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

BAKALÁŘSKÁ PRÁCE

Webový přehrávač komprimovaných dynamických sítí s implementací plného algoritmu EdgeBreaker

Plzeň 2008 Jan RUS

VLOŽENÝ LIST S ORIGINÁLNÍM

ZADÁNÍM PRÁCE

Prohlášení

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

V Plzni dne ---------- ----------

Anotace

Tato bakalářská práce se zabývá kompresí dynamických sítí algoritmem Coddyac, který ke kompresi topologie sítě využívá metodu EdgeBreaker.

Cílem práce je implementace algoritmu a vytvoření webové aplikace, která demonstruje jeho účinnost.

Na začátku je uveden popis důležitých algoritmů, které Coddyac používá, potom následuje popis programového řešení a diskuze výsledků testování programů.

Klíčová slova:
Komprese, dynamické sítě, EdgeBreaker, Coddyac

Abstract

This bachelor thesis deals with compression of dynamic meshes using the Coddyac algorithm which compresses the mesh topology by the EdgeBreaker method.

The purpose of this work is to implement the algorithm and a web application that demonstrates its efficiency.

In the beginning we introduce the important algorithms used by Coddyac, afterwards follows a description of the program solution and discussion of testing result.

Keywords:
Compression, dynamic meshes, EdgeBreaker, Coddyac

Obsah

1 Úvod 6

2 EdgeBreaker 7

2.1 CornerTable (CT) 7
2.2 Komprese 9
2.3 Dekomprese 10
2.4 Handles 12
2.5 Holes 14
2.6 Vícekomponentové modely 16

3 Principal Component Analysis (PCA) 17

4 Algoritmus Coddyac 18

5 Platformy pro webovou aplikaci 20

6 Popis programového řešení pro Mve-2 21

6.1 Compressor 22
6.1.1 Inicializace kompresního algoritmu 23
6.1.2 Průběh komprese 24
6.1.3 Odhad a komprese koeficientů trajektorie vrcholu 25
6.2 Decompressor 27
6.2.2 Průběh dekomprese 28
6.2.3 Metoda zip 30
6.2.4 Metoda checkHandle 31
6.2.5 Metoda createHoleTopology 31
6.3 Saver 33
6.4 Loader 33
6.5 PCA a ArithCoder 34

7 Popis programového řešení webové aplikace 36

8 Výsledky testů a měření 38

9 Závěr 40

Přehled zkratek 41

Použitá literatura 42

Přílohy 43

1 Úvod

Dnešní doba je dobou počítačů, internetu a dat v elektronické podobě. Díky nárůstu výpočetního výkonu současných počítačů se v několika posledních letech dostávají do popředí technologie pracující s daty popisujícími 3D prostor. Takovými daty mohou být například trojrozměrné mapy světa, trojrozměrné modely v počítačových hrách, počítačové modely výrobků strojního průmyslu, nebo výstupní data z počítačového tomografu.

Nároky na přesnost a množství dat neustále rostou a s nimi i objemy souborů, které tato data obsahují. Jelikož kapacity diskových jednotek a přenosové rychlosti počítačových sítí jsou omezené, je třeba využít kompresních algoritmů, které by objem 3D dat zmenšily a snížily tak hardwarové nároky na jejich skladování a sdílení.

Jistě jedněmi z nejznámějších kompresních algoritmů jsou algoritmy ZIP a RAR. Těmito algoritmy lze komprimovat libovolný druh dat, ale platí, že čím více víme o konkrétním typu dat, tím většího kompresního poměru můžeme dosáhnout. Bude tedy lepší použít nějaký specializovaný kompresní algoritmus, který dosáhne mnohem lepšího kompresního poměru.

Takovým kompresním algoritmem je například algoritmus Coddyac [9], který je specializovaný na kompresi trojrozměrných animací (dynamických trojúhelníkových sítí).

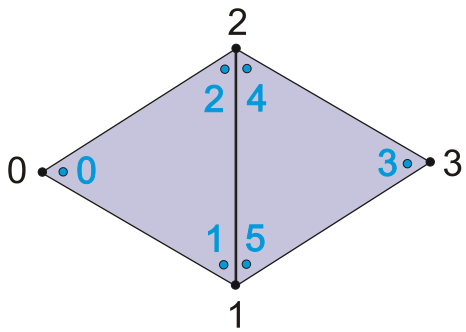
Cílem této práce je implementace kompresního algoritmu Coddyac ve formě modulu pro Mve-2, prostředí vyvíjeného na KIV/ZČU. Následně, po důkladném otestování správné funkčnosti, vytvoření webové aplikace, umožňující dekompresi a základní vizualizaci dynamických sítí komprimovaných algoritmem Coddyac, která by demonstrovala účinnost komprese a rychlost dekomprese tímto algoritmem.

2 EdgeBreaker

Algoritmus zvaný EdgeBreaker, slouží ke kompresi (dekompresi) trojúhelníkových sítí. Přesněji řečeno, ke kompresi (dekompresi) konektivity trojúhelníků dané sítě. Pro popis konektivity trojúhelníků používá tato metoda velmi jednoduchou strukturu, nazývanou CornerTable[1]. Algoritmus EdgeBreaker je jedním ze stěžejních algoritmů kompresního algoritmu Coddyac[8].

2.1 CornerTable (CT)

Základním pojmem, se kterým CT pracuje, je tzv. corner (česky roh). Označením corner je míněn jeden z vrcholů patřících konkrétnímu trojúhelníku. Máme-li dva trojúhelníky, s jednou společnou hranou, můžeme je sestavit ze 4 bodů trojúhelníkové sítě. Každý trojúhelník má 3 cornery, tedy máme 4 body, ale 6 cornerů, jak je znázorněno na obr. 2.1.



obr. 2.1

modře vyznačené cornery

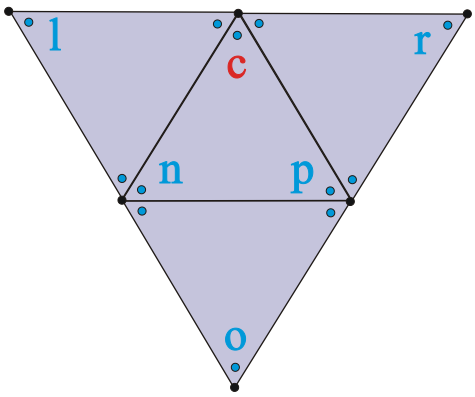
Všechny vrcholy sítě, cornery a trojúhelníky jsou strukturou CT indexovány nezápornými celými čísly. Každý trojúhelník je definován třemi cornery. Ty jsou v trojúhelníku postupně indexovány tak, že první trojúhelník obsahuje cornery 0, 1 a 2, druhý trojúhelník obsahuje cornery 3, 4, 5 atd. Díky tomu je snadné rozpoznat, kterému trojúhelníku patří zadaný corner. Stačí, když index corneru vydělíme beze zbytku číslem 3. Výsledek je indexem hledaného trojúhelníku.

Druhým významným pojmem je oposite corner. Corner a oposite corner jsou ty cornery, které patří dvěma sousedním trojúhelníkům a zároveň nejsou součástí jejich společné hrany. Na obr 2.1 corneru s indexem 0 přísluší oposite corner s indexem 3 a naopak. Tento vztah tvoří základ CT. Informace o corner a oposite cornerech jsou ukládány do dvou tabulek označovaných jako V a O. Tabulka O obsahuje indexy oposite corneru k corneru daného indexem tabulky O. Tzn. pokud O[2] = 7, znamená to, že corneru s indexem 2 náleží oposite corner s indexem 7. A tabulka V obsahuje ukazatele na prostorové souřadnice corneru daného indexem tabulky V. Tzn. pokud V[6] = 5, znamená to, že corner s indexem 6 má prostorové souřadnice uloženy v tabulce prostorových souřadnic všech vrcholů trojúhelníkové sítě v položce s indexem 5. Tato tabulka bývá označována jako G (geometry), nebo vertices.

Cornery trojúhelníku jsou indexovány postupně proti směru hodinových ručiček. Díky tomu, že je pořadí cornerů pevně dáno, můžeme nadefinovat i operace next a previous, které určí následující a předešlý corner ke corneru zadanému. V jazyce C# lze např. operaci next implementovat jako:

```
private int next(int corner) {
        if (corner % 3 == 2)            // 3. vrchol trojuhelniku, nasledujici je 1. vrchol
         return (c - 2); else  return (c + 1);
}
```

Mimo cornery oposite, next a previous, jsou využívané také cornery označené jako left (na obr.2.2 písmenko l) a right (na obr.2.2 písmenko r). Right corner lze popsat jako o(n(c)), kde o(c) je oposite corner k corneru c a n(c) je následující corner corneru c, tj. next corner. Analogicky lze popsat i left corner jako o(p(c)).

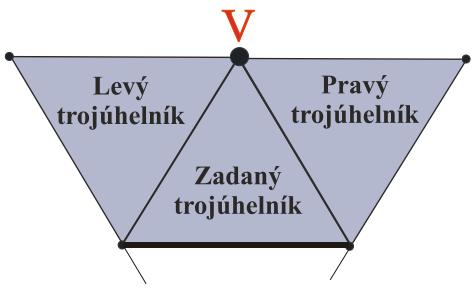


obr. 2.2

označení cornerů vzhledem k červeně označenému corneru *c*

2.2 Komprese

Během komprese algoritmus EdgeBreaker prochází trojúhelníkovou sítí z trojúhelníku na trojúhelník. Aby poznal, na který další trojúhelník může přejít, označuje všechny navštívené trojúhelníky a jejich vrcholy. Nechť levý trojúhelník a pravý trojúhelník, jsou dva trojúhelníky, které těsně přiléhají k danému trojúhelníku a vrchol V je vrchol, který je společný pro všechny tři trojúhelníky, jak je znázorněno na obr. 2.3.

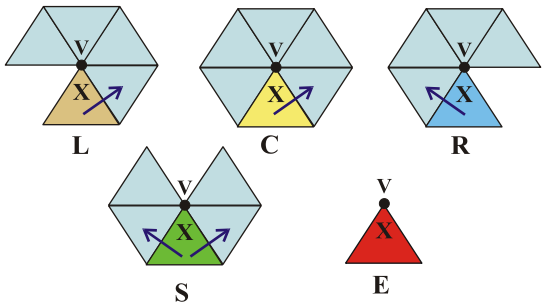


obr. 2. 3

označení trojúhelníků, tučně je vyznačena hrana překročená algoritmem EdgeBreaker

Při průchodu trojúhelníky algoritmus rozlišuje 5 případů. Pokud vrchol V ještě nebyl navštíven, pak nemohl být navštíven ani pravý a levý trojúhelník. Tento případ je označován jako C. Pokud byl vrchol V už navštíven, rozlišujeme dále zbývající 4 případy. Byl –li už navštíven levý trojúhelník, jedná se o případ L, pokud byl navštíven pravý trojúhelník, jedná se o případ R. Pokud byl navštíven levý i pravý trojúhelník, jedná se o případ E a pokud nebyl navštíven ani pravý ani levý trojúhelník, jedná se o případ S. Případy jsou znázorněny na obr. 4, již navštívené trojúhelníky nejsou vykresleny a směr následujícího průchodu trojúhelníky je vyznačen šipkou.

Výjimečný je případ S a to v tom, že algoritmus musí síť projít ve dvou směrech, to je řešeno tak, že pro směr vpravo je spuštěn algoritmus rekurzivně a po skončení rekurze, pokračuje vlevo. Jak algoritmus postupně prochází trojúhelníky v trojúhelníkové síti, jsou případy z obr. 2.4 zapisovány do tzv. CLERS řetězce, který tak popisuje konektivitu trojúhelníků v trojúhelníkové síti –topologii. Data zkomprimované sítě se tedy skládají z CLERS řetězce a seznamu prostorových souřadnic vrcholů trojúhelníkové sítě.



obr. 2.4

znázornění stavů při průchodu sítí, převzato z [3]

Podrobný popis komprese lze nalézt v [1], [3] a [4].

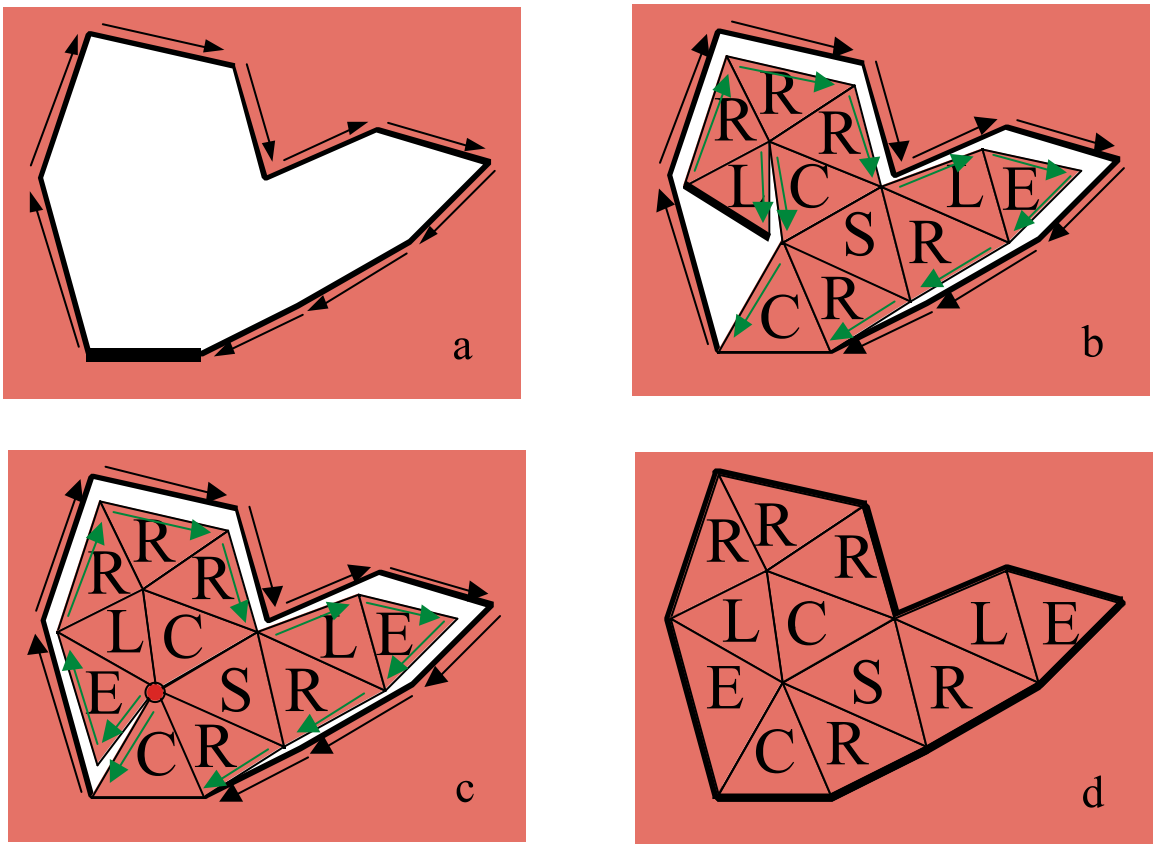
2.3 Dekomprese

Během dekomprese algoritmus EdgeBreaker postupně načítá symboly ze zadaného CLERS řetězce a seznamu prostorových souřadnic všech vrcholů trojúhelníkové sítě a sestavuje síť, která je shodná s původní trojúhelníkovou sítí. Tento proces se skládá ze dvou základních fází. Tyto fáze jsou nazývány wrapping a zipping. Fáze nazývaná wrapping je velmi podobná algoritmu komprese, jen namísto označování prošlých trojúhelníků, je s každým načteným symbolem z CLERS řetězce vytvořen nový trojúhelník a následně připojen k ostatním vytvořeným. Je-li vytvořen trojúhelník po načtení symbolu C, zároveň se načte i souřadnice jeho vrcholu, který dosud nebyl známý.

Protože je každý nový trojúhelník připojen jen ke svému přímému předchůdci, vznikají v povrchu sestavovaného modelu trhliny, které je třeba zacelit. K tomu slouží fáze zipping. Při této fázi jsou procházeny hrany trojúhelníků, které nejsou spojené s jiným trojúhelníkem a pro každou z nich je hledán jejich protějšek –hrana, se kterou bude spojena. Aby bylo možné rozpoznat, které dvě hrany k sobě patří, jsou tyto hrany již ve fázi wrapping označeny. Na následující obr. 2.5, na kterém je znázorněna celá sestavená síť, ve které je díra obr. 2.5 (a). Tato díra je v síti pouze z toho důvodu, že algoritmus dekomprese ještě nebyl dokončen.

Ze stavu znázorněném v obr. 2.5 (a), bude algoritmus pokračovat ze zvýrazněné hrany. Postupně bude připojovat trojúhelníky obr. 2.5 (b), až do chvíle, kdy načte z CLERS řetězce symbol E, kterým díru uzavře obr. 2.5 (c). V tuto chvíli je fáze wrap ukončena a s pomocí zippingu, je ve směru šipek zacelena vzniklá trhlina obr. 2.5 (d).

Na obr. 2.5 je také znázorněno, jak byly hrany během fáze wrap označeny. Původní hrany černě, nově vzniklé zeleně. Všimněte si, že během fáze zip, jsou spojovány pouze hrany s šipkami, které ukazují stejným směrem. Spojování hran začíná od vrcholu označeného červeným kolečkem a až poté, co byl dekomprimován trojúhelník typu E.



obr. 2.5

znázornění průběhu fází *wrap* a *zip*, převzato z [2]

Podrobný popis dekomprese lze nalézt v [2] a [3].

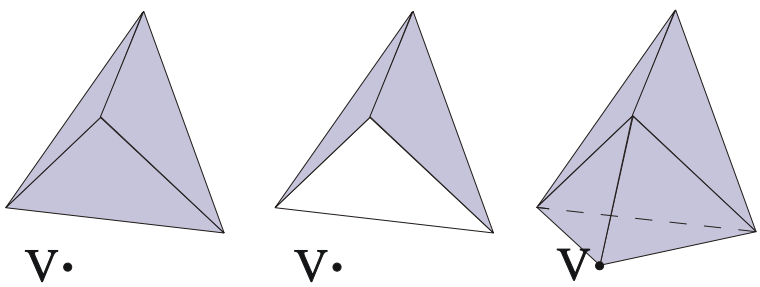
2.4 Handles

Ve stručném shrnutí algoritmů komprese a dekomprese, byl popisován pouze postup pro práci s modely (trojúhelníkovými sítěmi), které jsou typu simple mesh, tj. jejich topologie je shodná s topologií koule. To způsobuje značné omezení, co se tvaru komprimovaných objektů týče. Proto byla vymyšlena úprava EdgeBreaker algoritmu, která umožňuje pracovat s modely, které jsou nejen simple mesh, ale navíc obsahují tzv. handle. Handle si lze představit např. jako ucho u džbánu (obr. 2.7 (a)), nebo klasickou americkou koblihu (donnut), odborně se tento tvar nazývá torus. Objekty s handle tedy obsahují jednu nebo více děr, ale tyto díry nejsou vytvořeny pouhým odstraněním několika trojúhelníků z povrchu modelu, ale jsou do něj „vmodelovány". Takovéto objekty, ať už handle obsahují, nebo neobsahují, lze označit jako korektní. Korektní modely jsou takové modely, jejichž všechny trojúhelníky mají právě tři sousední trojúhelníky a každý z nich přiléhá k právě jedné hraně daného trojúhelníka.

V souvislosti s korektními modely stojí za zmínku výraz genus. Nejjednodušším simple mesh, který lze sestavit, je tetrahedron (čtyřstěn). Z tetrahedronu lze vytvořit libovolný složitější simple mesh jednoduchými úpravami. Jedna taková úprava obnáší přidání nového vrcholu V k modelu, odstranění jednoho trojúhelníku z povrchu modelu a jeho nahrazení třemi novými trojúhelníky se společným vrcholem V (obr. 2.6). Po této úpravě v modelu přibyl jeden vrchol, dvě stěny (trojúhelníky) a tři hrany. Z toho, vzhledem k předchozímu stavu modelu (v tomto případě tetrahedron), můžeme sepsat následující rovnost:

poč. vrcholů + poč. stěn = poč. hran + 2 => poč. vrcholů + poč. stěn – poč. hran – 2 = G

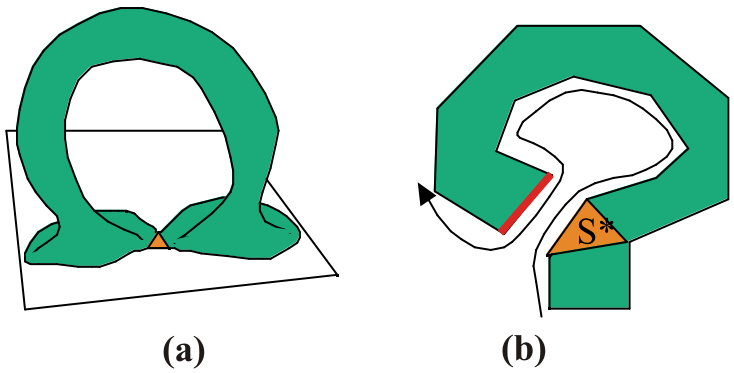
Hodnota G je genus. Z předešlého výpočtu lze zjistit, že genus simple mesh je roven 0. Pokud se v modelu vyskytne handle, je genus nenulový a jeho hodnota odpovídá počtu výskytů handle v modelu.



obr. 2.6
úprava tetrahedronu

Algoritmus EdgeBreaker dokáže handle rozpoznat relativně snadno. Při průchodu trojúhelníkovou sítí totiž pouze v případě výskytu handle může dojít k situaci, kdy je označen trojúhelník symbolem S, tím je algoritmus rekurzivně spuštěn přes pravý sousední trojúhelník a zakončen je u levé hrany trojúhelníku S, ze kterého algoritmus startoval, vizte obr. 2.7 (b). Pak je zbytečné po ukončení rekurzivního spuštění algoritmu pokračovat z daného S trojúhelníku přes jeho levého souseda. Případ výskytu handle musí být dobře ošetřen, protože zatím jediný případ, při kterém bylo možno ukončit rekurzivně spuštěný algoritmus, byl případ E. Proto navštíví-li algoritmus trojúhelník typu S, kontroluje po návratu z rekurzivního zkoumání přes pravého souseda i jeho levého souseda, který by mohl být již navštíven. Byl-li navštíven, algoritmus se ihned ukončí. Trojúhelník, kterým je rekurzivně spuštěný algoritmus ukončen, tedy nemusí být pouze typu E, ale může být i typu S.

Algoritmus EdgeBreaker si proto ukládá levou hranu (ve skutečnosti oposite cornery, mezi kterými hrana leží) takového S trojúhelníku do pole nazývaného handles.



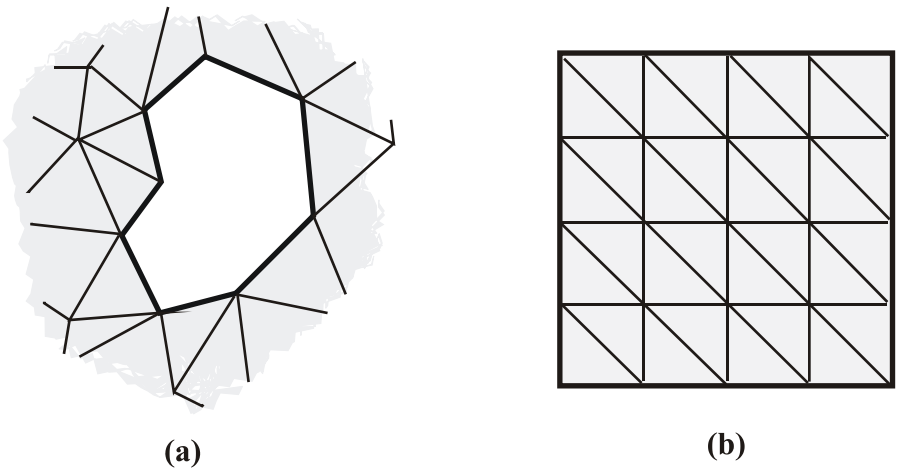
obr. 2.7

znázornění handle, převzato z [2]

Podrobný popis handles lze nalézt v [2] a [4].

2.5 Holes

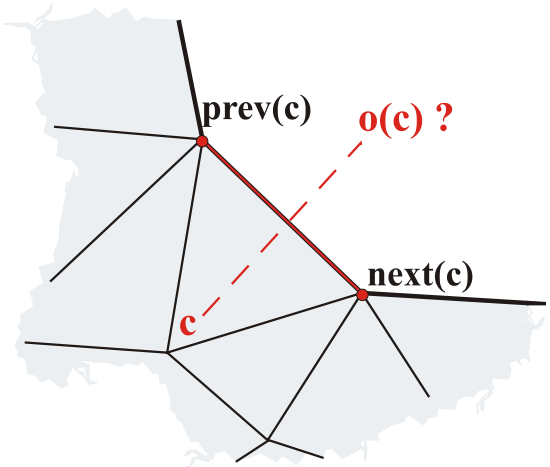
Dalším vhodným rozšířením algoritmu EdgeBreaker je umožnění práce s modely, jejichž povrch obsahuje tzv. hole (díru), popřípadě holes (díry). Jak může hole vypadat je zobrazeno na obr. 2.8 (a). Hole se dá rovněž chápat jako hranice (obr. 2.8 (b)). Narozdíl od handles nevzniká tato díra vmodelováním do modelu, ale odstraněním trojúhelníků z jeho povrchu. Model tedy obsahuje také takové trojúhelníky, k jejichž některým hranám již další trojúhelník nepřiléhá.



obr. 2.8

hole jako díra a hranice

Prvním důležitým krokem, který algoritmus musí udělat, je identifikace všech holes. Identifikace je velmi jednoduchá, hole je totiž obklopena trojúhelníky, jejichž některé cornery c nemají v tabulce O své oposite cornery. Případ je znázorněn na obr. 2.9. Z toho lze snadno vyvodit, že vrcholy v next(c) a prev(c) a hrana mezi nimi jsou součástí obvodu hole. K identifikaci holes zpravidla dochází již ve fázi vytváření CornerTable.

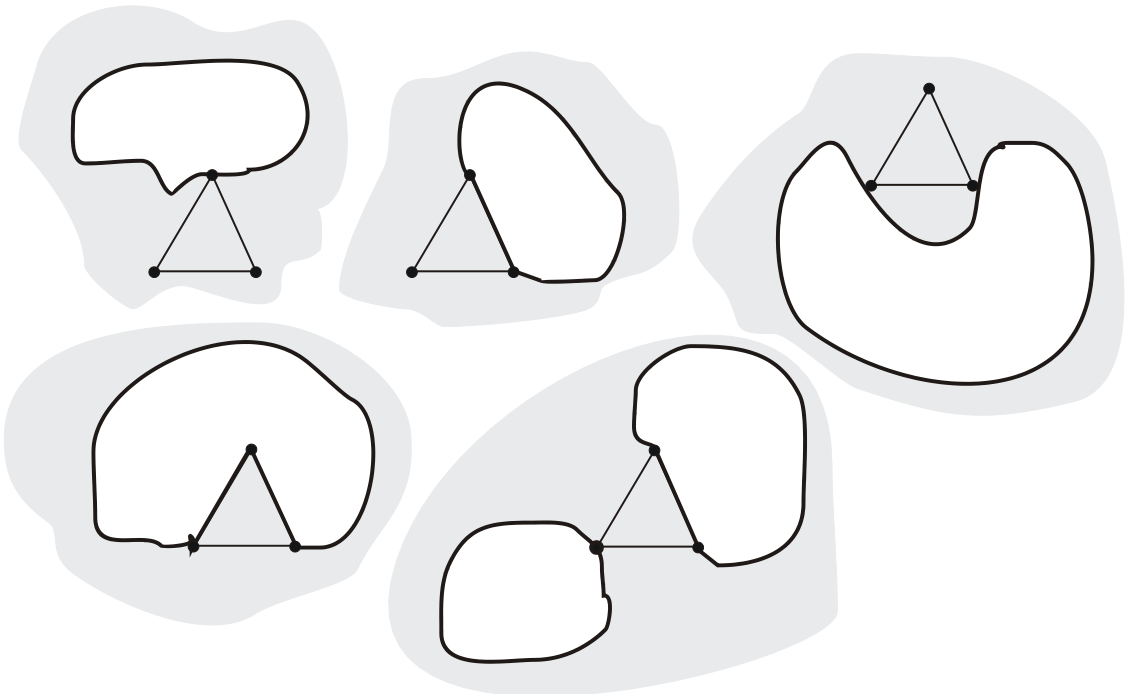


obr. 2.9

detekce hole

Následně, za běhu kompresního algoritmu, který prochází trojúhelníky povrchu modelu, je třeba na identifikované holes reagovat. Algoritmus EdgeBreaker to dělá tak, že kdykoliv narazí na trojúhelník typu C, zkontroluje, jestli nově označený vrchol náležící tomuto trojúhelníku náleží také obvodu hole. Pokud ano, algoritmus změní značení trojúhelníku z C na S, projde všemi vrcholy obvodu hole a označí je jako již navštívené. Algoritmus pak pokračuje v kompresi jako obvykle, jen k označené hole se už nechová jako k díře v povrchu, ale jako k plnému povrchu, který už byl algoritmem navštíven. Dostačujícími informacemi pro popis takovéto hole jsou délka obvodu (počet hran) a číslo, označující kolikátý S trojúhelník se této hole dotknul jako první.

Hole je vždy objevena C trojúhelníkem, tudíž jsou si všechny holes topologicky podobné. Výjimkou jsou případy, kdy trojúhelník, který se hole dotýká, je zároveň prvním označeným trojúhelníkem modelu. Taková situace je velmi nepříjemná, protože se trojúhelník může hole dotýkat mnoha různými způsoby, některé z si můžete prohlédnout na obr. 2.10, a každý je třeba ošetřit zvlášť. Způsob ošetření je popsán v kapitole 5.2.5 Metoda createHoleGeometry, nebo je k nahlédnutí ve zdrojových kódech.



obr. 2.10

některé způsoby dotyku prvního trojúhelníku a hole

Podrobný popis holes lze nalézt v [1] a [2].

2.6 Vícekomponentové modely

Poslední popisovanou, ale neméně užitečnou schopností algoritmu EdgeBreaker je schopnost komprese topologie modelů, které se skládají z několika komponent –topologicky nesouvisejících částí modelu. Takovým modelem může být například model sluneční soustavy, nebo automobilového motoru.

Z předešlého popisu algoritmu EdgeBreaker víme, že pokud pravý i levý sousední trojúhelník aktuálního trojúhelníku je označen jako navštívený (zkomprimovaný), je tato situace zapsána do CLERS řetězce jako E. Symbol E je tedy symbol, označující situaci, kdy není možné přejít na sousední trojúhelníky a lze tedy považovat povrch komponenty modelu za prozkoumaný (v případě dekomprese považujeme za rekonstruovaný). V takovém případě můžeme právě běžící větev algoritmu ukončit. Popis topologie komponenty může obsahovat více symbolů E, ale to je způsobeno rekurzivním spuštěním algoritmu při výskytu situace, označované symbolem S. Platí tedy, že nezávisle na počtu symbolů E, můžeme při každém jeho výskytu ukončit právě probíhající větev algoritmu a výsledkem bude, že se algoritmus sám ukončí právě po navštívení všech trojúhelníků povrchu komponenty. To platí pro kompresní i dekompresní algoritmus.

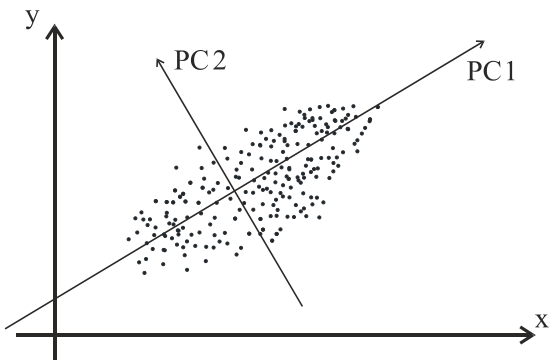
Této vlastnosti je možné velmi dobře využít při kompresi a dekompresi modelů, které se skládají z více komponent. Víme totiž, že po průchodu topologie jedné komponenty, se algoritmus sám ukončí a tím pádem není potřeba přenášet informace o délkách CLERS řetězců pro každou komponentu zvlášť. Vlastně není potřeba přenášet vůbec žádné další informace, kromě CLERS řetězce, pokud algoritmus komprese budeme spouštět tak dlouho, dokud model bude obsahovat nenavštívené trojúhelníky a všechny CLERS symboly všech komponent budeme zapisovat do jednoho společného řetězce. Analogicky platí pro dekompresní algoritmus, že má být spouštěn tak dlouho, dokud nebudou přečteny všechny symboly z CLERS řetězce.

Další detaily teorie algoritmu EdgeBreaker vizte v Použité literatuře.

3 Principal Component Analysis (PCA)

Do češtiny se dá PCA přeložit jako Analýza hlavních komponent. Tato statistická metoda je určena k hledání vzorů (závislostí) v souborech vícerozměrných dat. PCA je také vhodná ke kompresi takového souboru dat. Komprese metodou PCA je ztrátová a funguje následujícím způsobem.

PCA nejprve najde směry maximálního rozptylu hodnot zvoleného souboru dat. V těchto směrech pak souborem dat proloží pomyslné osy, které jsou navzájem kolmé a vytvoří tak nový souřadný systém (ortonormální bázi), jako třeba v případě na obr. 3.1. Tyto osy se nazývají komponenty, proto Analýza hlavních komponent.

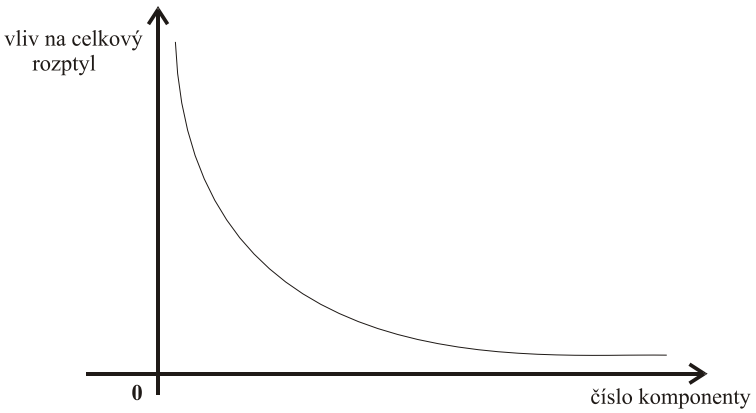


obr. 3.1

změna souřadného systému v závislosti na rozptylu dat

Jednotlivé komponenty jsou navzájem nezávislé a navíc jsou metodou PCA seřazeny podle míry, jakou ovlivňují celkový rozptyl hodnot v souboru dat. Teoreticky, jsou-li data náhodná, mohou komponenty ovlivňovat celkový rozptyl stejně velkou měrou, nebo měrou velmi podobnou. Pro reálná data, se kterými například pracuje algoritmus Coddyac, však platí, že první komponenta má na celkový rozptyl největší vliv a u následujících komponent, jak můžete vidět na obr. 3.2, vliv poměrně rychle klesá.

Protože vliv posledních komponent je prakticky zanedbatelný, můžeme je vypustit (snížit počet dimenzí) a tím v podstatě zkomprimovat data s minimálním vlivem na jejich celkový rozptyl.



obr. 3.2

typický průběh křivky vlivu komponent na celkový rozptyl reálných dat

Metoda PCA je využívána k mnoha účelům, mimo jiné k jednoduché kompresi počítačových obrázků, při počítačovém rozpoznávání lidských obličejů, nebo pro výpočet těsných boxů kolem počítačových modelů, které pak slouží k výpočtu kolizí.

Podrobný popis PCA lze nalézt v [7].

4 Algoritmus Coddyac

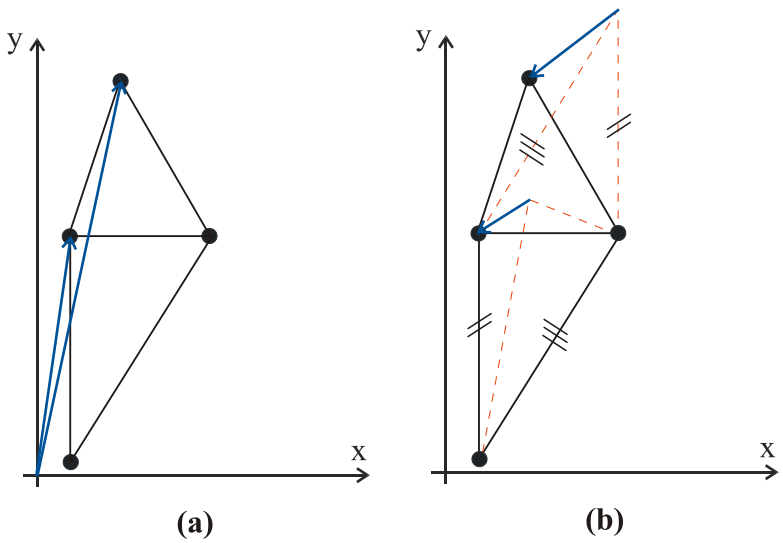
Coddyac [8] slouží ke kompresi 3D animací (dynamických trojúhelníkových sítí). Jeho nejdůležitějšími součástmi jsou kompresní algoritmy EdgeBreaker a PCA, které byly popsány výše. Komprese provedená tímto algoritmem je ztrátová, což je způsobeno použitím PCA algoritmu.

Animovaný model se skládá z vrcholů, jejichž souřadnice se v průběhu mění a tím dochází k jejich pohybu. Série takových souřadnic, patřící jednomu vrcholu, popisuje trajektorii vrcholu v průběhu animace. Algoritmus PCA slouží ke kompresi těchto trajektorií, z čehož vyplývá, že ztráta dat při kompresi má vliv především na pohyb animovaného modelu. Pokud by byla algoritmem Coddyac zkomprimována například animace běžícího člověka, nemělo by se stát, že vlivem komprese bude postavě člověka chybět prst na ruce, ale může se ztratit informace o tom, že se prst na ruce pokrčil a zase napřímil, protože je to z hlediska pohybu postavy nepodstatný pohyb.

EdgeBreaker je algoritmem Coddyac využíván k sestavení popisu topologie modelu, zjišťuje, jaké trojúhelníky spolu sousedí. Jak bylo popsáno výše, kdykoliv EdgeBreaker narazí na situaci C, zapíše C do CLERS a zároveň s tím si uloží souřadnice příslušného vrcholu modelu. V případě použití v algoritmu Coddyac se však nepoužívají souřadnice, ale vektory koeficientů, které byly získány algoritmem PCA z vektorů trajektorií vrcholů modelu. To proto, že topologie modelu se v průběhu animace nemění. Stačí tedy sestavit topologii modelu pouze v jednom snímku animace.

Kompresní algoritmus Coddyac dále hodnoty získané PCA kvantizuje a před uložením do souboru zakóduje aritmetickým (popřípadě Huffmanovým) kódováním, které se řadí do skupiny tzv. entropických kódování.

Entropické kódování je tím účinnější, čím menší je entropie hodnot, které kóduje. Z toho důvodu algoritmus Coddyac s pomocí PCA nekomprimuje přímo trajektorie (pozice) vrcholů (obr. 4.1 (a)). Místo toho provede v průběhu algoritmu EdgeBreaker jednoduchý odhad, jaká by trajektorie vrcholu (pozice v konkrétním snímku) mohla být a ke kódování poskytne vektory obsahující rozdíly mezi odhadem a skutečnou trajektorií. Tyto vektory jsou na obr. 4.1 (b) znázorněny modrými šipkami. V ideálním případě jsou si rozdíly velmi podobné a vlivem kvantizování je jim při entropickém kódování přiřazena stejná hodnota, jsou tedy účinněji komprimovány.



obr. 4.1

červeně -odhad pozice vrcholu (použit tzv. paralelogram)
modře -ukládaná informace (pozice / chyba odhadu)

Podrobný popis algoritmu Coddyac lze nalézt v [8].

5 Platformy pro webovou aplikaci

Jedním z cílů této práce je naprogramování webové aplikace, která by demonstrovala rychlost dekomprese a účinnost komprese algoritmu Coddyac. Protože aplikace bude muset vykreslovat vlastní grafiku (ne jen zobrazovat načtené obrázky), nabízí se pro tento úkol 3 pravděpodobně nejvhodnější platformy.

První z možností je naprogramování Java appletu. Jednou z výhod napsání této aplikace v Javě je, že je to poměrně rozšířený jazyk a tedy většině programátorů by nemělo dělat potíže aplikaci upravit. Java podporuje práci s 2D grafikou a dokonce i 3D grafikou počítanou na grafické kartě a je velmi dobře dokumentovaná, což je z pohledu programátora příjemné a urychluje to vývoj aplikací.

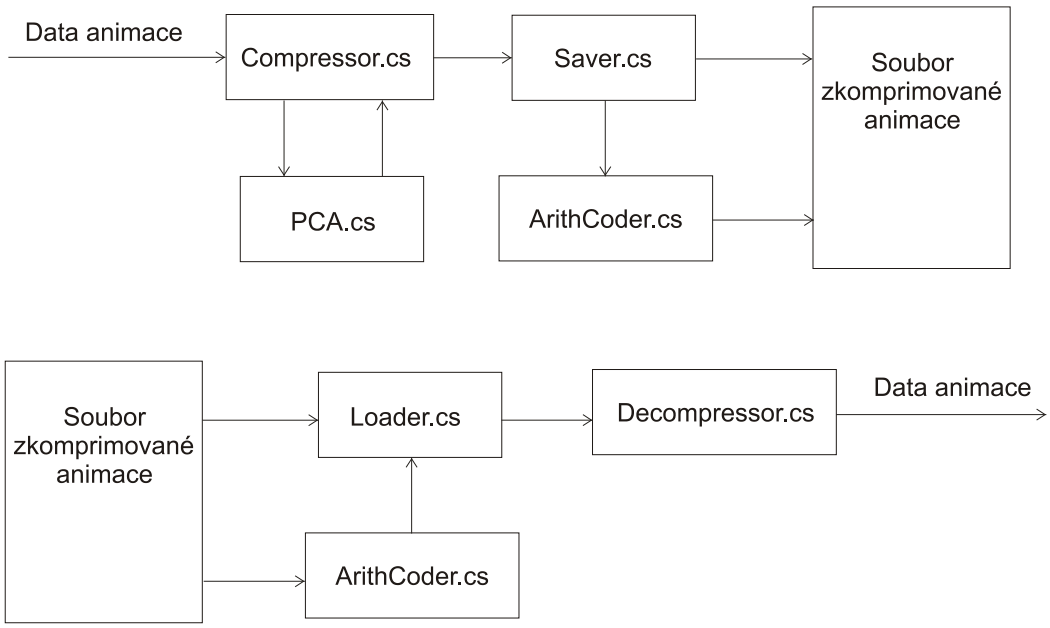
Druhou možností je použít pro napsání aplikace v prostředí Adobe Flash. Flashové aplikace jsou na internetu často k vidění, z čehož se dá usuzovat na jeho značnou oblibu mezi programátory. Díky tomu lze na internetu najít velké množství tutoriálů a diskuzí, které mohou pomoci při případných problémech během vývoje aplikace.

Poslední vybranou platformou je platforma Microsoft SilverLight. Jak je z názvu patrné, platforma je vyvíjena společností Microsoft a dá se tedy očekávat maximální podpora od ostatních jejích produktů (včetně operačního systému a Internet Exploreru). SilverLight by měl být podporován všemi druhy prohlížečů a na rozdíl od předchozích dvou platforem má obrovskou výhodu v tom, že umožňuje vývoj aplikací v jazycích platformy .NET včetně C#, tedy jazyku, ve kterém jsou psány kódy pro prostředí Mve-2. Po menších úpravách by mělo být možné použít už hotové kódy. Ukázky aplikací na domovských stránkách Microsoft SilverLight vypadají lákavě a jednou z nich je i engine pro renderování 3D grafiky. Dalším zajímavým faktem je podpora vývoje aplikací s více než jedním vláknem, což se dá využít třeba při vykreslování animací. Nevýhodou ale je, že SilverLight je nová technologie a na internetu je zatím jen málo tutoriálů

6 Popis programového řešení pro Mve-2

Vytvořený program slouží ke kompresi a dekompresi dynamických trojúhelníkových sítí algoritmem Coddyac. Je naprogramován v jazyce C# a sestaven jako knihovna (DLL) pro prostředí Mve-2. Důležité třídy, ze kterých se program skládá, jsou popsány níže, zjednodušený UML diagram tříd je zobrazen na obr. 6.1. Jako součást řešení byly vytvořeny také tři ukázkové mapy modulů pro Mve-2.

Program je v prostředí Mve-2 rozdělen do několika modulů, které je třeba vzájemně propojit a sestavit tak mapy modulů. Náhled ukázkových map je umístěn v příloze Příloha 1. V modulech lze nastavit například jméno souboru, ze kterého se má animace načíst, nebo koeficienty ovlivňující míru komprese. Jednou z výhod vytvoření tohoto programu právě pro prostředí Mve-2 je snadný přístup k poměrně kvalitní vizualizaci 3D dat a snadné znovupoužití modulů programu při sestavování komplexnějších aplikací.



obr. 6.1

zjednodušený UML diagram tříd
pro kompletní kompresi (nahoře) a dekompresi (dole)

6.1 Compressor

Vstupem tohoto modulu (třídy) je model dodaný prostředím Mve-2. Třída Compressor tento model zkomprimuje algoritmy EdgeBreaker a PCA a jako výstup do prostředí Mve-2 poskytne všechna data nezbytná pro dekompresi modelu. Konkrétně to jsou:

Informace o topologii využívané algoritmem EdgeBreaker

Numer of Components	-počet komponent, ze kterých se model skládá
Clers	-textový řetězec složený ze symbolů C, L, E, R a S, který popisuje sousednost trojúhelníků v trojúhelníkové sítě modelu
Handles	-pole které obsahuje páry oposite cornerů označujících výskyt handle
Holes	-pole které obsahuje informace o délce a počátečním bodu každé díry v povrchu modelu
Hole Starts	-pole které obsahuje indexy komponent, jejichž první označený trojúhelník se dotýká nejméně jedné díry v jejím v povrchu

Informace o geometrii využívané algoritmem PCA

Coefficients	-koeficienty trajektorií zkomprimovaných algoritmem PCA
Basis	-báze PCA nutná pro dekompresi trajektorií
Means	-průměrné hodnoty všech trajektorií
Deta d	-kvantizační konstanta pro kvantizaci trajektorií
Delta q	-kvantizační konstanta pro kvantizaci bázových vektorů
Coefficient Vector Length	-délka vektoru koeficientů zkomprimovaných trajektorií

Koeficienty trajektorií jsou algoritmem komprimovány v pořadí, v jakém byly navštíveny algoritmem EdgeBreaker. Algoritmus komprese využívá čtyři pole označená jako V, O, M a U. Pole V a O již byla zmiňována v 2.1 CornerTable(CT). Pole M je pole všech cornerů a zaznamenává se do něj, zda už byl corner s indexem rovným indexu položky pole M navštíven. Pole U je pole všech trojúhelníků a zaznamenává se do něj, zda byl daný trojúhelník algoritmem už navštíven.

6.1.1 Inicializace kompresního algoritmu

Při spuštění komprese nejprve dojde k inicializaci algoritmu metodou init. Ta ze zadaného vstupu načte všechny trojúhelníky a vrcholy modelu a nastaví velikosti polí V a O na trojnásobek počtu trojúhelníků, velikost pole M na hodnotu rovnající se počtu vrcholů modelu a velikost pole U na hodnotu rovnající se počtu trojúhelníků sítě. V zápětí nastaví hodnoty polí M a U na „0“, což indikuje, že vrcholy z M a trojúhelníky z U ještě nebyly algoritmem navštíveny. Pak je vytvořeno pole pro symboly CLERS, které popisuje sousednost trojúhelníků. Jeho délka je nastavena na „počet trojúhelníků -1“ prvků. Dalším krokem inicializace je označení vrcholů na obvodech holes, které se v modelu vyskytují. Nakonec inicializační metoda naplní pole V vrcholy všech trojúhelníků a pole O indexy oposite cornerů. Mají-li dva sousedící trojúhelníky společnou hranu, oposite cornery jsou cornery patřící vrcholům, které tyto dva sousedící trojúhelníky nemají společné. Tím jsou inicializovány všechny proměnné a struktury pro kompresi topologie modelu. Následně jsou metodou init inicializovány proměnné potřebné při kompresi geometrie modelu. Nejprve jsou sestaveny trajektorie vrcholů animovaného modelu metodou initTrajectories a pak jsou předány instanci třídy PCA, která provede většinu výpočtů potřebných k jejich kompresi. Dále už algoritmus nepracuje s trajektoriemi, ale s koeficienty těchto trajektorií, získaných od třídy PCA.

Po inicializaci metodou init je v cyklu spouštěna metoda startCompression, jejímž vstupním parametrem je index vrcholu corneru trojúhelníku, od nějž chceme kompresi začít. Hodnota tohoto parametru je volena tak, aby komprese pokaždé začala cornerem (tedy i trojúhelníkem), který ještě nebyl navštíven. Vykonáním každé smyčky cyklu tak dojde ke kompresi jedné komponenty modelu.

Tato metoda nejprve zkontroluje, jestli se první komprimovaný trojúhelník (obsahuje zadaný corner) dotýká nějaké hole a pokud ano, zjistí kolika stranami a vybere takový corner, jehož oposite corner nenáleží hraně hole. Pokud žádný takový neexistuje, tvoří trojúhelník samostatnou komponentu.

Potom jsou do pole s koeficienty trajektorií uloženy hodnoty 3 vrcholů trojúhelníka, z kterého pochází první zadaný corner. V poli M jsou tyto 3 vrcholy označeny hodnotou „1“, která značí, že byl příslušný vrchol modelu 1x navštíven algoritmem. Tento první trojúhelník je v poli U také označen jako navštívený. Na konec metoda startCompression spustí metodu compress se vstupním parametrem, kterým je oposite corner vybraného corneru.

6.1.2 Průběh komprese

První, co metoda compress udělá je, že trojúhelník, kterému náleží zadaný corner, označí jako navštívený. Poté zkontroluje pravý a levý sousedící trojúhelník. Všechny trojúhelníky jsou algoritmem označeny 1/0, podle toho, zda byly/nebyly navštíveny. Výjimku tvoří trojúhelníky typu S. Pokud byl algoritmem navštíven trojúhelník typu S, neobsahuje pole U číslo 1, ale hodnotu „počet navštívených trojúhelníků * 3 + 2“, která bude při dekompresi totožná s indexem jednoho z cornerů právě vytvořeného S trojúhelníka, konkrétně index prvního corneru proti směru hodinových ručiček od vrcholu, kterým byl S trojúhelník vytvořen. Pokud byla metodou compress u jednoho ze sousedních trojúhelníků tato hodnota (>1) nalezena, znamená to, že jeden z těchto sousedních trojúhelníků je S trojúhelník, který vytváří tzv. handle. Algoritmem je pak tato hodnota spolu s hodnotou oposite corneru z právě vytvořeného trojúhelníku uložena do seznamu handles.

Po kontrole sousedních trojúhelníků se zkontroluje corner, který byl metodě compress předán. Pokud jemu náležící vrchol ještě nebyl navštíven, označí jej algoritmus jako navštívený, provede odhad a kompresi jeho vektoru koeficientů trajektorie a zkontroluje, jestli náhodou neleží na obvodu nějaké hole. Pokud neleží, přidá na konec CLERS řetězce symbol C, který označuje nově vytvořený trojúhelník, který nemá ani pravého, ani levého souseda. Pokud leží na hranici hole, označí algoritmus všechny vrcholy ležící na obvodu hole jako navštívené, zkomprimuje jejich vektory koeficientů trajektorie a na konec CLERS řetězce přidá symbol S. Ještě se do pole holes zaznamená délka právě navštívené hole a počet S trojúhelníků, které byly dosud navštíveny. Počet S trojúhelníků funguje jako značka, která při dekompresi určuje, kde hole začíná. Algoritmus potom pokračuje navštívením vrcholu vpravo od aktuálního vrcholu.

V případě, že zadaný vrchol již byl navštíven, rozlišuje algoritmus 4 typy takto vytvořených (navštívených) trojúhelníků. Pokud už byl navštíven pravý i levý trojúhelník, jedná se o typ E, který funguje jako záplata. V tomto směru již není možné prozkoumávat povrch modelu a compress algoritmus je ukončen. Pokud byl navštíven pouze pravý trojúhelník, jedná se o typ R, algoritmus pak pokračuje prozkoumáváním levého sousedního trojúhelníku, tedy vrcholem vlevo od aktuálního vrcholu. Pokud byl navštíven pouze levý trojúhelník, jedná se o typ L, algoritmus pak pokračuje prozkoumáváním pravého sousedního trojúhelníku, tedy vrcholem vpravo od aktuálního vrcholu. Pokud nebyl navštíven ani pravý, ani levý sousední trojúhelník a přesto je zadaný vrchol označen jako navštívený, jedná se o typ S. Takový trojúhelník způsobí vytvoření dvou oblastí, napravo a nalevo, které jsou

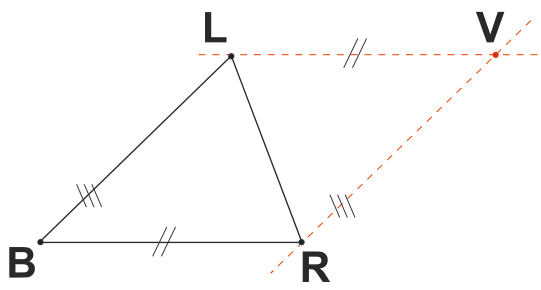
neprozkoumané a vytváří tak v prozkoumaném povrchu modelu „díry“, které je třeba algoritmem projít. Algoritmus tedy nejprve prozkoumá oblast napravo a poté i oblast nalevo.

Pokaždé, když algoritmus projde jedním ze 4 zmiňovaných typů trojúhelníků, přidá na konec CLERS řetězce symbol odpovídající typu trojúhelníku a v poli U označí trojúhelník symbolem 1 jako navštívený. Výjimkou je typ S, kdy se do pole U uloží výše popsaná hodnota, naznačující, že by se mohlo jednat o S trojúhelník vytvářející handle.

Všechny výše popsané akce metody compress se provádějí ve smyčce, s výjimkou případu S, kdy se algoritmus spouští přes pravý sousedící trojúhelník rekurzivně. Když metoda projde povrch celého modelu, smyčka se přeruší a na výstup třídy compressor je nastavena zkomprimovaná topologie a geometrie animovaného modelu. Ukázku zdrojového kódu si lze prohlédnout v příloze Příloha 2.

6.1.3 Odhad a komprese koeficientů trajektorie vrcholu

Pro odhad trajektorií (přesněji jejich koeficientů získaných PCA) vrcholů modelu jsou používány dva podobné postupy, jeden pro obecné případy a druhý pro odhad trajektorií vrcholů na obvodu hole. První zmiňovaný postup využívá vektorů 3 vrcholů trojúhelníku, který byl navštíven v předešlém kroku a předpokládá, že dva sousedící trojúhelníky spolu tvoří rovnoběžník (obr. 6.2). Tato metoda odhadu se anglicky nazývá Parallelogram Prediction.



obr. 6.2

odhad pozice vrcholu V

Odhad trajektorie je tedy spočítán jako:

$$v_i = l_i + r_i - b_i,$$

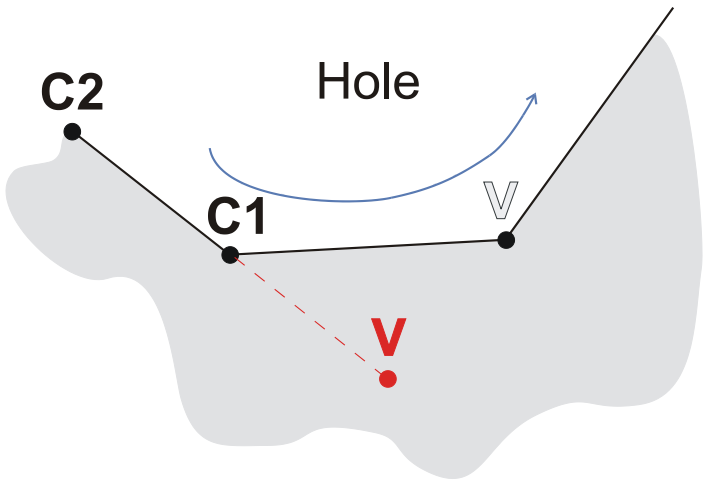
kde v, l, r a b jsou vektory koeficientů trajektorií příslušejících vrcholům V, L, R a B a hodnota indexu i nabývá hodnot od 0 do (délka vektoru koeficientů trajektorie – 1) a indexuje složky těchto vektorů.

Po provedení odhadu algoritmus zjistí skutečnou podobu vektoru koeficientů trajektorie vrcholu V a obyčejným odečtením vektorů vypočítá rozdíl mezi odhadovanými a skutečnými hodnotami, jak již bylo zmiňováno v kapitole 4. Tento rozdíl se nazývá reziduum. Reziduum je nakonec zkvantizováno tak, že jsou jeho složky nejprve vyděleny kvantizační konstantou delta d a pak zaokrouhleny na celá čísla. Kvantizací obvykle dojde k malé změně původní hodnoty rezidua, vznikne chyba. Aby při pozdější dekompresi nedocházelo k propagaci takto vzniklé chyby, vypočítá kompresní algoritmus nový vektor koeficientů příslušné trajektorie tak, jak by ho vypočítal dekompresní algoritmus (s kvantizovaným reziduem) a tento upravený vektor použije při dalších výpočtech místo původního vektoru.

Koeficienty trajektorií vrcholů na obvodu hole jsou komprimovány stejným způsobem. Rozdíl je pouze v odhadu vektoru koeficientů trajektorie. Když algoritmus prochází obvod hole, nemá k dispozici trojúhelníky které jí obklopují, ale jen vrcholy a proto koeficienty trajektorií nejsou odhadovány na základě vrcholů trojúhelníku navštíveného v předešlém kroku, ale na základě předešlé hrany a jejích krajních vrcholů C1 a C2:

$$v_i = 2 * c1_i - c2_i,$$

geometrický význam je nastíněn na následujícím obrázku (obr. 6.3).



obr. 6.3

odhad pozice vrcholu V, šipka ukazuje směr průchodu vrcholů hole

Stejný princip komprese jako v případě odhadu vektorů koeficientů trajektorií na obvodu hole je použit i na kompresi vektoru průměrných hodnot a vektorů báze, které jsou vytvořeny při kompresi algoritmem PCA.

6.2 Decompressor

Vstupní data tohoto modulu (třídy) jsou totožná s výstupními daty modulu pro kompresi a jsou popsána v kapitole 5.1 Compressor.cs. Třída Decompressor provede dekompresi topologie algoritmem EdgeBreaker a dekompresi geometrie algoritmem PCA. Z dekomprimovaných dat pak sestaví sérii modelů ve formátu, se kterým je prostředí Mve-2 schopné pracovat a nastaví jej na výstup.

6.2.1 Inicializace dekompresního algoritmu

Dekompresní algoritmus řeší dekompresi topologie a dekompresi geometrie odděleně. Důvodem pro toto rozdělení je to, že algoritmus dekomprese způsobuje v povrchu komponent trhliny, které je třeba zacelit. K zacelení ale dojde později, než to vyžaduje algoritmus dekomprese trajektorií vrcholů komponenty. Proto nejprve proběhne úplné sestavení topologie komponenty a až potom výpočty trajektorií.

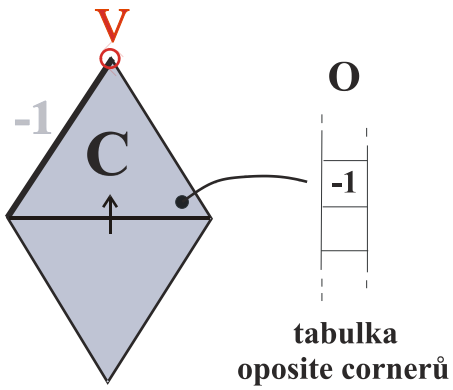
Prvním krokem dekompresního algoritmu je načtení dat ze vstupu a spuštění metody initTopology. Metoda initTopology zjistí počet trojúhelníků modelu a nastaví délku polí V a O na trojnásobek počtu trojúhelníku modelu. Pole V obsahuje indexy vrcholů všech trojúhelníků, pole O pak indexy jejich oposite cornerů. Následně je spuštěna metoda startTopologyDecompression, která vytvoří první trojúhelník, od kterého bude začínat dekompresní algoritmus. Indexy jeho vrcholů (0,1,2) uloží do pole V a zkontroluje, jestli se tento první trojúhelník právě dekomprimované komponenty nedotýká hole. Pokud se dotýká, provede se rekonstrukce topologie této hole. Princip rekonstrukce je popsán v kapitole 5.2.5 Metoda createHoleGeometry. Poté se spustí metoda decompressTopology, které jako vstupní parametr předáme index vrcholu trojúhelníka (corner), kterým má dekomprese začít. Metoda decompressTopology je spouštěna v cyklu a tím dochází k postupné dekompresi všech komponent, ze kterých se model skládá. Cyklus probíhá tak dlouho, dokud nejsou přečteny všechny symboly z CLRES řetězce.

6.2.2 Průběh dekomprese

Kód metody decompressTopology, je spouštěn v cyklu, který dekomprimuje všechny trojúhelníky dané komponenty. Výjimkou je případ S, kdy se algoritmus spouští přes pravý sousední trojúhelník rekurzivně. Po průchodu všech trojúhelníků komponenty se algoritmus ukončí.

Metoda decompress nejprve uloží do pole O aktuální corner a jeho oposite corner z posledního vytvořeného trojúhelníku. Hodnoty vrcholů nově vytvořeného trojúhelníku nastaví v poli V tak, aby indexy v nich uložené odpovídaly dvěma vrcholům, které mají nový trojúhelník a předchozí trojúhelník společné. Tím je zajištěno, že indexy společných vrcholů z V z různých trojúhelníků ukazují na stejné trajektorie při dekompresi geometrie modelu. Algoritmus poté přejde na corner ležící proti pravé hraně nově vytvořeného trojúhelníku. Výjimkou je případ R, kdy algoritmus přejde na corner ležící proti levé hraně.

Z CLERS řetězce je pak načten symbol, který říká, jaký typ trojúhelníku byl právě vytvořen. Podle typu trojúhelníku reaguje algoritmus následujícími způsoby. Je-li vytvořený trojúhelník typu C, přiřadíme levé hraně trojúhelníku hodnotu -1, která bude později využita metodou zip. Ve skutečnosti hodnota -1 není přiřazena levé hraně, ale uložena do tabulky oposite cornerů O na index corneru prev(V), jako je to vyobrazeno na obr. 6.4. Následně se do pole V pro nový vrchol uloží index ukazující v poli trajektorií na trajektorii tohoto vrcholu. Algoritmus dále pokračuje trojúhelníkem vpravo.



obr. 6.4

označení hran uložené v tabulce O

Je-li vytvořený trojúhelník typu L, přiřadí algoritmus levé hraně hodnotu -2, která bude později využita metodou zip. Dále zkontroluje, jestli levá hrana není uložena v poli handles –k tomu slouží metoda checkHandle. Pokud je, spustí na ni metodu zip, která spojí otevřené hrany označené symboly -2 a -1 a pak algoritmus pokračuje trojúhelníkem napravo.

Je-li vytvořený trojúhelník typu R, přiřadí algoritmus pravé straně hodnotu -2, zkontrolujeme, jestli je pravá hrana v poli handles a pak algoritmus pokračuje trojúhelníkem nalevo.

Je-li vytvořený trojúhelník typu E, označí algoritmus pravou i levou hranu trojúhelníka hodnotou -2 a zkontrolujeme, jestli je pravá nebo levá hrana uložena v poli handles. Pokud není levá hrana v poli handles, spustí metodu zip na tuto hranu. A přeruší nekonečnou smyčku prohledávání povrchu.

Je-li vytvořený trojúhelník typu S, musí algoritmus nejprve zjistit, jestli symbol S neoznačuje začátek hole. K tomu slouží metoda checkHole, která prozkoumá pole holes. Pokud pole holes obsahuje hole, která začíná právě načteným symbolem S, vrátí hodnotu true. Jestliže symbol S označuje začátek hole, rekonstuuje algoritmus její topologii a pokračuje průchodem přes pravou hranu trojúhelníku. Pokud symbol S neoznačuje začátek hole, spustí se rekurzivně průchod povrchem modelu přes pravý sousedící trojúhelník s pomocí opětovného spuštění metody decompress a po ukončení rekurze algoritmus pokračuje přes levý sousední trojúhelník.

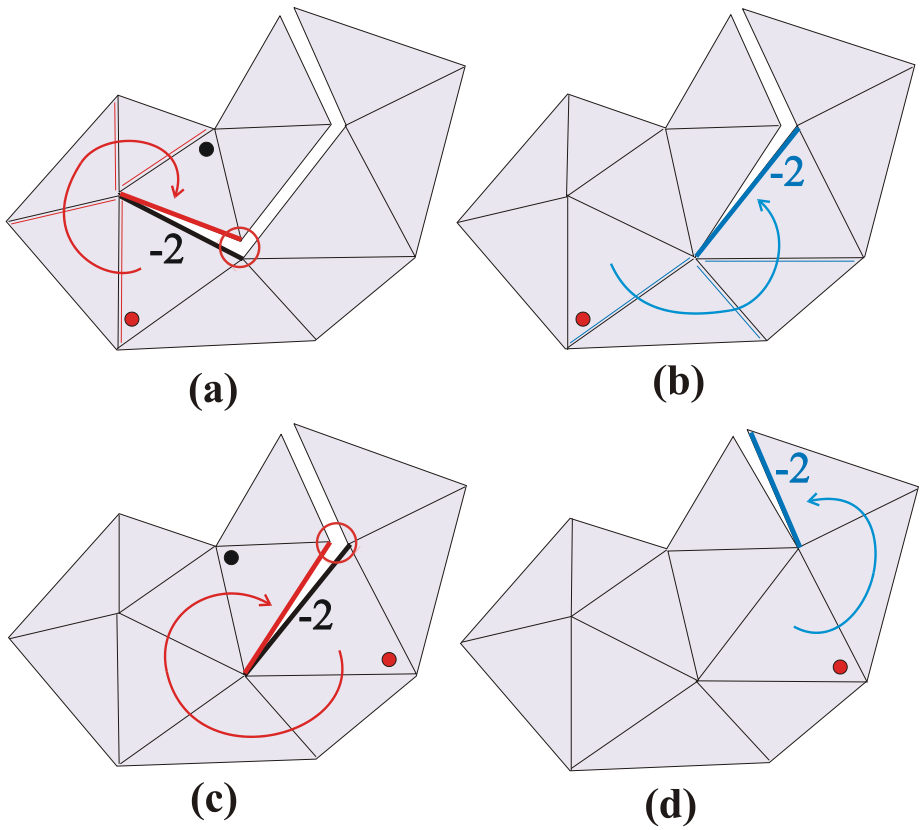
Pokud už byl ale tento levý trojúhelník navštíven (obvykle způsobeno výskytem handle), přerušíme nekonečnou smyčku algoritmu.

Po dokončení dekomprese topologie modelu se spustí metoda initGeometry, která nastaví vybrané proměnné tak, aby bylo možné aktuální komponentu sestavit (projít) ještě jednou. Potom jsou spuštěny metody startGeometryDecompression a decompressGeometry. Kódy a tedy i chování těchto metod jsou ekvivalentní jejich protějškům pro kompresi topologie. Rozdíl je pouze v tom, že prochází povrch komponenty, který je už rekonstruovaný a kdykoliv narazí na trojúhelník typu C, nebo na hole, dekomprimují trajektorie příslušných vrcholů a uloží je do pole editedCoefs. Po skončení dekomprese geometrie je třeba trajektorie vynásobit s bázovou maticí vytvořenou při kompresi algoritmem PCA a přičíst k výsledkům vektor průměrných hodnot (Means). Vznikne tak matice geometry s dekomprimovanými trajektoriemi vrcholů.

Na konec jsou pole V a geometry předána metodě createMesh, která jimi vyplní strukturu pole modelů používanou prostředím Mve-2. Tato struktura je pak předána na výstup modulu Decompressor.

6.2.3 Metoda zip

Metodě zip je jako vstupní parametr předána hrana jednoho trojúhelníku s označením -2, na obr. 6.5 vyznačena silnou černou čárou. Ve skutečnosti algoritmus nepracuje s hranami, ale protilehlými cornery – na obr. 6.5 zobrazeno jako červený bod. Tato metoda pak postupně prochází (ve směru hodinových ručiček) levé hrany levých sousedních trojúhelníků, dokud nenarazí na hranu s označením -1, nebo se nevrání do počátečního trojúhelníku – obr. 6.5 (a). Je-li nalezena hrana s označením -1, pak jsou červený a černý bod na obr. 6.5 uloženy do oposite corner tabulky O a odkazy cornerů zakroužkovaných vrcholů na jejich trajektorie jsou v poli V nastaveny na stejnou hodnotu. Tím dojde ke spojení červeně a černě vyznačené hrany – obr. 6.5(a). Algoritmus pak pokračuje prozkoumáním pravých hran pravých sousedních trojúhelníků (proti směru hodinových ručiček) a hledá hranu s označením -2, zobrazeno na obr. 6.5(b) jako silná modrá čára. Pokud je taková hrana nalezena, spustí algoritmus rekurzivně sám sebe a jako vstupní parametr předá nově objevenou hranu s označením -2 (obr. 6.5(c),(d)).



obr. 6.5

postup algoritmu metody *zip*

6.2.4 Metoda checkHandle

Tato metoda kontroluje, jestli je zadaná hrana v poli handles (tvoří handle). Pokud ne, vrací false. Pokud ano, zapíše oposite cornery dvou trojúhelníků se společnou handle hranou do pole O, pokusí se najít nejbližší volnou hranu označenou -2 z obou stran handle hrany a spustit na ně metodu zip. Pak metoda vrátí true.

6.2.5 Metoda createHoleTopology

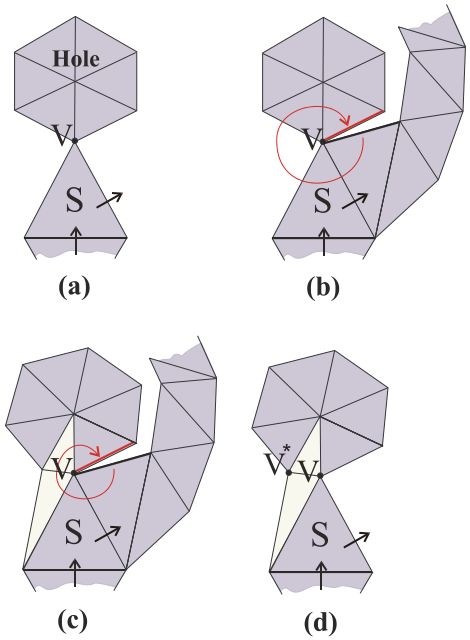
Tato metoda slouží k vytvoření topologie zadané hole. Prvním vstupním parametrem je počet hran hole, druhým je číslo corneru, kterým obvod hole začíná. Program obsahuje také metodu createHoleGeometry, která je metodě createHoleTopology velmi podobná, nebude zde tedy popisována.

Metoda createHoleTopology bývá spuštěna v okamžiku, kdy je vytvořen S trojúhelník, o kterém algoritmus ví, že se hole dotýká jako první. Metoda po svém spuštění vytvoří disk z tolika trojúhelníků, aby počet vnějších hran disku odpovídal délce (počtu hran) hole a připojí ho k trojúhelníku S. Vzniklý stav je zobrazen na obr. 6.6 (a).

Problém ale nastane ve chvíli, kdy se spustí metoda zip a pokusí se spojit trojúhelníky hole a trojúhelníky vytvořené průchodem přes pravý sousední trojúhelník trojúhelníku S. Metoda zip totiž potřebuje procházet trojúhelníky kolem vrcholu V, jako je to znázorněno na obr. 6.6 (b).

Problém je vyřešen tak, že při konstrukci disku hole, jsou vytvořeny ještě dva trojúhelníky, které jsou požadovány metodou zip. Ty jsou znázorněny na obr. 6.6 (c). Aby to nijak neovlivnilo výslednou podobu komponenty, mají tyto trojúhelníky nulovou výšku, tj. dva vrcholy se stejnými souřadnicemi (odkazy na vektory koeficientů) -V a V* na obr. 6.6 (d).

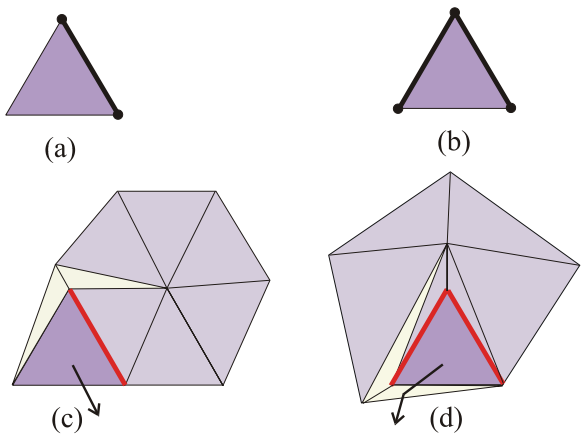
Indexy trojúhelníků disku hole a dvou přidaných trojúhelníků jsou uloženy do seznamu, který obsahuje index neviditelných trojúhelníků. V průběhu dekomprese se dekompresní algoritmus k těmto trojúhelníkům chová stejně, jako by tvořily normální povrch modelu. Avšak ve chvíli, kdy je vytvářeno pole modelů tvořících animaci, je každý trojúhelník kotrolován a je-li jeho index obsažen v poli neviditelných trojúhelníků, je tento trojúhelník algoritmem vynechán a tedy není vložen do trojúhelníkové sítě výsledného modelu. Tím vznikne požadovaná díra v povrchu modelu.



obr. 6.6

postup vytvoření hole, černé šipky ukazují směr průchodu trojúhelníků

V běžných situacích se trojúhelník S dotýká pouze jedné hole, pouze jedním vrcholem. Výjimečná situace nastává v případě prvního trojúhelníku komponenty, jak již bylo zmíněno v kapitole 2.5 Holes. Pokud se první trojúhelník dotýká hole, může nastat 9 různých druhů dotyku. Každý případ je nutné ošetřit zvlášť, příklady ošetření lze nalézt ve zdrojových kódech. Na obr. 6.7 jsou uvedeny ukázky dotyku dvou vrcholů a jedné hrany obr. 6.7 (a), tří vrcholů a dvou hran obr. 6.7 (b) a jejich ošetření obr. 6.7 (c, d), které se týká směru pokračování algoritmu a zipování hran hole k trojúhelníku. V obou případech je připojena hole délky 6.



obr. 6.7

ošetření dotyku více než jednoho vrcholu, šipkami jsou vyznačeny směry pokračování algoritmu, červeně zazipované hrany

6.3 Saver

Třída Saver má za úkol ukládat data zkomprimovaného modelu (trojúhelníkové sítě) na disk ve formě binárního souboru. Před spuštěním algoritmu jsou třídě předána všechna data, která vznikla při kompresi modelu a která jsou potřebná k jeho dekompresi. Potom je ze vstupu načteno jméno a umístění souboru, do kterého se mají data ukládat.

Jakmile je soubor vytvořen, zapíše se do něj počet komponent modelu, délka vektoru koeficientů trajektorií, kvantizační konstanty, pole handles, pole holes a pole holeStarts. K tomu je použita standardní třída BinaryWriter.

Následuje uložení řetězce CLERS, vektorů koeficientů, báze PCA a vektoru průměrných hodnot (Means). Protože jsou tato data velmi rozsáhlá a poměrně často se v nich opakují stejné hodnoty, jsou před uložení zakódována aritmetickým kódováním, čímž dojde k další kompresi dat. Metodu aritmetického kódování poskytuje třída ArithCoder.

Aby mohla být data aritmeticky zakódována, musí být celočíselná. Proto jsou symbolům (písmenům) v řetězci CLERS přiřazena čísla od 0 do 4 a celý řetězec je před kódováním a uložením do souboru převeden na pole čísel.

6.4 Loader

Účelem třídy Loader je zajistit správné načtení dat ze zadaného binárního souboru, který obsahuje popis topologii a geometrii zkomprimovaného modelu (trojúhelníkové sítě). Třídě je jako vstup zadáno jméno souboru. Řetězec CLERS a všechna data popisující geometrii modelu byla při ukládání zkomprimována aritmetickým kódováním, které poskytuje třída ArithCoder, proto je k načtení dat ze souboru potřebná i tato třída.

Třída Loader nejprve načte počet komponent, délku vektoru koeficientů trajektorií, kvantizační konstanty, pole handles, pole holes a pole holeStarts. K tomu stačí použít standardní třídu BinaryReader. Potom přijde na řadu načtení řetězce CLERS, vektorů koeficientů trajektorií, báze PCA a vektoru průměrných hodnot (Means). K tomu je použit již zmíněný ArithCoder. Význam popsaných dat je popsán v kapitole 5.1 Compressor.cs.

Po načtení jsou všechna data přetypována na datové typy, vhodné pro prostředí Mve-2 a vystavena na výstup.

6.5 PCA a ArithCoder

Tyto dvě třída byly vytvořeny panem ing. Liborem Vášou, proto bude jejich funkce popsána jen velmi stručně.

Třída PCA slouží ke kompresi trajektorií vrcholů modelu. Je tedy využívána jen v modulu Compress. Vstupem této třídy je pouze pole vektorů trajektorií, které chceme komprimovat. Délka tohoto pole je rovna počtu vrcholů modelu. Vektory které toto pole obsahuje mají délku (d * f), kde f je počet snímků animace a d je počet složek vrcholu. Obyčejně je d rovno 3, protože přenášíme pouze informace o poloze (souřadnice X, Y, Z).

Po provedení výpočtu uvnitř PCA je k dispozici pole vektorů koeficientů trajektorií se stejnými rozměry, jaké mělo pole zadané na vstupu. Vektory tohoto pole už ale neobsahují trajektorie vrcholů, ale jen jejich koeficienty seřazené podle míry vlivu na celkový rozptyl hodnot. Bližší popis lze nalézt v kapitole 3 Principal Component Analysis (PCA), nebo v [7]. Tyto vektory už nejsou při kompresi používány celé, ale jen několik prvních složek, což je důvodem ztrátové komprese trajektorií. Má-li například animace modelu 200 snímků, původní vektor trajektorie má 3 * 200 = 600 složek. Vektor koeficientů, se kterým později pracuje kompresní algoritmus, už může mít třeba jen 20 složek.

Aby bylo možné trajektorie opět dekomprimovat, je nutné si od PCA vyžádat jeho bázovou matici a vektor průměrných hodnot (Means). Délka vektorů báze a vektoru Means je rovna délce vektorů původních trajektorií, tedy (d * f). Počet vektorů báze, které je třeba při dekompresi použít, je roven délce vektorů koeficientů. Podle příkladu, který je uveden výše, by mělo být bázových vektorů 20.

Při dekompresi už není třeba používat třídu PCA, protože k získání kterékoliv původní trajektorie stačí když vektor koeficientů této trajektorie vynásobíme bázovou maticí a k výslednému vektoru přičteme vektor Means. Vzniklý vektor trajektorie má samozřejmě stejnou délku, jako původní vektor trajektorie.

Funkcí třídy ArithCoder je zakódovat zadané pole celočíselných hodnot aritmetickým kódováním a uložit ho do zadaného binárního souboru (streamu). Třída ArithCoder umí samozřejmě data z tohoto souboru načíst a dekódovat. Funkce této třídy jsou používány třídou Saver a třídou Loader.

Aritmetické kódování patří mezi tzv. entropická kódování. Základním rysem většiny typů entropických kódování je, že při kódování zjišťují pravděpodobnost výskytů symbolů, které mají být zakódovány. Symbolům jsou potom přiřazeny kódy, jejichž délka je úměrná pravděpodobnosti výskytů těchto symbolů. Čím pravděpodobnější je výskyt symbolu, tím kratší je jeho kód, kterým je symbol nahrazen. Entropická kódování jsou často používána ke kompresi dat, protože mohou několikabitové symboly nahradit mnohem kratším kódem.

Nejběžnějšími technikami entropického kódování jsou Huffmanovo kódování a aritmetické kódování. Algoritmus Huffmanova kódování používá binární stromy, kde listy zastupují kódová slova. Kódová slova jsou určena cestou, jakou je třeba projít z kořene stromu k listu (každá hrana grafu binárního stromu symbolizuje bit 1 nebo 0). Tyto kódy jsou prefixové, tzn. žádný kód není počátkem jiného kódu. Protože jsou kódy tvořeny z celých bitů, neodpovídá délka kódu přesně pravděpodobnosti výskytu a dochází tak k zaokrouhlení pravděpodobností směrem nahoru.

Tento problém řeší aritmetické kódování, které je schopno přiřazovat kódovaným symbolům desetinný počet bitů a tak dosahuje lepšího, nebo stejně dobrého kompresního poměru jako Huffmanovo kódování. Algoritmus aritmetického kódování reprezentuje celou kódovanou zprávu s pomocí intervalu (0, 1]. Interval je dělen v závislosti na pravděpodobnosti výskytu kódovaných symbolů. Postupným kódováním symbolů se interval zmenšuje, dokud nejsou zakódovány všechny symboly. Výsledkem kódování je pak jedno reálné číslo ze vzniklého intervalu.

7 Popis programového řešení webové aplikace

Pro vývoj webové aplikace byla zvolena platforma Microsoft SilverLight 2.0. Jedním z důvodů je možnost použití zdrojových kódů, které již byly napsány pro prostředí Mve-2, což značně urychlilo vývoj aplikace. Dalšími důvody byla možnost práce s vlákny a napsání kódu celé aplikace v jediném jazyce (C#). Pouze pro rozvržení komponent na okně internetového prohlížeče bylo nutné použít jazyk XAML.

Webová aplikace slouží pouze k dekompresi a zobrazení animovaných modelů. Z tříd naprogramovaných pro Mve-2 byly proto použity pouze třídy Decompressor, Loader a ArithCoder. Zdrojové kódy zůstaly bez významných změn, proto zde nebudou znovu popisovány. Jejich bližší popis lze nalézt v kapitole 6 Popis programového řešení pro Mve-2.

Jediná významná změna se týká výstupu poskytovaného třídou Decompressor. Původní verze této třídy v Mve-2 sestavovala sérii modelů z trojúhelníků. Protože webová aplikace zobrazuje pouze vrcholy modelů, není třeba trojúhelníky sestavovat a na výstup jsou posílány pouze série vrcholů modelu. V projektu webové aplikace byly třídy Loader a Decompressor sloučeny do jedné třídy, která se jmenuje Animation.

Naprogramovaná aplikace umožňuje přehrávání animací komprimovaných algoritmem Coddyac. Přehrávané animace lze pozastavovat (Pause), posouvat po jednotlivých snímcích, zvětšovat a zmenšovat. Animované modely jsou zobrazeny pouze jako množiny bodů, které představují vrcholy trojúhelníkové sítě modelu. Trojrozměrná data modelu jsou na dvojrozměrná převedena tak, že je z nich před vykreslením odstraněna hloubková souřadnice. Zobrazovaný model lze také rotovat kolem svislé osy, ale rotace je prováděna pouze kolem osy procházející bodem (0, 0, 0). Webová aplikace také po načtení modelu automaticky zarovná model na střed vykreslovací plochy tak, aby model během přehrávání animace nezmizel mimo tuto plochu. Stručná uživatelská příručka je umístěna v příloze Příloha 3.

Protože je množství vykreslovaných bodů veliké (řádově tisíce až desetitisíce) a vykreslování není akcelerováno grafickou kartou, přehrávání animací je pomalé. Proto pro zvýšení rychlosti přehrávání byla implementována možnost redukce vykreslovaných vrcholů modelu. Tato redukce je řešena velmi jednoduše. Navolí-li uživatel hodnotu redukce např. 10, bude algoritmus na vykreslovací plochu odesílat pouze každý 10. vrchol.

Vývoj aplikace na platformě SilverLight bohužel přinesl i nečekané potíže. SilverLight 2.0 nepodporuje práci s grafickou kartou a tedy ani renderování 3D grafiky, a proto jsou trojrozměrná data animací před vykreslením převáděna na dvojrozměrná. Údajně by se však podpora práce s grafickou kartou měla objevit v následujících verzích.

Druhou velkou nepříjemností je řešení obsluhy okna aplikace. Celé okno aplikace je obsluhováno pouze jedním vláknem. Pokud aplikace například právě zpracovává stisknutí tlačítka, vlákno okna aplikace se věnuje pouze tomuto úkolu a zablokuje tak možnost zpracování stisknutí ostatních tlačítek, nebo překreslení (aktualizace) okna. Vlákno, ve kterém probíhá cyklus výpočtu a vykreslení snímků animace, tak musí být pravidelně pozastavováno (sleep), protože při každém přístupu k vykreslovací ploše okna aplikace je znemožněno aktualizování okna samotným prohlížečem. Pokud není vlákno s cyklem pozastaveno a hlavní vlákno okna aplikace nemá čas na překreslení vykreslovací plochy okna, dochází sice k odesílání dat na vykreslovací plochu, ale na ploše se nic nevykresluje. Pozastavování vlákna má bohužel viditelný vliv na plynulost přehrávání animace.

Při programování bylo také nepříjemné, že dokumentace k SilverLight není ještě úplně hotová a na internetu je k dispozici jen velmi málo tutoriálů, ale psaní webových aplikací v jazyce C# a návrh grafického rozhraní s pomocí jazyka XAML je opravdu pohodlné.

Detaily implementace jsou popsány v komentářích v přiloženém zdrojovém kódu.

8 Výsledky testů a měření

Funkčnost a rychlost algoritmu Coddyac byla testována na osobním počítači s procesorem AMD 2,01GHz a 1GB RAM. Byly testovány moduly pro Mve-2 i webová aplikace. Níže uvedené časy komprese a dekomprese jsou aritmetickým průměrem 10 měření. Z výsledů měření je patrné, že čas dekomprese je mnohem kratší než čas komprese.

Dekompresní časy webové aplikace jsou několikanásobně lepší než dekompresní časy v prostředí Mve-2. To je způsobeno tím, že webová aplikace nesestavuje modely z trojúhelníků, ale používá pouze jejich vrcholy. Vytvoření trojúhelníků je časově náročnější.

Velikosti zkomprimovaných souborů by samozřejmě mohly být menší, ale při kompresi bylo dbáno na to, aby rozdíly mezi původní a zkomprimovanou animací nebyly viditelné. Kompresní poměr je ve všech měřených případech poměrně velký a u žádné komprimované animace nedošlo k viditelnému defektu (jak topologie, tak geometrie). Řešení úlohy tedy lze považovat za funkční.

Dance

počet vrcholů: 7061
počet snímků: 201
počet bázových vektorů: 20
původního velikost: 40.8MB
velikost po kompresi: 58.0kB
čas komprese (Mve-2): 40s
čas dekomprese (Mve-2): 9s
čas dekomprese web: 2s

Humanoid

počet vrcholů: 7646
počet snímků: 154
počet bázových vektorů: 20
původní velikost: 77.4MB
velikost po kompresi: 84.1kB
čas komprese (Mve-2): 23s
čas dekomprese (Mve-2): 7s
čas dekomprese web: 2s

Chicken

počet vrcholů: 3034
počet snímků: 400
počet bázových vektorů: 20
původní velikost: 30.3MB
velikost po kompresi: 71.3kB
čas komprese (Mve-2): 149s
čas dekomprese (Mve-2): 7s
čas dekomprese web: 2s

Jump

počet vrcholů: 15830
počet snímků: 222
počet bázových vektorů: 20
původní velikost: 90.4MB
velikost po kompresi: 115kB
čas komprese (Mve-2): 96s
čas dekomprese (Mve-2): 23s
čas dekomprese web: 6s

Walk

počet vrcholů: 35699
počet snímků: 187
počet bázových vektorů: 20
původní velikost: 187MB
velikost po kompresi: 283kB
čas komprese (Mve-2): 159s
čas dekomprese (Mve-2): 42s
čas dekomprese web: 11s

Z naměřených časů je přibližně 90% času potřebného ke kompresi stráveno výpočty třídy PCA a nejméně 50% času potřebného k dekompresi v prostředí Mve-2 je spotřebováno sestavováním trojúhelníků modelu metodou createMesh.

9 Závěr

Při kompresi a dekompresi implementovaným algoritmem Coddyac nedochází k poškození topologie modelů a geometrie modelů se mění pouze proto, že je komprimována ztrátovou kompresí. Algoritmus EdgeBreaker postihuje kompresi a dekompresi topologie trojúhelníkových sítí s hranicí i takové, které se skládají z více než jedné komponenty. Dá se tedy říct, že webová aplikace i moduly pro prostředí Mve-2 jsou plně funkční.

Zkomprimované animace jsou 300x až 1000x menší než původní animace, kompresní poměr je tedy dostatečně velký na to, aby se vyplatilo algoritmus Coddyac používat.

V budoucnu by bylo určitě vhodné se pokusit o takové úpravy zdrojového kódu, které by zkrátily čas komprese a dekomprese a zrychlily přehrávání animací webovou aplikací.

Až na potíže při implementaci webové aplikace v prostředí Microsoft SilverLight probíhalo zpracování této práce bez větších nepříjemností.

Přehled zkratek

EB EdgeBreaker, algoritmus sloužící ke kompresi topologie trojúhelníkových sítí.

PCA Principal Component(s) Analysis, statistická a kompresní metoda

CT Corner Table, datová struktura využívaná algoritmem EdgeBeaker

CLERS Řetězec znaků složených ze symbolů C, L, E, R a S, je používán algoritmem EdgeBreaker pro popis topologie trojúhelníkových sítí

Použitá literatura

[1] Edgebreaker: Connectivity compression for triangle meshes, J. Rossignac, IEEE Transactions on Visualization and Computer Graphics, Vol. 5, No. 1, January - March 1999

[2] WRAP&Zip decompression of the connectivity of triangle meshes compressed with edgebreaker, Jarek Rossignac, Andrzej Szymczak, November 1999 Computational Geometry: Theory and Applications, Volume 14 Issue 1-3

[3] 3D Compression Made Simple: Edgebreaker with Zip&Wrap on a Corner-Table, Jarek Rossignac, May 2001, Proceedings of the International Conference on Shape Modeling & Applications SMI 01

[4] Edgebreaker: A Simple Compression for Surfaces with Handles, J. Rossignac, Hélio Lopes, Alla Safanova, Geovan Tavares, Andrzej Szymczak

[5] 3D compression made simple: Edgebreaker with Zip&Wrap on a Corner-Table, Jarek Rossignac, College of Computing and GVU Center, Georgia Institute of Technology

[6] Examples of Input and Output files in ASCII format for Torus example
http://www.gvu.gatech.edu/%7Ejarek/edgebreaker/eb/Simple_Torus.html#_HANDLES

[7] A tutorial on Principal Component Analysis, Lindsay I Smith, February 26, 2002

[8] CoDDyAC: Connectivity Driven Dynamic Mesh Compression,
Libor Váša, Václav Skala, 3DTV Conference 2007.

[9] Microsoft Silverlight 2 Tutorials
http://silverlight.net/learn/tutorials.aspx

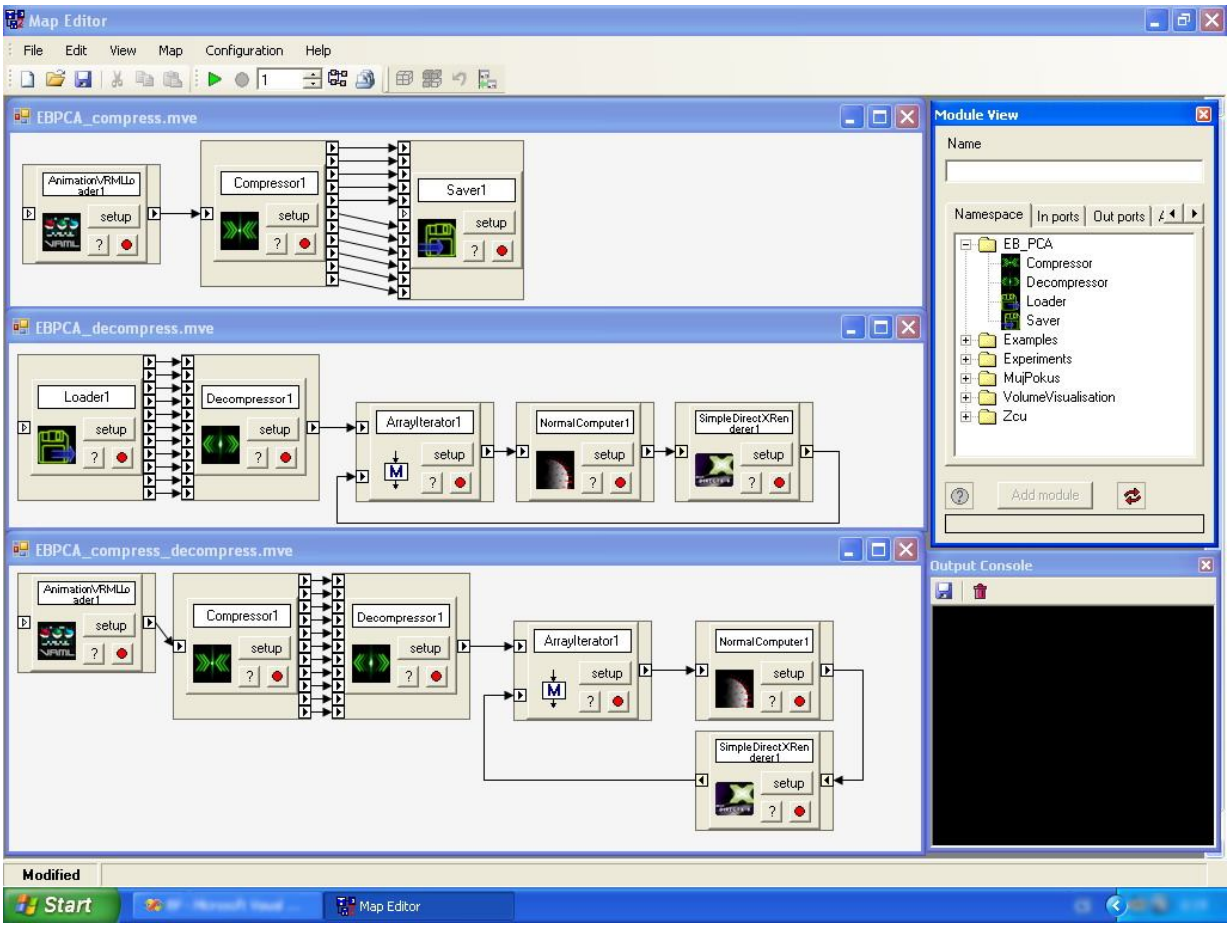
[10] Michael Sysnc's Silverlight 2.0 – Tutorials
http://michaelsync.net/2008/02/24/links-silverlight-2wpf-tutorials

[11] Kit3D - a 3D C# graphics engine for Microsoft Silverlight
http://www.codeplex.com/Kit3D

Přílohy

Příloha 1

Ukázka map v prostředí Mve-2



Na obrázku můžete vidět tři mapy, moduly Compressor, Decompressor, Saver a Loader byly vytvořeny v rámci této práce, zbylé moduly jsou součástí Mve-2.

Příloha 2

Ukázka zdrojového kódu metody compress algoritmu EdgeBreaker

```
/// <summary>
/// Projde topologii modelu a zkomprimuje ji
/// </summary>
/// <param name="c">Startovaci corner</param>
private void compress(int c)
{
  while (true)
  {
     U[tri(c)] = 1; //oznacime trojuhelnik jako navstiveny
     lastTriangle++;

     #region Kontrola handles
     //nejdriv se koukneme po handlech
     if (right(c) != -1)//-1 v pripade ze neexistuje (hole)
     {
         if (U[tri(right(c))] > 1) //pokud byl trojuhelnik vpravo "S -handle"
         {
             //ulozime do "handles" soucasny vrchol a jeho "oposite corner" ulozeny v "U"
             //musime pricist 3*numOfComponents, protoze Decompressor bude pro
             //kazdou novou komponentu pocitat + 1 trojuhelnik navic
             handles.AddLast(U[tri(right(c))]+3*numOfComponents);
             handles.AddLast((lastTriangle + numOfComponents
                              + invisibleTriangles) * 3 + 1); //pro
                                        decompressor to bude totez jako next(c)
         }
         else if (left(c) != -1) //-1 v pripade ze neexistuje (hole)
              //pokud byl trojuhelnik vlevo typu "S -handle"
              if (U[tri(left(c))] > 1)                          {
                 //ulozime do "handles" soucasny vrchol a jeho "oposite corner"
                   ulozeny v "U"
                 //musime pricist 3*numOfComponents, protoze Decompressor
                   bude pro
                 //kazdou novou komponentu pocitat + 1 trojuhelnik navic
                 handles.AddLast(U[tri(left(c))]+3*numOfComponents);
                 handles.AddLast((lastTriangle + numOfComponents
                                 + invisibleTriangles) * 3 + 2); //pro
                                   decompressor to bude totez jako prev(c)
               }
     }
     #endregion

     if (M[V[c]] <= 0) //pokud jsme vrchol jeste nenavstivili, -> novy trojuhlenik
     {
       #region Vypocet geometrie
       //ulozime vrchol modelu, protoze je novy -prave vytvoreny
       lastVertice++;
       //odhadneme koeficienty
       prediction = predictCoefs(c);

       //zjistime rozdil mezi odhadem a zkutecnosti
       reziduum = substractCoefs(pca.GetCoefs(trajectories[V[c]],
                                 coefficientLength), prediction);

       //upravime reziduum
       reziduumCoefs = modifyCoef(reziduum);

       //zapamatujeme si, jak by to odhadl Decompressor
       predictedCoefs[V[c]] = addCoefs(prediction,
                                  multiplyDeltaD(reziduumCoefs));
```

```
            if (lastVertice >= coefficients.Length)
            {
                bonusCoefs.AddLast((int[])reziduumCoefs);
            }
            else
            {
                //zjisteny rozdil upravime a ulozime
                coefficients[lastVertice] = reziduumCoefs;
            }
            #endregion

            //pokud je vrchol soucasti nenavstivene Hole
            // tak oznacime vsechny vrcholy hole za navstivene
            // a zapiseme do hole tabulky
            if (M[V[c]] == -1)
            {
                sCounter++;
                //ulozime znak z CLERS -konkretne "S"
                CLERS[lastTriangle - 1] = 'S';
                M[V[c]] = c + 1;
                markHole(c);
                c = right(c);
            }
            else
            {
                //ulozime znak z CLERS -konkretne "C"
                CLERS[lastTriangle - 1] = 'C';
                M[V[c]] = c+1;
                c = right(c);
            }
        }
        else  //vrchol uz jsme navstivili
        {
            if ((right(c) == -1) ||
                ((right(c) != -1) && (U[tri(right(c))] > 0))) //uz jsme
                                                    navstivili pravy trojuhelnik
            {
                if ((left(c) == -1) ||
                    ((left(c) != -1) && (U[tri(left(c))] > 0))) //uz
                                        jsme navstivili levy i pravy trojuhelnik
                {
                    CLERS[lastTriangle-1] = 'E';
                    return;  //jinam uz jit nemuzeme
                }
                else  //pravy uz byl navstiven, levy jeste nebyl navstiven
                {
                    CLERS[lastTriangle-1] = 'R';
                    c = left(c);
                }
            }
            else
            {
                if ((left(c) == -1) ||
                    ((left(c) != -1) && (U[tri(left(c))] > 0))) //uz
                                        jsme navstivili levy trojuhelnik a pravy ne
                {
                    CLERS[lastTriangle-1] = 'L';
                    c = right(c);
                }
                else  //levy ani pravy jeste nebyl navstiven
                {
```

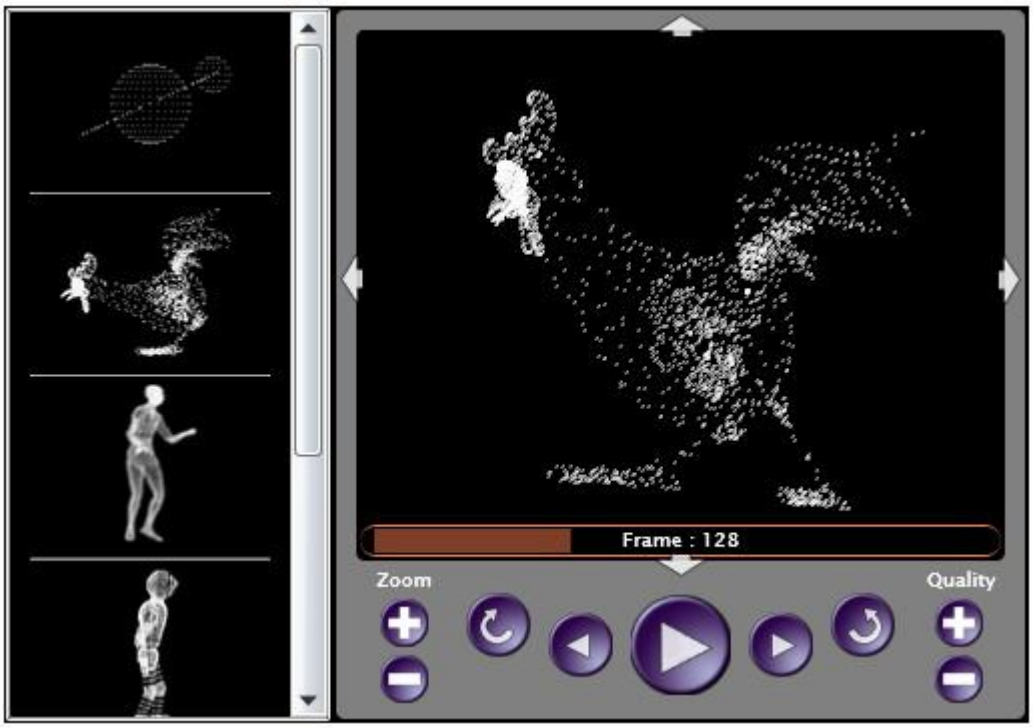
```
                    //zjistime, jestli je mozne se dostat k corneru, ktery vrchol oznacil
                    if (checkAvailability(c) == true)
                    {
                        sCounter++; // zaznamename celkovy pocet 'S' symbolu
                                    v CLERS
                        U[tri(c)] = (lastTriangle +
                                  invisibleTriangles) * 3 + 2; //pro
                                      decompressor to bude totez jako prev(c)
                        CLERS[lastTriangle - 1] = 'S';
                        compress(right(c));
                        c = left(c);

                        if (U[tri(c)] > 0) break;//pokud by tu tato
                                                    podminka nebyla,
                          //mohlo by dojit k vytvareni "E" -posloupnosti
                          //trojuhleniku se dostala z prave hrany az do leve -handle
                    }
                    else //pokud ne, misto S zavedeme C
                    {
                        //ulozime vrchol modelu, protoze je novy -prave vytvoreny
                        #region Vypocet geometrie
                        //ulozime vrchol modelu, protoze je novy -prave vytvoreny
                        lastVertice++;
                        //odhadneme koeficienty
                        prediction = predictCoefs(c);

                        //zjistime rozdil mezi odhadem a zkutecnosti
                        reziduum = substractCoefs(pca.GetCoefs(
                            trajectories[V[c]], coefficientLength),
                            prediction);

                        //upravime reziduum
                        reziduumCoefs = modifyCoef(reziduum);

                        //zapamatujeme si, jak by to odhadl Decompressor
                        predictedCoefs[V[c]] = addCoefs(prediction,
                                    multiplyDeltaD(reziduumCoefs));

                        if (lastVertice >= coefficients.Length)
                        {
                          bonusCoefs.AddLast((int[])reziduumCoefs);
                        }
                        else
                        {
                          //zjisteny rozdil upravime a ulozime
                          coefficients[lastVertice] = reziduumCoefs;
                        }
                        #endregion
                        //ulozime znak z CLERS -konkretne "C"
                        CLERS[lastTriangle - 1] = 'C';
                        // M[V[c]] = c + 1;
                        c = right(c);
                    }
                }
            }
        }
    }
}
```

Příloha 3

Uživatelská příručka k webové aplikaci

Uživatelská příručka

Tato webová aplikace slouží k dekompresi a přehrávání trojrozměrných animací zkomprimovaných algoritmem Coddyac. Ukázkový náhled aplikace můžete vidět na obr. (a).



obr. (a)
náhled webové aplikace

Ovládání aplikace je velice jednoduché. V levé části okna přehrávače se nachází sloupec s obrázky, který slouží k výběru animace, kterou chcete přehrát. Klepnutím na jeden z těchto obrázků se načte a dekomprimuje trojrozměrná animace a její první snímek se zobrazí v černém okénku přehrávače uprostřed. Modely trojrozměrných animací jsou reprezentovány bílými tečkami, které představují vrcholy trojúhelníků, z nichž se model skládá. V dolní části černého okénka můžete vidět číslo snímku, který je právě vykreslen a pruh, který ukazuje, jak velká část animace už je přehrána.

Přehrávání animace spustíte kliknutím na tlačítko Play (velká šipka směřující doprava), které se nachází uprostřed okna přehrávače. Pokud se animace přehrává, je na stejném místě zobrazeno tlačítko Pause, které přehrávání animace zastaví (pokud na něj kliknete).

Kromě tlačítka Play je možné snímky animace měnit dvěmi menšími tlačítky po stranách tlačítka Play. Kliknete-li na tlačítko s šipkou směřující vpravo, zobrazí se následující snímek, pokud kliknete na tlačítko s šipkou směřující vlevo, zobrazí se předchozí snímek.

Ještě dále od tlačítka Play se nachází tlačítka se symbolem zatočené šipky. Těmito tlačítky můžete trojrozměrný model rotovat kolem svislé osy. Kromě rotace je k dispozici ještě zvětšení a zmenšení modelu. To můžete provést kliknutím na tlačítka + a – pod nápisem Zoom.

Poslední funkcí, kterou přehrávač poskytuje, je možnost redukce zobrazovaných vrcholů modelu. Modely obvykle mají velké množství vrcholů a proto je přehrávání jejich animací pomalé. Redukcí počtu zobrazených vrcholů se přehrávání animace zrychlí. Snížit nebo zvýšit počet zobrazených vrcholů můžete tlačítky + a – pod nápisem Quality.

Chcete-li umístit tuto aplikaci na vlastní HTML stránky, postačí, když do zdrojového kódu stránky dopíšete následující kód:

```
<object data="data:application/x-silverlight,"
        type="application/x-silverlight-2-b1"
        width="492" height="358">

        <param name="source" value="APLIKACE.xap"/>
</object>
```

Text APLIKACE.xap představuje umístění a jméno souboru s touto aplikací. Do stejného adresáře, ve kterém se nachází soubor aplikace, je také třeba umístit soubor s jménem PlayList.txt, který obsahuje jména a umístění souborů s animacemi. Na konci každé takové položky musí být středník. Obsah souboru PlayList.txt může vypadat například takto:

```
Planet.cmp;Chicken.cmp;Dance.cmp;Humanoid.cmp;Jump.cmp;Walk.cmp;
```

Pokud je v playlistu aplikace více než jedna animace, zobrazí se vedle okna přehrávače jejich seznam. Každá animace je v tomto seznamu reprezentována obrázkem velikosti 120x90 pixelů ve formátu PNG, který je třeba vytvořit a umístit do adresáře, ve kterém se nachází soubor animace.