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

Bakalářská práce

Návrh systému vyrovnávací paměti pro grafické výpočty

Plzeň 2016 Lukáš Hruda

Prohlášení

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

V Plzni dne 2. května 2016

Lukáš Hruda

Abstract

In the context of this work, caching can be described as the process whereby a program stores calculated data into its memory or onto a hard drive in order to avoid recalculating it in case it is needed more than once. This document describes various known cache policies and contains a deeper analysis and description of a specific cache system designed and developed for the use in *MAF2* framework. It also describes the basic principles of the *VTK* visualization toolkit used by the *MAF2* framework. The system is therefore compatible with *VTK* data objects and it is also as generic as possible so that it can be used for caching almost any type of data.

Abstrakt

V kontextu této práce může *cachování* být popsáno jako proces, kdy si program v paměti či na pevném disku uchovává výsledky výpočtů z důvodu vyhnutí se jejich opětovnému počítání v případě, kdy je stejný výsledek použit více než jedenkrát. Tento dokument popisuje několik známých politik *cachování* a obsahuje hlubší analýzu a popis konkrétního *cachovacího* systému navrženého a vyvinutého pro potřebu využití v *MAF2 frameworku*. Zároveň jsou zde popsány základní principy vizualizačního nástroje *VTK*, který je *MAF2 frameworkem* využíván. Z tohoto důvodu je navržený systém kompatibilní s datovými objekty nástroje *VTK* a zároveň je dostatečně obecný, aby bylo možné ho použít pro *cachování* téměř libovolného typu dat.

Obsah

1 Úvod

Cílem této práce je navrhnout a implementovat knihovnu, s jejíž pomocí bude možné uchovávat výsledky složitých výpočtů a umožnit jejich znovupoužití bez nutnosti výpočty opakovat. Knihovna by měla nalézt využití ve vizualizační aplikaci *lhpBuilder*, na jejímž vývoji se podílí Katedra informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity v Plzni. Tato aplikace je tvořena s použitím *MAF2 frameworku*, který hojně využívá vizualizační nástroj *VTK*, je tedy nutné, aby knihovna byla kompatibilní s datovými objekty, které *VTK* definuje, protože právě tyto objekty bude nejčastěji potřeba uchovávat.

Mechanismus s touto funkcionalitou je již v aplikaci implementován, ovšem není využíván, protože trpí dvěma zásadními nedostatky. Prvním nedostatkem je fakt, že mechanismus je implementován pro jeden konkrétní vizualizační problém a pro nic jiného ho prakticky nelze využít. Druhým nedostatkem je nulový ohled na omezenou paměť. Mechanismus totiž uchovaná data nijak systematicky neodstraňuje z paměti, do které je ukládá, což může být velký problém především v případě, kdy tato data jsou nemalého objemu.

Z těchto nedostatků vyplývají i hlavní požadavky na návrh knihovny. Ta by měla být dostatečně obecná, aby bylo možné ji využít pro prakticky libovolný problém. Bude tedy nutné ji navrhnout tak, aby dokázala zpracovat data libovolného typu, nejen datové objekty nástroje *VTK*. Dále bude nutné navrhnout systém, který se postará o automatické odstraňování uložených dat z paměti na základě jejich velikosti a užitečnosti, protože je nežádoucí, aby v paměti zabírala místo velká data, která nejsou využívána.

Před samotným návrhem a implementací bude potřeba provést analýzu problému zahrnující nastudování známých *cachovacích* politik. Taktéž bude potřeba se seznámit se základními principy fungování nástroje *VTK*.

2 Cache obecně

Cache (česky vyrovnávací paměť) obecně slouží pro uložení dat, o kterých víme nebo předpokládáme, že budou často využívána. Pokud například program provádí složitý výpočet, jehož výsledkem jsou data, která budou využita několikrát, je výhodné tato data někam uložit, aby nebylo nutné výpočet opakovat. Podobně pokud program často využívá stejná data uložená na nějakém pomalém paměťovém médiu, například na pevném disku, vyplatí se zkopírovat tato data na médium rychlejší, například do operační paměti, a dále je číst odtud.

2.1 Využití cache

Princip popsaný v předchozím odstavci má mnoho různých využití. Jedním z nich je i již zmíněná aplikace *lhpBuilder*. Ta při procesu vizualizace provádí spoustu výpočtů, jejichž výsledkem jsou data, která jsou v aplikaci využita více než jednou. Bylo by tedy velmi vhodné tato data někam ukládat, a zamezit tak opětovnému provádění časově náročných výpočtů, což je v podstatě cílem této práce. Kromě tohoto konkrétního příkladu má ale *cache* mnoho dalších různých aplikací. Zde je základní popis několika z nich.

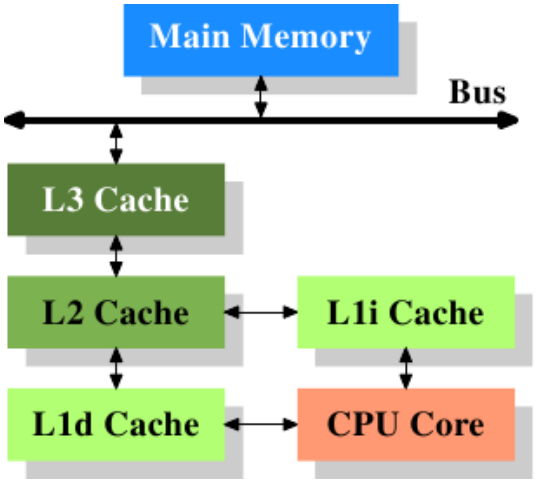
2.1.1 CPU cache

Většina moderních procesorů má vlastní vyrovnávací paměť. Ta slouží pro uchování kopie dat z operační paměti, u kterých procesor předpokládá, že je bude v blízké době potřebovat.

Vyrovnávací paměť procesorů je většinou víceúrovňová. Typicky má tři úrovně, označené jako L1, L2 a L3, kde L1 je úroveň nejnižší. Platí, že čím nižší je úroveň vyrovnávací paměti, tím je paměť rychlejší, zato však menší. Když procesor potřebuje pracovat s daty, zkontroluje nejdříve, zda se daná data nachází na nejnižší úrovni vyrovnávací paměti, tedy na úrovni L1. Pokud na této úrovni data nejsou, pokračuje procesor kontrolou vyšší úrovně. Pokud se data nenachází na žádné úrovni vyrovnávací paměti, musí

k nim procesor přistoupit do operační paměti.

Úroveň L1 bývá často rozdělena na dvě části. Jedna část je určena pro ukládání dat a druhá pro ukládání instrukcí programů. Systém s procesorem používajícím tříúrovňovou vyrovnávací paměť ukazuje obrázek 2.1.



Obrázek 2.1: Procesor s tříúrovňovou vyrovnávací pamětí (obrázek převzat z [5])

L1d je část L1 určená pro ukládání dat, L1i je část L1 určená pro ukládání instrukcí. Obrázek je pouze schématický. Ve skutečnosti data na cestě z procesoru do operační paměti nemusejí nutně projít přes všechny úrovně vyrovnávací paměti. Podrobný design závisí na konkrétním procesoru.

U procesorů s více než jedním jádrem většinou platí, že každé jádro má svoji vlastní vyrovnávací paměť úrovně L1 a L2. Úroveň L3 pak bývá společná pro všechna jádra.

Více informací o vyrovnávací paměti procesoru lze nalézt například zde [4] nebo zde [5].

2.1.2 Disková cache

Stejně jako procesory i pevné disky mívají svoji vyrovnávací paměť. Ta slouží především k vyrovnání toku dat mezi pevným diskem a zbytkem počítače. Pevný disk je ve srovnání s ostatními periferiemi počítače velmi pomalý a často nezvládá dostatečně rychle zapisovat příchozí data, ta jsou proto nejdříve uložena do jeho vyrovnávací paměti a odtud jsou postupně

čtena a zpracovávána. Zařízení, které data poslalo, tak nemusí čekat na jejich zápis a může pokračovat ve své činnosti, samozřejmě pouze v případě, kdy vyrovnávací paměť není plná. Tato paměť slouží také k uložení dat, u kterých je pravděpodobné, že budou od disku v blízké době vyžadována. Její velikost může mít znatelný vliv na rychlost celého počítače. Více informací lze nalézt například zde [6].

S pevným diskem souvisí také takzvaná *buffer cache*. Jedná se o paměť pod správou operačního systému, která se používá pro uložení kopie takových dat z pevného disku, která budou pravděpodobně v krátké době několikrát použita. Čtení těchto dat z *buffer cache* je pak znatelně rychlejší než jejich čtení z pevného disku. Takovými daty může být například obsah souboru, který byl nedávno otevřen v textovém editoru. Pokud operační systém předpokládá, že bude soubor brzy otevřen znovu, může si jeho obsah takto uchovat. Tato paměť se využívá i při zápisu dat. Často jsou totiž data po zápisu brzy znovu čtena, navíc je takto možné odložit zápis dat a provést jej později na pozadí bez toho, aby zpomaloval program, který ho vyvolal. Více o *buffer cache* lze nalézt například zde [3].

2.1.3 Webová cache

Webová *cache* slouží k ukládání obsahu webových stránek. Je běžnou součástí webových prohlížečů, které do ní ukládají obsah dříve navštívených webů. Při opětovném otevření nějaké webové stránky se tak nemusí znovu stahovat celý její obsah, ale načte se z disku, kam jej prohlížeč dříve uložil. To nejen šetří čas, ale také snižuje zátěž sítě. Více informací o *cache* v prohlížečích je zde [10].

Ve správě počítačových sítí se také používá *cache* server, který uchovává často vyžadovaný internetový obsah. Tento obsah tak při požadavku není nutné stahovat ze serveru, ze kterého původně pochází, ale stáhne se z bližšího *cache* serveru, což je rychlejší a snižuje to zátěž sítě. Více informací lze nalézt například zde [26].

Další *cache* související s počítačovými sítěmi je například DNS *cache*, která má velký význam v procesu překladu doménových jmen na příslušné IP adresy. Informace o ní lze nalézt zde [23].

2.2 Politiky cachování

Politika *cachování* je jakýsi soubor pravidel, podle kterých se provádí rozhodnutí o nahrazení dat uložených v *cache* v případě, kdy chceme uložit nová data, ale *cache* je plná. V takové situaci je potřeba nějaká data z *cache* odstranit. Ideálně bychom odstranili data, která budou v budoucnu použita nejméně často, popřípadě vůbec. Ovšem vzhledem k tomu, že budoucnost předpovědět nedokážeme, tuto informaci nemáme a právě k rozhodnutí, která data odstranit, slouží *cachovací* politika.

Zde je popis několika politik *cachování*. Více informací lze nalézt například zde [13].

2.2.1 Jednoduché politiky

Jednoduché politiky nepoužívají žádné statistické údaje ani další informace o uložených datech. Jejich výhodou je především snadná implementace.

RAND

RAND nebo *Random* je velmi jednoduchá politika, která určuje data k odstranění z *cache* zcela náhodně. Při nutnosti uvolnit místo v *cache* tedy jednoduše dojde k náhodnému výběru dat, která jsou pak odstraněna, popřípadě nahrazena novými daty. Snad jedinou výhodou této politiky je její velice snadná implementace.

FIFO

FIFO je zkratka anglického *First-In-First-Out*. Opět se jedná o velmi jednoduchou politiku. V případě nutnosti odstranění dat z *cache* jsou odstraněna ta data, která jsou v ní uložena nejdelší dobu, tedy data, která byla do *cache* uložena nejdříve. Pro implementaci může posloužit obyčejná fronta.

FIFO with second chance

FIFO with second chance, někdy označována jako *FIFO2*, je politika, která bere v úvahu informaci o tom, zda nějaká data byla použita. Využívá dvě fronty. Jedna slouží k uložení aktivních dat a druhá k uložení dat neaktivních. V případě potřeby odstranění dat z *cache* dojde k odstranění nejstarších neaktivních dat. Rozdíl oproti *FIFO* spočívá v tom, že pokud jsou od *cache* žádána data, která jsou neaktivní, přesunou se do fronty pro aktivní data a tím se zamezí jejich brzkému odstranění z *cache*. Fronta neaktivních dat je doplňována nejstaršími daty z fronty dat aktivních. K doplnění může dojít po každém odstranění dat nebo tehdy, když fronta neaktivních dat je prázdná [17].

CLOCK

Stejně jako *FIFO with second chance* i politika *CLOCK* využívá informaci o použití dat. Data jsou uložena v kruhovém *bufferu*, což znamená že pokud při jeho průchodu dojdeme na konec a chceme pokračovat, začneme ho procházet od začátku. U každého prvku uloženého v *cache* rozlišujeme dva stavy, a to *použit* a *nepoužit*. Kdykoliv jsou po *cache* žádána nějaká data, která jsou uložena v *bufferu*, je stav příslušného prvku nastaven na *použit*. V případě požadavku na odstranění dat z *cache* se začne *buffer* procházet a v průchodu se pokračuje do té doby, dokud není nalezen prvek, jehož stav je *nepoužit*, a ten je odstraněn. Každému prvku, který je při průchodu navštíven a jehož stav je *použit*, je stav změněn na *nepoužit* [20].

2.2.2 Politiky založené na statistice

Tyto politiky využívají statistické informace o četnosti přístupů k datům nebo o době, před jakou byla daná data naposledy použita.

LRU

LRU je zkratka anglického *Least Recently Used*. Při nutnosti odstranění dat z *cache* jsou odstraněna ta data, která byla naposledy použita před nejdelší dobou. Pro implementaci je možné využít prioritní frontu, kde priorita je

určena časovou značkou posledního přístupu k daným datům. *LRU* se zdá být vhodnou politikou pro *cachování* velkých datových souborů. Nevýhodou této politiky je fakt, že neuvažuje četnost přístupů k datovým položkám. Tedy pokud je jedna položka používána opakovaně, může být stále z *cache* odstraněna, pokud časový interval mezi jednotlivými přístupy je dlouhý [14], [11], [25].

Jako aproximace politiky *LRU* se někdy používají politiky *FIFO with second chance* a *CLOCK* [17], [20].

LFU

LFU je zkratka anglického *Least Frequently Used*. Každá datová položka uložená v *cache* má počítadlo přístupů. Při každém použití nějakých dat je počítadlo příslušné položky zvětšeno o 1. Při nutnosti odstranění dat z *cache* je odstraněna ta položka, jejíž počítadlo přístupů má nejnižší hodnotu. Nevýhodou této politiky je fakt, že data, která byla použita mnohokrát během krátké doby, nebudou pravděpodobně nikdy z *cache* odstraněna, i když již v budoucnu nebudou použita [14], [11], [25].

MRU

MRU je zkratka z anglického *Most Recently Used*. Tato politika je v podstatě přesným opakem politiky *LRU*. Při nutnosti odstranění dat z *cache* jsou jednoduše odstraněna ta data, která byla uložena před nejkratší dobou. Tato politika má využití v případě, kdy je přistupováno ke stále stejným datům v cyklu, tedy když známe pořadí, ve kterém budou data použita, a víme, že po použití poslední datové položky bude opět použita položka první. Je zřejmé, že v takovém případě bude za nejdelší dobu použita položka, která byla naposledy použita před dobou nejkratší [16]. Obecně je tato politika vhodná pro případy, kdy chceme mít snadný přístup k historickým informacím [11].

2.2.3 Hybridní politiky

Hybridní politiky jsou kombinací několika různých politik. Nejčastěji se jedná o kombinaci *LRU* a *LFU*.

2Q

Politika *2Q* je kombinací politik *FIFO* a *LRU*. Používá dvě fronty. Jednu pro tzv. *horká* data, spravovanou politikou *LRU*, a druhou pro tzv. *studená* data, spravovanou politikou *FIFO*. Při zápisu nové datové položky do *cache* je tato položka uložena do fronty pro *studená* data. V případě, že je libovolná položka v této frontě použita znovu, je přesunuta do fronty pro *horká* data. Pokud v jedné z těchto front dojde místo, jsou data odstraněna s respektováním příslušné politiky, tedy *LRU* pro frontu s *horkými* daty a *FIFO* pro frontu se *studenými* daty [21], [14].

LIRS

Při použití politiky *LIRS* dochází k měření vzájemné vzdálenosti dvou posledních přístupů k datům. Tato vzdálenost je označována jako *Inter-Reference Recency* (dále jen *IRR*) a její hodnota pro danou datovou položku je rovna počtu přístupů k jiným položkám, které nastaly mezi posledními dvěma přístupy k této položce. Dále se ke každé položce ukládá časová značka posledního přístupu. Celá *cache* je rozdělena do dvou částí. Jedna část je určena pro položky s nízkou *IRR*, nazývá se *Low Inter-Reference Recency set* (dále jen *LIR*) a obecně by měla tvořit velkou většinu celé *cache*. Druhá část je určena pro položky s vysokou *IRR*, ta se nazývá *High Inter-Reference Recency set* (dále jen *HIR*). Část *LIR* je naplněna datovými položkami s nejnižší hodnotou *IRR*, zbylé položky jsou uloženy v *HIR*. Datové položky, které se do *cache* nevejdou, v ní sice nejsou fyzicky přítomné, ale existují o nich záznamy v *HIR*, protože je potřeba znát jejich *IRR* a časovou značku jejich posledního použití. Když je potřeba v *cache* uvolnit místo, je odstraněna fyzicky přítomná datová položka z *HIR*, a to s použitím politiky *LRU* [19].

MQ

Politika MQ používá několik front spravovaných politikou *LRU*. Pokud je počet front stanoven na N, pak jsou jednotlivé fronty označeny jako $Q_0, Q_1, ..., Q_{n-1}$. Čím vyšší je index fronty, tím vyšší prioritu mají data v ní uložená. Priorita je určena jako počet přístupů k daným datům. V případě, kdy je potřeba v *cache* uvolnit místo, je odstraněna položka z neprázdné fronty s nejnižším indexem, a to s respektováním pravidel politiky *LRU*. Nové datové položky jsou vkládány do fronty Q_0. Politika dále obsahuje vý-

stupní frontu Q_{out}, spravovanou jako *FIFO*, do které jsou ukládány informace o odstraněných položkách [29].

FBR

Tato politika je kombinací *LRU* a *LFU*. *Cache* je rozdělena na tři části: *nová* část, *střední* část a *stará* část. Data jsou rozdělena do jednotlivých částí na základě doby, před kterou byla naposled použita, tedy nová data jsou uložena v *nové* části, *stará* část pak obsahuje data, která byla naposledy použita před nejdelší dobou. V případě nutnosti uvolnění místa v *cache* je odstraněna položka ze *staré* části, a to s respektováním pravidel politiky *LFU* [14].

LRFU

Tato politika je opět kombinací *LRU* a *LFU*. Při požadavku na uvolnění místa v *cache* je odstraněna položka s nejnižší hodnotou *CRF* (*Combined Recency and Frequency*). Hodnota *CRF* dané položky v daném čase je určena na základě počtu použití této položky a doby, před kterou byla položka použita naposledy. Podrobný způsob výpočtu hodnoty *CRF* lze nalézt například zde [22].

LRD

LRD je zkratka z anglického *Least Reference Density*. Tato politika vychází z politiky *LFU* a řeší její největší problém, kterým je fakt, že data, která byla použita mnohokrát během krátké doby, nebudou pravděpodobně nikdy z *cache* odstraněna. *LRD*, stejně jako *LFU*, zaznamenává pro jednotlivé položky počet jejich použití. Rozdíl je v tom, že tento počet se postupně snižuje. Ke snižování dochází vždy po určitém počtu přístupů do *cache* a je prováděno vždy na všech uložených datových položkách. Způsob snížení může být různý, používá se odečtení nebo vydělení předem určenou konstantou [18].

CRASH

CRASH je politika určená pro *cachování* datových bloků při čtení z pevného disku. Bloky jsou ukládány do různých množin, kde v jedné množině jsou vždy bloky, jejichž adresy jdou za sebou. Při nutnosti odstranění dat z *cache* je vybrána množina s nejvyšším počtem bloků a z této množiny je odstraněn blok s nejmenší adresou [14].

Další politiky

Za zmínku stojí také například politika *LRU-k*, která vychází z politiky *LRU*, nebo politika *ARC*, která je podobná politice *2Q*. Více informací o těchto politikách lze nalézt zde [13] nebo zde [14].

2.2.4 Analýza politik

Dostatečnou analýzu politik a výběr vhodné politiky, která by posloužila jako základ politiky pro výsledný *cachovací* systém, není možné provést před analýzou problému, k čemuž je potřeba nejdříve popsat knihovnu *VTK*. Důkladnější analýza politik *cachování* a výběr vhodné základové politiky se tedy nachází až v sekci 4.2.

3 Stručný popis VTK

Jak již bylo řečeno v předchozích kapitolách, *MAF2 framework* hojně využívá vizualizační knihovnu *VTK* (*The Visualization Toolkit*). Jedná se o knihovnu napsanou v programovacím jazyce C++, lze ji ovšem použít i v jiných programovacích jazycích, jako například Java nebo Python.

Knihovnu *VTK* lze použít pro vizualizaci nejrůznějších dat a informací od jednoduchých dvourozměrných grafů až po vícerozměrná vědecká data. *VTK* samozřejmě není pouze aparátem pro zobrazení výsledných dat, ale také pro jejich načítání, ukládání a pro různé způsoby jejich zpracování, mezi které patří například triangulace či decimace polygonové sítě, transformace objektů jako škálování nebo rotace a spousta dalších.

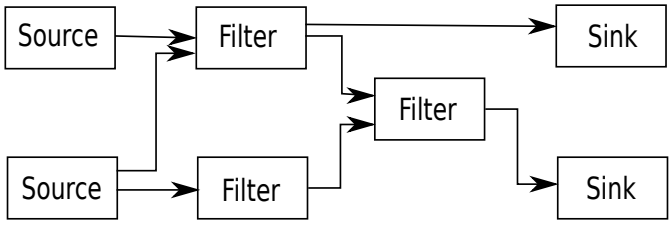
VTK bylo vytvořeno v roce 1993, je tedy ve vývoji již přes 20 let. Rozhraní knihovny se průběžně měnilo a verzi od verze se liší. Zdrojové kódy příkladů v této kapitole jsou napsány pro verzi 6.3.0. Vzhledem k použitým materiálům [27], [15] mohou ale některé obrázky či informace v textu příslušet verzím starším, což ovšem na vysvětlení principů knihovny nemá podstatný vliv.

3.1 Vizualizační pipeline

Vizualizační *pipeline* je pro *VTK* charakteristickým principem. Všechny operace nad daty jsou prováděny takzvanými filtry, kde každý filtr provádí pouze jednu specifickou operaci, například transformaci, jako je rotace. Vstupem filtru jsou grafická data, se kterými filtr provede příslušnou operaci, a výsledek této operace je výstupem filtru. Některé filtry mají pouze jeden vstup a jeden výstup, jiné mohou mít vstupů či výstupů více. Počet vstupů i výstupů může být pevně daný, nebo může být neomezený, vše záleží na operaci, kterou má daný filtr na starost. Filtry lze mezi sebou propojovat, což znamená, že výstup jednoho filtru může být napojen na vstup jiného filtru, čímž vzniká tzv. *pipeline*, česky roura.

Na samém počátku roury se vždy nacházejí datové zdroje (*sources*), což jsou procesy, které nemají žádný vstup a slouží k vytvoření nezpracovaných

dat. Může se jednat i například o načtení dat ze souboru. Na úplném konci roury jsou procesy, které jsou v angličtině označovány jako *sinks*. Vzhledem k tomu, že překlad slova *sink* do češtiny zní poněkud zvláštně (dřez, dno), budou v této práci tyto procesy dále nazývány jako koncové procesy. Koncové procesy nemají žádný výstup a jejich úkolem může být například zápis dat do souboru nebo zobrazení dat na obrazovku. Mezi datovými zdroji a koncovými procesy se může nacházet libovolný počet filtrů. Roura nemusí být přímočará, ale může se větvit, nebo dokonce obsahovat smyčky. Příklad roury ukazuje obrázek 3.1.



Obrázek 3.1: Ukázka jednoduché roury

Jednotlivé datové zdroje, filtry a koncové procesy jsou ve *VTK* reprezentovány instancemi příslušných tříd. Těchto tříd *VTK* obsahuje nespočet. Příkladem třídy definující filtr je *vtkElevationFilter*, jehož úkolem je přiřadit bodům v prostoru skalární hodnotu na základě jejich vzdálenosti od nějaké roviny. Příkladem datového zdroje je třída *vtkSphereSource*, jejíž instance generuje polygonovou síť koule. Příkladem koncového procesu je pak třída *vtkDataWriter*, pomocí které lze zapsat grafická data do souboru. Společným předkem všech datových zdrojů, filtrů a koncových procesů je třída *vtkAlgorithm*. Všechny třídy ve *VTK* včetně třídy *vtkAlgorithm* pak mají jednoho společného předka a tím je třída *vtkBaseObject*.

K provázání jednotlivých procesů slouží ve verzi 6 metody *SetInputConnection* a *GetOutputPort*, popřípadě ve starších verzích *SetInput* a *GetOutput*. Následující kus kódu v jazyce C++ ukazuje, jak může vypadat nasměrování výstupu jednoho filtru na vstup jiného.

```
filter2->SetInputConnection(filter1->GetOutputPort());
```

3.1.1 Spouštění procesů

Řízení spouštění procesů v rouře je takzvaně *demand-driven* (řízené požadavky). To znamená, že každý prvek roury poskytne data na výstup tehdy, když jsou žádána prvkem, do kterého je daný výstup nasměrován. Požadavek bývá typicky generován koncovým procesem (ovšem může být explicitně generován i jiným prvkem roury) a postupuje rourou v protisměru až k datovým zdrojům. Pak se směr průchodu rourou obrací a každý prvek poskytne data na své výstupy, takto se postupně ke koncovým procesům dostanou data, která byla žádána. Pro každý prvek se provádí kontrola, zda se od posledního požadavku změnil jeho stav nebo data na jeho vstupech. Pokud ne, prvek prostě poskytne na svých výstupech data, která má uložena od posledního požadavku. Pokud změna nastala, musí prvek spustit proces pro vytvoření výstupních dat a ta pak poslat na výstupy.

Pro vytvoření požadavku je používána metoda *Update*, která je volána nad konkrétním prvkem roury a která dá svým vstupním prvkům najevo, že od nich žádá výstup. Tato metoda je většinou volána automaticky koncovými procesy, ovšem lze ji zavolat i manuálně nad libovolným prvkem. Metoda *Update* je veřejná.

Pro spuštění procesu vytvářejícího výstupní data je vnitřně daným prvkem volána metoda, která se ve starších verzích jmenovala *Execute*, ve verzi 6 se nazývá *RequestData*. Tato metoda je chráněná, takže ji nelze volat manuálně jako metodu *Update*, ale je možné ji při dědění přepsat, čímž lze definovat, jakým způsobem budou výstupní data vytvořena.

Posílání požadavků a spouštění procesů může být popsáno na příkladu. Uvažujme následující C++ kód.

```
1.      vtkCubeSource* cube = vtkCubeSource::New();
2.      cube->SetXLength(1);
3.      cube->SetYLength(1);
4.      cube->SetZLength(1);
5.      vtkTransform* scalingTransform = vtkTransform::New();
6.      scalingTransform->Scale(1, 2, 1);
7.      vtkTransformFilter* transFilter =
8               vtkTransformFilter::New();
9.      ransFilter->SetInputConnection(cube->GetOutputPort());
10.     transFilter->SetTransform(scalingTransform);
11.     vtkXMLPolyDataWriter* writer =
12              vtkXMLPolyDataWriter::New();
13.     writer->SetInputConnection(transFilter->GetOutputPort());
```

```
14.     writer->SetFileName("output1.xml");
15.     writer->Write();
16.     writer->SetFileName("output2.xml");
17.     writer->Write();
```

Zde *cube* je datový zdroj pro vytvoření krychle, jeho výstup je napojen na vstup filtru, který provede transformaci, a jeho výstup je napojen na vstup koncového procesu, který zapíše výsledná data do xml souboru. Transformace, kterou filtr provádí, je nastavena na škálování. Metoda *New* slouží k vytváření instancí a o něco podrobněji bude popsána v sekci 3.2. Při prvním zavolání metody *Write* (viz řádek 15) nad koncovým procesem je vyslán požadavek na vstupní data transformačnímu filtru. Ten pošle požadavek na vstupní data datovému zdroji *cube*. Zdroj *cube* vytvoří data a pošle je na výstup, kde si je vyzvedne transformační filtr. Ten provede škálování a pošle výsledná data dále na svůj výstup, kde si je vyzvedne koncový objekt a provede zápis do souboru *output1.xml*. Při druhém volání metody *Write* (viz řádek 17) se požadavek opět dostane až k datovému zdroji *cube*, jelikož se ale od posledního požadavku na jeho stavu nic nezměnilo, pošle na výstup data, která má již vytvořená. Data si vyzvedne transformační filtr, na jehož stavu se také nic nezměnilo, zjistí, že data na vstupu jsou stejná jako při předchozím požadavku, a pošle také na výstup již vytvořená data. Výsledná data si vyzvedne koncový proces a provede jejich zápis do souboru *output2.xml*.

Uvažme nyní následující změnu v posledních několika řádcích kódu.

```
...
14.     writer->SetFileName("output1.xml");
15.     writer->Write();
16.     vtkTransform* rotatingTransform = vtkTransform::New();
17.     rotatingTransform->RotateX(20);
18.     transFilter->SetTransform(rotatingTransform);
19.     writer->SetFileName("output2.xml");
20.     writer->Write();
```

Při prvním volání metody *Write* (viz řádek 15) proběhne vše stejně jako v předchozím příkladu. Při druhém volání (viz řádek 20) ovšem nastane změna. Zdrojový objekt *cube* je stále ve stejném stavu, takže pouze poskytne transformačnímu filtru data vytvořená při prvním požadavku. Transformační filtr má sice nyní na vstupu stejná data jako při prvním požadavku, ale jeho stav se změnil. Při prvním požadavku prováděl transformaci škálování, teď má ale provádět rotaci. Filtr tedy spustí znovu proces pro vytvoření výstupních dat, která si pak koncový proces vyzvedne a zapíše do souboru.

3.1.2 Příklady

Princip funkčnosti vizualizační roury je nejlépe ukázat na konkrétních příkladech. Zde jsou uvedeny dva jednoduché příklady obsahující klíčovou část zdrojového kódu v jazyce C++, schéma roury a grafický výstup.

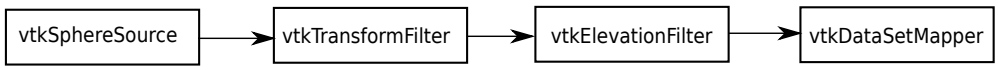
Příklad 1

V prvním příkladu je použita jednoduchá přímočará roura bez jakéhokoliv větvení. Zdrojový kód je následující.

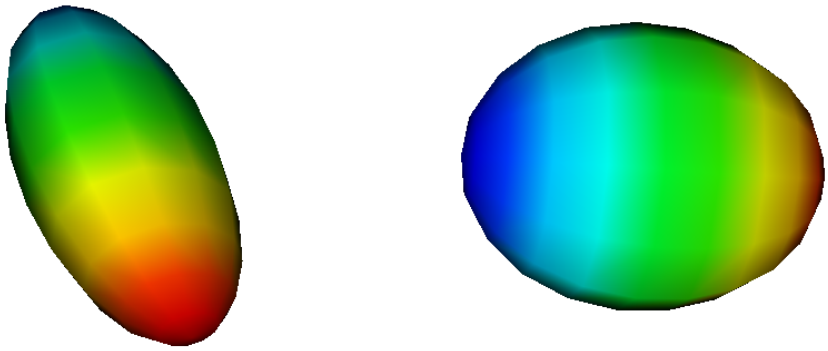
```
vtkSphereSource* sphere = vtkSphereSource::New();
sphere->SetPhiResolution(12);
sphere->SetThetaResolution(12);

vtkTransform* trans = vtkTransform::New();
trans->Scale(1, 1.5, 2);

vtkTransformFilter* transFilter = vtkTransformFilter::New();
transFilter->SetInputConnection(sphere->GetOutputPort());
transFilter->SetTransform(trans);

vtkElevationFilter* elevationfilter = vtkElevationFilter::New();
elevationfilter->SetInputConnection(transFilter->GetOutputPort());
elevationfilter->SetLowPoint(0, 0, -1);
elevationfilter->SetHighPoint(0, 0, 1);

vtkDataSetMapper* mapper = vtkDataSetMapper::New();
mapper->SetInputConnection(elevationfilter->GetOutputPort());

vtkActor* actor = vtkActor::New();
actor->SetMapper(mapper);
```

Jako datový zdroj zde slouží objekt *sphere*, který generuje nepříliš přesnou aproximaci koule. Výstup tohoto objektu je napojen na vstup transformačního filtru, který provede škálování v poměru 1:1 na ose x, 1:1,5 na ose y a 1:2 na ose z. Výstup transformačního filtru je dále napojen na vstup elevačního filtru, který přidělí každému bodu skalární hodnotu určenou vzdáleností od roviny, která je definována bodem (0, 0, -1) a normálovým vektorem $[0,0,1]^T - [0,0,-1]^T = [0,0,2]^T$. Výstup elevačního filtru je dále napojen na vstup *mapperu*, což je objekt, který obarví jednotlivé body na základě skalární hodnoty, která jim byla přidělena, a předá datovou reprezentaci výsledného objektu grafické knihovně. Instanci již vykreslitelného grafického

objektu reprezentuje *actor*, který je vytvořen z příslušného *mapperu*. Rouře v tomto příkladu odpovídá schéma na obrázku 3.2. Grafický výstup je vidět na obrázku 3.3. Výstup je ukázán ze dvou různých pohledů.



Obrázek 3.2: Schéma roury prvního příkladu



Obrázek 3.3: Grafický výstup prvního příkladu

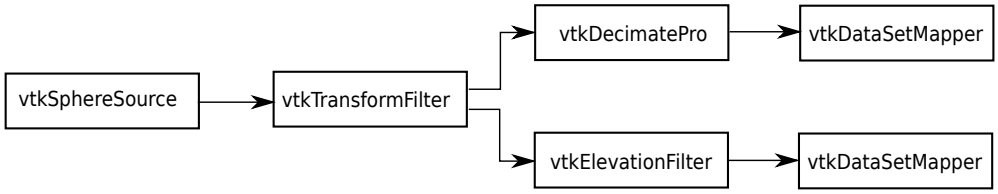
Příklad 2

Druhý příklad navazuje na předchozí a přidává jeden grafický objekt, který ale pochází ze stejného datového zdroje, pouze neprochází elevačním filtrem, nýbrž decimačním. Zdrojový kód je následující.

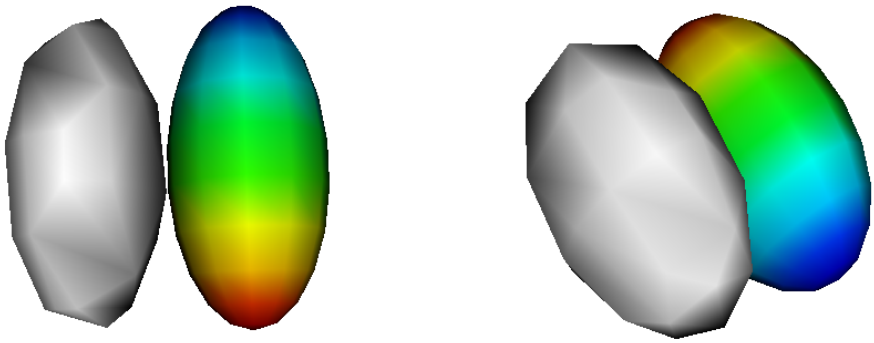
```
vtkSphereSource* sphere = vtkSphereSource::New();
sphere->SetPhiResolution(12);
sphere->SetThetaResolution(12);

vtkTransform* trans = vtkTransform::New();
trans->Scale(1, 1.5, 2);

vtkTransformFilter* transFilter = vtkTransformFilter::New();
transFilter->SetInputConnection(sphere->GetOutputPort());
transFilter->SetTransform(trans);
```

```
vtkDecimatePro* decimator = vtkDecimatePro::New();
decimator->SetInputConnection(transFilter->GetOutputPort());
decimator->SetTargetReduction(0.7);

vtkElevationFilter* elevationfilter = vtkElevationFilter::New();
elevationfilter->SetInputConnection(transFilter->GetOutputPort());
elevationfilter->SetLowPoint(0, 0, -1);
elevationfilter->SetHighPoint(0, 0, 1);

vtkDataSetMapper* mapper1 = vtkDataSetMapper::New();
mapper1->SetInputConnection(elevationfilter->GetOutputPort());
vtkDataSetMapper* mapper2 = vtkDataSetMapper::New();
mapper2->SetInputConnection(decimator->GetOutputPort());

vtkActor* actor1 = vtkActor::New();
actor1->SetMapper(mapper1);
actor1->SetPosition(-0.5, 0, 0);
vtkActor* actor2 = vtkActor::New();
actor2->SetMapper(mapper2);
actor2->SetPosition(0.5, 0, 0);
```

Zde původní grafický objekt z prvního příkladu je napojen na *mapper1*, z něhož je následně vytvořen *actor1*. Druhý objekt, který prošel decimačním filtrem, je napojen na *mapper2*, z kterého je vytvořen *actor2*. Decimační filtr provádí redukci původní aproximace koule a snižuje počet jejích polygonů o přibližně 70 %. Nakonec je oběma objektům nastavena různá pozice, aby byly vykresleny vedle sebe. Schéma roury v tomto příkladu je ukázáno na obrázku 3.4. Grafický výstup, opět ze dvou různých pohledů, ukazuje obrázek 3.5. Na neobarveném objektu je velmi dobře vidět jeho deformovaný tvar, který byl způsoben decimací jeho polygonové sítě.



Obrázek 3.4: Schéma roury druhého příkladu



Obrázek 3.5: Grafický výstup druhého příkladu

Oba příklady byly částečně převzaty z knihy [27]. Mnoho dalších příkladů lze nalézt zde [1].

3.2 Počítání odkazů

Instance tříd knihovny *VTK* se vytvářejí pomocí statické metody *New* volané nad konkrétní třídou a každou instanci je potřeba odstranit z paměti voláním metody *Delete*. Tento princip velmi nápadně napodobuje chování operátorů *new* a *delete* jazyka C++, ovšem poskytuje navíc něco, co tyto operátory poskytnout nemohou, a tím je počítání odkazů (*reference counting*). Objekt ve *VTK* může typicky být používán jiným objektem. Po vytvoření objektu metodou *New* má daný objekt vždy nastaven počet odkazů na hodnotu 1. Pokud tento objekt začne být používán jiným objektem, počet odkazů se o 1 zvýší. Při zavolání metody *Delete* nad daným objektem se jeho počet odkazů o 1 sníží. Pokud je metoda *Delete* zavolána nad objektem, jehož počet odkazů je 1, pak teprve je objekt skutečně smazán. Uvažujme následující kód.

```
vtkPoints* points = vtkPoints::New();
vtkPolyData* data1 = vtkPolyData::New();
data1->SetPoints(points);
vtkPolyData* data2 = vtkPolyData::New();
data2->SetPoints(points);
```

Zde objekt *points* je používán objekty *data1* a *data2*, jeho počet odkazů je tedy 3. Nyní přidejme následující tři řádky.

```
data1->Delete();
points->Delete();
data2->Delete();
```

V prvním řádku je smazán objekt *data1*, což způsobí snížení počtu odkazů na objekt *points* z hodnoty 3 na hodnotu 2. V druhém řádku je pouze snížen počet odkazů na objekt *points* z hodnoty 2 na hodnotu 1. V posledním řádku je smazán objekt *data2*, a jelikož počet odkazů na objekt *points* je 1, dojde zároveň i k smazání objektu *points*.

Tento mechanismus má jednu velkou přednost. Pokud máme vytvořen objekt a chceme ho zlikvidovat, nemusíme se starat, zda tento objekt není používán jinde. Prostě zavoláme metodu *Delete*, a pokud daný objekt již nikde jinde používán není, dojde k jeho smazání. Pokud objekt stále někde využíván je, dojde pouze ke snížení počtu odkazů a objekt bude smazán až tehdy, kdy skutečně již nikde využit nebude.

3.2.1 Chytré ukazatele

Počítání odkazů popsané výše nemá žádný vliv na následující situaci.

```
vtkPoints* points1 = vtkPoints::New();
vtkPoints* points2 = points1;
```

Objekt *points1* je vytvořen metodou *New* a objekt *points2* je vlastně jen přímým odkazem na objekt *points1*. Tento způsob přiřazení ovšem počítadlo odkazů nezvyšuje, pokud teď tedy zavoláme metodu *Delete* například nad objektem *points1*, rovnou tím tento objekt smažeme. Jelikož objekt *points2* je přímým odkazem na *points1*, tedy jedná se o ten samý objekt, bude po tomto volání smazán i on.

Tento neduh lze vyřešit takzvanými *smart pointery*, česky chytrými ukazateli. Následující kód demonstruje jejich použití.

```
vtkSmartPointer<vtkPoints> points1 =
        vtkSmartPointer<vtkPoints>::New();
vtkSmartPointer<vtkPoints> points2 = points1;
```

V tomto případě nejen, že počet odkazů se při přiřazení zvýší, a bude mít tedy hodnotu 2 pro objekt *points1* i objekt *points2*, ale navíc po skončení platnosti ukazatelů budou oba objekty automaticky zlikvidovány bez nut-

nosti volat metodu *Delete*. Programátor se tedy o likvidaci objektů prakticky vůbec nemusí starat. Více informací o chytrých ukazatelích lze nalézt zde [2].

4 Analýza problému

Způsob spouštění procesů ve vizualizační rouře knihovny *VTK*, popsaný v předchozí kapitole, vypadá na první pohled docela přívětivě. Uživatel se stará pouze o generování požadavků a při zpracovávání požadavků prvky roury dojde ke spuštění procesu pouze tehdy, když se stav daného prvku od posledního požadavku změnil. Tím je zajištěno, že se proces nespouští zbytečně vícekrát po sobě nad stejnými daty s produkcí stejného výstupu.

Způsob, jakým je dosaženo výše popsaného chování, je velmi jednoduchý, zároveň z něj ale plyne celý problém. Každý prvek obsahuje atribut udávající čas poslední změny jeho stavu, označme si ho *SMT* (*state modification time*), a čas posledního spuštění jeho procesu generujícího výstupní data, označme ho jako *ET* (*execution time*). [1] Atribut *SMT* je na začátku nastaven na nulu a v případě, že nastane změna dat na vstupu daného prvku nebo je změněna jeho konfigurace (například změna transformace u transformačního filtru nebo změna cílové hodnoty redukce u decimačního filtru), dojde k nastavení tohoto atributu na hodnotu aktuálního simulačního času. To samé platí pro atribut *ET* s tím rozdílem, že nastavení tohoto atributu na aktuální hodnotu simulačního času není vyvoláno změnou stavu daného prvku, ale spuštěním jeho procesu generování výstupních dat. V případě příchozího požadavku na výstupní data je proces jejich vygenerování spuštěn pouze tehdy, platí-li pro daný prvek roury, že $SMT > ET$, tedy pokud po posledním spuštění procesu nastala libovolná změna stavu. Uvažujme nyní následující kód.

```
1.      vtkSphereSource* sphere = vtkSphereSource::New();
2.      sphere->SetThetaResolution(100);
3.      sphere->SetPhiResolution(100);
4.      sphere->Update();
5.      sphere->SetPhiResolution(100);
6.      sphere->Update();
7.      sphere->SetPhiResolution(50);
8.      sphere->SetPhiResolution(100);
9.      sphere->Update();
```

Na řádcích 2 a 3 je nastaveno rozlišení polygonové sítě koule a atribut *SMT* je nastaven na aktuální simulační čas. Metoda *Update* (viz 3.1.1) volaná na řádku 4 způsobí prvotní vytvoření výstupu objektu *sphere* a nastavení atri-

[1] Důvodem zavedení alternativního označení těchto atributů místo použití jejich skutečných názvů je větší obecnost a možná nekonzistence skrze různé verze *VTK*.

butu *ET* na aktuální simulační čas. Je zřejmé, že v tuto chvíli je $SMT < ET$, protože metoda *Update* byla volána později než oba *settery*. Na řádku 5 je opět zavolána metoda pro nastavení φ-rozlišení, ovšem na stejnou hodnotu, která je aktuálně nastavena, tedy hodnota atributu *SMT* se nezmění. Jelikož je stále $SMT < ET$, při volání metody *Update* na řádku 6 nedojde ke spuštění procesu a výstup zůstane stejný. Zde je zatím vše v pořádku. Stav objektu *sphere* se od předchozího spuštění procesu nezměnil, takže proces nemusel být znovu spuštěn a také nebyl. Na řádku 7 je opět zavolána metoda pro nastavení φ-rozlišení a tentokrát již nová hodnota není stejná jako původní a atribut *SMT* je nastaven na aktuální simulační čas, tedy $SMT > ET$. Na řádku 8 je φ-rozlišení nastaveno zpět na hodnotu 100 a zřejmě stále zůstává $SMT > ET$. Když je tedy na řádku 9 zavolána metoda *Update*, proces se spustí i přesto, že od jeho posledního spuštění se stav objektu *sphere* vlastně vůbec nezměnil. Problém zde je zřejmý. Došlo ke zbytečnému znovuspuštění procesu, který vůbec spuštěn být nemusel, protože jeho výstup bude úplně stejný jako výstup při předchozím spuštění.

Dalším problémem může být například následující případ.

```
vtkSphereSource* sphere = vtkSphereSource::New();
sphere->SetThetaResolution(100);
sphere->SetPhiResolution(100);
sphere->Update();
sphere->SetPhiResolution(50);
sphere->Update();
sphere->SetPhiResolution(100);
sphere->Update();
sphere->SetPhiResolution(50);
sphere->Update();
sphere->SetPhiResolution(100);
sphere->Update();
sphere->SetPhiResolution(50);
sphere->Update();
...
```

Zde se neustále opakují dva různé stavy objektu *sphere* a po každé změně stavu je zavolána metoda *Update*. Na základě poznatků z předchozího odstavce je zřejmé, že proces pro generování výstupních dat se spustí při každém volání metody *Update*. Přitom by stačilo pouze někam uložit výstup pro oba stavy a při volání metody *Update* pak pouze zjistit, ve kterém z těchto stavů se objekt aktuálně nachází, a podle toho vybrat jeden z těchto dvou výstupů. Tím by bylo zajištěno, že proces pro vygenerování výstupu se spustí dohromady pouze dvakrát. Takto lze ovšem říci, že při n volání metody

Update se proces provede celkem n-krát, z toho $(n-2)$-krát zbytečně.

Situace popsaná v předchozím odstavci je bohužel velmi jednoduchá a příliš konkrétní. Obecně může počet různých stavů daného prvku roury být libovolně vysoký. Některé stavy mohou nastávat velmi často, některé méně často a jiné pouze jednou. Uvedené příklady jsou navíc uměle vykonstruované pro účely názorné demonstrace problému. V praxi však bude situace většinou složitější, protože prvky roury mají nastavitelných parametrů několik, stejně tak mohou mít více vstupů (viz 3.1).

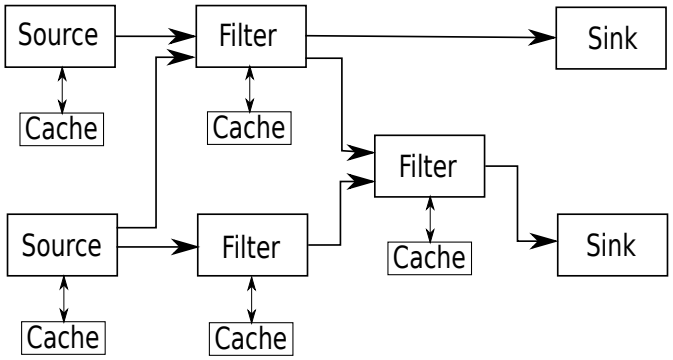
Za zmínku stojí také fakt, že knihovna *VTK* byla ve svých raných verzích špatně zdokumentována a princip vizualizační roury byl často programátory špatně pochopen. Ti pak například explicitně používali metodu *Modified*, která slouží k nastavení atributu *SMT* na aktuální simulační čas a je jinak volána interně v případě změny stavu nějakého prvku roury. Takto, v podstatě chybně, napsaný kód lze objevit i přímo v *MAF2*. Následující kód je výňatkem ze souboru *mafAxes.cpp*.

```
void mafAxes::SetPose( vtkMatrix4x4 *abs_pose_matrix )
{
        if(!m_Vme) return;
        assert(m_Coord);
        vtkMAFLocalAxisCoordinate *coord =
                (vtkMAFLocalAxisCoordinate*) m_Coord;
        if( abs_pose_matrix )
                coord->SetMatrix(abs_pose_matrix);
        else
                coord->SetMatrix(m_Vme->GetAbsMatrixPipe()->
                        GetMatrix().GetVTKMatrix());
        coord->Modified();
}
```

Metodu *Modified* nad objektem *coord* zde automaticky volá metoda *SetMatrix*, která se o jejím zavolání rozhodne podle toho, zda nově nastavovaná hodnota je rozdílná od té původní. Tedy ne vždy, kdy je volána metoda *SetMatrix*, bude automaticky volána i metoda *Modified*, což by mohlo ušetřit zbytečné spouštění procesu pro generování výstupních dat prvku *coord*. Protože je ovšem metoda *Modified* na konci metody *SetPose* zavolána explicitně, k žádnému ušetření nedojde.

Vzhledem k tomu, že proces generování výstupních dat je někdy časově poměrně náročný, může být jeho zbytečné opakování velmi nežádoucí. Bylo by tedy velmi vhodné nějakým způsobem umožnit uložení výstupů prvků

vizualizační roury nejen pro jejich aktuální stav (jak to *VTK* dělá), ale pro všechny stavy, ve kterých se daný prvek až doposud nacházel. Proces generování výstupu by pak byl spuštěn pouze tehdy, kdy pro daný stav není výstup uložen. K dosažení právě takového chování by měla posloužit *cachovací* knihovna, jejíž návrh a implementace jsou cílem této práce. Vizualizační roura by pak měla vypadat podobně jako na obrázku 4.1.



Obrázek 4.1: Ukázka jednoduché roury používající *cache*

Programátor sestavující vizualizační rouru by měl mít možnost k libovolnému prvku připojit *cache* objekt, který by se staral o ukládání a poskytování výstupních dat pro jednotlivé stavy. Připojení *cache* objektu je možné provést v metodě *Execute*, resp. *RequestData* (viz 3.1.1), kterou programátor daného prvku roury přepíše a přizpůsobí.

4.1 Obecnost a transparentnost

MAF2 framework je sice postaven na knihovně *VTK*, ovšem mnoho jeho logiky se nachází mimo vizualizační *VTK* rouru. Z tohoto důvodu je nutné, aby *cachovací* knihovna dokázala pracovat nejen s objekty tříd *VTK*, ale s prakticky libovolnými daty, z čehož plyne požadavek na obecnost knihovny.

Dále se předpokládá, že knihovna bude *cachování* provádět zcela automaticky s nutností jen minimální interakce s programátorem. Tím se myslí, že *cache* objekt by měl sám zjistit, zda se v něm daná data již nacházejí, pokud tomu tak není, měl by se postarat o jejich vytvoření a uložení a také by

měl zařídit poskytnutí daných dat na svůj výstup. Z tohoto plyne požadavek na transparentnost.

Je celkem zřejmé, že oba požadavky jsou ve vzájemné kontradikci. Čím méně toho knihovna ví o vstupních a výstupních datech, tím více informací jí musí být poskytnuto programátorem. Konkrétně jde například o informace o tom, jakým způsobem zkopírovat data do *cache*, jak data z *cache* odstranit nebo jak spolu data porovnávat. Vzhledem k tomu, že zde data mohou mít naprosto libovolnou, strukturu a knihovna o nich tedy neví vůbec nic, bude nutné přinejmenším tyto a jiné podobné operace definovat programátorem. Z toho vyplývá, že úplné transparentnosti není možné dosáhnout. Výsledná knihovna by se jí ale měla co nejvíce přiblížit.

4.2 Politika cachování

Operační paměť počítače není nekonečná a data, pro jejichž uchovávání by *cache* knihovna měla být využívána, mohou být nemalého objemu. Z tohoto důvodu bude potřeba navrhnout efektivní *cachovací* politiku, která se bude starat o systematické odstraňování uložených dat z paměti v případě, že bude zaplněna. Politika může být taktéž použita pro zabránění nahromadění příliš vysokého počtu datových položek v *cache*, protože vysoký počet položek, nezávisle na jejich velikosti, může *cache* zpomalovat.

Politika by měla zohlednit několik informací. Nejdůležitějšími údaji pro politiku budou pravděpodobně četnost a nedávnost použití daných dat. Na základě těchto údajů lze odhadnout, zda budou data v blízké době použita a jak často budou použita. Další užitečnou informací je objem dat, protože se například může vyplatit odstranit jeden větší datový objekt místo několika menších. Dále pak může být užitečná naměřená doba procesu generování dat, protože je pravděpodobně výhodnější vyřadit z *cache* data, která se vytvářejí krátkou dobu, než vyřadit data, která se vytvářejí relativně dlouho. Zde je potřeba brát v úvahu, že tato doba nemusí být vždy změřena přesně, protože záleží na aktuálním zatížení procesoru. Navíc proces generování dat může zahrnovat vlastní *cachování* a pak například jeho první spuštění může trvat mnohem delší dobu než další spuštění a podobně. Měřit lze i například dobu vyhledání dat v *cache*, dobu jejich přesunutí na výstup nebo dobu jejich zkopírování do *cache* po dokončení procesu jejich vytvoření. Všechny tyto informace mohou být užitečné, obecně je ovšem potřeba se všemi naměřenými

časy zacházet opatrně, protože chyba měření může být prakticky libovolně velká. Při návrhu politiky se lze inspirovat některými z politik uvedených v sekci 2.2.

4.2.1 Politiky založené na statistice

Ze statistických politik lze zvážit *LRU*, či *LFU*. Každá z těchto dvou politik používá jednu ze dvou klíčových informací.

LRU vyřazuje data, která byla naposledy použita před nejdelší dobou (viz 2.2.2). Ovšem to, že data nebyla nějakou dobu použita, ještě neznamená, že nebudou opět použita v budoucnu, třeba i několikrát (stačí, aby byl jeden datový objekt používán opakovaně, pouze s dlouhými časovými rozestupy). Stejně tak jako to, že data byla nedávno použita, neznamená, že budou brzy použita znovu.

LFU vyřazuje data, která byla použita nejméněkrát (viz 2.2.2). Zde je problém, že data, která byla použita mnohokrát někdy dávno, zůstanou v *cache* i přes to, že aktuálně již nejsou používána vůbec nebo téměř vůbec.

4.2.2 Hybridní politiky

Zaveďme nyní termín „priorita“, jakožto hodnotu, na jejímž základě se politika rozhoduje, zda vyřadit, či nevyřadit konkrétní datovou položku. Tedy v případě potřeby uvolnění místa v *cache* bude vyřazena položka, jejíž priorita je menší než priorita všech ostatních položek v *cache*. U politiky *LFU* je priorita určena jen jako počet přístupů k dané položce. Pro politiku *LRU* pak platí, že nejnižší prioritu má položka, která byla naposledy použita před nejdelší dobou, jako prioritu lze tedy použít časovou značku posledního přístupu k dané položce.

Výhoda politik z předchozí sekce je ta, že datové položky lze ukládat do prioritní fronty, protože v případě přístupu do *cache* se změní priorita pouze jedné položky. Po vyřazení nebo po změně priority nějaké položky stačí pouze prioritní frontu opravit, což lze typicky provést s časovou složitostí $O(log(n))$. Není tedy nutné procházet všechny položky vždy, když chceme nějakou vyřadit, nebo dokonce vždy, když nastane přístup do *cache*.

Tato vlastnost může někdy být užitečná a některé hybridní politiky ji dále rozvíjejí. Například politika *2Q* je rozdělená na dvě části, kde jedna je spravovaná politikou *LRU* a druhá politikou *FIFO* (viz 2.2.3). Prioritní fronta tedy obsahuje pouze data z té části, která je spravovaná politikou *LRU*, a samozřejmě čím menší je prioritní fronta, tím rychlejší je její oprava. Část spravovaná politikou *FIFO* pak může být implementována jako spojový seznam, kde oprava má časovou složitost $O(1)$. Dalším příkladem je politika *MQ*, která je rozdělena hned do několika částí, jejichž množství může být prakticky libovolné (viz 2.2.3). V případě přístupu do *cache* je pak většinou potřeba projít pouze data v jedné z těchto částí.

Navrhovaná *cache* bude určena pro ukládání výsledků časově velmi náročných grafických výpočtů. Tedy předpokládá se, že počet položek uložených v *cache* prakticky nikdy nebude tak vysoký, aby iterace všemi těmito položkami a spočtení jejich priorit trvala tak dlouho, že by tato doba byla významná ve srovnání s časy jiných akcí, které bude *cache* provádět. Především se jedná o akci samotného vytváření dat, ale také o jejich vyhledávání v *cache* a jejich kopírování. Z tohoto důvodu bude vhodné se zaměřit především na co nejkvalitnější výpočet priority a nesnažit se například zbytečně navrhovat nějakou sofistikovanou datovou strukturu pro snížení režie politiky. Nejvhodnější tedy bude inspirovat se politikou, která bere v úvahu jak počet přístupů k datům, tak nedávnost jejich použití, a která tyto informace používá stejnou měrou nad všemi položkami, čímž zajistí objektivní výběr kandidáta pro vyřazení. Tyto vlastnosti, zdá se, splňují politiky *LRFU* a *LRD*.

Politika *LRFU* vyřazuje data s nejnižší hodnotou *CRF*, která je spočtena z doby, před kterou byla data naposledy použita, a z počtu přístupů k datům (viz 2.2.3). Tedy priorita datové položky je tím větší, čím je kratší doba, před kterou byla položka naposledy použita, a čím vícekrát byla v minulosti použita. Zde je problém v tom, že pokud datová položka byla někdy dávno použita mnohokrát během krátké doby a následně se frekvence jejího používání rapidně snížila, její priorita zůstane stále poměrně vysoká. Počet použití této položky je totiž stále vysoký a položka je stále využívána, i když už jen velmi řídce, takže nedávnost použití této položky nemusí nikdy klesnout tak, aby celková priorita byla dostatečně malá pro vyřazení. Výsledkem je, že i přes to, že jsou daná data využívána již jen velmi málo, nikdy nebudou z *cache* odstraněna, protože jejich priorita byla kdysi rapidně navýšena vysokým počtem přístupů během krátké doby.

Problém politiky *LRFU* řeší politika *LRD*. Ta sice zaznamenává pouze počet přístupů k datům, ale postupně tento počet pro všechny položky snižuje (viz 2.2.3). Tímto lze zařídit, že nejvyšší prioritu budou mít pouze data, která byla často využívána v blízké minulosti. Prioritu tedy nekazí dávné přístupy, které aktuálně již nejsou významné. *LRD* sice přímo nevyužívá nedávnost přístupu k datům, ovšem vzhledem k postupně klesající prioritě je zřejmé, že dlouhodobé nepoužití datové položky povede k jejímu vyřazení. Tyto poznatky naznačují, že právě politika *LRD* by mohla být vhodným základem pro návrh politiky *cachovací* knihovny.

4.3 Ukládání na disk

Některá data, která budou ukládána do *cache*, mohou být poměrně velká a paměť se může rychle zaplnit. Navíc *MAF2 framework* je 32-bitový a s jeho migrací na 64-bit se v blízké době nepočítá, takže i s dostatkem fyzické paměti není v tomto případě možné využít více než 4 GB. Rychle se zaplňující paměť je samozřejmě problém, protože to vede k neustálému odstraňování datových položek a nahrazování jinými, takže proces generování dat je prováděn velmi často. Z tohoto důvodu by bylo velmi vhodné po odstranění dat z paměti zvážit jejich zapsání na disk, odkud by bylo možné je v případě potřeby opět načíst. Data zapsaná na disku by samozřejmě byla součástí *cache* stejně jako data v paměti. V případě, že by byla tato data od *cache* vyžadována, měla by být na disku nalezena a poskytnuta na výstup, stejně jako data z paměti.

Zápis na disk se bohužel nemusí vždy vyplatit, protože ne vždy je pravda, že doba generování dat je delší než doba jejich vyhledání a přečtení z disku. Z tohoto důvodu bude nejspíše potřeba obohatit politiku *cachování* o rozhodnutí, která data na disk uložit a která nikoliv. Bude tedy vhodné provádět měření doby čtení datových položek z disku a na základě naměřeného času a jeho porovnání s časem generování dat odhadnout, zda se uložení dané položky na disk vyplatí.

Když už budeme data za běhu aplikace ukládat na disk, nic nebrání tomu, abychom data nechali na disku i po vypnutí aplikace a při opětovném zapnutí je pouze připojili k příslušnému *cache* objektu. Takto by bylo možné zařídit, že při jiném než prvním spuštění aplikace se v mnoha případech proces generování dat nespustí ani jednou, protože data již budou k dispozici

z předchozího běhu aplikace.

4.3.1 Redis

Za zmínku stojí systém jménem *Redis*. *Redis* je datový server, často používaný jako *cache*, který umožňuje ukládání jednoduše serializovatelných dat, jako jsou číselné hodnoty, textové řetězce, seznamy, množiny atd. Každá datová položka je uložena pod klíčem, s jehož pomocí je možné danou položku později přečíst. Server je ovládán příkazy, zde je jednoduchý příklad.

```
> SET text "Hello"
> GET text
```

První příkaz uloží textový řetězec *Hello* pod klíčem *text*. Druhý příkaz je vlastně jednoduchý dotaz, který žádá hodnotu, která je uložena pod klíčem *text*. Odpovědí na tento příkaz je řetězec *Hello*.

Redis má všechna data uložená v paměti a pro vyřazování používá politiku *LRU*. Umožňuje ovšem také zápis dat na disk z důvodu bezpečnosti, jakožto prevenci před ztrátou dat. *Redis* poskytuje dva různé přístupy k zapisování dat na disk, přičemž oba tyto přístupy by případně mohly být inspirací pro navrhovanou *cachovací* knihovnu. Prvním přístupem je takzvaný *snapshotting*, kde se jednou za čas (např. každou minutu) zapíší na disk do souboru všechna data, která jsou aktuálně v paměti uložena. V případě pádu serveru je pak možné z tohoto souboru obnovit stav paměti alespoň na stav, který byl zachycen v posledním *snapshotu*. Druhým přístupem je *append only file*, kde jsou do jednoho souboru zapisovány všechny změny, které jsou nad daty provedeny. Zápis se provádí typicky každou sekundu nebo po každém spuštěném příkazu. Z tohoto souboru je pak možné po pádu obnovit stav paměti na stav, ve kterém paměť byla těsně před pádem.

Více informací o sytému *Redis* lze nalézt zde [7].

4.4 Programovací jazyk

Výběr programovacího jazyka je nedílnou součástí vývoje každého programového vybavení. Jelikož *MAF2 framework* je napsaný v jazyce C++ a jedním z hlavních požadavků je kompatibilita s knihovnou *VTK*, která je taktéž

napsána v C++, je jedině logické tento jazyk použít i pro vývoj *cachovací* knihovny. Bylo by sice možné použít jiný jazyk a výsledek připojit jako dynamicky či staticky linkovanou knihovnu, ovšem tento postup je poměrně nepraktický a není k tomu žádný dobrý důvod. C++ navíc používá manuální správu paměti, což poskytuje dobrou kontrolu nad daty.

5 Návrh knihovny

Ačkoliv hlavním důvodem návrhu a implementace *cache* systému jsou problémy vizualizační roury knihovny *VTK* uvedené na začátku kapitoly 4, musí systém umět pracovat i s jinými datovými typy, než jsou třídy knihovny *VTK*, a musí být použitelný i mimo vizualizační rouru *VTK* (viz 4.1). Z tohoto důvodu nepřipadá v úvahu zabudování *cache* přímo do knihovny *VTK*, ačkoliv by to pravděpodobně bylo jednodušší, ale bude potřeba navrhnout a implementovat *cache* systém jako samostatně stojící knihovnu, která bude ve výsledku použita tam, kde to bude potřeba.

Návrh *cachovací* knihovny sestává z několika částí. V první řadě je potřeba navrhnout rozhraní knihovny, protože to určuje, jakým způsobem se bude knihovna používat. Dále je nutné navrhnout datovou strukturu, kterou bude knihovna používat pro uložení *nacachovaných* datových položek. Následuje návrh *cachovací* politiky, která se bude starat o vyřazování dat z paměti v případě jejího zaplnění. Nakonec bude potřeba navrhnout systém, který umožní zápis a čtení *nacachovaných* dat z pevného disku.

5.1 Rozhraní

Ačkoliv se tak nemusí zdát, návrh rozhraní je dost možná nejdůležitější částí celé práce. Rozhraní je jediná věc, pomocí které lze ovlivnit obecnost a transparentnost knihovny. Špatně navržené rozhraní může způsobit velké problémy při budoucích pokusech o rozšíření. Naopak, pokud rozhraní bude navrženo dobře, bude možné kteroukoliv část implementace kdykoliv změnit bez nutnosti zásahu do kódu, ve kterém bude knihovna používána. Z těchto důvodů lze usoudit, že návrh rozhraní nebude zcela triviální a bude třeba mu věnovat dostatek pozornosti.

Shrňme si nejdříve to, co je od knihovny vlastně požadováno. Knihovna by měla umožnit vytvoření libovolného počtu *cache* objektů. Každý *cache* objekt by měl být schopný na základě konkrétního vstupu poskytnout korektní výstup. Tuto činnost by měl každý objekt provádět zcela automaticky tak, že v případě existence příslušných dat v *cache* tato data poskytne na výstupu, v opačném případě se postará o jejich vytvoření příslušným al-

goritmem, uloží je do *cache* a následně poskytne na výstupu. Algoritmus pro vytváření výstupních dat dodá programátor, který vytváří daný *cache* objekt, a to v podobě ukazatele na funkci (resp. metodu).

Vzhledem k tomu, že výstup *cache* objektu musí pro daný vstup být stejný jako výstup funkce, která provádí algoritmus generování výstupu, je možné *cache* objekt navrhnout tak, že v podstatě simuluje chování dané funkce. *Cache* objekt bude obsahovat metodu například s názvem *call* a v konstruktoru bude přebírat ukazatel na funkci, která má být použita pro generování výstupních dat. V případě, kdy programátor bude chtít získat data pro konkrétní vstup, zavolá nad daným objektem metodu *call*, která se z jeho pohledu zachová stejně, jako kdyby volal samotnou původní funkci. Rozdíl bude v tom, že samotná funkce bude vnitřně zavolána pouze tehdy, nebudou-li data pro daný vstup v *cache* k dispozici. Třídu, jejíž instancemi jsou jednotlivé *cache* objekty, nazvěme *CachedFunction*. Důvodem tohoto názvu je fakt, že jednotlivé objekty mají za úkol simulovat volání nějaké funkce a výstupy jednotlivých volání uchovávat v *cache*. Princip demonstruje následující pseudokód v jazyce C++.

```
int function(int input)
{
        return input * 10;
}

...

CachedFunction cachedFunction = CachedFunction(function);

...

int output1 = function(5);
int output2 = cachedFunction.call(5);
```

Na samém začátku je definována funkce mající jeden vstupní parametr typu *int* a jeden výstup typu *int* v podobě návratové hodnoty. Dále je vytvořen *cache* objekt *cachedFunction*, kterému je v konstruktoru předán ukazatel na tuto funkci. Následně je jednou funkce přímo zavolána a její výstup uložen do proměnné *output1* a jednou je její chování nasimulováno voláním metody *call* nad *cache* objektem a výsledek je uložen do proměnné *output2*. Proměnné *output1* a *output2* by měly obě mít hodnotu 50. Informace o tom, zda metoda *call* vnitřně zavolala funkci *function*, či pouze poskytla data, která byla v *cache* k dispozici z nějakého předchozího volání, je pro programátora v zásadě nepodstatná, protože on toto nemusí řešit. Metoda *call* to vyřeší za

něj a tak či onak poskytne data, o která programátor stojí.

5.1.1 Generický cache objekt

Je zřejmé, že funkce generující výstupní data nebude vždy přebírat parametr typu *int*, taktéž nebude vždy vracet hodnotu typu *int*, ale že datové typy mohou být obecně libovolné. Z tohoto důvodu je potřeba, aby třída *CachedFunction* byla generická, tedy aby dokázala pracovat s libovolnými datovými typy. Jazyk C++ k těmto účelům poskytuje takzvané šablony. Jedná se o část kódu obsahující proměnné datové typy či konstanty. Proměnné v tom smyslu, že při každém použití daného kódu mohou být různé, ale v rámci jednoho použití neměnné. Programátor při použití šablony třídy či šablony funkce (resp. metody) dodá dané datové typy a hodnoty konstant ve špičatých závorkách. Více informací o šablonách v C++ a o jejich použití lze nalézt například zde [28], případně v knize [24]. Upravený kód s použitím šablony třídy *CachedFunction* může být následující.

```
int function1(int input)
{
        return input * 10;
}

double function2(float input)
{
        return input / 0.5;
}

...

CachedFunction<int, int> cachedFunction1 =
        CachedFunction<int, int>(function1);
CachedFunction<double, float> cachedFunction2 =
        CachedFunction<double, float>(function2);

...

int output1 = function1(5);
int output2 = cachedFunction1.call(5);
double output3 = function2(5);
double output4 = cachedFunction2.call(5);
```

První parametr šablony určuje návratový typ, druhý parametr pak určuje typ parametru funkce. Hodnoty proměnných *output1* a *output2* by měly být

opět 50. Proměnné *output3* a *output4* by měly obě mít hodnotu 2,5.

Tímto je vyřešen problém obecných datových typů. Stále je zde ale problém, že parametr funkce nemusí být pouze jeden, ale může jich být obecně prakticky libovolný počet a všechny mohou být různých typů. Naštěstí C++ od verze 11 poskytuje tzv. variadické šablony, neboli šablony s proměnným počtem parametrů, které jsou pro tento problém jako dělané. Více informací o nich lze nalézt například zde [9] nebo zde [12]. Kód s použitím variadické šablony třídy *CachedFunction* může být následující.

```
int function1(int input1, int input2)
{
        ...
}

double function2(float input1, int input2,
      double input3, int input4)
{
        ...
}

...

CachedFunction<int, int, int> cachedFunction1 =
      CachedFunction<int, int, int>(function1);
CachedFunction<double, float, int, double, int> cachedFunction2 =
      CachedFunction<double, float, int, double, int>(function2);

...

int output1 = cachedFunction1.call(5, 10);
double output2 = cachedFunction2.call(1.5, -3, 7.45, 12);
```

Prvním parametrem šablony je stále návratový typ funkce. Dále následují typy parametrů v pořadí, v kterém jsou uvedeny u příslušné funkce.

5.1.2 Výstupní parametry

V jazycích C a C++ se poměrně běžně používají takzvané výstupní parametry. Jde o situaci, kdy funkce (resp. metoda) v parametru přebírá ukazatel na adresu, na které leží například nějaký objekt či pole. Tato funkce pak data na dané adrese nějakým způsobem změní a v podstatě v nich předává volající rutině svůj výstup. Výstupních parametrů může být samozřejmě libovolné

množství. Tento přístup je celkem hojně používán i knihovnou *VTK*. Bylo by tedy velmi vhodné, možná dokonce nezbytné, kdyby metoda *call* třídy *CachedFunction* byla schopná simulovat i toto chování.

Zřejmě bude potřeba rozšířit třídu *CachedFunction* o možnost nějaké konfigurace, pomocí které bude možné zvolit, jaké parametry jsou vstupní a jaké výstupní. Použití pak bude například následující.

```
int function(int inputParam, int* outputParam)
{
        *outputParam = inputParam + 5;
        return inputParam * 10;
}

...

CachedFunction<int, int, int*> cachedFunction =
        CachedFunction<int, int, int*>(function);
cachedFunction.setParamType(0, Input);
cachedFunction.setParamType(1, Output);

...

int output1A;
int output1B = function(5, &output1A);
int output2A;
int output2B = cachedFunction.call(5, &output2A);
```

Na začátku je vytvořena funkce s jedním vstupním a jedním výstupním parametrem. Dále je vytvořen *cache* objekt a jeho konfigurace je nastavena tak, že první parametr (parametr s indexem 0) je vstupní a druhý parametr (s indexem 1) je výstupní. Po zavolání funkce *function* a metody *call* by měly proměnné *output1A* a *output2A* mít hodnotu 10 a proměnné *output1B* a *output2B* hodnotu 50.

Tento způsob konfigurace trpí jedním drobným nedostatkem, kterým je fakt, že programátor může konfiguraci změnit kdykoliv za běhu aplikace, což by mohlo způsobit nekonzistenci dat uvnitř *cache*. Možná by tedy bylo vhodné změnu za běhu nějakým způsobem znemožnit. To lze zařídit celkem jednoduše, a to tak, že místo aby konfigurace byla nastavována přímo nad *cache* objektem, vytvoří se nejdříve konfigurační objekt, který bude následně *cache* objektu předán v konstruktoru. *Cache* objekt si vytvoří kopii konfiguračního objektu a tuto kopii bude používat interně, tedy programátor k ní po vytvoření *cache* objektu již nebude mít přístup. Jelikož jde pouze o kopii,

změna v původním konfiguračním objektu již na chování *cache* objektu po jeho vytvoření nebude mít žádný vliv. Vytvoření *cache* objektu by mohlo vypadat například takto.

```
...

CacheConfiguration conf;
conf.setParamType(0, Input);
conf.setParamType(1, Output);
CachedFunction<int, int, int*> cachedFunction =
        CachedFunction<int, int, int*>(function, conf);

...
```

5.1.3 Manipulace s daty

Cache objekt bude s daty, s nimiž bude pracovat, provádět mnoho operací, které nelze definovat obecně bez znalosti konkrétních datových typů. Jako příklad lze uvést situaci, kdy má být do *cache* zkopírována nová položka a jedná se o ukazatel na nějaký objekt. Má být zkopírován pouze ukazatel, či má být vytvořena kopie daného objektu? Má se jednat o hlubokou kopii, či o mělkou kopii? Jakým způsobem má být objekt překopírován? Toto je pouze jeden z několika případů, kdy operace, která má být s daty provedena, není jasně definována. Jako další příklady lze uvést kopírování dat z *cache* do výstupního parametru, získání dat z *cache* návratovou hodnotou nebo porovnání vstupního parametru s hodnotou uloženou v *cache* při hledání příslušného výstupu pro daný vstup. Zřejmě tedy bude nutné rozšířit konfigurační třídu a zařídit, aby programátor měl možnost tyto operace definovat. Vzhledem k tomu, že podobných operací může být poměrně velké množství (některé další se pravděpodobně odhalí až v pozdějších fázích návrhu či až při implementaci), by bylo vhodné, kdyby alespoň pro primitivní datové typy a pro často používané třídy knihovny *VTK* byly tyto operace předdefinovány a programátor je mohl pouze dosadit do konfigurace. Otázkou zůstává, jakým způsobem by tyto operace měly být konfiguračnímu objektu předány. Nabízejí se dvě možnosti.

Abstraktní třída

Prvním způsobem je vytvoření abstraktní šablonové třídy, která bude obsahovat hlavičky metod, které by měly provádět všechny potřebné operace.

Jedna funkce pro překopírování objektu do *cache*, další pro porovnání dvou objektů atd. Programátor by pak pro každý parametr a návratovou hodnotu vytvořil potomka této třídy s daným datovým typem a všechny metody by korektně definoval dle svého uvážení. Nevýhoda tohoto přístupu spočívá v tom, že pokud by programátor chtěl použít předdefinované operace dodané s knihovnou, musel by vždy použít celou třídu pro jeden parametr či návratovou hodnotu. Pokud by chtěl pouze například jednu z těchto operací provést jinak, než jak ji daná třída definuje, musel by vytvořit nového potomka a přepsat příslušnou metodu. Další nevýhodou je fakt, že se šablonovou třídou se zachází trochu jinak než s běžnou třídou a přesnou znalost šablon jazyka C++ lze považovat spíše za znalost pokročilou. Tedy programátor, který s jazykem C++ nemá příliš velké zkušenosti, by mohl mít s tímto principem problémy a mohl by snadno udělat chybu, pokud by definoval vlastního potomka abstraktní šablonové třídy.

Ukazatele na funkce

Druhý způsob je jednoduchý. Každá operace pro daný parametr či návratovou hodnotu bude programátorem dodána v podobě ukazatele na funkci, která příslušnou operaci provede dle jeho uvážení. Tento způsob sice tak trochu kazí objektový návrh celé knihovny, ovšem netrpí problémy prvního přístupu a z tohoto důvodu se zdá být vhodnější. Následující kus kódu ukazuje možný způsob definování operace porovnání vstupního parametru s hodnotou uloženou v *cache*.

```
class Object
{
private:
        int a;
        int b;

public:
        ...
        int getA() const { return a; }
        int getB() const { return b; }
};

...

bool equalFunction(const Object & o1, const Object & o2)
{
        return o1.getA() == o2.getA() && o1.getB() == o2.getB();
}
```

```
...

CacheConfiguration conf;
conf.setParamEqualFunction(2, equalFunction);
```

Zde je vytvořena třída *Object*, která obsahuje dva atributy *a* a *b*. Dále je vytvořena funkce, která přebírá odkaz na dva objekty typu *Object*, a v případě, že oba tyto objekty mají oba atributy shodné, vrátí hodnotu *true*. V opačném případě vrátí *false*. Konfiguračnímu objektu je pak předána informace, že *cache* objekt, kterému bude konfigurace předána, má pro porovnávání parametru na indexu 2 s uloženými hodnotami používat právě funkci *equalFunction*. [1]

5.1.4 Závislosti mezi parametry

Někdy se může stát, že mezi jednotlivými parametry funkce jsou určité závislosti. Příkladem může být situace, kdy prvním parametrem je ukazatel na nějaké pole a druhým parametrem je velikost tohoto pole. U takovýchto parametrů není možné provést například jejich porovnání s jinými hodnotami samostatně, protože při porovnávání prvního parametru neznáme velikost pole, a nevíme tedy, kolik prvků máme vlastně porovnávat. Stejný problém samozřejmě nastane i například při kopírování.

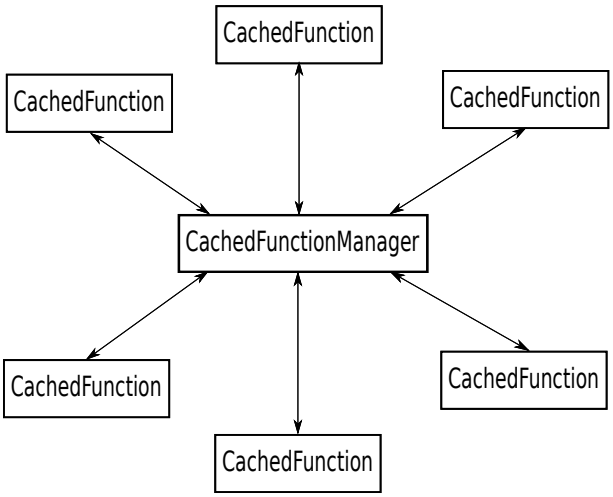
Tento problém lze vyřešit tak, že programátor při vytvoření *cache* objektu předá konstruktoru ukazatel na libovolnou adresu. Na této adrese může být například nějaká datová struktura, kterou programátor vytvořil. Daná adresa pak bude předávána do každé funkce, kterou je potřeba pro jednotlivé parametry v nějaké situaci zavolat. Tedy například funkce pro porovnání dvou objektů nebude v parametrech přebírat pouze dané objekty, ale navíc jeden parametr typu *void**, ve kterém bude vždy předána adresa, kterou programátor při vytvoření zvolil. Dále bude pro každou funkci jasně stanoveno

[1] Vhodné je definovat podobné operace jako metody přímo u příslušných tříd a ve funkcích předávaných konfiguračnímu objektu pak pouze tyto metody zavolat. Možné je také použít přetížené operátory = a == či kopírovací konstruktor. Použití přetížených operátorů nemusí být vždy přehledné a vhodné, ovšem používání kopírovacího konstruktoru autor práce důrazně doporučuje. V případě nepoužití kopírovacího konstruktoru a operátoru = autor doporučuje jejich explicitní zakázání (deklarací jako *private*). Kopírovací konstruktor i operátor = jsou totiž v C++ implicitně definovány tak, že provádějí mělkou kopii objektu, což může někdy způsobovat velmi těžko nalezitelné chyby.

pořadí jejího volání pro všechny parametry. Tedy například bude dáno, že funkce pro porovnání bude vždy volána pro jednotlivé parametry v pořadí, ve kterém jsou tyto parametry uvedeny ve funkci pro generování výstupních dat. Berme stále v úvahu příklad s ukazatelem na pole jakožto prvním parametrem a velikostí pole jakožto druhým parametrem. Programátor pak může napsat funkci pro porovnání prvního parametru tak, že ukazatele, které mají být porovnány, pouze uloží do datové struktury, která se nachází na předané adrese a vrátí vždy hodnotu *true*. Funkce pro porovnání druhého parametru si pak pouze tyto ukazatele z dané struktury vyzvedne, a jelikož již má všechny potřebné informace, provede skutečné porovnání daných polí s ohledem na jejich velikost a podle toho vrátí hodnotu *true*, nebo *false*.

5.1.5 Spolupracující cache objekty

Vzhledem k tomu, že fyzická paměť je pro všechny *cache* objekty sdílená, bylo by velmi vhodné, kdyby navržená politika *cachování*, která se bude starat o vyřazování dat z *cache*, nepracovala vždy pouze nad jedním *cache* objektem, ale se všemi objekty v aplikaci, nebo ještě lépe, nad konkrétní skupinou objektů, kterou si programátor sám určí. Z tohoto důvodu bude dobré nevytvářet jednotlivé *cache* objekty přímo, ale přes nějakou tovární třídu, kterou nazveme například *CachedFunctionManager*. Tato třída bude obsahovat šablonovou metodu s názvem například *createCachedFunction*, která vytvoří samostatný *cache* objekt a zaregistruje ho u příslušného manažera (viz obr. 5.1).



Obrázek 5.1: Schéma spolupracujících *cache* objektů pomocí manažera

Metoda *createCachedFunction* bude v parametrech přebírat všechny informace potřebné k vytvoření *cache* objektu. Vytvoření *cache* objektu by pak mohlo vypadat například takto.

```
CachedFunctionManager manager;
CachedFunction<int, double> cachedFunction1 =
        manager.createCachedFunction<int, double>(...);
CachedFunction<int, int, int> cachedFunction2 =
        manager.createCachedFunction<int, int, int>(...);
```

Zde je vytvořen jeden manažer, kterým jsou následně vytvořeny dva *cache* objekty. Třída *CachedFunctionManager* bude obsahovat informaci o tom, jaký je maximální možný součet velikostí všech dat v *cache*. Tato maximální hodnota bude samozřejmě nastavitelná programátorem. Pokud součet velikostí dat uložených ve všech *cache* objektech, které patří jednomu manažerovi, dosáhne dané maximální hodnoty a bude potřeba nějaká data odstranit, bude kandidát pro odstranění vybírán ne z jednoho konkrétního *cache* objektu, ale ze všech objektů daného manažera. Toto seskupování *cache* objektů může být použito i například pro hromadnou konfiguraci. Počet manažerů v aplikaci nebude nijak omezen.

Bylo by taktéž možné vytvářet hierarchické struktury, kde jeden manažer by nemusel spravovat pouze přímo jednotlivé *cache* objekty, ale další manažery. Tento princip by mohl ještě více rozšířit možnosti konfigurace.

5.1.6 Definovatelná politika cachování

V předchozí kapitole bylo řečeno, že součástí výsledné knihovny by měla být i politika starající se o odstraňování dat z *cache* v případě jejího zaplnění. Před návrhem samotné konkrétní politiky *cachování* (bude popsáno níže) bude vhodné navrhnout způsob, jakým bude daná politika do knihovny zakomponována. Bylo by totiž velmi vhodné, kdyby uživatel měl možnost politiku upravit dle svých představ či ji úplně vyměnit za jinou politiku, kterou si sám definuje. Toho lze docílit definováním abstraktní třídy s názvem například *CachePolicy* reprezentující konkrétní politiku *cachování*. Tato třída bude obsahovat metody provádějící všechny potřebné akce, které politice *cachování* přísluší. Jedna taková metoda bude volána při každém přístupu k libovolné položce v *cache*, další bude sloužit k výpočtu priority dané položky a bude volána pro všechny položky v případě, kdy bude potřeba nějakou položku z *cache* vyřadit. Metod může být více, další budou pravděpodobně přidány

při samotné implementaci dle potřeby. Každá z těchto metod bude v parametru přebírat identifikátor konkrétní položky a součástí třídy pak může být nějaká datová struktura, kde klíčem bude zmíněný identifikátor a hodnotou přidruženou k danému klíči bude cokoliv, co bude uživatel chtít pro jednotlivé položky zaznamenávat. Parametrů metod může také být více.

Součástí konfigurace třídy *CachedFunctionManager* (viz 5.1.5) bude instance třídy dědící od *CachePolicy*. Metody této instance pak budou použity pro rozhodnutí, kterou položku v případě potřeby z *cache* odstranit. Následující kód ukazuje možný způsob definice vlastní politiky, konkrétně se jedná o politiku *LFU*.

```
class MyCachePolicy : public CachePolicy
{
private:
        std::map<unsigned int, double> priorities;
protected:
        void cacheHitEvent(unsigned int id)
        {
                priorities[id]++;
        }

        double getPriority(unsigned int id)
        {
                return priorities[id];
        }
        ...
};
...
MyCachePolicy cachePolicy;
cachedFunctionManagerConfig.setCachePolicy(cachePolicy);
```

Metoda *cacheHitEvent* bude volána vždy, kdy nastane přístup ke konkrétní položce, přičemž identifikátor položky bude předán parametrem. Metoda *getPriority* bude použita k výpočtu priorit jednotlivých položek, přičemž identifikátor položky bude opět předán parametrem. Obě tyto metody jsou přepsanými metodami třídy *CachePolicy*. Objekt, který je zde pojmenován *cachedFunctionManagerConfig*, je pak instance nějaké konfigurační třídy.

Součástí knihovny bude třída s názvem například *DefaultCachePolicy* odděděná od *CachePolicy*, která bude definovat politiku navrženou v rámci této práce. Instance třídy *DefaultCachePolicy* pak bude dosazena do konfigurace automaticky v případě, kdy uživatel nedodá žádnou vlastní politiku.

Persistence dat

Představme si situaci, kdy uživatel chce ušetřit čas při kopírování dat z *cache* na výstup a rozhodne se předávat data na výstup pouze odkazem, tedy například jako ukazatel na daná data. Zde může nastat problém, pokud se politika *cachování* rozhodne odstranit z *cache* data, která si uživatel dříve vyžádal a stále je někde používá. V případě, kdy se jedná o data v podobě objektu libovolné třídy knihovny *VTK*, není potřeba tento problém nijak zvlášť řešit, protože *VTK* používá počítání odkazů (viz 3.2) a k tomu navíc poskytuje chytré ukazatele (viz 3.2.1). Nicméně podobně se chovající chytré ukazatele poskytuje ve svých standardních knihovnách i samotný jazyk C++, konkrétně jde o šablonovou třídu *shared_ptr* (viz například [8]). Tyto chytré ukazatele může uživatel použít místo klasických ukazatelů, a zabránit tak úplnému odstranění dat v případě, kdy se politika *cachování* rozhodne data odstranit z *cache*. Samozřejmě je nutné, aby uživatel patřičně přizpůsobil funkci (resp. metodu) pro vytváření výstupních dat.

5.2 Datová struktura

K uchování položek v *cache* bude potřeba použít nějakou datovou strukturu, která umožní co nejefektivnější nalezení výstupních dat příslušících konkrétním vstupním datům. Jako vhodnou strukturu by bylo možné zvolit *hash* tabulku, kde klíčem je množina vstupních parametrů a *hashovací* funkci dodá uživatel. Vzhledem k tomu, že porovnání dvou různých klíčů může být poměrně časově náročné, je velmi žádoucí, aby v tabulce nastávalo co nejméně kolizí. Kolize v *hash* tabulce může nastat nejen v případě, kdy jsou *hash* hodnoty dvou různých klíčů stejné, ale také v případě, kdy se shodují pouze zbytky po dělení *hash* hodnot dvou různých klíčů velikostí tabulky. Vhledem k tomu, že vstupní data budou většinou poměrně heterogenní, lze předpokládat, že uživatel bude téměř vždy schopen dodat takovou *hashovací* funkci, aby dva různé klíče neměly stejnou *hash* hodnotu téměř nikdy. Při nedostatečné velikosti tabulky ovšem mohou stále kolize nastávat v případě stejných zbytků po dělení. Tento problém lze vyřešit tak, že *hash* hodnota původního klíče bude sama sloužit jako klíč v tabulce, ke kterému bude přiřazen seznam kolizních hodnot. Vzhledem k tomu, že *hash* hodnota je celé číslo, pak nalezení správného seznamu v tabulce bude pouze záležitostí porovnávání celých čísel, což lze většinou oproti porovnávání původních klíčů považovat za časově zanedbatelné.

Celý proces nalezení správných výstupních dat pro daná vstupní data bude následující. Nejdříve se spočte *hash* hodnota vstupních dat. Tato hodnota bude použita jako klíč v tabulce a tím bude získán seznam kolizních vstupních dat, přičemž ke všem těmto datům budou samozřejmě přiřazena příslušná data výstupní. Vstupní data v seznamu budou postupně porovnávána se vstupními daty, pro která hledáme data výstupní, a to tak dlouho, dokud nenastane shoda, nebo dokud nedojdeme na konec seznamu. V případě, že ke shodě dojde, budou na výstupu poskytnuta příslušná výstupní data, v opačném případě bude patřičně indikováno, že k daným vstupním datům nejsou výstupní data k dispozici. Způsob implementace tabulky není příliš podstatný, lze použít například standardní šablonovou třídu *unordered_map*. Pro seznam kolizních hodnot může být použit například spojový seznam.

5.3 Politika cachování

V kapitole 4 (konkrétně 4.2) bylo uvedeno, že jako základ politiky *cachování* pro navrhovanou knihovnu bude použita politika *LRD* (viz 2.2.3). Bude tedy potřeba navrhnout, jakým způsobem bude u jednotlivých položek v *cache* postupně snižována hodnota vyjadřující počet přístupů. Dále bude vhodné brát v úvahu velikost položek, protože se jistě více vyplatí odstranit z *cache* položku, která zabírá velké množství místa, než položku, která ho zabírá malé množství. Také by bylo vhodné do politiky zahrnout dobu generování jednotlivých položek, protože položka, která se vytvářela dlouhou dobu, bude jistě mít pro uživatele větší cenu, než položka, která byla vytvořena téměř okamžitě. V případě, kdy budeme chtít do *cache* vložit položku, pro kterou již nezbývá dostatek místa, budou postupně vyřazovány položky s nejnižší prioritou do té doby, dokud nebude uvolněn požadovaný prostor pro novou položku. Priorita každé položky bude vypočtena podle rovnice 5.1, kde i je index položky, P_i je výsledná priorita, R_i je hodnota podle politiky *LRD*, tedy počet přístupů k položce, který je postupně snižován, f_S je klesající funkce určující přírůstek priority na základě velikosti položky, S_i je velikost položky, f_T je rostoucí funkce určující přírůstek priority na základě naměřené doby vytváření položky, T_i je naměřená doba vytváření položky v milisekundách a k_R, k_S, k_T jsou konstanty určené uživatelem, jejichž součet musí být 1.

$$P_i = k_R \cdot R_i + k_S \cdot f_S(S_i) + k_T \cdot f_T(T_i) \tag{5.1}$$

Konstanty k_R, k_S, k_T budou implicitně všechny nastaveny na hodnotu $\frac{1}{3}$. Zbývá určit způsob, jakým bude snižována hodnota R_i a stanovit funkce f_S a f_T.

5.3.1 Funkce f_S a f_T

U funkce $f_S(S)$ by jistě bylo vhodné, kdyby její argument S nebyl absolutní velikostí dané položky v *bytech*, ale spíše relativní hodnotou určující, jak velkou část celkové paměti položka zabírá. Budeme tedy velikost položek mapovat na interval $< 0; 1 >$ podle vztahu 5.2, kde S je výsledným argumentem funkce f_S, C je paměť, kterou má *cache* k dispozici a M je skutečná velikost položky v *bytech*.

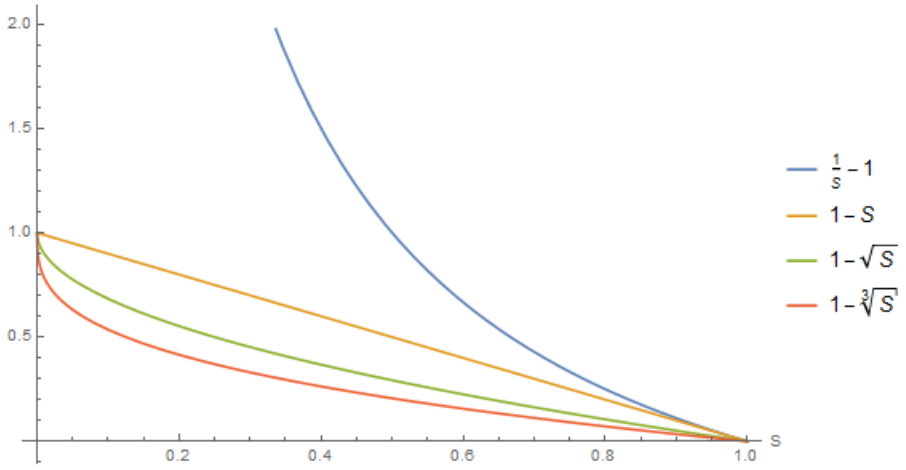
$$S = \frac{M}{C} \tag{5.2}$$

Je zřejmé, že hodnota S pro žádnou položku nemůže být větší než 1, protože na takovou položku nikdy nebude v *cache* dostatek místa, a tudíž nikdy nebude do *cache* uložena.

Pro samotnou funkci $f_S(S)$ bychom mohli použít například funkci $\frac{1}{S}$, tedy spíše $\frac{1}{S} - 1$, aby pro $S = 1$ byla hodnota funkce nulová. Tato funkce má ovšem jednu nepříjemnou vlastnost, a to tu, že pro velmi malé hodnoty S bude její funkční hodnota velmi vysoká. To může způsobovat nestabilitu ve výpočtech z důvodu nepřesné reprezentace desetinných čísel a také, a to především, pro malé položky by výsledná priorita závisela v podstatě jen na jejich velikosti a ostatní aspekty, které hrají roli při výpočtu priority, by byly téměř bezvýznamné. Vhodnější by mohla být funkce $1 - S$. Ta ale pro malé hodnoty S klesá velmi pomalu, což znamená, že například položka, která zabírá jednu tisícinu celé paměti by měla téměř stejnou prioritu jako položka, která zabírá jednu desetinu celé paměti. Toto lze napravit například tak, že místo samotné hodnoty S ve funkci použijeme $\sqrt{S}$, tedy dostaneme $1 - \sqrt{S}$. Není nutno se omezovat pouze na druhou odmocninu a vzhledem k tomu, že pro druhou odmocninu funkce stále klesá poměrně pomalu, vhodnější bude odmocnina třetí. Dostáváme tedy funkci ve tvaru 5.3.

$$f_S(S) = 1 - \sqrt[3]{S} \tag{5.3}$$

Srovnání všech zmíněných funkcí ilustruje graf na obrázku 5.2.



Obrázek 5.2: Srovnání různých funkcí f_S

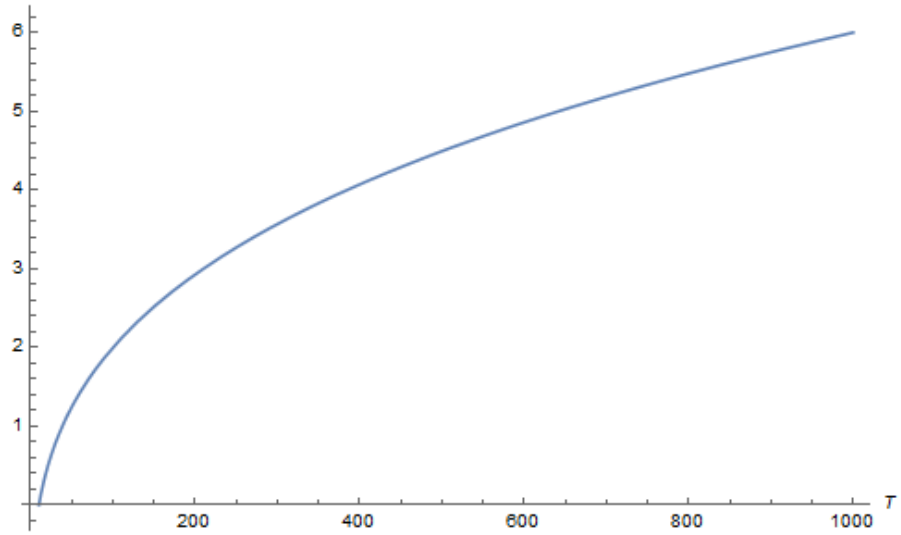
Při návrhu funkce $f_T(T)$ je potřeba brát v úvahu fakt, že měření doby vytváření dat může být zatíženo prakticky libovolně velkou chybou. Z tohoto důvodu bude vhodné funkci f_T(T) definovat tak, aby znatelný rozdíl ve funkčních hodnotách byl pouze tehdy, když bude velký rozdíl v hodnotách parametrů. Rozumná se zdá být taková funkce $f_T(T)$, pro kterou by přibližně platilo, že když se argument T zvětší desetkrát, pak se funkční hodnota zdvojnásobí. Takovou podmínku zjevně splňuje funkce 5.4.

$$f_T(T) = 2^{log_{10}(T)} \tag{5.4}$$

Vzhledem k tomu, že logaritmus není definován v případě, kdy argument má hodnotu 0, což se vzhledem k nepřesnosti měření teoreticky může stát, bude vhodné definovat nějakou umělou dolní mez pro argument T. Jako rozumná dolní mez se zdá být hodnota 10, tedy pokud naměřený čas bude menší než 10 ms, do funkce bude dosazena hodnota 10, jinak bude samozřejmě dosazena naměřená hodnota. Dále by bylo vhodné, aby pro danou dolní mez měla funkce hodnotu 0, čehož lze snadno docílit tak, že od celé funkce odečteme hodnotu 2. Výsledná funkce f_T tedy bude ve tvaru 5.5.

$$f_T(T) = 2^{log_{10}(T)} - 2 \tag{5.5}$$

Graf funkce f_T je zobrazen na obrázku 5.3.



Obrázek 5.3: Graf funkce f_T

Definujme nyní novou funkci f_{ST} dvou proměnných S a T, vyjadřující příspěvek velikosti položky a času jejího generování k celkové prioritě. Předpokládejme, že všechny uživatelské konstanty k_R, k_S, k_T jsou nastaveny na implicitní hodnotu $\frac{1}{3}$. Funkce f_{ST} bude tedy ve tvaru 5.6.

$$f_{ST}(S,T) = \frac{1}{3}f_S(S) + \frac{1}{3}f_T(T) \tag{5.6}$$

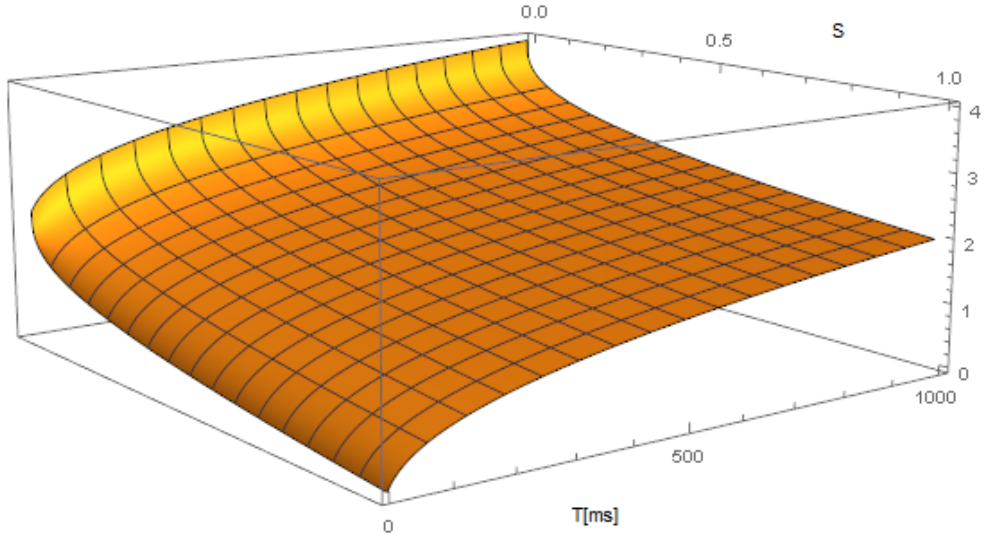
Pokud budeme předpokládat, že naměřené časy T pro jednotlivé položky ve většině případů budou přibližně spadat do intervalu $< 10; 1000 >$, pak je zřejmé, že příspěvky funkcí f_S a f_T jsou v nepoměru, protože $f_S(0) = 1$, ale $f_T(1000) = 6$. Bylo by vhodné toto napravit, což lze provést jednoduše tak, že funkci $f_S(S)$ celou vynásobíme hodnotou 6. Dostáváme tedy výslednou funkci f_S ve tvaru 5.7.

$$f_S(S) = 6 \cdot (1 - \sqrt[3]{S}) \tag{5.7}$$

Tedy funkce f_{ST} pro $k_S = k_T = \frac{1}{3}$ bude ve tvaru 5.8.

$$f_{ST}(S,T) = \frac{6 \cdot (1 - \sqrt[3]{S})}{3} + \frac{2^{log_{10}(T)} - 2}{3} \tag{5.8}$$

Graf funkce $f_{ST}(S,T)$ pro $S \in < 0,1 >, T \in < 10, 1000 >$ ukazuje obrázek 5.4.



Obrázek 5.4: Graf funkce $f_{ST}(S, T)$

5.3.2 Snižování počtu přístupů

Hodnota R_i konkrétní položky by měla být zvýšena o 1 pokaždé, když nastane přístup k dané položce, tedy když si uživatel danou položku od *cache* vyžádá. Zároveň by ale hodnota R_i měla být postupně pro všechny položky snižována. Toho můžeme dosáhnout v podstatě dvěma způsoby. Prvním způsobem je vynásobení hodnoty R_i nějakým číslem z intervalu $(0; 1)$. Druhým způsobem je odečtení nějakého nezáporného čísla od hodnoty R_i. První způsob se nezdá být příliš vhodný, protože při násobení nemůže hodnota R_i nikdy klesnout pod nulu. Pokud by pak příspěvek k prioritě spočtený z hodnot S_i a T_i byl příliš vysoký, daná položka by nikdy nemusela být z *cache* odstraněna, ani kdyby k ní nebylo přistoupeno velmi dlouhou dobu. Při použití druhého způsobu, tedy odečítání, může hodnota R_i klesnout do záporných čísel. Tím je zaručeno, že priorita každé položky, ke které nebude dlouho přistupováno, dříve či později klesne dostatečně nízko na to, aby daná položka odstraněna byla.

Odečítání by bylo nejvhodnější provádět při každém přístupu do *cache*, tedy vždy, když přistoupíme k nějaké položce, pak by hodnota R_i dané položky měla být o 1 zvýšena a zároveň pro všechny ostatní položky o něco snížena. Pro velmi vysoký počet položek by ovšem tento přístup mohl být poněkud neefektivní. Z tohoto důvodu bude vhodnější provést odečítání vždy najednou až ve chvíli, kdy bude potřeba nějakou položku vyřadit a bude pro

všechny položky počítána priorita. Odečtená hodnota pak může být vyjádřena jako $r \cdot k$, kde r je počet přístupů do *cache*, které nastaly mezi aktuálním a předchozím výpočtem priority pro danou položku, a k je hodnota odečítaná pro jeden přístup. Pokud by hodnota k byla konstantní, pak je takovýto způsob odečítání v pořádku. Ovšem daleko vhodnější by bylo, kdyby odečítaná hodnota byla tím menší, čím více položek je v *cache* aktuálně uloženo, protože nedává příliš velký smysl, aby byla od všech položek odečtena stejná hodnota, pokud budou v *cache* 3 položky, jako když jich bude 1000. Vhodné bude odečítat pro každý přístup do *cache* například hodnotu $\frac{1}{n}$, kde n je aktuální počet položek v *cache*. Zde je ovšem nutné počítat s tím, že při odečtení $r \cdot \frac{1}{n}$ bude výsledná priorita zatížena určitou chybou, protože ztrácíme informaci o tom, kolik položek bylo v *cache* uloženo při každém z r přístupů, a bereme v úvahu pouze počet položek, který je aktuální. Ideální by bylo, kdyby uživatel měl možnost volby a sám by určil, zda má být priorita přepočítána při každém přístupu do *cache*, nebo až při hledání položek k vyřazení, přičemž v druhém případě je nutné počítat s určitou chybou.

5.3.3 Shrnutí

Výsledný způsob výpočtu priority v navržené politice *cachování* vyjadřuje rovnice 5.9.

$$P_i = k_R \cdot R_i + 6k_S \cdot (1 - \sqrt[3]{S_i}) + k_T \cdot (2^{log_{10}(T_i)} - 2) \tag{5.9}$$

Všechny použité symboly jsou popsány na začátku této sekce a v podsekci 5.3.1, způsob výpočtu hodnoty R_i je popsán v podsekci 5.3.2.

5.4 Použití pevného disku

V kapitole 4, konkrétně 4.3, bylo řečeno, že při odstranění dat z *cache* můžeme zvážit jejich zapsání na pevný disk a v případě potřeby je opět načíst do paměti. V takovém případě by uživatel musel dodat funkci pro serializaci a deserializaci dat, která by pak byla knihovnou automaticky používána při ukládání a načítání. K jednotlivým datovým položkám by byla uložena informace o tom, zda jsou data právě v paměti, nebo na disku, případně v jakém souboru. Problémem použití pevného disku je fakt, že přístup k němu je většinou oproti přístupu k operační paměti poměrně dost pomalý. Tedy ne vždy se ukládání dat vyřazených z paměti na pevný disk vyplatí, protože

doba zápisu, případně čtení dat, může být větší, než doba vygenerování nových dat. Z tohoto důvodu není možné obecně rozhodovat, zda daná data uložit na disk či nikoliv, vždy při jejich vyřazení z paměti, protože před tím, než zápis skutečně provedeme, netušíme, jak dlouho potrvá, a netušíme ani, jak dlouho potrvá opětovné načtení dat.

Jedním z možných řešení tohoto problému by bylo ukládat v každém *cache* objektu zpočátku všechna data a při každém zápisu i načtení provést časové měření. Po načtení určitého počtu datových položek z disku by byl spočten průměr všech naměřených časů načítání a ukládání dat a v případě, že by tento průměr byl tak vysoký, že se ukládání na disk očividně nevyplatí, by bylo ukládání na disk trvale zakázáno pro celý *cache* objekt. V případě, kdy by ukládání na disk zakázáno nebylo, by byl spočtený průměr aktualizován s každými uloženými a načtenými daty a ukládání na disk by mohlo být dodatečně zakázáno, pokud by se průměr vyšplhal příliš vysoko. Toto řešení zcela jistě není dokonalé a jeho výhodou je spíše implementační jednoduchost. Jeho hlavním problémem je fakt, že dokud nebude k dispozici dostatek naměřených hodnot, budou na disk ukládány všechny položky vyřazené z paměti a načteny budou všechny položky z disku, které budou od *cache* objektu vyžadovány. Další nevýhodou je, že pokud bude zápis na disk jednou zakázán, už nikdy nebude znovu povolen, a to ani pokud by nastala situace, kdy se po nějaké době začne zápis na disk vyplácet, protože například data ukládaná do *cache* budou menší.

Dalším možným řešením je neprovádět zápis na disk až tehdy, když budou data vyřazena z paměti, ale již po jejich uložení do *cache*. Tento zápis by proběhl v paralelním vlákně, aby zbytečně nezdržoval běh aplikace, a doba zápisu by byla změřena. Okamžitě po zápisu by bylo, taktéž v paralelním vlákně, provedeno načtení daných dat, a to čistě za účelem změření doby tohoto načtení. Pokud by v průběhu tohoto procesu byla do *cache* uložena další data, pak by jejich uložení na disk a měřené načtení bylo provedeno až po uložení a načtení dat předchozích. Takto by bylo možné pro každou datovou položku zvlášť rozhodnout, zda po jejím vyřazení z paměti má být uložena na disk, či nikoliv. Problém by mohl nastat ve chvíli, kdy se politika *cachování* rozhodne odstranit z paměti data, jejichž proces zápisu na disk a načtení z disku v paralelním vlákně ještě nebyl dokončen. Zde by uživatel měl mít možnost nastavit, zda v takové situaci budou data rovnou odstraněna a proces jejich zápisu a čtení bude zrušen, či zda se má čekat, až proces čtení a zápisu bude dokončen. Možné by bylo také čekat pouze určitou dobu, a pokud do skončení této doby proces zápisu a čtení daných dat nedoběhne,

bude ukončen.

Dalším řešením by mohlo být provádění měření rychlosti zápisu na disk a čtení z disku již při instalaci aplikace. Výsledky měření by mohly být uloženy do konfiguračního souboru, který by byl načten při každém spuštění aplikace a naměřené časy by posloužily jako určité referenční hodnoty, s jejichž pomocí by bylo možné odhadnout dobu čtení a zápisu každé datové položky. Měření by bylo provedeno tak, že by byl na disk zapsán blok dat o určité velikosti a následně by byl opět přečten. Z doby zápisu a čtení celého bloku by byla spočtena doba zápisu a čtení jednoho *bytu* dat. Odhad doby zápisu dat by pak byl spočten jako $t_w \cdot n$, kde t_w je doba zápisu jednoho *bytu* a n je počet *bytů* serializovaných dat k zápisu. Obdobně odhad doby čtení dat by byl spočten jako $t_r \cdot n$, kde t_r je doba čtení jednoho *bytu* a n je počet *bytů* serializovaných dat k přečtení. Měření by bylo prováděno i při každém zápisu dat na disk a čtení dat z disku v rámci *cache* při běhu aplikace, přičemž tyto nově naměřené hodnoty by posloužily ke zpřesnění hodnot uložených v konfiguračním souboru, výkon pevného disku se totiž může měnit například z důvodu jeho různého zaplnění. Nevýhodou tohoto řešení je fakt, že do naměřených časů nemůže být započtena doba serializace a deserializace dat, protože ta bude pravděpodobně pro každou datovou položku odlišná.

Z výše popsaného vyplývá, že využití pevného disku nemusí vždy přinést výhody. I kdyby byla na disk skutečně vždy zapsána jen data, jejichž doba zápisu na disk i čtení z disku je menší než doba jejich vytvoření, stále může v určitých situacích použití pevného disku přinést zpomalení. Uvažujme například situaci, kdy na vstup přijdou nová data, jejichž *hash* hodnota se shoduje s *hash* hodnotou nějakých dat uložených na disku, ovšem samotná data se neshodují. Proběhne tedy načtení dat z disku, které je nejspíše znatelně pomalejší než čtení z paměti, jen proto, aby bylo provedeno porovnání těchto dat s novými daty na vstupu, přičemž toto porovnání stejně skončí neshodou, tedy data byla načítána prakticky zbytečně. Uživatel by tedy rozhodně měl mít možnost využívání pevného disku úplně vypnout.

Nejspíše je vhodné podotknout, že s implementací využití pevného disku při *cachování* se v rámci této práce nepočítá. Jedná se spíše o návrh případného budoucího rozšíření.

6 Testování

V průběhu implementace byla knihovna testována na jednoduchých příkladech. Nakonec byla knihovna otestována na dvou konkrétních scénářích, které zde budou popsány. Prvním scénářem je uměle vykonstruovaný experiment, který byl vytvořen pro ověření funkcionality a kompatibility s knihovnou *VTK*. Druhým scénářem je integrace *cachovací* knihovny do reálné aplikace a testování s reálnými daty. Testování bylo prováděno na počítači s procesorem *Intel® Pentium® Processor N3540* (taktovací frekvence 2.16 GHz, 4 jádra, L1 *cache* 224 kB, L2 *cache* 2 MB), grafickou kartou *NVIDIA GeForce 920M* a 4 GB operační paměti s taktovací frekvencí 666 MHz.

6.1 Umělý experiment

V rámci prvního testovacího scénáře byly *cachovány* výsledky decimace polygonové sítě koule, resp. elipsoidu, vytvořené pomocí třídy *vtkSphereSource* z knihovny *VTK*.

Funkcionalita je demonstrována na jednoduché animaci, kde se vedle sebe nachází několik koulí, které jsou postupně škálováním roztahovány a smršťovány ve směru osy y. V každém kroku animace je každá koule zobrazena po aplikaci transformace škálování o vektor $[1, h_i, 1]$, kde i je index kroku animace a $h_i = h_{i-1} + \triangle h$ je pak výška vzniklého elipsoidu v daném kroku animace. Hodnota $|\triangle h|$ je konstantní po celou dobu animace, ale různá pro jednotlivé koule, hodnota $\triangle h$ tedy pro každou kouli pouze mění znaménko. Pokud hodnota h_i překročí určitou hranici, změní se na zápornou a koule se začne smršťovat, pokud pak hodnota h_i klesne pod určitou hranici, změní se opět na kladnou a koule se začne roztahovat. Horní hranice každé koule se sama stejným způsobem snižuje a zvyšuje, tedy není konstantní, ovšem její hodnoty se periodicky opakují, a to pro každou kouli s jinak velkým krokem a jinak velkým maximem i minimem. Pro některé koule je hodnota $\triangle h$ zpočátku kladná, pro jiné záporná. Každá koule je také generována s jiným rozlišením její polygonové sítě. Výše popsanými vlastnostmi animace je dosaženo toho, že všechna *cachovaná* data nejsou stejně velká, nevytvářejí se přibližně stejnou dobu a neopakují se stejně často.

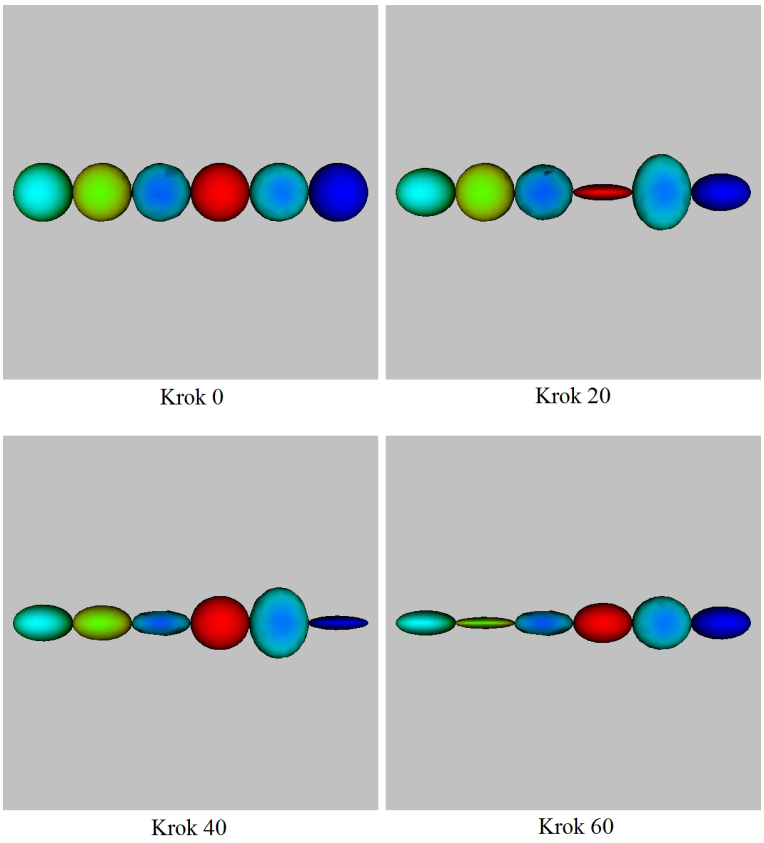
V každém kroku animace je po škálování provedena decimace vzniklé polygonové sítě o přibližně 90% a teprve tato decimovaná síť je vykreslena. Pro decimaci je použita třída knihovny *VTK vtkDecimatePro*. Vzhledem k tomu, že decimace pomocí této třídy trvá poměrně dlouho a celou animaci velmi zpomaluje (což v tomto umělém experimentu je záměr), právě její výsledky jsou *cachovány*. Jako vstup je zde do *cache* ukládána polygonová síť každé koule po provedení škálování a jako výstup je ukládána daná síť po decimaci. Obrázky 6.1 a 6.2 ukazují schéma roury jedné koule pro případ bez *cache* a případ s *cache*.

Obrázek 6.1: Roura umělého experimentu v případě bez *cache*

Obrázek 6.2: Roura umělého experimentu v případě s *cache*

Třída *CachingDecimator* je definována v rámci tohoto experimentu a dědí od třídy *vtkDecimatePro*, jejíž funkcionalitu využívá a zároveň implementuje použití *cachovací* knihovny. Třída *vtkTransform* provádí transformaci škálování pro jednotlivé kroky animace a třída *vtkElevationFilter* slouží pouze pro barevné odlišení jednotlivých koulí.

Vzhledem k tomu, že animace pro každou kouli se periodicky opakuje, dříve či později se začnou opakovat i vstupní sítě, což s použitím *cache* způsobí znatelné zkrácení doby decimace. Obrázek 6.3 ukazuje výstup experimentu ve čtyřech různých krocích animace.



Obrázek 6.3: Ukázka čtyř různých kroků animace umělého experimentu

Samotné testování probíhalo tak, že bylo spuštěno prvních 3000 kroků animace a byl změřen čas všech těchto kroků, zároveň byl změřen počet úspěšných přístupů do *cache*. Za úspěšný přístup do *cache* je považována situace, kdy se na vstupu ocitnou vstupní data, pro která jsou aktuálně v *cache* k dispozici data výstupní, v angličtině je tento jev označován jako *cache hit*, neúspěšný přístup je pak označován jako *cache miss*. Za jeden krok animace je považováno provedení jednoho kroku pro všechny koule. Vzhledem k tomu, že animace obsahuje celkem 6 koulí, bylo v rámci jednoho měření prováděno celkem $3000 \cdot 6 = 18000$ decimací. V průběhu celé animace může být do *cache* uloženo celkem 74 653 324 *bytů* různých dat. Nejdříve byly testovány dva základní případy, tedy animace bez jakékoliv *cache* a animace používající *cache* s neomezenou kapacitou. Dále, z důvodu otestování *cachovací* politiky (viz 5.3) a nalezení vhodné kombinace uživatelských konstant k_R, k_S, k_T, bylo provedeno několik měření s *cache* o kapacitě 60 MB a několik měření s *cache* o kapacitě 30 MB. Výsledky jsou uvedeny v tabulce 6.1, procentuálně uvedené počty úspěšných přístupů jsou zaokrouhleny na jedno desetinné místo.

Kapacita *cache*	k_R	k_S	k_T	Čas	Počet úspěšných přístupů
Žádná	-	-	-	1302 s	0 (0 %)
Neomezená	-	-	-	125 s	17587 (97.7 %)
60 MB	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	167 s	16886 (93.8 %)
60 MB	1	0	0	173 s	16911 (94 %)
60 MB	0	1	0	216 s	16867 (93.7 %)
60 MB	0	0	1	328 s	12589 (69.9 %)
60 MB	0.5	0.5	0	174 s	16919 (94 %)
60 MB	0.5	0	0.5	174 s	16886 (93.8 %)
60 MB	0	0.5	0.5	336 s	13212 (73.4 %)
60 MB	0.7	0	0.3	182 s	16924 (94 %)
60 MB	0.3	0	0.7	190 s	16821 (93.5 %)
60 MB	0.7	0.3	0	184 s	16914 (94 %)
60 MB	0.3	0.7	0	184 s	16938 (94.1 %)
30 MB	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	745 s	8854 (49.2 %)
30 MB	1	0	0	735 s	9065 (50.4 %)
30 MB	0	1	0	1025 s	8796 (48.9 %)
30 MB	0	0	1	912 s	7323 (40.7 %)
30 MB	0.5	0.5	0	711 s	9298 (51.7 %)
30 MB	0.5	0	0.5	684 s	8728 (48.5 %)
30 MB	0	0.5	0.5	862 s	5273 (29.3 %)
30 MB	0.7	0	0.3	716 s	8936 (49.6 %)
30 MB	0.3	0	0.7	664 s	8337 (46.3 %)
30 MB	0.15	0	0.85	643 s	8345 (46.4 %)
30 MB	0.05	0	0.95	614 s	8161 (45.3 %)
30 MB	0.02	0	0.98	692 s	7525 (41.8 %)
30 MB	0.035	0	0.965	672 s	7736 (43 %)

Tabulka 6.1: Výsledky měření umělého experimentu

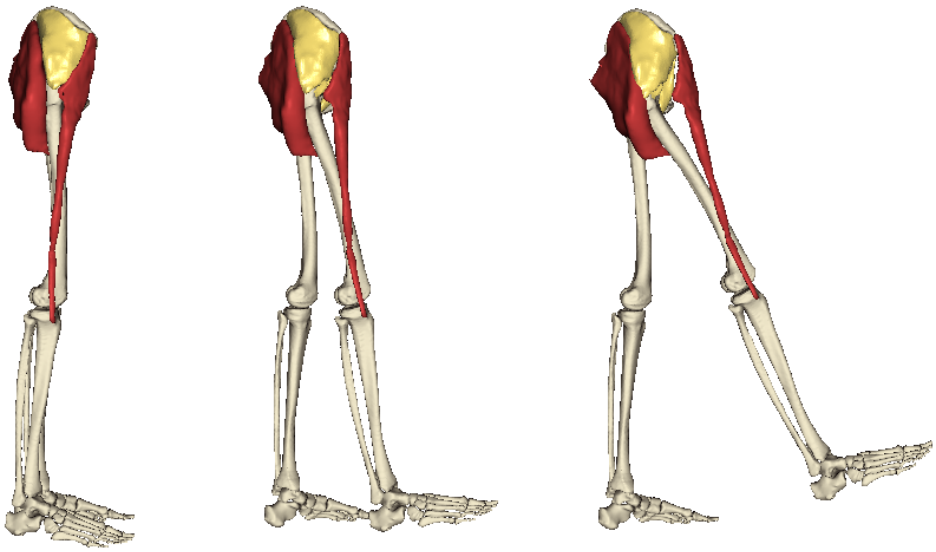
Z výsledků je zřejmé, že rozdíl mezi animací s žádnou *cache* a animací s neomezeně velkou *cache* je velmi znatelný. Pro *cache* o kapacitě 60 MB snaha o dosažení lepšího času nepřinášela znatelné zlepšení, z tohoto důvodu lze konstatovat, že výchozí konfigurace $k_R = k_S = k_T = \frac{1}{3}$ se pravděpodobně v tomto případě blíží konfiguraci optimální. Pro *cache* o kapacitě 30 MB se jako optimální zdá být taková konfigurace, kde dominantní roli ve výpočtu priority hraje doba generování výstupních dat, tedy konstanta k_T je vysoká, ovšem zároveň musí alespoň malou roli hrát počet referencí, tedy konstanta k_R je sice nízká, ale neblíží se nule. Zajímavý je také fakt, že nejvyšší počet

úspěšných přístupů do *cache* nastává tehdy, kdy jsou ve výpočtu s podobnou váhou zastoupeny počet referencí a velikost položky (konstanty k_R, k_S), přičemž toto platí jak pro kapacitu 30 MB, tak pro kapacitu 60 MB.

6.2 Testování na reálných datech

V rámci druhého testovacího scénáře byla *cachovací* knihovna integrována do aplikace *lhpBuilder*, na jejímž vývoji se podílí Katedra informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity v Plzni.

Jednou z funkcí této aplikace je deformace modelů svalů do různých poloh s ohledem na překážky, jako jsou lidské kosti, a jejich vizualizace. Testování bylo prováděno na scénáři, kde je možné zobrazit několik svalů z pánevní oblasti a oblasti pravého stehna. Tyto svaly je možné zobrazit v 19 různých polohách, přičemž v poloze 0 jsou obě nohy vedle sebe na zemi a s každou další polohou se natažená pravá noha zvedá o 5 stupňů směrem nahoru. V poloze 18 je pak pravá noha rovnoběžná se zemí. Uživatel si může jednotlivé polohy nechat zobrazit automaticky za sebou a sledovat celý proces jako animaci, kde se pravá noha postupně zvedá. Pro každou polohu je potřeba provést deformaci všech zobrazených svalů a právě tato deformace trvá velmi dlouho (řádově sekundy až desítky sekund). Pokud by si uživatel zobrazil například polohu 0, následně polohu 1 a pak znovu polohu 0, všechny zobrazené svaly by se musely deformovat třikrát, jednou pro polohu 0, pak pro polohu 1 a následně opět pro polohu 0, a to i přes to, že pro polohu 0 byla již jednou deformace provedena. Integrace *cache* právě tomuto opakování výpočtu deformací měla zabránit. Jako vstup se do *cache* ukládají modely původních nezdeformovaných svalů spolu s informacemi o překážkách (lidské kosti) a poloze, do které má být sval deformován. Jako výstup se pak ukládají pouze modely zdeformovaných svalů. Obrázek 6.4 ukazuje pět různých svalů ve třech různých polohách. Konkrétně jde o svaly *iliacus*, *gluteus maximus*, *gluteus medius*, *gluteus minimus* a *tensor fascia latae*.



Obrázek 6.4: Ukázka tří různých poloh svalů

Samotné testování probíhalo tak, že bylo naráz zobrazeno pět svalů, stejných jako na obrázku 6.4, a to postupně v polohách 0, 3, 9, 3, 9, 3, 9, 6, 9, 6, 3, 0, 3, 6, 9. Pro každou polohu byla změřena doba deformace všech svalů a výsledkem je pak součet všech těchto změřených dob pro všechny polohy a všechny svaly. Zároveň je zaznamenáván počet úspěšných přístupů do *cache*. V rámci dané posloupnosti poloh může být do *cache* uloženo celkem 24 886 292 *bytů* různých dat. Měření bylo opět provedeno na základních případech, tedy na případu bez *cache* a na případu s *cache* o neomezené kapacitě. Dále bylo testování provedeno s *cache* o kapacitě 20 MB a s *cache* o kapacitě 10 MB. Oproti umělému experimentu (viz 6.1) jsou zde dva zásadní rozdíly. První rozdíl je v tom, že datových položek je zde mnohem méně, ale každá položka zabírá mnohem větší část kapacity *cache* (na jednu položku připadá v průměru $\frac{1}{20}$ celkového objemu dat, která mohou být do *cache* uložena). Druhým rozdílem je fakt, že doba vytváření dat je zde v průměru mnohem delší. Výsledky měření ukazuje tabulka 6.2, procentuálně uvedené počty úspěšných přístupů jsou zaokrouhleny na jedno desetinné místo.

Kapacita *cache*	k_R	k_S	k_T	Čas	Počet úspěšných přístupů
Žádná	-	-	-	779 s	0 (0 %)
Neomezená	-	-	-	200 s	55 (73.3 %)
20 MB	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	274 s	45 (60 %)
20 MB	1	0	0	300 s	45 (60 %)
20 MB	0	1	0	336 s	48 (64 %)
20 MB	0	0	1	299 s	43 (57.3 %)
20 MB	0.5	0.5	0	326 s	45 (60 %)
20 MB	0.5	0	0.5	275 s	47 (62.7 %)
20 MB	0	0.5	0.5	288 s	44 (58.7 %)
20 MB	0.7	0	0.3	348 s	39 (52 %)
20 MB	0.3	0	0.7	295 s	44 (58.7 %)
10 MB	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	545 s	13 (17.3 %)
10 MB	1	0	0	716 s	9 (12 %)
10 MB	0	1	0	699 s	21 (28 %)
10 MB	0	0	1	520 s	16 (21.3 %)
10 MB	0.5	0.5	0	664 s	13 (17.3 %)
10 MB	0.5	0	0.5	573 s	9 (12 %)
10 MB	0	0.5	0.5	473 s	17 (22.7 %)
10 MB	0	0.3	0.7	473 s	16 (21.3 %)
10 MB	0	0.7	0.3	463 s	18 (24 %)
10 MB	0	0.85	0.15	481 s	20 (26.7 %)

Tabulka 6.2: Výsledky měření experimentu v aplikaci *lhpBuilder*

Je vidět, že i v tomto experimentu je velmi znatelný rozdíl mezi testovacím případem s *cache* a případem bez *cache*, i když tento rozdíl není tak zásadní jako v umělém experimentu, což je ovšem dáno tím, že zde je poměrně malý počet výpočtů a úspěšný přístup do *cache* může nastat jen u 55 z 75 těchto výpočtů, což je pouhých 73.3 %. Při testování s vyšším počtem poloh a třeba i s více svaly bychom téměř jistě dosáhli mnohem lepších výsledků, vzhledem k době deformací svalů by ale takové testování bylo velmi časově náročné. Stejně jako u umělého experimentu pro *cache* s větší kapacitou, tedy zde 20 MB, snaha o dosažení lepšího času nepřinášela žádné znatelné zlepšení, a lze tedy opět konstatovat, že konfigurace $k_R = k_S = k_T = \frac{1}{3}$ se v tomto případě pravděpodobně blíží konfiguraci optimální. Pro *cache* s kapacitou 10 MB se zdá být vhodná konfigurace, kde konstanta k_R je nulová a konstanta k_S má o něco vyšší hodnotu než konstanta k_T. Dále konfigurace, kde $k_R = k_T = 0$ a $k_S = 1$, ačkoliv se nezdá být časově výhodná, dosahuje pro obě kapacity *cache* nejvyššího počtu úspěšných přístupů.

6.3 Shrnutí

Výsledky testování popsané v této kapitole lze shrnout do následujících poznatků týkajících se uživatelských konstant *cachovací* politiky.

V případech, kdy kapacita *cache* není zásadně menší než celkový objem dat, která mohou být do *cache* uložena, je, zdá se, dobře použitelná výchozí konfigurace $k_R = k_S = k_T = \frac{1}{3}$. Tedy v takovém případě očividně hrají roli všechny veličiny, které přispívají k výpočtu priority položek. V případech, kdy kapacita *cache* naopak zásadně menší je, již situace není tak jednoznačná, lze ale odhadovat, že zde vždy určitou roli hraje doba generování výstupních dat, tedy konstanta k_T by neměla být nulová.

Samozřejmě nelze tvrdit, že výše zmíněné poznatky jsou platné obecně pro všechny případy. Pro nalezení takové konfigurace, která by se alespoň blížila obecně optimální konfiguraci, by bylo potřeba otestovat mnohem více scénářů, což přesahuje rámec této práce.

7 Závěr

V rámci této práce byla navržena a implementována knihovna, s jejíž pomocí je možné provádět *cachování* výstupních dat libovolných výpočtů či procesů odpovídajících libovolným vstupním datům. Knihovna byla navržena tak, aby byla kompatibilní s nástrojem *VTK*, aby ji bylo možné využít pokud možno kdekoliv a aby kromě počáteční konfigurace vyžadovala jen minimální interakci s programátorem. Pro tuto knihovnu byla dále navržena a implementována *cachovací* politika starající se o vyřazování dat z *cache* v případě jejího zaplnění, a to s ohledem na frekvenci a nedávnost přístupů k datovým položkám, velikost položek a dobu vytváření položek.

Knihovna byla úspěšně otestována na dvou konkrétních scénářích, přičemž jeden ze scénářů byl zcela uměle vykonstruovaný a v rámci druhého scénáře byla knihovna integrována do reálné aplikace. Testování ukázalo, že knihovna je kompatibilní s datovými objekty nástroje *VTK* a že její nasazení může skutečně způsobit velké zrychlení konkrétního výpočtu či procesu (viz kapitola 6). Vzhledem k tomu, že knihovna byla až do samého konce navrhována a vyvíjena bez jakékoliv znalosti aplikace *lhpBuilder*, do které byla v rámci testování integrována, a jen se základní znalostí nástroje *VTK*, a zároveň oba scénáře, na kterých byla knihovna testována, jsou poměrně odlišné (jedinou společnou vlastností je využití *VTK*), lze alespoň do jisté míry konstatovat i její obecnost.

Do budoucna by bylo jistě vhodné doimplementovat například využití pevného disku (viz 5.4), zabudovat systém pro sběr statistik či systém pro logování z důvodu pohodlnějšího ladění a také implementovat načítání konfigurace ze souboru. Dále, protože se předpokládá budoucí využití knihovny ve vizualizačních procesech používajících nástroj *VTK*, by bylo vhodné navrhnout a implementovat systém, který by dokázal automaticky provést úpravu zdrojového kódu libovolné třídy z *VTK* tak, aby při své hlavní činnosti prováděla *cachování* s použitím této knihovny. Takový systém by mohl velmi usnadnit integraci knihovny například do jiných částí aplikace *lhpBuilder*.

Závěrem lze říci, že knihovna, která vznikla jako výstup této práce, může dobře posloužit původně zamýšlenému účelu (integrace *cache* do aplikace *lhpBuilder*) a, zdá se, splňuje všechny požadavky na ni kladené.

Literatura

[1] [online]. . [cit. 9.11.2015]. Dostupné z: http://www.vtk.org/Wiki/VTK/Examples/Cxx.

[2] [online]. . [cit. 10.11.2015]. Dostupné z: http://www.vtk.org/Wiki/VTK/Tutorials/SmartPointers.

[3] *The buffer cache* [online]. [cit. 9.10.2015]. Dostupné z: http://www.tldp.org/LDP/sag/html/buffer-cache.html.

[4] *cache memory definition* [online]. 2014. [cit. 6.10.2015]. Dostupné z: http://searchstorage.techtarget.com/definition/cache-memory.

[5] *Memory part 2: CPU caches* [online]. 2007. [cit. 6.10.2015]. Dostupné z: https://lwn.net/Articles/252125/.

[6] *What's a Disk Cache And Why Is It Important?* [online]. [cit. 9.10.2015]. Dostupné z: http://www.acronis.com/en-us/resource/tips-tricks/2004/disk-cache.html.

[7] [online]. [cit. 22.11.2015]. Dostupné z: http://redis.io/.

[8] [online]. 2015. [cit. 19.2.2016]. Dostupné z: http://en.cppreference.com/w/cpp/memory/shared_ptr.

[9] *Ellipses and Variadic Templates* [online]. [cit. 6.12.2015]. Dostupné z: https://msdn.microsoft.com/en-us/library/dn439779.aspx.

[10] *Cache v prohlížečích* [online]. 2008. [cit. 9.10.2015]. Dostupné z: https://www.fio.cz/navody/cache/.

[11] Arora, K. – Ch, D. R. Web Cache Page Replacement by Using LRU and LFU Algorithms with Hit Ratio: A Case Unification, 2014.

[12] Bendersky, E. *Variadic templates in C++* [online]. 2014. [cit. 6.12.2015]. Dostupné z: http://eli.thegreenplace.net/2014/variadic-templates-in-c/.

[13] Bžoch, P. Zvyšování výkonu a udržování konzistentnosti dat v mobilních zařízeních pro distribuované systémy souborů, 2014.

[14] Bžoch, P. et al. Towards Caching Algorithm Applicable to Mobile Clients, 2012.

[15] Brown, A. *The Architecture Of Open Source Applications*. 2008.

[16] Chou, H.-T. – Dewitt, D. J. An Evaluation of Buffer Management Strategies for Relational Database Systems, 1985.

[17] Draves, R. P. Page Replacement and Reference Bit Emulation in Mach, 1991.

[18] EFFELSBERG, W. – HAERDER, T. Principles of Database Buffer Management, 1984.

[19] Jiang, S. – Zhang, X. LIRS:An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance, 2002.

[20] Jiang, S. – Chen, F. – Zhang, X. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement, 2005.

[21] Johnson, T. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm, 1994.

[22] Lee, D. et al. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies, 2001.

[23] Ng, Y. *In the World of DNS, Cache is King* [online]. 2014. [cit. 9.10.2015]. Dostupné z: `http://blog.catchpoint.com/2014/07/15/world-dns-cache-king/`.

[24] Prata, S. *Mistrovství v C++*. 2011.

[25] Reed, B. – Long, D. D. E. Analysis of Caching Algorithms for Distributed File Systems, 1996.

[26] Rouse, M. *cache server* [online]. 2009. [cit. 9.10.2015]. Dostupné z: `http://whatis.techtarget.com/definition/cache-server`.

[27] Schroeder, W. – Martin, K. – Lorensen, B. *The Visualization Toolkit*. 2002.

[28] Vijayvargiya, A. *An Idiot's Guide to C++ Templates - Part 1* [online]. 2013. [cit. 5.12.2015]. Dostupné z: `http://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Templates-Part`.

[29] Zhou, Y. – James F. Philbin, K. L. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches, 2001.