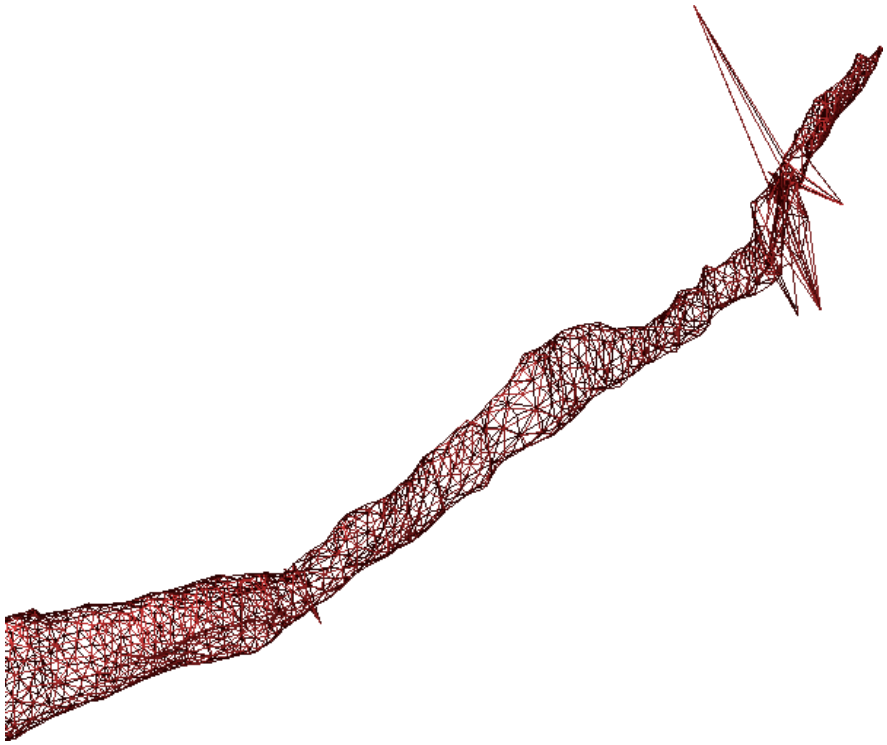


(a) (b)

Obrázek B.4: Výsledky deformace pomocí filtru `vtkMEDPolyDataDeformationPK` pro *Stehenní kost* (*Femur*). (a) Pomocí původní hrubé sítě (b) Pomocí mého decimačního filtru



(a)



(b)

Obrázek B.5: Výsledky deformace pomocí filtru `vtkMEDPolyDataDeformationPK` pro *Sval krejčovský* (*Sartorius*). (a) Pomocí původní hrubé sítě (b) Pomocí mého decimačního filtru