Vizualizace s grafickými akcelerátory

Diplomová práce

Petr Kadlec

Fakulta elektrotechnická
České vysoké učení technické v Praze
Katedra počítačů

Praha 2004

Anotace

Tato práce předkládá knihovnu některých základních funkcí používaných při vizualizaci vědeckotechnických dat v reálném čase, jako jsou vizualizace dat v kartézské mřížce, vizualizace vektorového pole, nebo částicové systémy. Knihovna pracuje s rozhraním OpenGL, pomocí kterého využívá některé pokročilé vlastnosti moderních grafických akcelerátorů. Implementace této knihovny byla použita v existujícím simulačním systému, kde pomohla zvýšit kvalitu vizualizace.

Abstract

This thesis presents a library of basic tools used in real-time scientific visualization, e.g. data stored in a Cartesian grid, vector field visualization, or particle systems. The library is based on OpenGL and it uses some advanced features of modern graphics accelerators. The library has been implemented and is used in an existing simulation system, where it has helped to improve the quality of visualization results.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v kapitole Literatura.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 zákona č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 15. ledna 2004

Obsah

1 Úvod ... 3
2 Vizualizace vědeckotechnických dat ... 4
2.1 Mřížka hodnot ... 4
2.1.1 Interpolace ... 4
2.1.2 Zobrazení ... 5
2.2 Pole mřížek ... 6
2.3 Částicové systémy ... 6
2.4 Vektorové pole ... 7
2.4.1 Přímé zobrazení ... 8
2.4.2 Spot Noise, LIC, OLIC, FROLIC ... 8
2.4.3 IBFV ... 9
2.5 Potřeba akcelerace ... 10
3 Moderní grafické akcelerátory ... 11
3.1 Minulost ... 11
3.2 Současnost ... 12
3.3 Architektura grafických akcelerátorů ... 12
3.3.1 Lokální videopaměť ... 13
3.3.2 AGP ... 14
3.3.3 Grafický procesor ... 14
3.4 Schopnosti ... 15
3.4.1 Transformace a osvětlování ... 15
3.4.2 Texturování ... 18
4 Rozhraní pro grafické akcelerátory ... 20
4.1 Úvod ... 20
4.2 OpenGL 1.x ... 20
4.2.1 Schopnosti OpenGL ... 21
4.3 Rozšíření OpenGL ... 21
4.3.1 Extenze použité v knihovně Visla ... 22
4.4 OpenGL 2.0 ... 24
4.5 Jiná rozhraní ... 25
5 Vizualizační knihovna Visla ... 27
5.1 Zadání diplomové práce ... 27
5.2 Motivace řešení ... 27
5.2.1 Základní charakteristika MyPCC ... 28

5.2.2 Vizualizace částic ... 28
5.2.3 Vizualizace mřížky dat ... 28
5.2.4 Vizualizace rychlosti proudění ... 28
5.3 Stávající řešení ... 28
5.3.1 Vizualizace částic ... 28
5.3.2 Vizualizace mřížky dat ... 29
5.4 Schopnosti knihovny ... 29
6 Implementace knihovny ... 30
6.1 Požadavky ... 30
6.2 Struktura ... 30
6.3 Popis implementace ... 31
6.3.1 Základní informace ... 31
6.3.2 Všeobecně matematické funkce ... 31
6.3.3 Funkce pro správu a konfiguraci ... 31
6.3.4 Funkce pro práci s texturami ... 32
6.3.5 Funkce pro vykreslování částic ... 32
6.3.6 Funkce pro vykreslování základní mřížky bodů ... 32
6.3.7 Mřížky s interpolací pomocí spline křivek ... 33
6.3.8 Funkce pro vykreslování sady mřížek ... 35
6.3.9 Funkce pro vizualizaci metodou IBFV ... 35
6.3.10 Interní funkce ... 35
7 Testování knihovny ... 36
7.1 Metodologie testování ... 36
7.2 Ukázkové příklady ... 36
7.3 MyPCC ... 36
7.3.1 Porovnání kvality vizualizace ... 37
7.3.2 Zcela nové schopnosti ... 39
7.3.3 Porovnání rychlosti ... 40
8 Závěr, budoucí práce ... 42

1 Úvod

V běžném životě, v technické praxi, ale i ve vědeckém výzkumu se setkáváme s mnoha jevy, které lze jen těžko srozumitelně vyjádřit čísly, ale jsou snadno pochopitelné v grafické podobě. Počítač byl původně navržen tak, aby byl velmi dobře schopen pracovat s čísly; k tomu, aby těmto číslům rozuměl i člověk, je ovšem třeba je vhodným způsobem zobrazit. Tomuto cíli se věnuje odvětví počítačové grafiky, nazývané *vizualizace vědeckotechnických dat* (*scientific visualization*).

Tato práce se zabývá některými vybranými metodami vizualizace dat a prezentuje způsoby, jak je co nejlépe provádět na moderním počítačovém vybavení. Protože metod vědeckotechnické vizualizace existuje nepřeberné množství, není možné pokrýt celou šíři této problematiky v jediné práci daného rozsahu. Zájemce o další používané metody pak může nalézt některé další informace ve zdrojích uvedených v kapitole o použité literatuře. V této práci jsem se věnoval hlavně těm metodám, které jsou využívány v programu MyPCC, který je nejdůležitějším testovacím subjektem této práce. Základním informacím o používaných metodách je věnována druhá kapitola.

Existuje mnoho již klasických metod vizualizace, které jsou schopny v dodaných datech nalézt a zobrazit takové vlastnosti, které jsou nějak zajímavé pro příslušnou aplikaci. Jiné metody slouží ke snazšímu pochopení celého děje, který se v modelu odehrává. Významná část všech těchto metod má ovšem jistý nedostatek – kvůli omezením, daným limitovanou výpočetní silou dostupného počítačového vybavení, je třeba volit kompromis mezi tím, co všechno a v jaké kvalitě se bude zobrazovat, a tím, jak rychle se to bude zobrazovat. Zvláště u rychlosti zobrazování je zřejmé, že ideálním cílem je tzv. *vizualizace v reálném čase*, tzn. že vizualizace má okamžitý efekt, uživatel může interaktivně měnit parametry zobrazování a ihned vidět vliv těchto změn. Výhody takové možnosti jsou patrné: např. při využití vizualizace při výuce má interaktivita nesmírný význam. Možnosti vizualizovat v reálném čase napomáhají schopnosti moderních grafických adaptérů, které jsou schopny některé úlohy provádět samostatně, čímž uvolňují hlavní procesor pro další činnost (např. pro simulaci zobrazovaného modelu). Grafickým akcelerátorům se věnuje třetí kapitola, programátorským rozhraním pro práci s nimi pak čtvrtá.

Výsledkem řešené práce je knihovna v programovacím jazyce C++, kterou jsem nazval Visla. Knihovně se věnují kapitoly 5–6, jejímu použití a testování se věnuje kapitola 7. V době, kdy vznikalo původní zadání této diplomové práce, ještě nebylo známo, že výsledky této diplomové práce se okamžitě využijí v doktorandské práci ing. Gayera, konkrétně v jeho projektu nazvaném „My Pulverized Coal Combustion“ (MyPCC) [1], [2]. Program MyPCC provádí simulaci spalovacích procesů v průmyslových uhelných kotlích a výsledky v reálném čase zobrazuje, k čemuž původně využíval samotné OpenGL. V průběhu řešení této práce se do programu připojila knihovna Visla, což rozšířilo jeho schopnosti a zvýšilo kvalitu vizualizace. Součástí práce je porovnání výsledků dříve používaných vizualizačních metod a výsledků při použití metod implementovaných v této práci.

2 Vizualizace vědeckotechnických dat

Metod vizualizace dat je snad stejně velké množství, jako je jejich aplikací, a jako je způsobů získávání těchto dat. Jelikož je tato práce zaměřena na vylepšení vizualizace v projektu MyPCC, budu se zde zabývat hlavně tam použitými metodami a v příkladech budu odkazovat na použití dané metody v MyPCC.

2.1 Mřížka hodnot

Model, který je zdrojem vizualizovaných dat, je často diskrétní, tzn. poskytuje hodnoty pouze v některých bodech, ne v celém objemu. I tehdy, když dostupný model je teoreticky schopen vypočítat sledované údaje v libovolném místě simulovaného prostoru, jsme obvykle omezeni přesností a výpočetním výkonem počítače, který může zpracovat jenom konečné množství údajů. Výsledkem je, že typickou formou vizualizovaných dat je množina bodů a k nim příslušných hodnot. Rozložení těchto bodů v prostoru samozřejmě závisí na použitém modelu, ale velmi často se jedná o nějakou formu mřížkového uspořádání, z nichž nejjednodušší (a zřejmě i nejčastější) je *kartézská mřížka*, kdy body jsou rozmístěny ve vrcholech pomyslných pravidelných kvádrů. Ještě základnější formou této mřížky je *2D kartézská mřížka*, kdy simulovaný prostor je dvourozměrný.

Typickým příkladem použití je zobrazování sledované skalární hodnoty, např. teploty, v simulovaném modelu. Model nám poskytuje hodnoty teploty v bodech kartézské mřížky, cílem vizualizace je zobrazit rozložení teploty v celém simulovaném prostoru. K tomu jsou potřeba dva základní kroky: interpolace a zobrazení. Účelem interpolace je získat hodnoty v bodech uvnitř mřížky, zobrazení musí zajistit převod původních údajů (které jsou např. ve stupních Celsia) na nějak zobrazitelné parametry (např. na barvu).

2.1.1 Interpolace

Pokud musíme interpolovat dodaná dvourozměrná data, máme na výběr mnoho různých interpolačních metod: lineární (resp. bilineární, tzn. lineární ve dvou rozměrech), interpolace polynomem (kvadratickým, kubickým, obecným), interpolace spline křivkami, specializovaná interpolace, která využívá známých vlastností konkrétního modelu, …

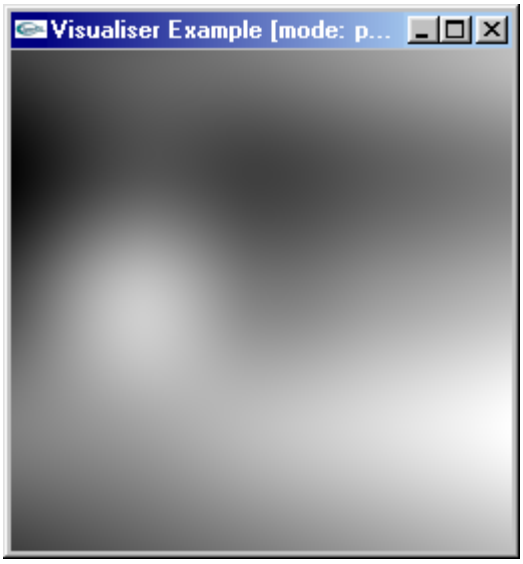
Zřejmě nejčastěji voleným způsobem interpolace je lineární interpolace, jejíž hlavní výhodou je vysoká rychlost. Přesnost této metody je velice závislá na konkrétních datech (což ovšem platí u všech metod), ale je zřejmé, že se hodí hlavně pro modely s vysokým rozlišením, u kterých jsou všechny vlastnosti modelu zřejmé z hodnot v mřížce, protože lineární interpolace nedokáže zobrazit žádné detaily menší než rozteč mřížky. Dalším problémem lineární interpolace je, že v bodech mřížky dochází ke skokové změně derivace, na což je lidské oko poměrně citlivé a vnímá v obrázku takto vzniklé nepřirozené hrany. Navíc, hodnoty v rozích jednoho pole mřížky nemusí patřit do jedné „roviny“, tzn. derivace na protějších hranách

mřížky se mohou lišit. Tento problém ovšem lineární interpolace také nezvládá dobře a výsledkem jsou další nepřirozené hrany, které ve výsledném obrázku tvoří často viditelný trojúhelníkový vzor.

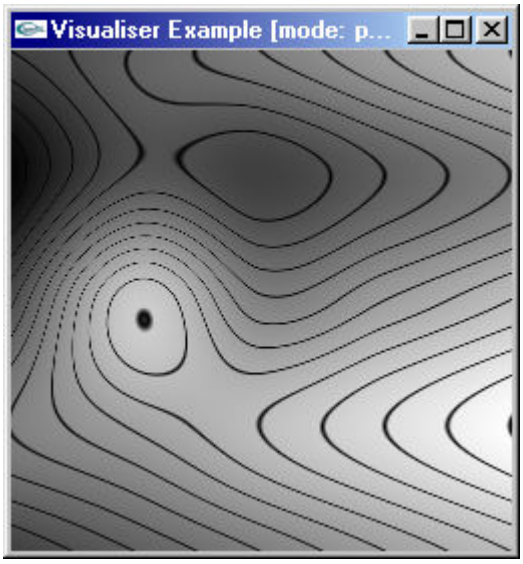
Interpolace s použitím polynomů vyššího stupně je schopna zvýraznit i menší rysy, ovšem za cenu vyšší výpočetní náročnosti. Zde je možno využít interpolace pomocí různých druhů spline křivek, které při použití kubických polynomů zajišťují přinejmenším C_1-spojitost, což se obvykle u běžných dat projeví dostatečnou vizuální kvalitou. Jsou tím totiž odstraněny problémy s uměle vznikajícími hranami, neboť na nespojitost vyšších derivací již lidské oko není tolik citlivé.

2.1.2 Zobrazení

Model nám poskytuje číselné hodnoty sledovaných parametrů. Abychom je mohli prezentovat uživateli, musíme je převést na nějaký zobrazitelný parametr. Metoda, kterou to můžeme provést, závisí na typu dat. Pokud jsou vizualizovaná data skalární (např. teplota), je asi nejuniverzálnější metodou převod na barvu (u použitého příkladu teploty můžeme využít snadno pochopitelné mapování barev od temně rudé pro nejchladnější místa po bílou pro nejteplejší). Pro vícerozměrná (vektorová) data máme mnoho možností, mezi kterými se musíme rozhodnout podle konkrétní aplikace. Buď můžeme rozšířit převod na barvu tak, že jednotlivé složky hodnoty budeme převádět na jednotlivé složky barvy (např. teplotě bude odpovídat světlost, tlaku odstín), výsledkem čehož ovšem může u některých dat být naprosto nepochopitelná barevná plocha. Nebo můžeme v daném místě zobrazit nějaký složitější objekt, jehož vzhled bude záviset na hodnotě příslušného vektoru; takovým objektem může pro běžná vektorová data (reprezentující např. rychlost) být klasická reprezentace vektoru – šipka. Vizualizaci vektorového pole se podrobněji věnuje kapitola 2.4.



obrázek 1 – Mřížka dat



obrázek 2 – Izokřivky

Pokud pro zobrazení dat používáme mřížku hodnot, přičemž hodnoty zobrazujeme pomocí převodu na barvu, výsledný obrázek je sice poměrně snadno srozumitelný, ale často se v něm ztrácí detaily, protože oko nedokáže postřehnout drobné rozdíly barvy. Proto se obrázek ještě často nějak doplňuje. Asi

nejpřehlednější metodou pro zvýraznění detailů je použití *izokřivek*, tzn. křivek, které v obrázku spojují místa se stejnou hodnotou (viz obrázky).

2.2 Pole mřížek

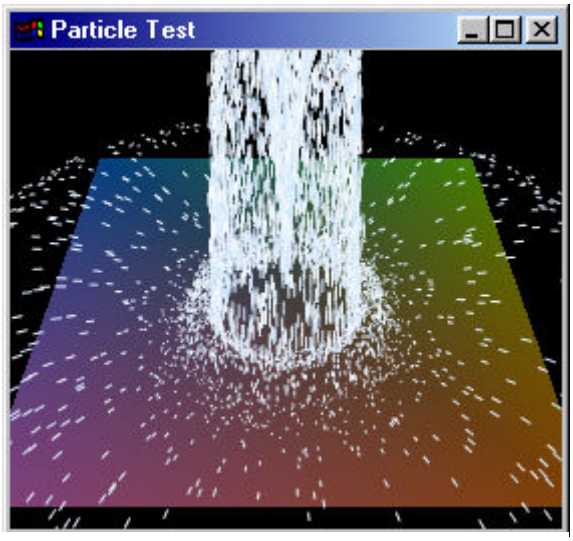
Pokud jsou modelovaná data trojrozměrná, ale opět strukturovaná do kartézské mřížky, můžeme se na výslednou mřížku dívat jako na sadu řezů, které tvoří pole dvourozměrných mřížek. Výsledné řezy pak můžeme zobrazovat buď jednotlivě, nebo, pokud je např. vykreslujeme poloprůhledně, i najednou. (Další techniky, jako např. rekonstrukce povrchu pomocí metody *marching cubes* zde neuvádím, neboť se typicky jedná o metody neinteraktivní vizualizace. Zájemce odkazuji např. na [3].)

I v tomto případě je samozřejmě zapotřebí interpolace. Navíc k předchozí interpolaci uvnitř jednoho řezu, která je stále používána, je zde ještě zapotřebí vyřešit, jak získávat údaje z bodů, které leží mezi jednotlivými řezy. V principu se jedná o tutéž úlohu, jako v případě jednoho řezu, jenom pootočenou do jiné (kolmé) roviny. V tomto případě máme ovšem dvě možnosti, mezi kterými se musíme rozhodnout podle konkrétní úlohy: Buď nejprve získáme hodnoty v mřížce ležící v mezilehlém řezu, čímž vyrobíme novou mřížku, kterou poté běžným způsobem interpolujeme a zobrazíme, nebo nejprve interpolujeme obě sousední mřížky, načež z příslušných sousedících bodů interpolujeme jednotlivé body mezilehlého řezu. První metoda je typicky výrazně rychlejší a méně paměťově náročná (nemusí se vypočítávat a uchovávat dva interpolované řezy), proto se také většinou používá.

Pokud jsme tedy schopni vypočítat libovolný mezilehlý řez, můžeme celý model vizualizovat pomocí animace, „průletu“ modelovaným prostorem. Také můžeme získat „šikmé“ řezy, které nejsou rovnoběžné s žádnou rovinou mřížky. Výsledkem je, že interaktivní vizualizace umožňuje model snadno zkoumat a nalézt tu část modelu, která je pro uživatele zajímavá.

2.3 Částicové systémy

Pokud jsou součástí vizualizovaného modelu nějaké konkrétní objekty, které se účastní simulace (např. částečky uhlí a vzduchu), může být vhodné je přímo zobrazovat. I v situacích, kdy zkoumaný jev sám o sobě



obrázek 3 – Ukázka částicového systému

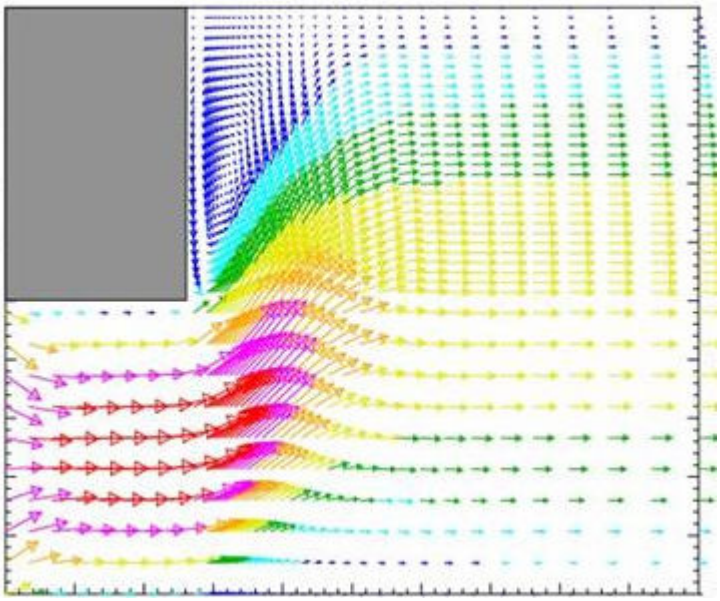
žádné takové diskrétní objekty nemá, je často vhodné je v rámci modelu vyrobit, protože se tím simulace často zjednodušuje a také to umožňuje snazší pochopení vizualizovaného jevu, protože pak není nutné vyrábět abstraktní reprezentaci probíhajících dějů, když jsou tyto děje přímo zobrazeny. Ve všech těchto případech se pro zobrazování drobných objektů (typicky částic) používá obecná skupina metod, nazývaných *částicové systémy* [4]. Pod tímto pojmem je možno si představit široké rozmezí různých technik, od základního vykreslování pevné množiny pohybujících se bodů po proměnlivé skupiny částic, z nichž z některých jsou za určitých podmínek vypouštěny další částice, všechny částice na sebe navzájem působí silami, atd.

Z hlediska vizualizace jako takové (tzn. pokud nebudeme uvažovat schopnost částicového systému ovlivňovat simulaci) má částicový systém méně možností než mřížkové zobrazení z předešlé části. U jednotlivých částic můžeme ovlivňovat barvu, příp. velikost. Změny dalších parametrů (např. tvar částice) nebývají příliš časté.

Základní indikací kvality implementace částicového systému je rychlost, tzn. počet částic, které je schopen zobrazit každou sekundu. Na dnešních počítačích není problém používat částicové systémy s řádově desítkami až stovkami tisíc částic, za jistých podmínek i více.

2.4 Vektorové pole

V oblasti vizualizace vědeckotechnických dat se často setkáváme s potřebou vizualizovat vektorová data, nejčastěji přímo ve formě vektorového pole. Velmi častým zdrojem takového vektorového pole je



obrázek 4 – Přímé zobrazení vektorového pole

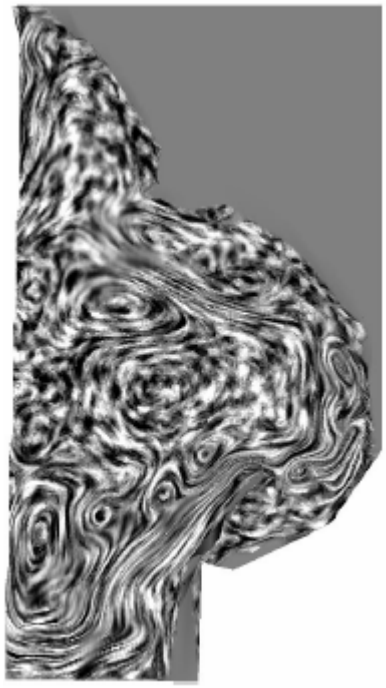
modelování proudění. (V programu MyPCC je např. zapotřebí vykreslovat rychlost proudění v kotli.) Na vstupu tedy máme hodnoty vektorového pole v jistých bodech prostoru (typicky opět ve vrcholech kartézské mřížky, i když to není nezbytně nutné) a cílem je nějakým vhodným způsobem prezentovat strukturu

vektorového pole uživateli. Jelikož se tento problém vyskytuje v mnoha podobách, existuje i mnoho způsobů, jak jej řešit.

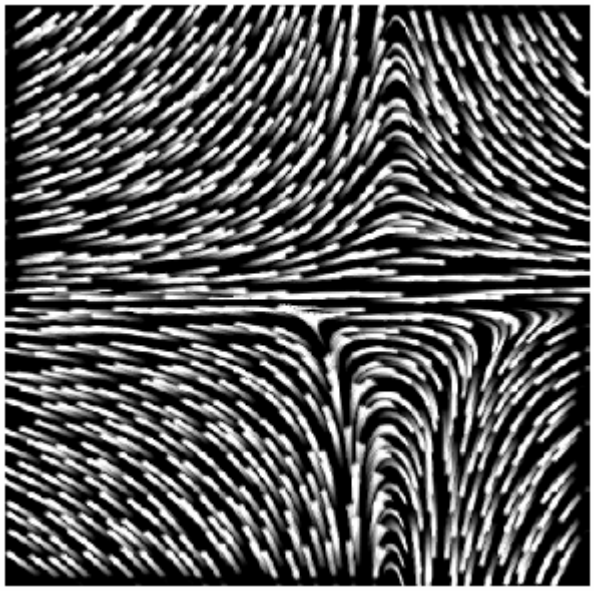
2.4.1 Přímé zobrazení

Jednou z možností je přímo zobrazit vektory získané ze vstupních dat tím způsobem, že některou z metod z části 2.1.2 převedeme vektor na zobrazitelné parametry (např. jednotlivé složky barvy) a tyto parametry pak zobrazíme. Nejsrozumitelnějším způsobem je zřejmě zobrazení každého známého vektoru pomocí běžné „vektorové šipky“, ze které je ihned zřejmý směr i velikost vektoru (viz obrázek). Výhodou tohoto způsobu (nazývaného *arrow plot*) je snadná pochopitelnost, nevýhodou je, že pro hustší mřížky začnou vektory splývat, čímž zobrazení ztrácí přehlednost někdy až k nepoužitelnosti. Také např. u turbulentního proudění se hodnoty vektorového pole mění v prostoru tak prudce, že se zobrazené šipky kříží, což zásadně snižuje srozumitelnost vizualizace.

Místo klasické šipky je možno použít i jiné symboly, např. zužující se úsečku, úsečku s tečkou (nebo jiným symbolem) na jednom konci, nebo i složitější ikony, které mohou reprezentovat další údaje spojené s příslušným bodem.



obrázek 5 – *Spot noise*



obrázek 6 – OLIC

2.4.2 Spot Noise, LIC, OLIC, FROLIC

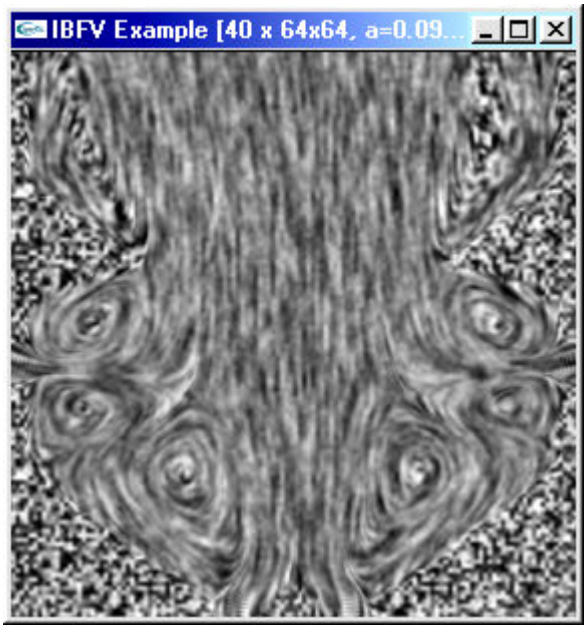
Další možností, jak zobrazit vektorové pole, je vzít nějakou víceméně náhodnou množinu bodů a v každém z nich vhodným způsobem zobrazit hodnotu, kterou tam vektorové pole má. Sloučením všech zobrazení pak může vzniknout obrázek, který dobře popisuje celkovou strukturu vektorového pole. Jednotlivé zmíněné techniky se pak od sebe odlišují způsobem, jak zobrazují hodnotu v jednom konkrétním bodě.

Technika *spot noise* [5] vykresluje skvrny (*spots*) náhodné intenzity na náhodně vybraná místa. Každá skvrna (původně kruhová) je však protažena ve směru hodnoty vektorového pole v příslušném bodě. Na obrázku, který vznikne vykreslením velkého množství takových skvrn, je pak zřetelně vidět jak globální struktura pole, tak i detaily, vzniklé např. turbulentním prouděním (viz obr. 5).

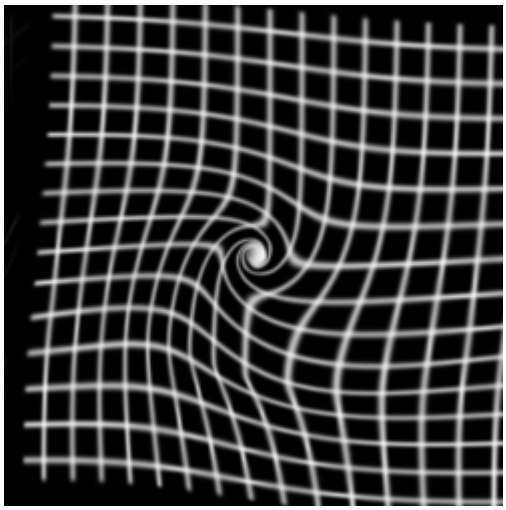
Teoretickým základem technik *LIC* (*Line Integral Convolution*) [6], *OLIC* (*Oriented Line Integral Convolution*) [7], *FROLIC* (*Fast Rendering of Oriented Line Integral Convolution*) [8] je filtrování textury s bílým šumem podél úseků proudnic vektorového pole. U techniky LIC se používá hustá šumová textura a symetrický filtr, u techniky OLIC řídká textura a asymetrický filtr, což má v praxi výsledek, který budí dojem kapek barviva uvolněných do vektorového pole (viz obr. 6), jejichž stopy vytvoří obrázek. Technika FROLIC z důvodu urychlení nepoužívá filtraci šumové textury, ale přímo vykresluje aproximaci stop pomyslných kapek barviva. Ačkoliv tím dochází k jisté ztrátě přesnosti, je tato poměrně zanedbatelná, ovšem výkonnostní nárůst je značný.

2.4.3 IBFV

Technika *IBFV* (*Image Based Flow Visualization*) [9] je relativně nový způsob vizualizace proudění, který pracuje poměrně odlišně od předešlých, i když vizuální výsledky mohou být podobné. Celý proces vizualizace probíhá v obrazovém prostoru, tzn. pracuje se přímo s výslednou bitovou mapou, která se v každém kroku transformuje podle hodnot vektorového pole a smíchá s filtrovaným bílým šumem. Tato technika se velice hodí pro použití na grafických akcelerátorech, neboť velmi dobře využívá schopnosti těchto akcelerátorů.



obrázek 7 – IBFV připomínající spot noise



obrázek 8 – Deformace mřížky v IBFV

Vizualizace probíhá v reálném čase a skládá se ze dvou základních kroků: nejprve se vezme obrázek z předchozího vizualizačního kroku, rozdělí se mřížkou, načež se každá buňka mřížky transformuje a deformuje podle hodnot, které vektorové pole má v jejích rozích a vykreslí se *přes* původní obrázek. Ve druhém kroku se s obrázkem smíchá (s využitím hardwarově podporované poloprůhlednosti) jeden obrázek

ze sekvence šumových textur, které byly při inicializaci vytvořeny. Výsledný obrázek se jednak vykreslí, jednak uschová pro použití v dalším kroku animace.

Další výhodou (kromě vhodnosti pro hardwarové urychlení) této techniky je možnost rozsáhlého nastavení. V procesu vizualizace se dají měnit mnohé parametry, které významně ovlivňují výsledek. Namátkou to jsou třeba koeficient průhlednosti šumové textury, velikost a faktor zmenšení šumové textury, atd. Vhodným nastavením těchto parametrů lze dosáhnout toho, že výsledek připomíná např. techniku spot noise (viz obr. 7), techniku OLIC, atd. Zásadním přínosem techniky IBFV oproti těmto technikám je ovšem rychlost, která vyplývá z už zmiňované vhodnosti pro dnešní grafický hardware.

Metoda IBFV má i další možnosti, jako snadnou rozšiřitelnost o vykreslování stop barviva, které je do proudění vloženo. Jelikož IBFV pracuje stále s touž bitovou mapou, kterou pouze postupně transformuje, stačí do této bitové mapy vykreslit libovolný objekt (ať už kapku barviva, nebo třeba pravidelnou čtvercovou mřížku – viz obr. 8) a pokračovat v animaci. Výsledkem je, že vidíme, jak se tento objekt mění v proudění, které vektorové pole popisuje, přičemž jsme nemuseli implementovat *žádné* další rozšíření algoritmu, tzn. ani požadavky na výkon se nijak nezměnily!

2.5 Potřeba akcelerace

Z předešlých částí je zřejmé, že vizualizace v reálném čase přináší velké výhody, ovšem vzhledem k tomu, že u mnoha vizualizačních metod se způsob zpracování dat liší od tradičního způsobu zpracování geometrie grafickým systémem, není často možné využít optimalizovaných funkcí dostupné grafické knihovny, neboť ta je navržena pouze pro obecnou 2D nebo 3D grafiku. To má za následek sníženou výkonnost, neboť veškeré vizualizační výpočty se musí provádět v aplikačním programu (běžícím na hlavním procesoru), který ovšem má na starosti i důležitější činnost, totiž provádět samotné modelování vizualizovaného procesu. Navíc, nejspíše z historických důvodů není využívání akcelerátorů ve vědeckotechnické vizualizaci zatím zcela běžnou záležitostí (viz např. [10]).

Je tedy zřejmé, že je záhodno využít schopností moderních grafických akcelerátorů, které svou flexibilitou umožňují využít dodatečný paralelismus dalšího procesoru i v úlohách, které nejsou podobné těm tradičním, pro které byly tyto schopnosti původně navrhovány (např. pro hry). Tomuto cíli se snaží pomoci i vizualizační knihovna řešená v rámci této práce.

3 Moderní grafické akcelerátory

V této kapitole se seznámíme s dřívějšími a současnými grafickými akcelerátory, s jejich architekturou a schopnostmi. Následující kapitola se pak bude věnovat programátorským rozhraním, pomocí kterých lze tyto schopnosti využít.

3.1 Minulost

Původním cílem grafického adaptéru (jelikož se v této práci věnuji platformě IBM PC, pominu zde obvykle neužívané technologie, jako např. vektorové displeje; také přeskočím z hlediska této práce zcela nezajímavé adaptéry schopné pracovat pouze v textovém režimu) bylo prostě grafiku alespoň nějakým způsobem zobrazit. Nikdo nevyžadoval, aby adaptér jakkoli pomáhal hlavnímu procesoru s vytvářením grafického obsahu a ani by to nebylo možné, protože grafická karta obvykle neobsahovala žádný procesor dostatečně univerzální pro takovou úlohu. Schopnosti takového adaptéru byly poměrně primitivní: ze své paměti číst tam připravené snímky, převést je do podoby vhodné pro displej a vyvést signály na vnější rozhraní. A při rozlišení kolem 320x200 při čtyřech barvách (CGA) nelze ani očekávat žádné pokročilejší grafické funkce. S příchodem VGA zařízení, která umožňovala až 256 barev, anebo rozlišení až 640x480 (ne však oboje zároveň), a zvláště poté s příchodem různých druhů tzv. Super VGA zařízení, která dokázala zpracovávat až 16 milionů barev při vysokých rozlišeních, začalo být zřejmé, že na běžném domácím PC bude možno používat detailní grafiku, což zaujalo kromě výrobců her i návrháře používající CAD programy.

Na počátku hardwarově akcelerované grafiky byli profesionální grafici, kteří si při práci v CAD systémech uvědomili potřebu co nejrychleji vykreslovat čáry, kružnice a další grafické prvky. Proto první grafické akcelerátory (nazývané 2D akcelerátory) byly vybaveny procesorem pro interpolaci čar a kružnic, která se využívala hlavně v 2D grafických aplikacích, jako jsou CAD programy, programy pro vektorové kreslení, atd.

Základní nevýhodou těchto zařízení (jako ostatně snad všech specializovaných zařízení v té době) byla nejednotnost. Pro akcelerátory každého konkrétního výrobce existovalo speciální rozhraní, takže programy musely podporovat každý akcelerátor zvlášť. V čase kolem příchodu Microsoft Windows 3.0 se mezi funkcemi akcelerátorů objevily i funkce na vyplňování polygonů, což se velice hodilo na urychlení práce s okénkovým systémem. To vedlo i k jistému zjednodušení rozhraní pro 2D akceleraci, protože operační systém Windows umožňoval abstrahovat od detailů konkrétního zařízení a pracovat jenom s Windows API, přičemž výrobcem dodaný ovladač zařídil zbytek.

Jenže rozmach 2D akcelerátorů byl velice krátkodobý. Jakkoliv je trh s CAD systémy velký, nemůže se ani v nejmenším rovnat trhu počítačových her. A byly to právě počítačové hry, které vedly k obrovskému rozšíření, zdokonalování a současně zlevňování grafických akcelerátorů. Na úplném počátku byly hry jako Wolfenstein 3D (přestože přízvisko „3D“ je poněkud neoprávněné) a Doom (obě vytvořené

společností Id Software, která měla a stále má na vývoj grafických akcelerátorů velký vliv), které, ač stále ještě vykreslované plně softwarově, ukázaly, že běžný počítač je schopen v reálném čase vykreslovat 3D grafiku v takové kvalitě, že je možno ji používat v počítačových hrách. Po krátké době se na trhu objevily rozšiřující adaptéry, které, po zasunutí do dalšího rozšiřujícího slotu PCI sběrnice a propojení s grafickou kartou (adaptér vůbec neobsahoval základní 2D funkce, takže k funkci potřeboval „běžnou" grafickou kartu, tato technika se slangově nazývala *piggybacking*), umožňovaly prostřednictvím speciálního aplikačního rozhraní (nejvýznamnější z nich se nazývalo GLIDE, nejspíše pro svou podobnost s OpenGL) urychlovat vykreslování 3D grafiky.

Tyto akcelerátory, jejichž nejvýznamnějším výrobcem byla firma 3dfx, patřily k nutné výbavě počítačového hráče, protože většina 3D her podporovala právě jejich rozhraní a nárůst výkonu byl takový, že bez akcelerátoru bylo mnoho her buď přímo nepoužitelných, nebo přinejmenším hratelných pouze s výrazně sníženou kvalitou zobrazování. Úplně zpočátku byly ovšem jejich schopnosti nízké (takže první opravdu 3D hra, Quake, byla často rychlejší se softwarovým vykreslováním, než s využitím akcelerátoru první generace), ovšem s postupem času do nich bylo integrováno čím dál více funkcí využívaných ve hrách. Jenže, když začalo speciálních efektů přibývat, bylo jasné, že se speciální funkcí pro každý efekt by bylo rozhraní zanedlouho nepoužitelné. Proto došlo k dalšímu zásadnímu obratu ve vývoji akcelerátorů – začaly být programovatelné.

3.2 Současnost

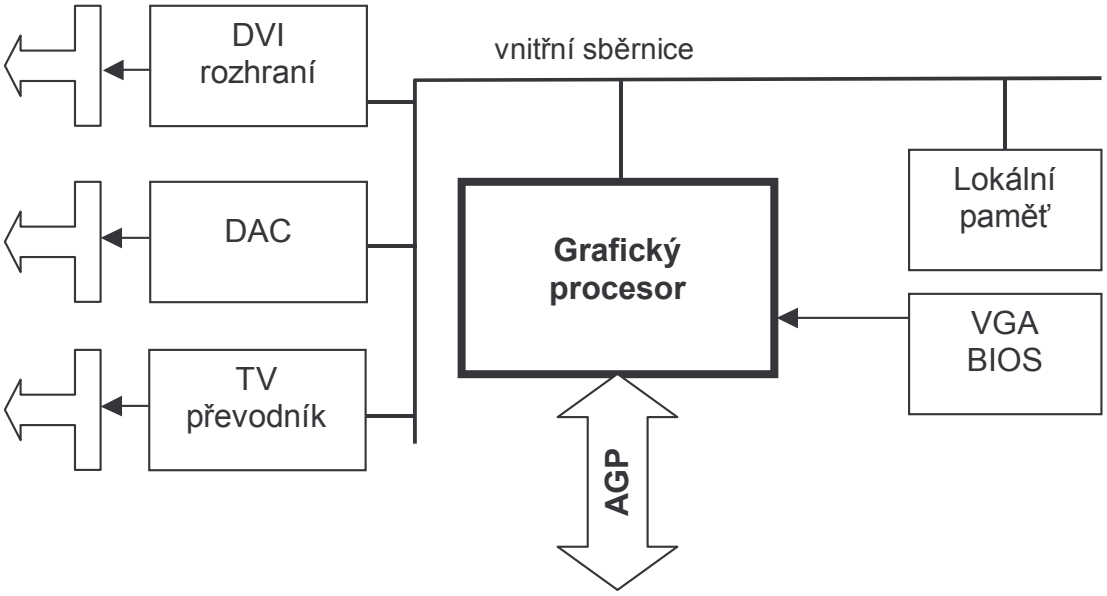
Základním rysem, který moderní grafické adaptéry odlišuje od předešlých generací, je jejich programovatelnost. Jednou z hlavních součástí dnešního akcelerátoru je programovatelný procesor (tzv. grafický procesor, *Graphics Processing Unit, GPU*), který zpracovává veškerou geometrii, provádí rasterizaci, mapování textur, výpočty osvětlení, atd. V předešlých generacích byly všechny tyto úkony vykonávány pevně stanoveným způsobem, na kterém mohl programátor měnit pouze hodnoty parametrů (transformační matice, parametry osvětlení, atd.). Dnes není prakticky žádná fáze zobrazování prováděna pevně daným postupem, ale naopak, programátor může nahradit standardní zpracování vlastním programem, který s daty může provádět prakticky libovolné výpočty. Klasický způsob zobrazování je pak dosažen tím, že ovladač grafického adaptéru je schopen dodat pro každou fázi zpracování takový program, který provádí všechny úkony původním způsobem, takže aplikace, které neumí, nebo nechtějí nové možnosti využít, nemusí svoje chování měnit.

3.3 Architektura grafických akcelerátorů

Základní blokové schéma prvků akcelerátoru je na následujícím obrázku. Na něm je vidět, že nejdůležitější součástkou je programovatelný grafický procesor, kromě kterého už akcelerátor v podstatě obsahuje jen rozhraní pro různá výstupní zařízení (klasický analogový monitor, digitální monitor, např. s rozhraním DVI, a také dnes velice často nabízený výstup na běžnou televizi, nejčastěji pomocí rozhraní S-Video), lokální videopaměť a také paměť (typicky typu ROM, popř. některé programovatelné varianty) s VGA BIOSem.

3.3.1 Lokální videopaměť

Lokální videopaměť má na dnešním akcelerátoru velikost až několik stovek megabajtů (typicky 128 MB, špičkové akcelerátory více) a používají se pro ní specializované typy RAM pamětí, které umožňují několikanásobný současný přístup. Lokální videopaměť původně sloužila pro uchovávání obrazového rámce (*frame buffer*). K tomu stačí několik málo megabajtů paměti (např. cca 3 MB pro režim 1024x768x32,



obrázek 9 – Základní blokové schéma akcelerátoru

obvykle se používají nejméně dva takové rámce (*front* a *back*) pro odstranění trhání animací – tato technika se nazývá *double buffering*). S vývojem grafických adaptérů se rozšiřoval seznam toho, co se ještě do lokální paměti adaptéru ukládá. Jednou z prvních takových položek byla paměť hloubky (*depth buffer*, *z-buffer*), což je prakticky jediná technika (kromě triviálního zahazování zadních stěn – *backface culling*) řešení viditelnosti, která se v hardwaru provádí. Pro tuto paměť je potřeba zhruba stejné množství paměti jako pro obrazový rámec (v závislosti na přesnosti – 16-bitový z-buffer je naprosté minimum, dnes se obvykle používá 32-bitový). Součástí a rozšířením paměti hloubky někdy bývá paměť šablony (*stencil buffer*), typicky 8-bitová. Ovšem stále nemáme důvod, proč bychom měli adaptér vybavovat stovkami megabajtů paměti. Tím důvodem jsou další objekty, které můžeme na dnešních akcelerátorech uložit do lokální paměti. Nejdůležitější skupinou takových objektů jsou textury. Jelikož je akcelerátor zaměřen na 3D grafiku (s důrazem na počítačové hry) a dnešní 3D grafika se bez textur neobejde, obsahuje i jen průměrný program využívající trojrozměrné grafiky obrovské množství textur. Z důvodu velkého objemu těchto textur pak není vhodné, aby se data textur neustále přesunovala po systémové sběrnici, protože i rychlá AGP sběrnice by byla úzkým hrdlem. Proto se pokud možno všechny používané textury ukládají do lokální paměti adaptéru. Tím je možno obhájit prakticky libovolnou velikost paměti.

Přesto to ještě není vše. Později se objevily i další možnosti využití lokální paměti: pokud zobrazujeme nějaký statický (neměnný) model, nemusíme každý snímek posílat jeho geometrická data do adaptéru, protože stačí poslat je tam jednou, adaptér si je uloží do lokální paměti, odkud pak probíhá celé vykreslování bez další zátěže systémové sběrnice a hlavního procesoru. Taková geometrická data obsahují

kromě skutečné geometrie (tzn. souřadnice vrcholů, texturovací souřadnice, atd.) také indexovací data – pro zvýšení výkonnosti se totiž modely zadávají jako lineární seznam vrcholů, ze kterých se trojúhelníky staví pomocí indexů do tohoto seznamu. To umožňuje uchovávat transformované vrcholy v mezipaměti, což zvláště u některých způsobů zadávání trojúhelníků (nejvíce u indexovaných trojúhelníkových pásů, *triangle strips*) vede k rapidnímu nárůstu výkonu.

Na lokální paměť na kartě je teoreticky možno uložit i jiná data (teoreticky je možno využít ji třeba i k rozšíření hlavní paměti počítače, např. jako ramdisk), tato možnost se však příliš často nevyužívá.

3.3.2 AGP

Dnešní grafické akcelerátory jsou až na skutečné výjimky navrženy pro použití rozhraní AGP (AGP = *Accelerated Graphics Port*). Adaptéry pro sběrnici PCI se vyrábí pouze pro upgrade zastaralých počítačů nevybavených AGP, popř. pro systémy s více grafickými kartami, neboť systém nemůže být vybaven více než jednou grafickou kartou s rozhraním AGP. Technologie AGP nenahrazuje PCI, v systému stále existuje sběrnice PCI pro všeobecné použití, rozhraní AGP je dodatečným vysokorychlostním kanálem specializovaný pouze pro grafický adaptér, mezi jehož výhody patří hlavně následující:

- **Vyšší rychlost (propustnost)** – přenosová rychlost technologie AGP 4x je až 1 GB za sekundu (rychlost PCI je max. 136 MB). Navíc je rozhraní AGP odděleno od sběrnice PCI, takže při AGP přenosech je sběrnice PCI volná pro použití ostatními zařízeními (a naopak, takže přenos dat čtených z disku se navzájem s AGP transakcemi nebrzdí). Novější technologie AGP 8x disponuje ještě vyšší přenosovou rychlostí, ovšem existují pochyby, jestli taková rychlost má za dané situace smysl, neboť AGP sběrnice již v takovém případě není úzkým hrdlem, takže zdvojnásobení přenosové rychlosti se zatím neprojevuje stejným zvýšením výkonu celého grafického systému.
- **AGP Texturing** – je technologie, která AGP adaptéru umožňuje přímo pracovat s hlavní pamětí (zvláště s texturami uloženými v hlavní paměti) stejně snadno, jako by byly uloženy přímo v lokální paměti akcelerátoru. Jelikož není pro takové přenosy zapotřebí účast hlavního procesoru a také není nutné přenášet celou texturu, ale pouze požadované vzorky, je možný kompromis mezi cenou adaptéru a jeho rychlostí, kdy, přestože je adaptér vybaven menším množstvím lokální paměti, poskytuje relativně dostatečný výkon. Textury uložené v lokální paměti jsou stále výrazně výhodnější, protože přístup k nim je rychlejší, ale také proto, že uvolňuje přenosové pásmo AGP sběrnice pro další data, např. dynamickou geometrii, kterou je nutno přenášet každý snímek.

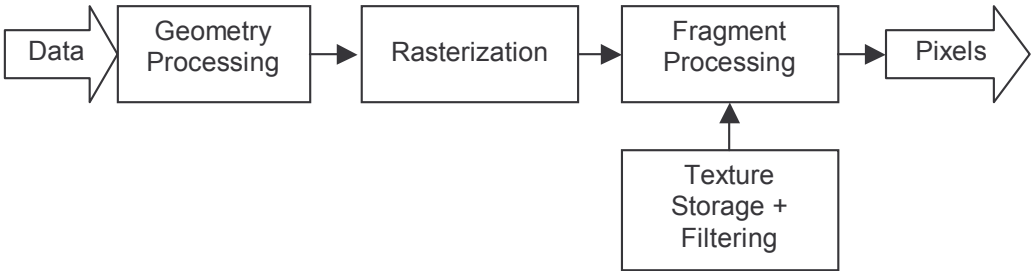
3.3.3 Grafický procesor

Grafický procesor je nejdůležitější částí grafického akcelerátoru. Na následujícím obrázku je schéma proudového zpracování, které probíhá celé na tomto procesoru (zleva přicházejí po AGP sběrnici data z hlavního procesoru, vpravo se výsledek ukládá do frame bufferu). V jednotlivých fázích se provádějí následující funkce:

- **Geometry processing** – zpracování geometrie, tzn. transformace geometrických dat podle dodaných transformačních a projekčních matic, osvětlovací výpočty, atd. Také zde probíhá ořezávání (*clipping*) do

zobrazovaného objemu. Tato fáze zpracovává jednotlivé vrcholy, tzn. zde běží vertex program (viz část 3.4.1).

- **Rasterization** – zde se jednotlivá primitiva (trojúhelníky) rasterizují, tzn. převádí se na fragmenty. Zde probíhá interpolace všech hodnot (barvy, texturovací souřadnice, atd.). Zde probíhá také odstraňování odvrácených stěn (*backface culling*).



obrázek 10 – Proudové zpracování v grafickém procesoru

- **Texture Storage + Filtering** – čte textury z příslušné paměti (systémová/lokální) a aplikuje na ně filtraci podle platného nastavení.
- **Fragment Processing** – zpracování fragmentů, tzn. aplikace textur, mlhy, výpočet výsledné barvy fragmentu, atd. V této fázi běží fragment program (viz část 3.4). Po této fázi se provádí operace jako hloubkový test (*depth test*), test alfahodnoty (*alpha test*), míchání barev (*blending*) atd., načež je výsledný fragment (pokud nebyl v průběhu této fáze zahozen), uložen do frame bufferu.

Jak vidno, hlavní procesor provádí velké množství operací. Detailní informace o jednotlivých činnostech jsou součástí následující kapitoly.

3.4 Schopnosti

Moderní grafické akcelerátory jsou schopny samostatně provádět velkou část zpracování trojrozměrné grafiky. Tato kapitola obsahuje informace o tom, jaké jsou nejdůležitější schopnosti takových akcelerátorů a jak se těchto schopností v principu dosahuje.

Na počátku zpracování máme aplikaci, která chce zobrazovat (obecně) trojrozměrná data. Tato data má ve formě oddělené geometrické a topologické informace, tzn. má seznam vrcholů s geometrickými daty, ke kterému má pole indexů do tohoto seznamu, které teprve udává, které vrcholy tvoří jednotlivé trojúhelníky. V průběhu zpracování se tato data transformují až do podoby pixelů, které na monitoru tvoří osvětlenou, texturovanou scénu, ve které se objevují i speciální efekty jako mlha. Toho všeho se dosahuje pomocí následujících schopností akcelerátoru:

3.4.1 Transformace a osvětlování

Anglická zkratka *HW TnL* (znamenající *Hardware Transform and Lighting*, někdy psáno *HW T&L*) popisuje, že daný akcelerátor je schopen sám, bez pomoci hlavního procesoru, provádět výpočty potřebné pro transformaci a projekci geometrických dat z původního souřadného systému do výsledného prostoru zařízení a provádět i všechny osvětlovací výpočty (samozřejmě pouze jednoduchý Phongův osvětlovací model, se

kterým se později při rasterizaci použije pouze konstantní nebo Gouraudovo stínování, pokročilejší způsoby jako např. Phongovo stínování nejsou přímo podporovány). Tato schopnost byla základní výhodou minulé generace akcelerátorů. Aktuální generace postoupila ještě o krok dál, umožňuje totiž tuto TnL jednotku programovat.

3.4.1.1 Klasické zpracování

Při běžném zpracování provádí TnL jednotka následující kroky:

- převod souřadného systému ze souřadnic modelu do světových souřadnic (*world transformation*)
- převod souřadného systému ze světových souřadnic do souřadnic relativních vůči kameře (*view transformation*)
- projekci, tzn. převod trojrozměrných souřadnic na dvojrozměrné pomocí násobení projekční maticí (*projection transformation*)
- výpočet osvětlení vrcholu (Phongův osvětlovací model)
- výpočet parametrů mlhy
- převod souřadnic do normalizovaných souřadnic zařízení podle aktuálně nastaveného okna (*viewport*)
- ořez do okna
- perspektivní dělení (*perspective divide*), tzn. převod z homogenních souřadnic [x,y,z,w] na nehomogenní [x/w, y/w, z/w], se kterými se dále (do rasterizéru) posílá ještě tzv. RHW hodnota (*reciprocal of homogenous W coordinate*) rovná 1/w, která se může využít v paměti hloubky

(V praxi jsou první tři transformační kroky provedeny najednou, pomocí násobení jednou složenou maticí.) Toto zpracování vyhovuje každému programu, který se snaží zobrazovat běžnou trojrozměrnou grafiku standardním způsobem (s tím, že některé kroky se nemusí v daném případě uplatnit, např. při vypnuté mlze je příslušný krok přeskočen). Pokud by ovšem program chtěl některý krok provést podstatně jiným způsobem, nemá u pevného zpracování jinou možnost, než *celý* proces transformace a osvětlení provést sám na hlavním procesoru, s patřičnými požadovanými úpravami, a již transformovanou a osvětlenou geometrii poslat k dalšímu zpracování. Je zřejmé, že tím přichází o veškeré výhody HW TnL

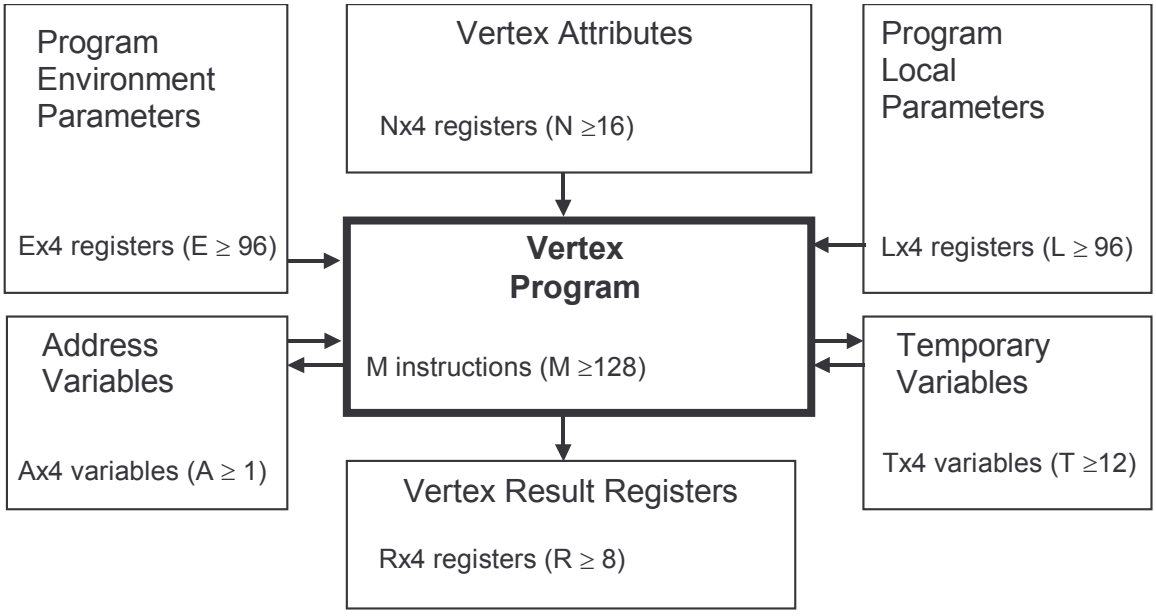
3.4.1.2 Vertex programy

Jak tento problém vyřešit lépe? V moderních akcelerátorech to možné je: místo toho, aby hardware pevným způsobem prováděl popsané operace, provádí obecný program. Tento program může popisovat přesně takovou činnost, na kterou jsme zvyklí z tradičního zpracování, ale může popisovat také něco velice odlišného. Takový program pak řídí transformaci každého vrcholu, proto se nazývá *vertex program* (někdy též *vertex shader*, což je původní terminologie DirectX, odkud tuto schopnost OpenGL převzalo). Pro každý vrchol předaný akcelerátoru je vyvolán vertex program, který může data vrcholu libovolným způsobem transformovat. Vertex program *nemůže* měnit počet vrcholů (ani vytvářet nové, ani zahazovat existující) a nedisponuje žádnými topologickými informacemi (neví např. s kolika/kterými trojúhelníky daný vrchol inciduje, ani jakého typu primitiva je vrchol částí, atd.).

Základní schéma činnosti vertex programu je na následujícím obrázku (u každé položky je uvedena minimální hodnota limitu požadovaná specifikací OpenGL rozšíření ARB_vertex_program, které vertex

programy do OpenGL zavádí). Vertex program konceptuálně běží na jednoduchém SIMD procesoru, u kterého každá instrukce pracuje najednou se čtveřicí hodnot v pohyblivé řádové čárce, které mohou reprezentovat homogenní vektor (x,y,z,w), barvu (r,g,b,a), nebo libovolné jiné údaje podle volby programátora. Tyto hodnoty se vyskytují v následujících typech registrů:

- **Vertex Attributes** – pomocí těchto registrů může program číst atributy, které program zadává u každého vrcholu. Patří sem tradiční atributy jako poloha, barva, normálový vektor, texturovací souřadnice, …, ale program sem může uložit i jiné, jako např. hustotu, rychlost, váhy, atd. Hodnoty těchto registrů jsou na začátku provádění vertex programu nastaveny vždy na hodnoty pro aktuální vrchol. Vertex program je může pouze číst.



obrázek 11 – Schéma činnosti vertex programu

- **Program Local Parameters** – tyto registry obsahují parametry společné pro mnoho vrcholů, kterými se typicky konfiguruje některá schopnost programu. Je možné sem uložit např. transformační matice, koeficienty pro výpočty mlhy, atd. Tyto registry nastavuje aplikace mimo glBegin()/glEnd(). Vertex program je může pouze číst.
- **Program Environment Parameters** – tyto registry mají stejnou funkci jako Program Local Parameters, rozdíl spočívá v tom, že parametry prostředí jsou sdíleny všemi vertex programy, lokální parametry se nastavují nezávisle pro každý program. Do této skupiny se může zařadit také aktuální stav OpenGL, neboť program může číst některé z jeho hodnot, např. barvu mlhy, nastavenou velikost bodu, transformační matice, koeficienty ořezových rovin, atd.
- **Address Variables** – speciální typ registrů (vyžadován je nejméně jeden), který implementuje jakousi obdobu ukazatelů. Narozdíl od všech ostatních registrů je zde využita jenom jedna složka čtyřprvkového vektoru, která obsahuje index do tabulky registrů. Pomocí tohoto typu je umožněno relativní adresování registrů.

- **Temporary Variables** – registry pro libovolné použití, např. pro uložení mezivýsledků, atd. Na začátku vertex programu mají nedefinované hodnoty, po dokončení vertex programu se jejich obsah zahazuje. Program je může číst i do nich zapisovat.
- **Vertex Result Registers** – registry, do kterých se ukládá výstup programu. V případě nepoužití fragment programu (viz dále) obsahují hodnoty pro klasické výsledky transformace a osvětlení: homogenní polohu, barvy (primární a sekundární), souřadnice mlhy, texturovací souřadnice, velikost bodu. Z těchto hodnot je povinná pouze poloha, ostatní závisí na nastavení (např. souřadnice mlhy jen tehdy, je-li mlha zapnuta, velikost bodu pouze při vykreslování bodů, atd.). Pokud je použit fragment program, mohou výstupní registry obsahovat i zcela jiná data, která poté slouží jako vstup fragment programu.

Každá instrukce programu operuje nad některými z těchto registrů, přičemž může zcela zdarma (aniž by to mělo vliv na dobu provádění instrukce) využít modifikační operátory pro přístup k registrům: *swizzling*, kdy se jednotlivé složky vektoru libovolně přeskupí, *masking*, kdy dochází k zápisu jenom do určitých složek vektoru, *negation*, kdy jsou všem zvoleným složkám otočeno znaménko.

Instrukční sada vertex programů obsahuje různé matematické funkce (jako např. sčítání, absolutní hodnotu, skalární součin, …), specializované funkce pro 3D grafiku (např. výpočet osvětlovacích koeficientů), instrukce pro porovnání vektorů i běžnou operaci přesunu (MOV). Pro ukázku vertex programu viz přílohu. Co u dnešních akcelerátorů zatím chybí nejvýrazněji je podmíněné zpracování – neexistují zde žádné podmíněné skoky (ostatně ani nepodmíněné), ani jiná možnost, jak přeskočit vykonávání některých instrukcí. Veškeré podmínky se musí rozepsat do matematických operací (pro nalezení maxima a minima existují přímo instrukce MAX, MIN; jiné podmínky se mohou zapsat pomocí instrukcí porovnání, jejichž výsledkem je číslo 1 nebo 0, kterým poté můžeme např. násobit příslušné hodnoty). Možnost provádět podmíněné zpracování se očekává v další generaci akcelerátorů.

Pokud je vertex program použit, je velká část operací z výše uvedeného seznamu přeskočena. Některé operace jsou ovšem prováděny i tak (a nelze je vertex programem přímo ovlivnit). Jsou to: transformace a ořez do okna, perspektivní dělení, oříznutí barev do použitelného rozsahu (neboť vertex programu nic nebrání nastavit hodnotu barvy např. na (1234;–1000;1111;1), z čehož se v tomto kroku vyrobí (1;0;1;1), což už je smysluplná hodnota) a zbytek zpracování počínaje sestavením primitiv a rasterizací.

3.4.2 Texturování

Jakmile začaly akcelerátory být schopné provádět mapování textury přímo v hardwaru, stalo se to okamžitě standardem, který rapidně zvýšil výkonnost grafických programů využívajících této funkce. Jenže takové programy (mezi kterými měly významný vliv – opět – počítačové hry) zanedlouho vyžadovaly další speciální funkce v souvislosti s texturováním. Jednou z prvních takových technik bylo tzv. *multitexturování (multi-texturing*), tzn. aplikace několika textur na jeden povrch. Cílem této techniky je dosáhnout efektů (např. osvětlovacích), které by jinak byly velice náročné. (V podstatě poprvé se „veřejně“ objevilo multitexturování v počítačové hře Quake, ve které se touto technikou dosahovalo vynikajícího detailního osvětlení, přestože použitý model měl poměrně málo detailů. Technika multitexturování byla v té době relativně exotická a byla hardwarem podporována jen na špičkových grafických pracovních stanicích. Úspěch hry Quake a jejích

následovníků, kteří používali stejnou techniku, způsobil, že multitexturování teď patří k základním standardním vlastnostem běžného akcelerátoru.)

S využitím (multi-)texturování lze provádět i různé další efekty, jako např. tzv. *bumpmapping*, kdy se modulací normály dosahuje zdání nerovností povrchu, přestože původní model takové detaily neobsahuje (a je to vidět na obrysu předmětů). I hardwarová podpora pro bumpmapping byla začleněna do některých akcelerátorů. Jenže je zde opět zřejmý problém, že pokud se bude postupně začleňovat podpora pro jednotlivé efekty, zanedlouho může počet speciálně podporovaných efektů přerůst rozumnou mez. Proto se, se značným využitím inspirace vertex programy, vytvořila druhá programovatelná vrstva.

3.4.2.1 Fragment programy

Při zpracování následují po fázi rasterizace fáze aplikace textur, míchání barev a zápisu do framebufferu. Právě aplikaci textur (která ještě obsahuje i výpočet výsledné barvy s využitím zadané primární/sekundární barvy a barvy mlhy) je možno nahradit uživatelským programem. Tento program ovlivňuje výslednou barvu každého zpracovávaného fragmentu, proto se nazývá *fragment program*. Pro každý fragment, který vznikne jako výstup rasterizéru, je vyvolán program, jehož výstupem je barva, která je poté předána do konečných fází zpracování.

Princip činnosti fragment programů je prakticky totožný s činností vertex programů (např. stejný model činnosti, stejné typy registrů, atd.), pouze s několika důležitými odchylkami:

- fragment program *může* zahodit fragment (ani on však nemůže vytvořit nový)
- v současné verzi fragment programů neexistují adresové registry (a tedy ani nepřímá adresace)
- instrukční sada je prakticky shodná, až na nové instrukce pro čtení textury (viz níže) a instrukci KIL, pomocí které lze způsobit již zmíněné zahození fragmentu

Základním cílem fragment programu je správně aplikovat textury a další efekty (např. per-pixel osvětlení, pomocí kterého lze dosáhnout Phongova stínování). K tomu potřebuje umět číst textury. Proto jsou v instrukční sadě fragment programů instrukce, které podle dodané pozice na textuře a identifikátoru texturovacího objektu (který odkazuje na nastavenou texturu, filtrovací režim a další stavové proměnné OpenGL související s texturováním) získají odpovídající hodnotu (obvykle barvu, ale textury lze využít jako zcela obecné tabulky konstant pro libovolné výpočty). Pomocí těchto instrukcí lze dosáhnout i tzv. nepřímého texturování, kdy je hodnota z jedné textury použita pro výpočet souřadnice v jiné textuře. (Je ovšem omezen počet úrovní takového nepřímého adresování.) V druhé příloze je ukázka jednoduchého fragment programu.

Vertex a fragment programy jsou v principu nezávislé. Aplikace může použít standardní způsob zpracování, jenom vertex programy, jenom fragment programy, nebo oba. V případě, že použije oba druhy programů zároveň, je možno dosáhnout velice komplexních efektů (např. existuje ukázkový program, který implementuje Conwayovu hru Life, která ovšem celá běží výhradně na grafickém procesoru). K tomu lze využít fakt, že hodnoty, které jsou výstupem vertex programu, jsou v podstatě též vstupem fragment programu a není na ně kladeno žádné významné omezení, takže si mohou předávat téměř libovolná potřebná data a tím rozdělit práci mezi vertex a fragment program.

4 Rozhraní pro grafické akcelerátory

V minulé kapitole jsme se seznámili s architekturou a schopnostmi akcelerátorů, tato kapitola se bude věnovat způsobům, jak těchto schopností můžeme využít.

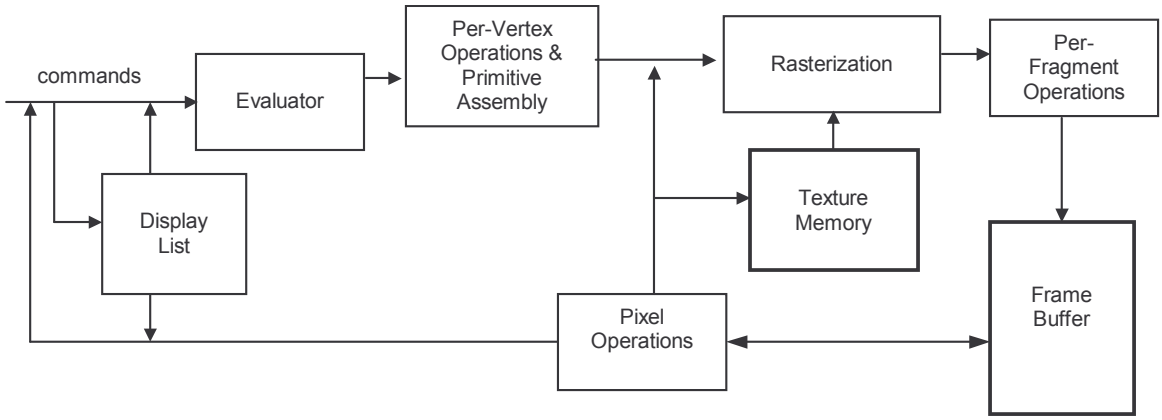
4.1 Úvod

Jak už bylo v minulé kapitole zmíněno, zpočátku existovaly speciální způsoby práce s akcelerátory každého výrobce, takže každý program musel používat specializované knihovny pro to rozhraní, které bylo v tom konkrétním počítači nainstalováno. To se naštěstí změnilo a existuje několik způsobů, jak detaily nejnižších vrstev přístupu k hardware přenechat ovladačům operačního systému. Asi nejvýznamnějším dnešním rozhraním, které nám toto umožňuje, je OpenGL.

4.2 OpenGL 1.x

OpenGL je dnes nejpoužívanějším standardním rozhraním pro práci s grafikou. Jeho implementace existují na rozsáhlé škále platforem, od průměrného PC po výkonné pracovní stanice SGI. Pokud je u nějakého osobního počítače podporována 3D grafika, je takřka jisté, že daný počítač podporuje alespoň základní funkce OpenGL.

Původní standard OpenGL (tzv. OpenGL 1.0) byl vydán 1. července 1992. Od té doby se schopnosti OpenGL rozšiřovaly, vždy nejdříve pomocí tzv. rozšíření (extenzí, angl. *extensions*), kterým se věnuje kapitola 4.3, některá úspěšná rozšíření se posléze stala součástí základního standardu OpenGL v jedné z jeho vyšších verzí (poslední vydaná verze standardu je verze 1.5 z 29. července 2003 [11], o očekávané verzi 2.0 pojednává kapitola 4.4).



obrázek 12 – Logické schéma činnosti OpenGL 1.x

4.2.1 Schopnosti OpenGL

OpenGL je schopno zobrazovat 2D a 3D grafická primitiva (body, čáry, polygony), aplikovat na ně textury (1D, 2D, 3D, *cube map* textury), podporuje speciální efekty jako např. mlhu, atd. Veškeré tyto funkce jsou poměrně nízkoúrovňové; OpenGL je procedurální systém, kdy programátor musí určit nejen *co* se má vykreslovat, ale i *jak* se to má vykreslovat. Celý návrh rozhraní OpenGL má za cíl vytvořit univerzální rozhraní, které neobsahuje žádné funkce specializované pro nějakou aplikační oblast (neobsahuje např. specializované funkce pro nějakou konkrétní problematiku CAD systémů). Pokud programátor vytváří program využívající OpenGL pro nějakou specializovanou aplikační oblast, může využít některých existujících rozšiřujících knihoven (odkazy na nejvýznamnější takové knihovny je možno nalézt na internetových stránkách OpenGL konsorcia na adrese http://www.opengl.org/).

Tato diplomová práce si klade za cíl vytvořit takovou knihovnu pro některé důležité funkce využívané ve vizualizaci vědeckotechnických dat, přičemž se snaží využívat schopností moderních grafických adaptérů, které jsou často opomíjeny.

4.3 Rozšíření OpenGL

Jelikož OpenGL je standardizované rozhraní (ovšem nejedná se o ISO standard, jako např. u systému PHIGS/PHIGS-PLUS), na jehož specifikaci se usnáší oficiální komise (tzv. ARB, *Architecture Review Board*), trvá poměrně dlouho, než jsou nové schopnosti dostupného hardware zabudovány do OpenGL. Pokud by se ovšem s každou novinkou muselo čekat na novou verzi OpenGL standardu, bylo by to neúnosné. Proto v OpenGL existuje mechanismus, kterým může dodavatel OpenGL implementace rozšířit funkčnost OpenGL o dodatečné funkce. Tyto funkce nejsou specifikovány přímo v OpenGL standardu, ale v jakýchsi dodatcích nazývaných extenze. Pokud programátor aplikace zná a podporuje konkrétní extenzi, může jejích služeb využít, pokud je extenze na příslušném počítači podporována (jak hardwarem, tak i jeho ovladači). Nevýhodou extenzí je evidentní fakt, že se tím ztrácí jednotnost standardu, což je jasný krok zpět. Přesto, jelikož doba mezi revizemi OpenGL standardu je dlouhá (a podpora nových verzí je velice špatná), jsou extenze jediným použitelným způsobem, jak v OpenGL využít novějších schopností moderního grafického hardware a prakticky se bez jejich využití nelze obejít.

Když chce program použít funkce nějaké extenze, musí nejprve zjistit, jestli je na daném počítači podporována, a pouze v případě, že je, ji může začít využívat. Pokud požadované rozšíření podporováno není, může program zkusit využít nějaké jiné, které třeba podporuje podobné funkce, nebo musí vystačit se standardními funkcemi OpenGL (anebo se odmítne spustit). Pro zjišťování podpory extenzí má každá extenze svůj jednoznačný název a součástí základních funkcí OpenGL je funkce pro zjištění seznamu názvů podporovaných extenzí.

Extenze existují v několika formách: první formou, což je forma, ve které se obvykle objevují zcela nové extenze, je rozšíření specifické pro jednoho dodavatele (*vendor-specific extension*). Je to rozšíření, které najdete nejspíše jenom v počítačích vybavených hardwarem příslušného dodavatele. Název takové extenze začíná identifikací tohoto dodavatele (NV = nVidia, ATI, APPLE, …). Druhá skupina extenzí jsou taková rozšíření, které několik výrobců shledalo jako užitečné, takže se shodli na jejich podpoře (tzv. *multivendor extension*). V názvu těchto extenzí je místo identifikace dodavatele zkratka EXT. Oproti předchozí skupině

jsou tato rozšíření většinou podporována ve vyšší míře. Třetí skupinou jsou rozšíření, která byla v podstatě shledána jako vhodná pro budoucí začlenění do standardu OpenGL. Jelikož se ovšem standardy vydávají sporadicky, v mezidobí je nová funkcionalita vydána jako rozšíření, na kterém se usnesla celá komise ARB. Název takových extenzí začíná zkratkou ARB. Po vydání ARB extenze (a zapracování extenze do nových verzí ovladačů) je velice pravděpodobné, že každý hardware, který je schopen příslušné funkce zvládnout, tuto extenzi podporuje.

Jakkoliv mají extenze obecně kratší dobu schvalování, stále jsou součástí striktně standardizovaného OpenGL, takže i ony mají svoje specifikace, které vytváří a zveřejňuje příslušný tvůrce (tzn. výrobce, skupina výrobců, nebo komise ARB). Tyto specifikace mají podobu doplňků k základnímu standardu OpenGL, který pozměňují a doplňují. Každý vývojář se může seznámit se specifikací libovolné extenze ve volně dostupném registru OpenGL extenzí (*OpenGL Extension Registry*) [12].

4.3.1 Extenze použité v knihovně Visla

V této kapitole se stručně zmíním o extenzích použitých v knihovně Visla. U každé extenze je uvedeno, do jaké míry je pro použití knihovny vyžadována.

4.3.1.1 ARB_vertex_program, NV_vertex_program

Tyto extenze slouží k rozšíření funkcionality OpenGL o použití vertex programů. Extenze umožňují nahrát program zapsaný ve speciálním nízkoúrovňovém jazyce, povolovat/zakazovat jeho používání a předávat mu parametry v mnoha formách.

Knihovna Visla vyžaduje, aby alespoň jedna z obou extenzí byla na hostitelském systému podporována, jinak se odmítne spustit.

4.3.1.2 ARB_vertex_buffer_object, NV_vertex_array_range

Od verze 1.1 umožňuje OpenGL zadávat geometrická data pomocí takzvaných *vertex arrays*, což jsou pole souřadnic vrcholů, texturovacích souřadnic, barevných hodnot a dalších atributů, která jsou předána takto najednou (na rozdíl od běžného zadávání po jednom pomocí volání funkcí glVertex, glTexCoord, atd.). Výhodou používání těchto polí je zvýšení rychlosti zpracování, neboť odpadá režie volání mnoha funkcí (původně muselo být pro každý vrchol zavoláno několik funkcí, nyní stačí konstantní množství volání pro libovolně složitou geometrii).

Jenže veškerá data jsou stále uložena na libovolném místě systémové RAM paměti (jehož umístění závisí na libovůli aplikačního programu), takže při zpracování je potřeba je posílat po systémové sběrnici, což dostupný výkon omezuje. Také nelze aplikovat příliš optimalizací, protože model chování standardních vertex arrays je velice volný – program pouze označí, kde každé pole začíná. To znamená, že OpenGL nemůže vědět, že data nebyla změněna, ani neví, jak velká část paměti se bude používat, takže nemůže data např. ukládat v nějaké rychlejší mezipaměti přímo na grafickém adaptéru, ani nesmí předpokládat, že program data nepoškodí ihned po příslušném volání glEnd, takže je musí zpracovat okamžitě a zdržet vykonávání programu do té doby, než všechna data přečte.

Pro odstranění těchto problémů existuje několik extenzí, z nichž Visla podporuje ARB_vertex_buffer_object a NV_vertex_array_range (k jehož využití ještě Visla vyžaduje přítomnost extenze NV_fence, která poskytuje dodatečné funkce pro řízení přístupu k poli). Konkrétní způsob řešení zmiňovaných nedostatků se v obou extenzích mírně liší, ale podstatou je striktnější model přístupu k polím: je třeba přesně specifikovat jejich velikost a není možné je libovolně měnit (každá změna musí být oznámena OpenGL). To umožňuje ovladači umístit toto pole do rychlé paměti, případně optimalizovat práci s ním.

Pokud ani jedna z těchto extenzí není systémem podporována, Visla bude pro vertex arrays používat standardní funkce OpenGL 1.1, což ovšem může vést ke snížené výkonnosti.

4.3.1.3 ARB_fragment_program, NV_fragment_program

Tyto extenze slouží k rozšíření funkcionality OpenGL o použití fragment programů. Extenze umožňují nahrát program zapsaný ve speciálním nízkoúrovňovém jazyce, povolovat/zakazovat jeho používání a předávat mu parametry v mnoha formách.

Pokud systém nepodporuje ani jednu z obou extenzí, nelze v programu využívajícím knihovnu Visla používat žádné funkce založené na fragment programech, zbylé funkce fungují bez omezení.

4.3.1.4 ARB_pbuffer

Tato extenze se oproti ostatním zde zmiňovaným odlišuje (striktně vzato do této kapitoly ani nepatří), neboť se nejedná o OpenGL extenzi, ale o WGL extenzi (tzn. celý název nezní GL_ARB_pbuffer, ale WGL_ARB_pbuffer). WGL je nižší vrstva pro podporu OpenGL na platformě Microsoft Windows, která zajišťuje základní funkce jako např. vytvoření OpenGL kontextu. (Protějškem WGL na platformě X Window System je rozhraní GLX, které obvykle podporuje obdobné extenze.)

Cílem tohoto rozšíření je umožnit vytvoření pomocného bufferu (*p-buffer = pixel buffer*), do kterého je možno vykreslovat, ale výsledky vykreslování nejsou automaticky vidět. Použití p-bufferů umožňuje vykreslování do textury, neboť po vykreslování do bufferu je možno pomocí funkcí glCopyTexImage obsah bufferu použít jako texturu, kterou lze poté aplikovat v běžném vykreslování.

Podpora extenze ARB_pbuffer (a jí vyžadovaných extenzí ARB_extensions_string a ARB_pixel_format) je vyžadována při použití knihovních funkcí pro IBFV.

4.3.1.5 NV_texture_rectangle

Toto rozšíření zavedené firmou nVidia umožňuje použití textur, které nemají rozměry omezené na mocniny dvou, což je obvyklý požadavek na textury v OpenGL. Pokud je podporována tato extenze, je možno použít nový texturovací objekt TEXTURE_RECTANGLE_NV, jehož textury jsou adresovány nenormalizovanými souřadnicemi v pixelech (tzn. nikoliv normalizovanými souřadnicemi v rozmezí ‹0;1›), přičemž rozměry textur tohoto objektu nejsou omezeny na mocniny dvou.

Existuje také ARB extenze s obdobnou funkčností (ARB_texture_non_power_of_two), se kterou ovšem Visla dosud nedokáže pracovat, neboť jsem neměl k dispozici žádný systém, kde by již byla podporována. V budoucnosti by bylo vhodné podporu této extenze doplnit, neboť se dá očekávat, že bude fungovat na systémech vybavených hardwarem od většího množství výrobců.

Pokud tato extenze není podporována, je při IBFV vizualizaci možno použít pouze p-buffer o rozměrech mocnin dvou, neboť obsah p-bufferu se používá jako textura.

4.3.1.6 ARB_point_sprite, NV_point_sprite

OpenGL umožňuje vykreslování bodů, u kterého ještě dovoluje nastavit několik parametrů jako velikost bodu a použitý způsob vykreslování (v nejjednodušší verzi se vykreslují jako malé čtverce, dále je možno čtvercům zaoblit rohy, nebo konečně vykreslovat dokonale kruhové body). U každého bodu je možno nastavit jeho barvu. Co ovšem základní OpenGL neumožňuje, je vykreslit místo bodu celou texturu. Tato technika, nazývaná *point sprite*, je velice používaná např. u částicových systémů, kdy se každá částice vykreslí s pomocí textury jako drobná osvětlená koule. Samozřejmě není možné použít skutečný model koule, neboť vykreslujeme mnoho tisíc částic, ale pokud máme texturu s předrenderovanou osvětlenou koulí, kterou použijeme, je vizuální dojem vynikající a výkonnost zcela dostatečná. Bohužel, jediný způsob, jakým OpenGL umožňuje vykreslit texturu, je pomocí texturovaného čtyřúhelníku (nebo, zcela ekvivalentně, dvou trojúhelníků). To ovšem vede k potřebě přenášet a transformovat čtyřnásobné množství geometrie (musíme pro každý bod zadat čtyři vrcholy, třebaže by stačilo specifikovat jeden, neboť souřadnice vrcholů čtyřúhelníku získáme pouze posunutím již transformovaných souřadnic středu o předem dané rozměry).

Pro umožnění využití techniky point sprites existují zmiňované extenze. Extenze umožňuje nahradit každý bod celou texturou o velikosti dané aktuální velikostí bodu (nastavené pomocí glPointSize).

Knihovna Visla využívá této extenze při jednom z režimů vykreslování částic. Pokud extenze není podporována, knihovna vykresluje částice pomocí čtyřúhelníků, ovšem pro všechny vrcholy každého čtyřúhelníka předává stejné souřadnice (středu bodu), přepočet těchto souřadnic se provádí pomocí vertex programu, což alespoň částečně odlehčuje zátěž procesoru. Je zřejmé, že chybějící podpora extenze vede ke snížení výkonnosti vykreslování částic.

4.4 OpenGL 2.0

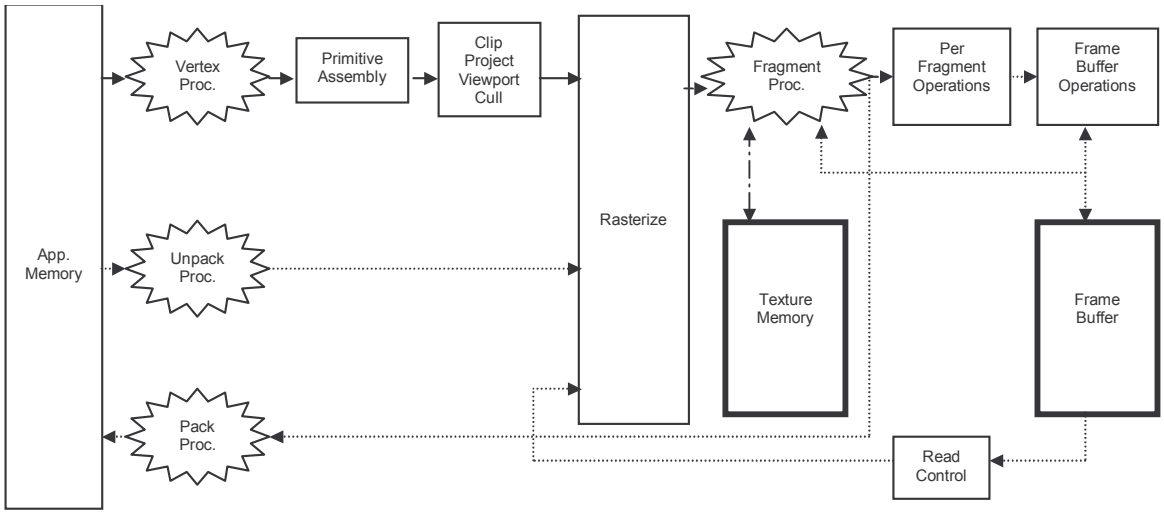
Jak už bylo řečeno, původní specifikace OpenGL pochází z roku 1992. Od té doby došlo k dalekosáhlým změnám v architektuře používaných grafických adaptérů i architektuře počítače jako celku. Extenze a nové verze OpenGL sice přidávají podporu pro některé nové funkce, ovšem základní struktura rozhraní je stále stejná. Proto existuje úsilí o vytvoření zcela nové verze OpenGL [13], která by měla OpenGL umožnit co nejlépe využívat schopností dnešního a budoucího grafického hardware.

Základní cíle nové verze jsou:

- Zahrnout schopnosti grafického hardware.
- Ovlivňovat směr vývoje dalších generací programovatelných grafických akcelerátorů.
- Omezit potřebu OpenGL extenzí pomocí široké programovatelnosti.
- Umožnit existujícím aplikacím pozvolný přechod na novou verzi OpenGL díky zachování zpětné kompatibility.
- Ale na druhou stranu umožnit aplikacím, které zpětnou kompatibilitu nevyžadují, co nejjednodušší vývoj aplikací s úplným využitím čistého OpenGL 2.0 (*Pure OpenGL 2.0)*.
- Zahrnout problematiku multimediálních aplikací související s projektem OpenML 1.0.

Nová verze OpenGL je silně zaměřena na využití programovatelnosti moderních grafických adaptérů. Jeho logické schéma obsahuje čtyři programovatelné jednotky (viz obrázek), jejichž činnost je možno upravit pomocí vysokoúrovňového jazyka, podobného programovacímu jazyku C.

Pokud se OpenGL 2.0 podaří prosadit, bude se jistě jednat o značný krok kupředu, který umožní programům pro OpenGL skutečně využívat moderních grafických adaptérů.



obrázek 13 – Logické schéma činnosti OpenGL 2.0 (hvězda označuje programovatelný procesor)

4.5 Jiná rozhraní

Prakticky jediným dnešním dostatečně veřejným (některé firmy používají speciální soukromá rozhraní, jenže ta nepředstavují alternativu pro běžného vývojáře) konkurentem rozhraní OpenGL je rozhraní DirectX firmy Microsoft (přesněji řečeno, jedna z jeho částí, tzv. DirectX Graphics).

Základním rozdílem je jeho proprietárnost a omezenost na platformu Microsoft Windows (i když existují jisté projekty, jak zprostředkovat použití rozhraní DirectX alespoň v prostředí Linuxu, zvláště pro programy spouštěné pod systémem WINE; tyto projekty však dosud nejsou příliš úspěšné). Proto vývojáři požadující přenositelnost svých programů nemohou DirectX použít. Na druhou stranu má tato proprietárnost tu výhodu, že zde neexistuje jistá zkostnatělost OpenGL a novinky se v DirectX objevují velice rychle (někdy až příliš rychle, takže se často stává, že novinky nějaké verze se začnou skutečně využívat až ve chvíli, kdy je vydána verze ještě novější), jelikož odpadá dlouhý schvalovací proces. I když v jistých situacích se může i jistá modularita OpenGL extenzí brát jako přednost, protože pro využití některé novinky není třeba zcela přejít na novou verzi, stačí jen využít příslušné rozšíření, což v DirectX není možné. Jako zajímavost je možno podotknout, že rozhraní DirectX Graphics se používá v herní konzoli X-Box (která je ovšem opět založena na platformě Windows/Intel).

Ze starších rozhraní jmenujme systémy GKS a PHIGS, které, jelikož jsou schváleným ISO standardem, jsou využívány např. v některých starších projektech financovaných vládou USA nebo velkými

firmami. V nových projektech se však prakticky nepoužívají, neboť nemají proti OpenGL žádné důležité výhody.

Vzhledem k tomu, že OpenGL je průmyslovým standardem a je podporováno na mnoha platformách, jsem ho zvolil jako rozhraní, pomocí kterého budu vizualizační knihovnu implementovat.

5 Vizualizační knihovna Visla

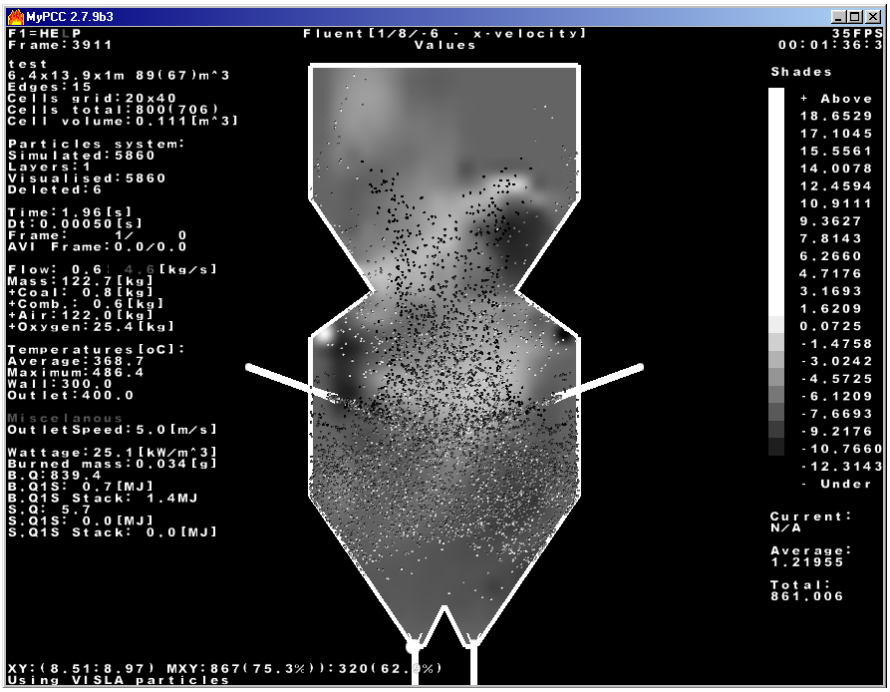
5.1 Zadání diplomové práce

Původním zadáním diplomové práce bylo implementovat vhodné části vizualizačního systému s využitím funkcí dostupných na nejnovějších grafických akcelerátorech. V okamžiku vytvoření tohoto zadání nebylo známo, že vytvořená vizualizační knihovna bude ihned zapojena do existujícího projektu obsahujícího vizualizační část (již zmiňovaný projekt MyPCC).

Kvůli zaměření k tomuto projektu leží výsledné těžiště práce (a podle mého názoru i přínos práce) hlavně ve zvýšení kvality vizualizace prováděné systémem MyPCC a v podpoře nových vizualizačních technik.

5.2 Motivace řešení

Tato diplomová práce se snaží vylepšit výsledky vizualizace v projektu MyPCC (My Pulverized Coal



obrázek 14 – základní obrazovka programu MyPCC

Combustion), který tvoří součást řešení doktorandské práce ing. Gayera [1]. V této části uvedu charakteristiku projektu a požadavky, které systém klade na metody vizualizace.

5.2.1 Základní charakteristika MyPCC

MyPCC je interaktivní systém pro simulaci a vizualizaci spalovacích procesů v průmyslových práškových kotlích. Simulační jádro pracuje v reálném čase díky použití fluidního simulátoru a systému virtuálních částic (které nereprezentují skutečné částečky uhlí, ale pouze reprezentují odpovídající množství uhlí v dané části prostoru). Simulací procesů v kotli program vypočítává a případně zobrazuje mnoho parametrů modelu, jako např. rychlost proudění, teplotu, množství kyslíku, atd. Program obsahuje z vizualizačního hlediska dvě základní části – vizualizaci částic a vizualizaci sledovaných parametrů modelu.

5.2.2 Vizualizace částic

MyPCC zobrazuje virtuální částice pomocí jednoduchého částicového systému, ve kterém každá zobrazovaná částice svou barvou indikuje hodnotu s částicí asociovanou, např. podíl spalitelné hmoty, průměr, hmotnost, atd.

5.2.3 Vizualizace mřížky dat

Zřejmě nejdůležitější vizualizací prováděnou programem MyPCC je vizualizace hodnot sledovaných programem v objemu celého kotle, v pravidelné mřížce. Tyto hodnoty (jako např. teplota, tlak, podíl kyslíku, velikost rychlosti proudění, atd.) jsou vizualizovány barevným pozadím vykresleným pod částicovým systémem. K tomu účelu sleduje program aktuální minimální a maximální hodnotu, kterou použije pro krajní body barevné palety. Je možné přepínat mezi několika přednastavenými barevnými paletami (světelné spektrum, úrovně šedé, atd.). Je možné též zobrazit původní mřížku, což umožňuje zvýraznit rozlišení dat.

5.2.4 Vizualizace rychlosti proudění

Rychlost proudění v kotli je možno vizualizovat buď jednoduše pomocí předešlé techniky, kdy se pomocí mřížky dat vizualizuje přímo velikost rychlosti nebo velikost jedné z jejích složek. Druhou možností je specializovaná vizualizace vektorového pole. Původně umožňoval program MyPCC pouze přímou vizualizaci pomocí krátkých úseček orientovaných podle vektorů rychlosti, vykreslovaných v bodech mřížky. Po zapojení knihovny Visla umožňuje i vizualizaci proudění pomocí metody IBFV (viz kapitola 2.4.3).

5.3 Stávající řešení

Původně prováděl program MyPCC veškerou vizualizaci s využitím pouze základních funkcí OpenGL (a knihovny MGL pro podpůrné funkce 2D grafiky, jako např. zobrazování textu). Program nevyužíval žádných pokročilejších schopností grafických akcelerátorů a výstupy vizualizace obsahovaly některé viditelné nedostatky.

5.3.1 Vizualizace částic

Pro vizualizaci částic bylo používáno základní vykreslování bodů z OpenGL, u kterého umožňovala nastavení kvality zobrazování mezi nejjednoduššími (čtvercovými) body, body se zaoblenými hranami a dokonale kruhovými body.

5.3.2 Vizualizace mřížky dat

Pro zobrazování mřížky dat bylo použito vykreslování čtyřúhelníků z OpenGL, přičemž k interpolaci hodnot uvnitř čtyřúhelníků bylo využito schopnosti OpenGL interpolovat barvu, takže u všech vrcholy byla nastavena barva, odpovídající hodnotě sledovaného parametru v příslušném bodu mřížky, a vyplnění čtyřúhelníku bylo ponecháno na OpenGL, které provedlo interpolaci v prostoru barev RGB.

5.4 Schopnosti knihovny

Knihovna Visla poskytuje aplikacím následující funkce:

- Vykreslování částicových systémů
- Vykreslování kartézské 2D mřížky bodů při použití lineární interpolace hodnot
- Vykreslování kartézské 2D mřížky bodů při použití interpolace hodnot pomocí spline křivek
- Interpolaci a vykreslování mezilehlých vrstev v poli kartézských mřížek, vše pomocí spline křivek
- Provádění a vykreslování vizualizace vektorového pole metodou IBFV

V následující kapitole jsou tyto schopnosti popsány podrobněji.

6 Implementace knihovny

6.1 Požadavky

Knihovna Visla pro své zkompilování vyžaduje následující:

- Kompilátor jazyka C++, vyhovující normě ANSI (včetně standardní knihovny STL)
- Programátorské rozhraní OpenGL nejméně verze 1.1, spolu s rozhraním pro všechny používané extenze
- Knihovna MGL [14], kterou program používá pro práci s texturami
- Kvůli použití knihovny MGL je vyžadována také knihovna GLUT

Zdrojové texty knihovny jsou komentovány za použití systému Doxygen, pomocí kterého lze ze zdrojových kódů automaticky vygenerovat programátorskou příručku ke knihovně.

6.2 Struktura

Funkce knihovny Visla můžeme rozdělit do několika logických celků podle toho, jakému účelu daná skupina funkcí slouží:

- **Všeobecně matematické funkce** – jsou obsaženy v třídě `TVector3`, kterou knihovna Visla používá pro práci s 3D vektory. Můžeme sem zařadit i soukromé třídy `TVector4` a `TMatrix4x4`, které jsou používány v implementaci knihovny.
- **Funkce pro správu a konfiguraci** – sem patří inicializační a konfigurační funkce, které nastavují parametry vizualizace, dále pak funkce, které zprostředkovávají základní správu související s požadavky OpenGL, nebo také speciální statická metoda `whatIsSupported()`, která poskytuje informace o schopnostech akcelerátoru na daném systému v souvislosti s požadavky a schopnostmi knihovny.
- **Funkce pro práci s texturami** – kromě funkcí pro nastavení používané textury (které můžeme zařadit i do předešlé skupiny) sem patří hlavně dvě funkce pro generování textury pomocí funkce pro převod hodnoty na barvu (jedna funkce generuje texturu pro vykreslování s izokřivkami, druhá bez nich). Kromě nich obsahuje knihovna i pomocnou třídu `ValueTextureCache`, která zjednodušuje proces generování textur a zajišťuje vysokou výkonnost díky uchovávání jednou vygenerovaných textur, takže nemusí být znovu vyrobeny při dalším použití.
- **Funkce pro vykreslování částic**
- **Funkce pro vykreslování základní mřížky bodů** – tyto funkce jsou určeny pro jednodušší vykreslování menšího množství dat, např. pro vytvoření náhledu, apod.
- **Funkce pro vykreslování mřížek bodů s interpolací pomocí spline křivek** – tyto funkce jsou určeny pro většinu vykreslování kartézských mřížek bodů
- **Funkce pro interpolaci a vykreslování sady mřížek**

- **Funkce pro vizualizaci pomocí IBFV** – vzhledem k tomu, že tyto funkce tvoří poměrně oddělenou část knihovny, a také proto, že pro použití vyžadují alokovat systémové zdroje, je tato skupina funkcí vyjmuta do samostatné třídy `IBFVHelper`.
- **Interní funkce pro práci s vertex programy, fragment programy, …**

6.3 Popis implementace

V této kapitole je popsán způsob implementace jednotlivých funkcí knihovny, popř. důvody, proč jsem zvolil takové řešení.

6.3.1 Základní informace

Celá knihovna je psána v C++, s plným využitím jeho objektových vlastností. Všechny deklarace knihovny jsou v prostoru jmen (*namespace*) nazvaném `Visla`, což předchází možnosti konfliktu identifikátorů. Knihovna neexportuje žádné globální funkce, všechny funkce jsou implementovány jako metody uzavřené do tříd. Nejdůležitější z těchto tříd je `Visualiser`, což je obalová třída, pomocí které se provádí prakticky veškerá vizualizační činnost knihovny. V případě chyby detekované některou částí knihovny je vygenerována výjimka, patřící do některé z tříd definovaných knihovnou.

Aplikační program musí při použití knihovny použít hlavičkový soubor `visla.h`, který obsahuje nebo importuje (pomocí dalších direktiv `#include`) deklarace celého rozhraní knihovny.

6.3.2 Všeobecně matematické funkce

Pro snadnou práci s vektory obsahuje Visla jednoduchou implementaci základních funkcí pro práci s 3D vektory, jako např. sčítání, skalární součin, atd. K tomuto účelu existuje třída `TVector3`, jejíž implementace je zcela přímočará.

Součástí vnitřních struktur implementační části knihovny jsou třídy `TVector4` a `TMatrix4x4`, které tvoří podporu pro zcela elementární operace se čtyřprvkovými vektory a maticemi rozměrů 4×4. Obě třídy jsou používány pouze uvnitř knihovny, nejsou součástí jejího rozhraní a nejsou exportovány.

6.3.3 Funkce pro správu a konfiguraci

Při použití knihovny je základním krokem vyrobit instanci třídy `Visualiser`, pomocí které bude poté probíhat veškerá další činnost. Jelikož se již v konstruktoru této třídy nahrávají některé použité vertex programy, může při jeho vykonávání dojít k výjimce (typu `not_supported_error`) v případě, že systém nepodporuje žádné známé rozšíření pro vertex programy. Proto je vhodné před voláním konstruktoru použít statickou funkci `Visualiser::whatIsSupported()`, která vrátí (formou bitové masky) seznam podporovaných vlastností. V případě, že podpora pro vertex programy není nalezena, nelze knihovnu Visla na daném systému použít.

Jako parametr se konstruktoru předává cesta, na které je možno nalézt vertex a fragment programy používané knihovnou. Visla všechny použité programy čte ze souborů, přestože by bylo možné je pevně zapsat do zdrojových textů. Důvodem této volby je snadná modifikovatelnost, která se hodí zvláště při

odlaďování problémů. V budoucnosti by bylo vhodné umožnit konfigurovatelnost, tzn. možnost předat knihovně programy přímo v paměti, čímž by se umožnila tvorba jednosouborových aplikací, což současná verze neumožňuje.

Konstruktor nastaví implicitní hodnoty všech parametrů, ovšem všechny hodnoty je poté možné změnit voláním metod `setXXX`.

6.3.4 Funkce pro práci s texturami

Knihovna Visla používá textury pro dva základní účely: jako point sprite při jednom z režimů vykreslování bodů (toto je malá textura, znázorňující částici, tzn. obvykle předrenderovaná stínovaná koule), nebo jako tabulku barev pro převod zobrazované hodnoty na barvu. První druh textury se obvykle získá pomocí nahrání připraveného externího souboru. To je možné i u druhého typu, ale tam je častější jiné řešení: jelikož je tato textura pouhý jeden řádek pixelů, je jednoduše možné ji vygenerovat programem. K tomu slouží dvě statické metody, které podle dodaných parametrů (jako velikost nebo druh texturovacího filtru OpenGL) využijí zadanou funkci k tomu, aby zaznamenaly, na jakou barvu se převedou různé hodnoty, přičemž jsou tyto barvy ukládány do textury, kterou je ihned poté možno použít. Obě funkce se liší v podpoře izokřivek. Pokud k vytvoření převodní textury použijeme funkci `genValueTexIso()`, bude každý n-tý texel nastaven na barvu izokřivky, což při použití takové textury způsobí, že na vykresleném obrázku se objeví soustava izokřivek.

Pro zjednodušení a zrychlení práce s texturami používanými pro převod hodnot na barvu je součástí knihovny Visla třída `ValueTextureCache`, která umožňuje programu jednoduše měnit používanou paletu, přičemž tato třída automaticky generuje textury odpovídající dané paletě a již vygenerované textury ukládá a znovupoužívá v případě, že je opět vyžadována dříve vytvořená textura.

6.3.5 Funkce pro vykreslování částic

Částice je možno vykreslovat několika způsoby: buď je možno přímým voláním funkcí `drawPoint()` a `drawPoints()` způsobit okamžité vykreslení zadaných bodů, nebo, což je obvyklejší alternativa, lze pomocí trojice funkcí `beginParticles()`, `particle()`, `endParticles()` zajistit vykreslení mnoha bodů zároveň, obdobným způsobem, jako pracuje OpenGL (podle paralely s `glBegin()`, `glVertex()`, `glEnd()`). Přesné chování obou způsobů je závislé na aktuálně nastaveném režimu vykreslování bodů, neboť body lze vykreslovat buď klasicky, jako v běžném OpenGL, nebo vykreslováním point sprites (viz též 4.3.1.6). Funkce pracují tak, že při volání `particle()` se pouze uloží data do připraveného vertex array a celé pole se vykreslí jedním voláním při `endParticles()`, což má za cíl zvýšení výkonu.

6.3.6 Funkce pro vykreslování základní mřížky bodů

Tyto funkce slouží pro méně přesné (ale často o něco rychlejší) vykreslování jednodušších dat. Pracují pouze s lineární interpolací, ale poskytují velice jednoduché rozhraní pro případy, kdy není potřebné dosáhnout vysoké kvality vizualizace a je pohodlnější např. zobrazovat mřížku po jedné buňce. Jelikož používají pouze

lineární interpolaci, mohou být o něco rychlejší, ovšem všechna data se vykreslují jednotlivě, takže výkonnostní nárůst je tím minimalizován.

6.3.7 Mřížky s interpolací pomocí spline křivek

Funkce, které slouží pro vykreslování kartézských mřížek dat s interpolací pomocí spline křivek jsou asi nejběžněji používanými funkcemi knihovny. Proto je zde popíši podrobněji:

Aplikace dodává knihovně data tím způsobem, že mezi voláními funkcí `beginGrid()` a `endGrid()` postupně volá funkci `gridPoint()` pro jednotlivé body mřížky, přičemž tato funkce pouze ukládá dodávané hodnoty do připraveného pole. Teprve po zavolání funkce `endGrid()` je vykreslena celá mřížka, a to tak, že pro každou buňku mřížky je předpočítána matice koeficientů spline křivek a s využitím této matice je buňka vykreslena.

6.3.7.1 Interpolace

Interpolace hodnot se, jak už bylo uvedeno, provádí s využitím spline křivek. Jelikož se nám jedná o interpolaci ve dvourozměrné mřížce, musíme místo přímého použití křivek použít plát (*patch*), definovaný těmito křivkami. Jako základ interpolace jsem zvolil Catmullovy-Romovy spline křivky [15], což jsou kubické křivky, které se ideálně hodí pro interpolaci hodnot zadaných čtyřmi okolními body (vyšší počet by byl nepraktický, protože by všechny hodnoty ovlivňovaly i příliš vzdálené body, což není vyžadováno a pouze by vedlo k vyšší výpočetní náročnosti). Navazováním těchto křivek takovým způsobem, že se sousední křivky shodují ve dvou řídících bodech, dosáhneme v řídích bodech zaručené spojitosti prvních derivací (na zbytku křivky jsou spojité i druhé derivace). Když z těchto křivek vytvoříme plochu, získáme tím bikubický plát, což je přesně ta metoda, kterou Visla používá pro interpolaci hodnot uvnitř mřížky. Catmullova-Romova křivka je definována následující rovnicí:

$$\mathbf{P}(t) = \begin{pmatrix} 1 & t & t^2 & t^3 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 0 \\ -0{,}5 & 0 & 0{,}5 & 0 \\ 1 & -2{,}5 & 2 & -0{,}5 \\ -0{,}5 & 1{,}5 & -1{,}5 & 0{,}5 \end{pmatrix} \begin{pmatrix} \mathbf{P_0} \\ \mathbf{P_1} \\ \mathbf{P_2} \\ \mathbf{P_3} \end{pmatrix}$$

kde $\mathbf{P}_i$ jsou kontrolní body křivky (jelikož se jedná o interpolační křivku, jsou to body, kterými křivka prochází v bodech, kde t je rovno postupně –1, 0, +1, +2). Při dosazení této rovnice za každý kontrolní bod získáme rovnici pro výpočet hodnoty plátu (tato rovnice je opět principiálně jednoduchá, ale je poměrně velká, takže ji zde neuvádím).

Knihovna Visla provádí takto definovanou interpolaci s využitím schopností moderních akcelerátorů – přímo na grafickém procesoru. Funguje to tak, že program pro každou buňku mřížky (pro kterou platí jedna rovnice, daná sítí 4x4 hodnot v okolních bodech) vypočítá podle předchozí rovnice koeficienty u mocnin složek proměnné **t** (která je zde již vektorem, neboť se interpoluje na dvojrozměrném plátu). Tyto koeficienty (tvořící 16 hodnot v matici 4x4) jsou, jako lokální parametry, předány fragment programu. Fragment program, vyvolaný pro každý pixel v buňce, vypočítá hodnotu, které tam mřížka dosahuje.

Právě popsaný postup je ovšem pouze jedna ze dvou možností, implementovaných knihovnou. Druhou možností je, provádět přesnou interpolaci pouze v několika bodech buňky a zbytek dopočítat lineární

interpolací, výsledkem čehož je vyšší rychlost. Výhodou této druhé metody je (kromě rychlosti) její škálovatelnost – můžeme si zvolit, jak často budeme vyhodnocovat přesnou hodnotu: čím častěji, tím vyšší přesnost a také vyšší nároky na výkon. Ukazuje se, že pro běžné situace tato metoda plně dostačuje a při vyšším výkonu dosahuje zcela vyhovující kvality.

Jak tedy funguje tato alternativa? Výpočet koeficientů plátu je shodný s předchozím krokem, ale poté se nepoužívá fragment program, ale vertex program, který ve zvolených bodech (nakonfigurovaných předem) spočítá hodnotu, kterou tam vizualizovaná data mají, přičemž pro zbytek (lineární) interpolace se využije přirozené schopnosti akcelerátoru mapovat (a interpolovat) texturu. Výhodou této metody je fakt, že u per-pixel interpolace musí fragment program v každém bodě vypočítat z dostupné hodnoty **t** potřebné druhé a třetí mocniny, což je relativně náročné při provádění pro každý pixel. Naproti tomu je možno u omezeného, předem daného počtu vrcholů tyto mocniny předpočítat a uložit ve vertex array, takže vertex program již pouze tyto hodnoty lineárně zkombinuje z dodanými koeficienty plátu.

Zůstává ještě jeden problém – pokud vykreslujeme mřížku pomocí kubické interpolace, potřebujeme u každé buňky mřížky znát hodnoty ve vrcholech sousedních buněk. To ovšem nemůžeme v případě, že daná buňka leží na okraji mřížky. Tento problém lze vyřešit dvěma způsoby: buď nevykreslovat krajní buňky (a používat je pouze pro interpolaci), čímž ovšem uživateli upíráme mnoho náročně vypočítaných dat, nebo, což je způsob, který je použit v knihovně Visla, na krajní body neaplikujeme stejnou interpolační metodu. V mé implementaci se toho dosahuje tím, že po zadání celé mřížky jsou lineární extrapolací vypočítány body, které leží mimo zadanou mřížku, takže i původně okrajové buňky nyní mají dostatek sousedních bodů na to, aby se dala použít kubická interpolace.

6.3.7.2 Izokřivky

Při vykreslování mřížky hodnot mají pro dobré pochopení velký význam izokřivky, tzn. křivky spojující na obrázku místa se stejnou hodnotou. Jejich vykreslování je však obecně poměrně složité a často se neprovádí v reálném čase. Knihovna Visla umožňuje vykreslovat izokřivky v reálném čase, ale tato schopnost má samozřejmě jistá omezení – izokřivky vykreslované knihovnou nejsou zcela přesné a v jistých případech se může stát, že jsou velmi nezřetelné.

Metoda vykreslování izokřivek je velice jednoduchá: jelikož se pro převod zobrazované hodnoty na barvu používá textura, můžeme tuto texturu vhodně upravit – pokud se např. místo barvy, která odpovídá hodnotě 0,5, do textury vloží černá barva, bude při vykreslování zobrazena černá barva všude tam, kde se interpolací získá hodnota 0,5 – což je ovšem právě definice izokřivky. Jenže takto získaná izokřivka není dokonalá, protože vykreslujeme diskrétní množinu bodů, takže se může stát, že jeden bod má hodnotu o něco nižší než hodnota izokřivky, sousední pak o něco vyšší, ale žádný bod nemá přesnou hodnotu nastavenou pro izokřivku (přestože původní data byla spojitá). Stejně tak je konečný i počet hodnot textury, takže černý bod na textuře neodpovídá jenom hodnotě 0,5, ale i např. hodnotě 0,45, atp. To odpovídá situaci, že izokřivka je mírně rozmazaná mezi okolními body. V extrémních případech může tato jednoduchá metoda vést k tomu, že „izokřivka“ zabírá celou zobrazovanou plochu, ovšem vzhledem k nízkým požadavkům na výkon (přesněji řečeno, zobrazování izokřivek nemá *žádné* dodatečné nároky oproti základnímu vykreslování mřížky dat) je tato metoda vhodná pro vizualizaci v reálném čase.

6.3.8 Funkce pro vykreslování sady mřížek

Tyto funkce jsou velice jednoduché – buď lineárně, nebo v předchozí kapitole popsanou metodou spline interpolace se vyrobí požadovaný mezičez, který je poté právě popsanou metodou zcela normálně zobrazen.

6.3.9 Funkce pro vizualizaci metodou IBFV

Pro vizualizaci vektorového pole nabízí knihovna Visla skupinu funkcí sdruženou ve třídě IBVFHelper. Vizualizace dynamického vektorového pole probíhá po dvou krocích: nejprve se předají hodnoty vektorů v celé mřížce (metodou obdobnou předávání hodnot při vizualizaci běžné mřížky), které se uloží do připraveného pole. Poté se vytvoří nový snímek animace pomocí vykreslení čtvercové mřížky, deformované v každém svém bodě podle hodnoty, kterou tam vektorové pole má, přičemž se minulý snímek používá jako textura. Tím se dosáhne toho, že se minulý snímek transformuje podle vektorového pole. Poté je do snímku vmíchán bílý šum. Výsledný snímek je pak vykreslen.

Pokud vizualizujeme statické vektorové pole, stačí hodnoty pole předat jednou, na začátku. Poté se animace, popisující toto statické pole, vytváří naprosto stejným způsobem.

6.3.10 Interní funkce

Knihovna Visla obsahuje třídy a funkce, které umožňují pracovat s různými funkcemi OpenGL extenzí nehledě na to, která konkrétní známá extenze ze známých je na daném systému podporována. To umožňuje zbytku knihovny oprostit se od rozdílů mezi jednotlivými extenzemi. V aktuální verzi knihovny jsou podporovány NV a ARB verze extenzí (neboť mě dostupný hardware podporoval právě tyto), ovšem přidání podpory jiných extenzí s obdobným rozhraním je jednoduché.

Tuto část knihovny je možno též použít odděleně v jiných aplikacích, neboť poskytuje poměrně užitečné funkce pro programy, které se při použití OpenGL extenzí chtějí vyhnout nutnosti vybrat si jednu konkrétní k implementaci.

7 Testování knihovny

Tato kapitola pojednává o způsobech, jakými byla knihovna Visla testována, jsou zde také výsledky těchto testů, srovnání kvality vizualizace, atp.

7.1 Metodologie testování

Testování je rozděleno na dvě základní složky: jednak se na několika ukázkových příkladech, dodávaných s knihovnou (v adresáři examples), odděleně zkoušejí všechny schopnosti knihovny na umělých datech. V této části testování se odlaďovaly základnější chyby. Druhou složkou je použití knihovny v programu MyPCC, kde se všechny schopnosti knihovny používají současně, na reálných datech. V této fázi se odlaďovaly nenápadnější chyby, které se projevily až po zapojení do komplexní aplikace (jako např. vzájemné ovlivňování stavu OpenGL mezi aplikací a knihovnou).

7.2 Ukázkové příklady

S knihovnou Visla se distribuuje několik příkladů, které ukazují jednotlivé funkce knihovny:

- Příklad **e_basic** – ukazuje základní inicializaci knihovny a použití knihovny pro vykreslení mřížky hodnot
- Příklad **e_points** – ukazuje vykreslování bodů (částic), umožňuje přepínat mezi režimy zobrazování částic
- Příklad **e_layers** – ukazuje vykreslování a interpolaci vrstev při použití sady mřížek
- Příklad **e_cache** – ukazuje přepínání palet a použití paměti pro jednou vygenerované textury, umožňuje zapínat a vypínat i zobrazování izokřivek
- Příklad **e_vecfield** – předvádí vizualizaci vektorového pole pomocí metody IBFV, zobrazuje jednoduchý vír, do kterého je možno vstřikovat barvivo pro zvýraznění.

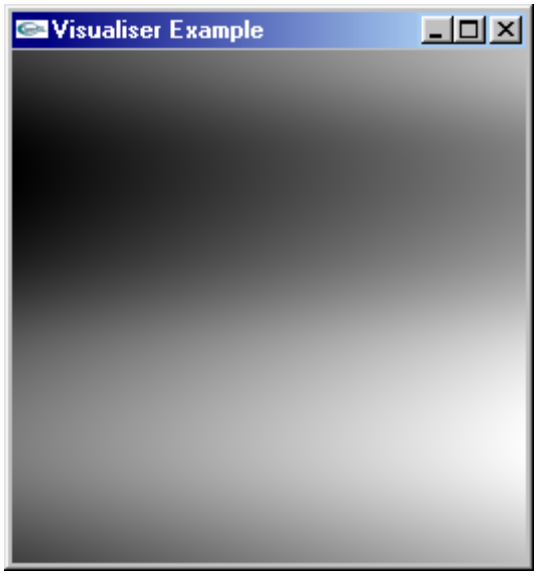
Všechny příklady (až na nejjednodušší e_basic) umožňují nastavovat různé parametry zobrazení, k čemuž lze použít buď klávesové zkratky, nebo kontextové menu systému GLUT (zobrazené pomocí pravého tlačítka myši).

7.3 MyPCC

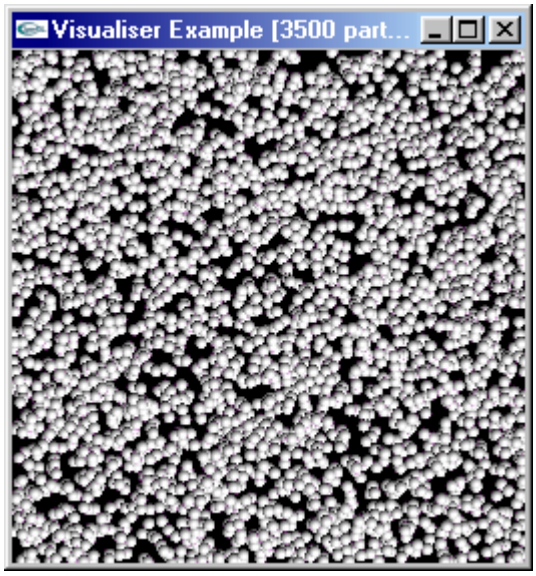
Primárním cílem a měřítkem této diplomové práce je právě program MyPCC, do kterého byla knihovna zapojena. Proto se v této kapitole zaměřím na výsledky testování knihovny v programu MyPCC, porovnání původní a nynější vizualizace, atd.

7.3.1 Porovnání kvality vizualizace

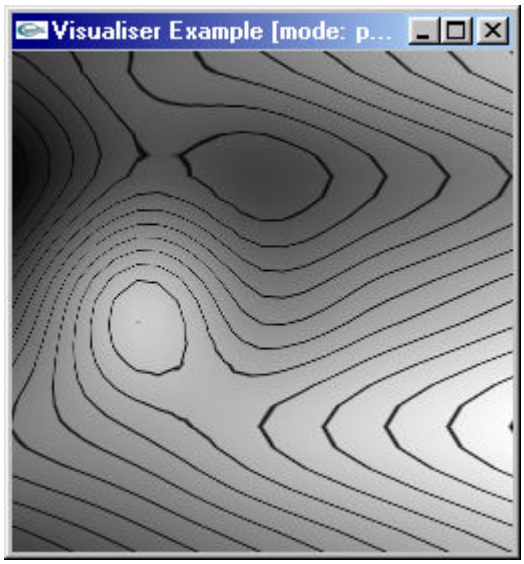
Kvalitu vizualizace můžeme posuzovat z několika hledisek, z nichž nejdůležitější jsou zřejmě vizuální



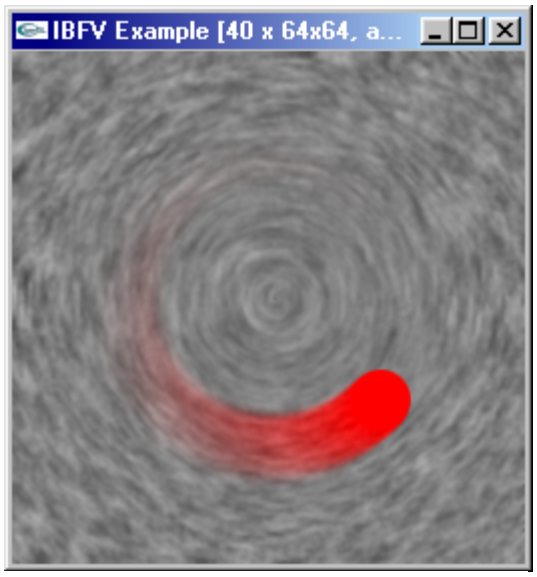
obrázek 15 – příklad e_basic



obrázek 16 – příklad e_points



obrázek 17 – příklad e_cache



obrázek 18 – příklad e_vecfield

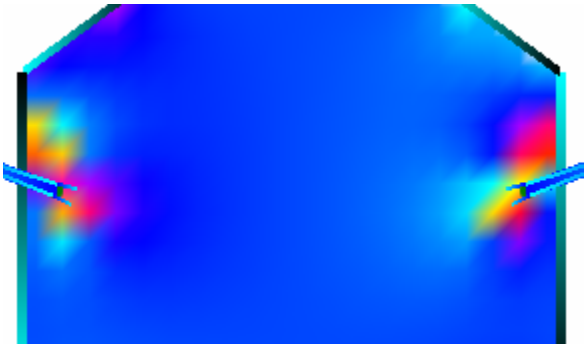
kvalita (a s ní související přesnost) a rychlost vizualizace.

Původní vizualizace trpěla především v situacích, kdy je v nějakém místě velký gradient hodnoty, neboť tehdy vedla RGB interpolace k tomu, že na takových místech se objevovaly zcela špatné barvy (při použití implicitní palety byla tato místa výrazně tmavá a šedá). Tento problém je dobře vidět pouze na barevném obrázku, alespoň ilustrován je na obrázku 19, na obrázku 20 je pak vidět, jak stejné místo vypadá při použití vizualizace nabízené knihovnou.

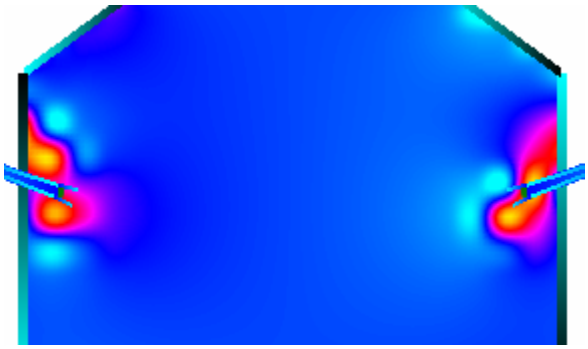
Dalším nedostatkem původní metody vizualizace byl viditelný šestiúhelníkový vzor v situacích, kdy vizualizovaná data byla výrazněji nelineární. Tehdy bylo již na první pohled zřejmé, že data tvoří čtvercovou

(přesněji řečeno trojúhelníkovou, kvůli způsobu, jakým OpenGL provádí interpolaci) mřížku, neboť body této mřížky byly zvýrazněny viditelnými nepravými hranami. Tato vada je vidět na obrázku 21, na následujícím obrázku je pro srovnání vidět výsledek nové vizualizace.

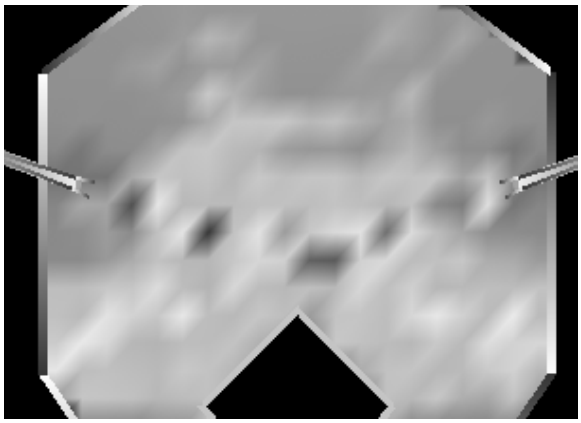
Původní metoda vizualizace byla prakticky nepoužitelná při zvětšení detailu vizualizovaných dat, což je vidět na obrázku 23. Oproti tomu je na dalším obrázku vidět, že bikubická interpolace je schopna vysoké kvality vizualizace i v takových situacích.



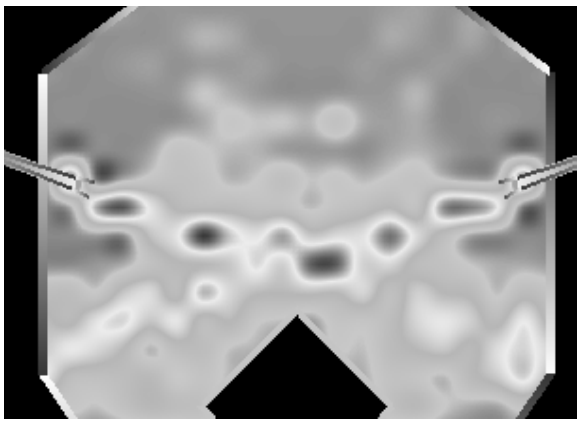
obrázek 19 – Velký gradient, původní



obrázek 20 – Velký gradient, Visla

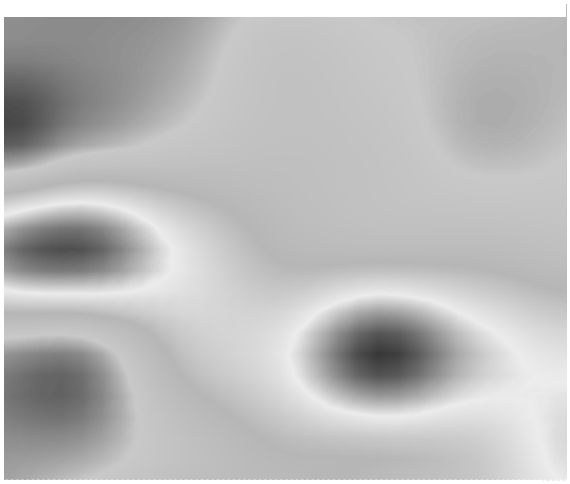


obrázek 21 – Nelineární data, původní



obrázek 22 – Nelineární data, Visla

obrázek 23 – Zvětšený detail, původní

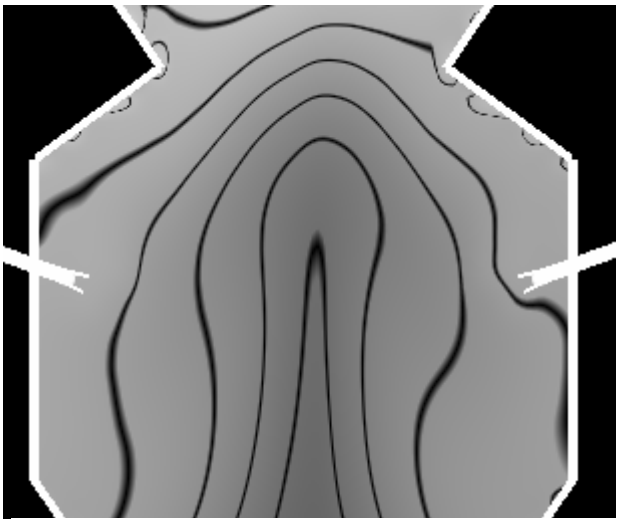


obrázek 24 – Zvětšený detail, Visla

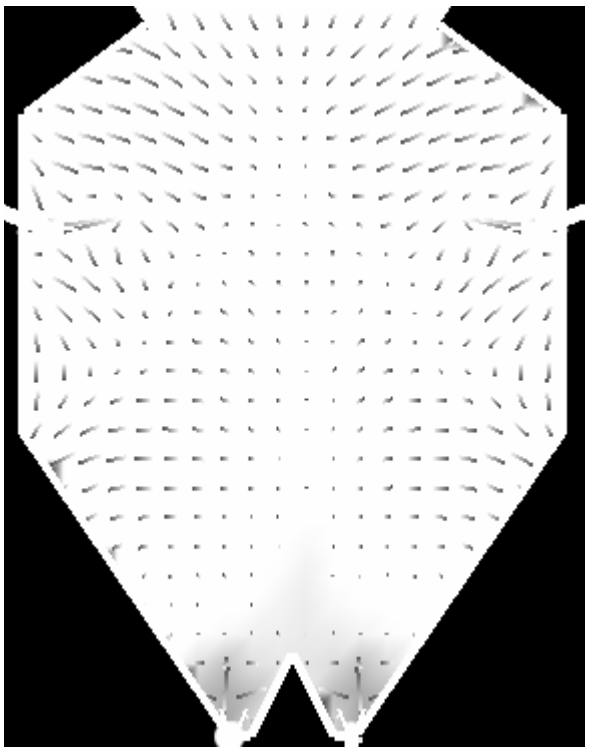
7.3.2 Zcela nové schopnosti

Použití knihovny Visla přineslo aplikaci MyPCC některé zcela nové schopnosti:

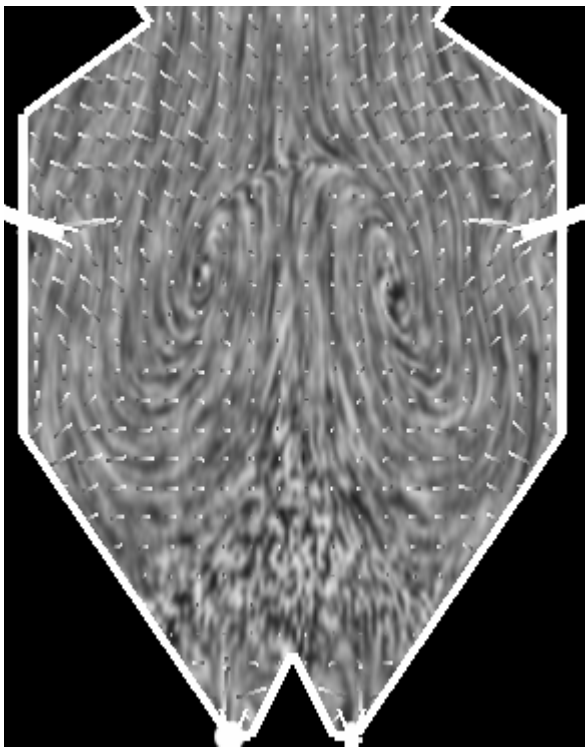
- Vykreslování izokřivek (viz obrázek 25)
- Vizualizace proudění metodou IBFV – původně se proudění zobrazovalo pouze pomocí orientovaných úseček, natočených ve směru proudění, viz obrázek 26. Výsledek IBFV je na obrázku 27.



obrázek 25 – Izokřivky pomocí knihovny Visla



obrázek 26 – Vizualizace proudění, původní



obrázek 27 – Vizualizace proudění, Visla

7.3.3 Porovnání rychlosti

Porovnání původního a současného řešení z hlediska rychlosti je evidentně velice obtížné – je zřejmé, že značný nárůst kvality musí být doprovázen jistým nárůstem výpočetní náročnosti. Přesto se zde pokusím o porovnání, s jehož interpretací je ovšem třeba být obezřetný.

V tabulce 1 jsou uvedeny naměřené rychlosti na testovacích počítačích, které jsem měl k dispozici (popis počítačů viz níže). Rychlosti jsou udávány ve fps (počet vykreslených snímků za sekundu). Měření probíhalo přímo v aplikaci MyPCC, která tento údaj zobrazuje a jelikož použití v tomto programu je hlavním cílem práce, je vhodné sledovat právě výkon tohoto programu. Program běžel v různých režimech, přičemž měření probíhalo při použití standardního testovacího kotle `test-pi.blr`, který je dodáván v distribuci programu MyPCC.

V tabulce jsou jednak hodnoty při použití původní vizualizace (pouze lineární interpolace, atp.), jednak hodnoty při použití knihovny Visla (bikubická interpolace, …). U obou variant jsou pak ukázány hodnoty ve dvou situacích: sloupec *běžící* obsahuje hodnoty pro současnou vizualizaci a výpočet modelu (tzn. při spuštěné simulaci). Sloupec *zastavena* obsahuje hodnoty pro případ, kdy je simulace zastavena a provádí se pouze vizualizace konstantních dat.

Konfigurace	**Původní vizualizace**		**Visla**	
	běžící	**zastavena**	**běžící**	**zastavena**
1	18	100	16	64
2	7	64	6	25
3	4	59	3	15
4	2	3	1	1
5	3	29	*nelze*	*nelze*

Tabulka 1 – Výsledky měření rychlosti zobrazování

Konfigurace 1: AMD Athlon XP 1700+, 256 MB RAM, nVidia GeForce FX5200 128 MB, AGP 4x
Konfigurace 2: AMD Athlon 1200 MHz, 512 MB RAM, nVidia GeForce4Ti 32 MB, AGP 2x
Konfigurace 3: Intel Pentium II 350 MHz, 256 MB RAM, nVidia GeForce4 MX 440 64 MB, AGP 2x
Konfigurace 4: AMD Athlon XP 1700+, 256 MB RAM, plně softwarové vykreslování
Konfigurace 5: Intel Pentium 4M 1,6 GHz, 256 MB RAM, integrovaný adaptér SiS se sdílenou pamětí

Konfigurace 5 je v tabulce uvedena pouze pro možnost srovnání výkonu původní vizualizace na počítači se špatným grafickým hardwarem, neboť grafická karta integrovaná v tomto počítači (jedná se o notebook) není vhodná pro 3D akceleraci a nepodporuje funkce vyžadované knihovnou Visla.

Konfigurace 4 pak ukazuje, jak důležitý je grafický hardware. V této konfiguraci bylo použito plně softwarové vykreslování a výkon klesl na prakticky nepoužitelné minimum.

7.3.3.1 Interpretace výsledků

Je zřejmé, že při použití původní lineární interpolace je samotná vizualizace (při zastavené simulaci) citelně rychlejší (porovnání sloupců *zastavena*). Takové srovnání ovšem jednak zanedbává podstatný rozdíl v kvalitě vizualizace, ale především opomíjí hlavní cíl knihovny – interaktivní simulaci a vizualizaci. Pokud totiž zastavíme všechny další výpočty a pouze necháme daná konstantní data zobrazit, nepotřebujeme na to vysoký výkon – stačí totiž spočítat jeden snímek. Oproti tomu vidíme při porovnání sloupců *běžící*, že při spuštěné simulaci je výkon aplikace MyPCC při použití knihovny Visla téměř stejný, jako u původní vizualizace. To je velice důležitý výsledek, ve kterém se projevuje paralelismus, kterého jsme dosáhli při použití grafického akcelerátoru: grafický procesor provádí výpočty potřebné pro vizualizaci, zatímco centrální procesor může provádět simulaci. Proto se zvýšení kvality vizualizace sice projevuje ve zvýšení zátěže grafického procesoru (kterou ovšem bohužel nemůžeme změřit, pro poskytování takových informací nemá procesor přístupné žádné rozhraní), ale jelikož ten byl dosud prakticky nevyužit, navenek to vypadá tak, že toto zvýšení kvality nemá nároky na vyšší výkon. Mírné snížení celkové rychlosti, ke kterému samozřejmě došlo, je dáno tím, že některé vizualizační výpočty musí být provedeny vždy na centrálním procesoru.

Proto můžeme říct, že pokud disponujeme moderním grafickým hardwarem, jsme schopni zvýšit kvalitu vizualizace tak, že schopností tohoto hardware využijeme. V opačném případě totiž velká část jeho výkonu leží ladem.

8 Závěr, budoucí práce

Tato diplomová práce nabízí obecně použitelnou knihovnu základních nástrojů pro vizualizaci vědeckotechnických dat v reálném čase, založenou na OpenGL a využívající některých vlastností moderních grafických akcelerátorů.

Knihovna byla úspěšně zaintegrována do existujícího simulačního a vizualizačního systému MyPCC, kde se při testování osvědčila, a ve kterém výrazně zvýšila kvalitu vizualizace při minimálním nárůstu výpočetní náročnosti.

Další rozvoj této knihovny by měl spočívat jednak v doplnění některých implementačních detailů (jako podporu extenzí od dalších výrobců, nebo podporu pro zahrnutí vertex a fragment shaderů přímo do spustitelného souboru s aplikací), jednak v rozšiřování schopností knihovny zahrnutím dalších vizualizačních metod. Autor plánuje knihovnu zveřejnit pro volné použití a úpravy, což umožní její další rozvoj.

Literatura

[1] Gayer, M. *Vizualizace technologických procesů*. Praha, 2004. Doktorandská práce na FEL ČVUT. Vedoucí práce Pavel Slavík.

[2] Gayer, M. *My Pulverized Coal Combustion* [online]. http://www.cgg.cvut.cz/~xgayer/

[3] Lorensen, W., Cline, H. *Marching Cubes. A High Resolution 3-D Surface Construction Algorithm*. In *Computer Graphics*. volume 21, str. 163–169, 1987. ACM SIGGRAPH.

[4] Reeves, W. T. *Particle Systems – A Technique For Modelling a Class of Fuzzy Objects*. In *Proceedings of SIGGRAPH '83*, volume 17, n. 3, str. 359–376. ACM Computer Graphics, 1983.

[5] van Wijk, J. J. *Spot Noise – Texture Synthesis For Data Visualization*. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, str. 263–272, 1991. ACM SIGGRAPH.

[6] Cabral, B., Leedom C. *Imaging Vector Fields Using Line Integral Convolution*. In *SIGGRAPH 93 Conference Proceedings*, str. 263–270, 1993. ACM SIGGRAPH.

[7] Wegenkittl, R., Gröller, E., Purgathofer, W. *Animating Flowfields: Rendering of Oriented Line Integral Convolution*. In *Computer Animation '97 Proceedings*, str. 15–21, 1997. IEEE Computer Society.

[8] Wegenkittl, R., Gröller, E. *Fast Oriented Line Integral Convolution for Vector Field Visualization via the Internet*. Institute of Computer Graphics, Vienna University of Technology, 2003.

[9] van Wijk, J. J. Image Based Flow Visualization. In *ACM Transactions on Graphics (SIGGRAPH 2002 Proceedings)*, special issue, 2002. ACM SIGGRAPH.

[10] Murdoch, A. A. *Potential of Commodity Graphics Hardware for Scientific Computation* [online]. http://www.ukhec.ac.uk/publications/reports/gfx_report.pdf. The University of Edinburgh, 2002.

[11] *The OpenGL® Graphics System: A Specification (Version 1.5)* [online]. Editors Mark Segal, Kurt Akeley. http://www.opengl.org/documentation/specs/version1.5/glspec15.pdf

[12] *OpenGL Extension Registry* [online]. http://oss.sgi.com/projects/ogl-sample/registry/

[13] *OpenGL 2.0 Overview* [online]. http://www.3dlabs.com/support/developer/ogl2/whitepapers/OGL2_Overview_1.2.pdf

[14] Gayer, M. *Grafická knihovna MGL* [online]. http://www.cgg.cvut.cz/~xgayer/mgl/

[15] Catmull, E., Rom, R. *A Class of Local Interpolating Splines*. Computer Aided Geometric Design, 1974. Academic Press.

Další využívané WWW stránky

Vývojářské stránky společnosti nVidia – http://developer.nvidia.com/

Vývojářské stránky společnosti ATI – http://www.ati.com/developers/

X-Zone – stránky o programování DirectX – http://www.mvps.org/directx/

A Ukázka vertex programu

Jako ukázku, jak vypadá vertex program, zde uvádím zkrácenou verzi programu pro výpočet texturovacích souřadnic při použití *point sprites* bez jejich speciální podpory (viz odstavec 4.3.1.6):

```
!!ARBvp1.0
TEMP R0, R1, R2;

ATTRIB vPos = vertex.position;
ATTRIB vTex0 = vertex.texcoord[0];
ATTRIB vCol0 = vertex.color;

PARAM cMatrix[4] = { program.local[0..3] };
PARAM cSize = program.local[4];
PARAM cConsts = program.local[5];

# Transformace vrcholu do homogenního prostoru
DP4   R0.x, cMatrix[0], vPos;
DP4   R0.y, cMatrix[1], vPos;
DP4   R0.z, cMatrix[2], vPos;
DP4   R0.w, cMatrix[3], vPos;

# Výpočet souřadnic pro point sprite
MOV   R1, vTex0;
MAD   R2, R1, cSize.xyww, R0;
MOV   R0, cConsts;
MAD   result.position, -R0.yyyy, cSize.xxww, R2;

# Uložení texturovacích souřadnic
MOV   result.texcoord[0], R1;

# Uložení barvy
MOV   result.color.front.primary, vCol0;

END
```

B Ukázka fragment programu

Jako ukázku, jak vypadá fragment program, zde uvádím smyšlený příklad, jak by se dala aplikovat jakási detailová textura, k čemuž se využije multitexturování. Program je velice jednoduchý, ovšem ukazuje jediný podstatný rozdíl mezi obsahem vertex a fragment programů – texturovací instrukci TEX – zbytek je na první pohled k nerozeznání od vertex programů.

```
!!ARBfp1.0

TEMP R0, R1;

PARAM c0 = {10, 0, 0, 0};

# Aplikace základní textury
TEX   R0, fragment.texcoord[0], texture[0], 2D;

# Aplikace detailové textury
MUL   R1.xy, fragment.texcoord[0], c0.x;
TEX   R1, R1, texture[1], 2D;
ADD   R1, R0, R1;

# Modulace dodanou barvou
MUL   result.color, R1, fragment.color.primary;
END
```