Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Softvérové inžinierstvo

Bc. Ondrej Hirjak

SIMULÁCIA TEKUTÍN POMOCOU GRAFICKÉHO PROCESORA V REÁLNOM ČASE

Diplomová práca

Vedúci diplomovej práce: Ing. Peter Drahoš
máj, 2008

Tu bude kópia zadania

Analyzujte možnosti využitia moderných grafických procesných jednotiek (GPU) v simulácii nestlačiteľných tekutín v reálnom čase na základe Navier-Stokes rovníc. Porovnajte existujúce metódy v 2D i 3D simuláciách so zameraním na výkon v reálnom čase. Implementujte aplikáciu využívajúcu GPU na simuláciu a vizualizáciu turbulencií tekutiny v obmedzenom 2D priestore s prekážkami.

Čestne prehlasujem, že túto prácu som vypracoval samostatne a použil som len citované zdroje.

Ondrej Hirjak

Anotácia

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Študijný program: Softvérové inžinierstvo
Autor: Bc. Ondrej Hirjak
Diplomová práca: Simulácia tekutín pomocou grafického procesora v reálnom čase
Vedúci práce: Ing. Peter Drahoš
máj, 2008

Práca analyzuje možnosti programovania grafického hardvéru, jeho vývoj a súčasný stav. Na zvýraznenie špecifík a rozdielov sú uvedené príklady klasického programovania CPU a príklady paradigmy paralelného počítača. V práci je diskutovaná vhodnosť rôznych GPU technológií a metód na účely všeobecného programovania. Špeciálne sú rozobraté programovacie jazyky OpenGL. Ako ukážková aplikácia je použitá CPU a GPU implementácia simulácie tekutín. Sú prezentované základné matematické a numerické metódy na riešenie diferenciálnych rovníc a sústavy lineárnych rovníc. Práca uvádza zmeny a rozšírenia algoritmu oproti známym metódam. Špecifikácia vymenováva zoznam požiadaviek na aplikáciu a časť návrhu rozoberá štruktúru programu a riešenie požiadaviek špecifikácie. Výkon CPU a GPU je porovnávaný na základe testovania programu s meraním výkonnosti procesorov. Zhrnutie výsledkov práce a výstupov doterajšej činnosti s možnosťami ďalšieho rozširovania sú obsiahnuté v závere.

Annotation

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Degree Course: Software Engineering
Author: Bc. Ondrej Hirjak
Diploma project: Real-time fluid simulation on GPU
Supervisor: Ing. Peter Drahoš
2008, May

This paper presents analysis of graphics hardware programmability, its history and current status of utilization. Examples of classic CPU programming and data-parallel computer paradigm are provided to emphasize the differences. Various GPU related technologies and methods are analyzed for their suitability for general purpose programming. Specific OpenGL programming languages are examined. CPU and GPU implementation of fluid simulation is used as a sample application. Basic math and numerical methods for solving differential equations and systems of linear equations are discussed. Extensions to known methods are presented to satisfy assignment. Requirements for application are listed and structure of program and solution of requirements is proposed. Testing and benchmarking provide comparison of CPU and GPU performance. Status of current results and future plans are presented in conclusion.

Obsah

Zoznam obrázkov

1 Úvod

Práca sa zaoberá využitím grafických procesných jednotiek – grafických kariet – GPU na všeobecné výpočty, príkladom ktorých je simulácia nestlačiteľných tekutín. Historický vývoj a súčasný stav vývoja a použitia grafických kariet je prezentovaný v pokračovaní tejto kapitoly. Analýzou existujúcich prác, možnosťami využitia PC na výpočty a oblasťou ich využitia sa zaoberá časť Analýza problémovej oblasti. V časti Použitie GPU na všeobecné výpočty sú rozobraté spôsoby, akými je možno výpočty realizovať, technológie, ktoré sa využívajú, ich porovnanie a diskusia ich obmedzení. Príkladom využitia GPU na všeobecné výpočty sa zaoberá kapitola Simulácia tekutín. V nej je uvádzaný potrebný matematický aparát na riešenie diferenciálnych rovníc. Zároveň je rozobrané využitie Navier-Stokes rovníc na riešenie 2D metód simulácie tekutín. Nasleduje kapitola špecifikácia programu, jeho návrh a hrubý opis implementácie. Náplňou ďalšej kapitoly je uvedenie spôsobu testovania aplikácie, spracovania dát a vyhodnotenia výstupov testov. V závere je uvedené zhrnutie výsledkov dosiahnutých počas práce na projekte. Ďalej sú uvedené možnosti rozšírenia práce a jej smerovania. Práca obsahuje prílohy: Príklady existujúcich aplikácií, Technická dokumentácia, Používateľská príručka, Zoznam použitého hardvéru a Obsah elektronického média.

1.1 Historický vývoj

Súčasná doba sa vyznačuje rýchlym technickým a technologickým pokrokom. Tento trend informačné technológie neobchádza, naopak, rast objemu prostriedkov vkladaných do IT tento fakt len potvrdzuje. Prudký rozvoj technológie so sebou prináša množstvo nových myšlienok a postupov, ktoré dovtedy neboli mysliteľné. Zároveň sa neraz stáva, že technológie sa využívajú spôsobmi, na ktoré ich tvorcovia ani nepomysleli. Takéto prípady sa stávajú tak pri softvéri, ako aj pri hardvéri. Práve prípadom inovatívneho (z hľadiska pôvodného určenia) využitia hardvéru, konkrétne grafických kariet, sa zaoberá táto práca.

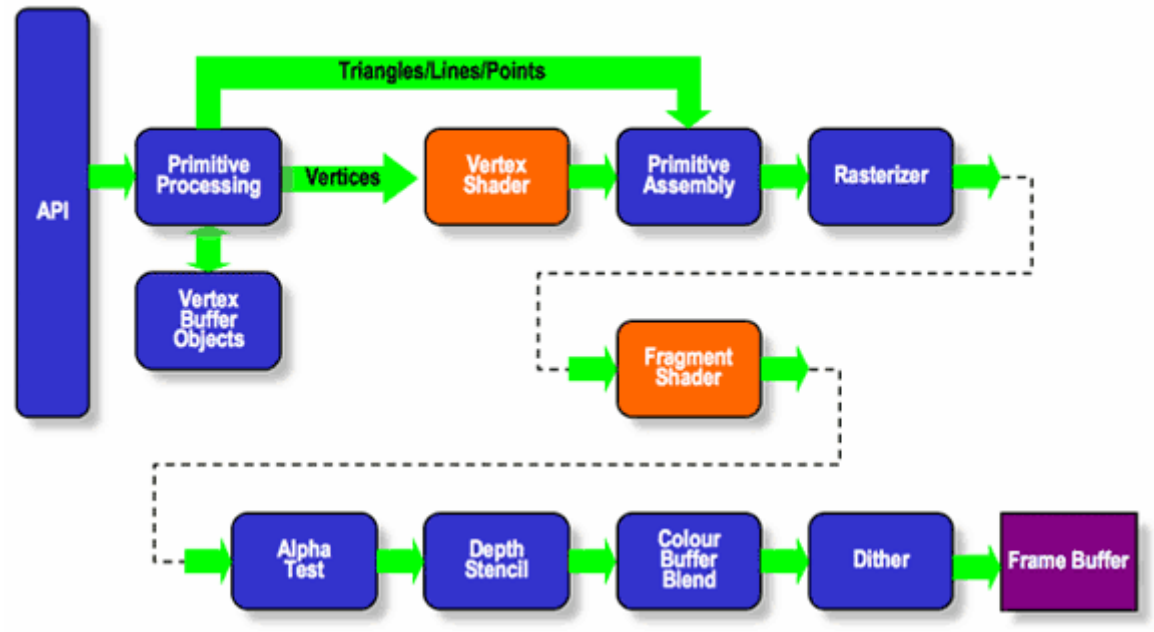
Počítačová grafika a jej využitie ešte v nedávnej minulosti patrili do domény profesionálnych grafických štúdií a výskumných pracovísk, ktoré si mohli dovoliť financovať toho času drahé grafické pracovné stanice (napr. SGI). Zlacňovanie hardvéru a jeho sprístupnenie širokej verejnosti postupne zotrelo hranice medzi profesionálnym a amatérskym využitím takýchto staníc a grafických kariet. Takto sa medzi širokú bázu spotrebiteľov dostali grafické karty, ktoré zďaleka neplnili len ich pôvodnú funkciu, t.j. prevod digitálneho signálu do jeho analógovej formy, ale disponovali aj potrebným výkonom na náročné grafické operácie.

Začiatkom 90-tych rokov prišli na trh prvé grafické karty, ktoré sa začali nazývať akcelerátormi. Akcelerácia spočívala v prebraní vykresľovania základných grafických primitív od CPU, teda jeho odľahčenia a zároveň urýchlenia vykresľovania. Príkladom akcelerátorov boli karty ako S3 Virge, ATI Rage alebo Matrox Mystique. Tieto karty však podporovali iba 2D akceleráciu, na 3D akceleráciu bolo potrebné mať separátny 3D akcelerátor. Jedným z nich bola karta 3dfx Voodoo.

Prvým skutočným akcelerátorom 3D grafiky bola karta NVIDIA GeForce 256. Táto umožňovala hardvérovú akceleráciu výpočtov transformácií a osvetlenia (známe pod T&L – Transformation and Lightning), ktoré bolo dovtedy potrebné vykonávať na CPU. Odvtedy vývoj kariet priniesol zväčša zvýšenie rýchlosti grafického čipu, rozšírenie pamäťovej zbernice alebo zvýšenie pamäťovej kapacity. Vyskytli sa však aj iné smery vývoja, ktoré neboli zamerané len na zvyšovanie hrubého výkonu, ale aj na pridanie ďalších vlastností, tzv. programovateľnosti grafických čipov. Úvodná čiastočná programovateľnosť vyústila až do stavu, kedy je možné každú fázu spracovania obrazu programovo riadiť.

1.2 Súčasný stav

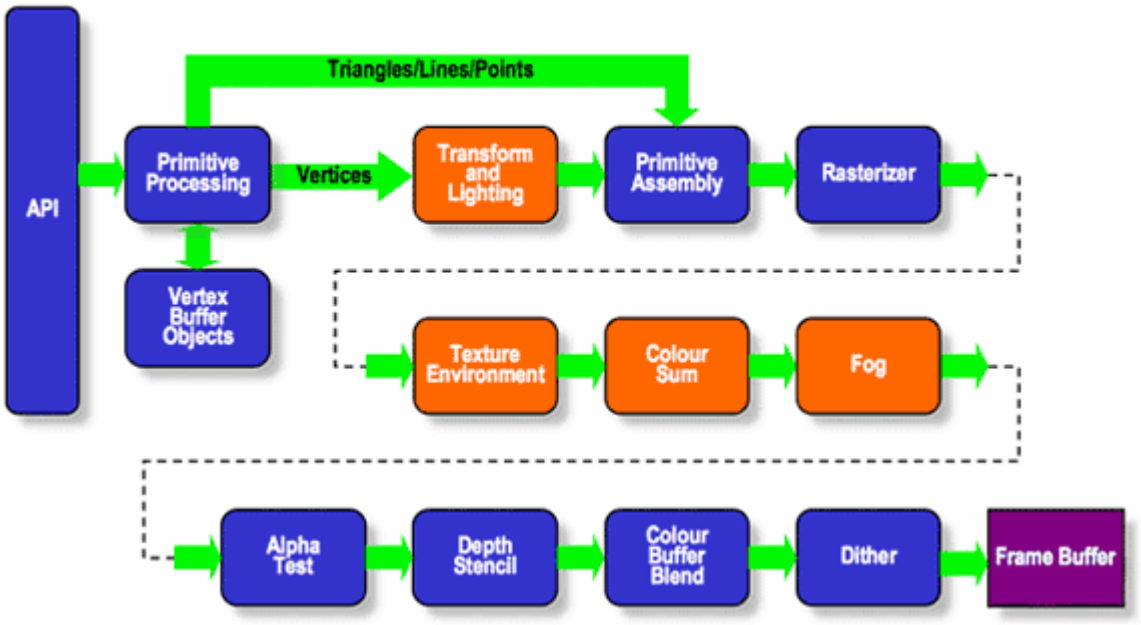
Výsledkom vývoja grafických kariet je generácia kariet s tzv. *programmable pipeline* (Obrázok 1. Programmable graphics pipeline [4]), teda kariet s programovateľným spracovaním geometrie (*vrcholov*) a obrazových elementov (*fragmentov*). To znamená, že pre každý krok spracovania dát je možné vytvoriť program, ktorý bude určovať konkrétny spôsob spracovania. Rozdielom oproti predchádzajúcej generácii, t.j. generácii kariet s tzv. *fixed pipeline* (Obrázok 2. Fixed graphics pipeline [4]) je práve voľnosť vytvoriť akýkoľvek program. *Fixed pipeline* síce umožňovala zmenu parametrov spracovania, ale táto množina a zároveň funkcionalita bola obmedzená. Podobne každá zmena špecifikácie spracovania *fixed pipeline* alebo každý výrobca prinášali nové parametre, čím sa situácia len komplikovala. Konkrétne umiestnenie programovateľných jednotiek je na Obrázok 3. Programmable OpenGL pipeline.



Obrázok 1. Programmable graphics pipeline [4]

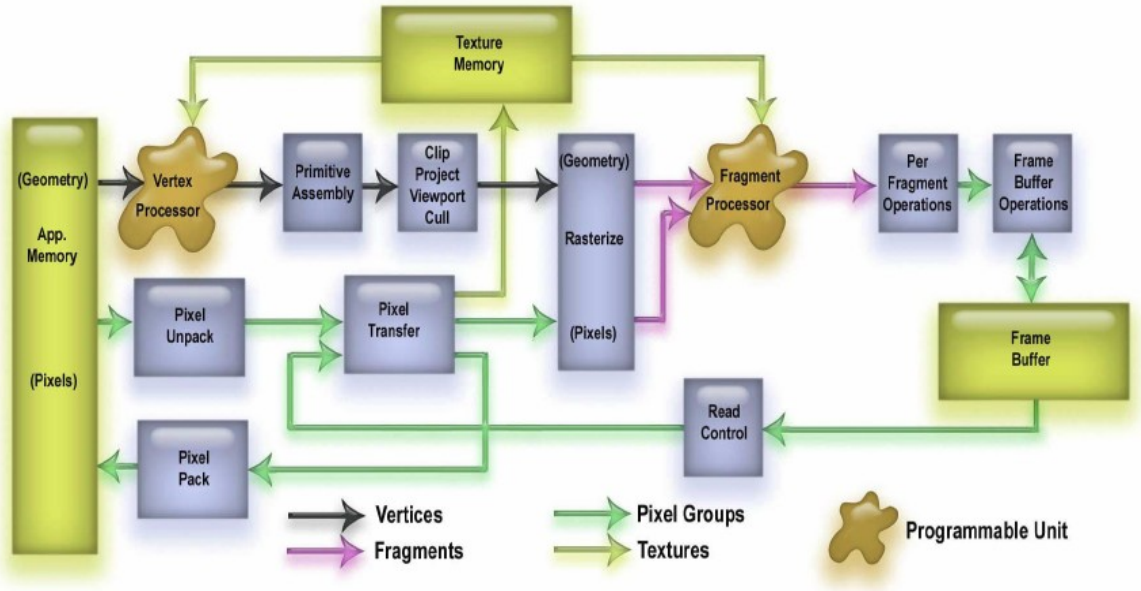
Pomenovanie samotných programov pre jednotlivé programovateľné časti *graphics pipeline* sú napr. *shader* alebo *program*. Z dôvodov neskôr uvedených sa použije pomenovanie *shader*. *Shadery* sa rozdeľujú na tri druhy: *vertex shader*, *geometry shader*

a *fragment*. *Vertex shader* riadi spracovanie geometrických dát, napr. transformáciu zo súradníc objektu do súradníc kamery. *Fragment shader* zväčša riadi výpočet osvetlenia alebo textúrovanie. *Geometry shader* sa v *graphics pipeline* nachádza za *vertex* a pred *fragment shaderom*. Umožňuje manipuláciu s vrcholmi v zmysle mazania existujúcich alebo pridávania nových.



Obrázok 2. Fixed graphics pipeline [4]

Prvými kartami prinášajúcimi *vertex shadery* boli 3. generácia kariet GeForce ako NVIDIA GeForce 3Ti alebo jej náprotivky od ATI. Vytváranie *fragment shaderov* umožňovala až 5. generácia kariet GeForce FX. Súčasná, 8. generácia grafických kariet GeForce (8x00) prináša *geometry shader*.



Obrázok 3. Programmable OpenGL pipeline

V súčasnosti je rozdelenie programov na *vertex*, *geometry* a *fragment shaderov* čisto len záležitosť pohľadu na využitie týchto kariet (GeForce 8x00, Radeon HD 2x00), t.j. na spracovanie a zobrazenie grafiky. Špecifikácia týchto čipov totiž hovorí, že to sú všeobecné *prúdové (stream)* procesory [13][14]. Práve softvérová nadstavba (ovládače) hovorí o tom, ako sa budú tieto procesory správať. Ako je možné túto skutočnosť využiť, rozoberá nasledujúca kapitola.

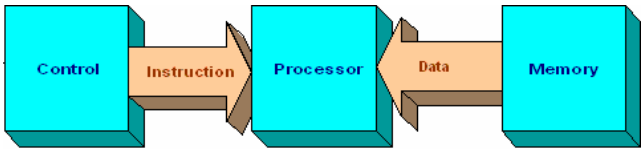
2 Počítač ako výpočtový stroj

Táto kapitola sa zaoberá využitím počítača ako výpočtového stroja. Uvádza rozličné prístupy pri spracovávaní dát, vývoji a spôsobe zvyšovania výkonu počítačov ako aj úlohu grafického procesora v tomto procese. Ponúka príklady a oblasti využitia výpočtového výkonu GPU a v krátkosti spomína existujúce práce na podobné témy.

2.1 Možnosti využitia jednotlivých komponentov

2.1.1 Výpočty na CPU

Hlavnou úlohou počítačov je uskutočňovať výpočty, ktoré realizuje centrálna procesorová jednotka – CPU. Keďže architektúra počítača je založená na von Neumann-ovom modeli, mikroprocesory boli dlhú dobu typu SISD, čiže Single Instruction, Single Data. Cesta zvyšovania výkonu sa uberala smerom zvyšovania pracovných frekvencií, pridávaní zložitejších inštrukcií, ktoré zvládali komplikovanejšie operácie (architektúra CISC) alebo kombináciou oboch spôsobov. Avšak čas ukázal, že táto cesta nie vždy vedie k očakávaným výsledkom alebo prináša dodatočné komplikácie, napr. zvyšovanie zložitosti kompilátorov pridávaním inštrukcií alebo narazenie na fyzické vlastnosti materiálov pri zvyšovaní frekvencií.



Obrázok 4. Model SISD [15]

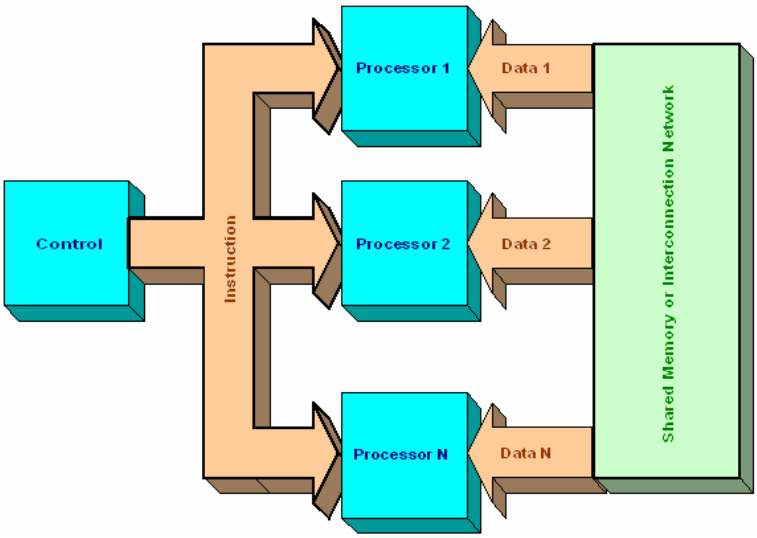
2.1.2 SIMD

Vtedy sa všetky operácie, ktoré bolo potrebné vykonať, vykonávali sekvenčne. Príchod procesora Intel Pentium MMX znamenal koniec tejto éry. Inštrukčná sada procesora rozšírená o inštrukcie MMX zmenila klasickú koncepciu mikroprocesora zo SISD na SIMD – Single Instruction, Multiple Data. Inštrukcie MMX umožňujú v rámci jedinej inštrukcie spracovanie viacerých dát – ich počet závisí od ich veľkosti. Ďalšie rozšírenia ako SSE, SSE2 atď. od Intel-u alebo 3DNow! od AMD nemajú charakter koncepčnej zmeny, oproti MMX, „iba“ rozširujú inštrukčnú sadu o nové typy operandov a operácií.

Princípom je použitie existujúcich registrov procesora (prípad MMX, použitie dolnej časti 80 – bitových FPU registrov fp0 až fp7 ako 64 – bitové mm0 až mm7 registre) alebo pridanie rozširujúcich registrov (SSE a pod., pridanie nových – 128 bitových registrov xmm0 až xmm7). Nad nimi je možné vykonávať operácie, ktoré definujú spôsob ich použitia. Napr. nad MMX registrami je možné vykonať jednu 64 – bitovú operáciu, dve 32 – bitové, štyri 16 – bitové alebo osem 8 – bitových, čiže de facto SIMD operáciu.

Efektívne využitie SIMD a rozšírení MMX, SSE atď. však vyžaduje znalosť inštrukcií assemblera. Pokiaľ nepotrebujeme implementovať špecifické výpočty, zväčša

vystačíme s použitím novšieho prekladača. Ten obsahuje ich podporu a program bez akejkoľvek zmeny bude využívať výhody SIMD.



Obrázok 5. Model SIMD [15]

2.1.3 Multijadrové procesory

Multijadrové alebo *multicore* procesory sú od určitého času bežnou výbavou pracovných staníc, preto je o nich možné uvažovať ako o strojoch s paralelným spracovaním. Podobným prípadom sú multiprocesorové počítače, tie však nie sú natoľko rozšírené. Z hľadiska paralelného spracovania údajov ich však možno považovať za rovnocenné.

Existujú rozličné prístupy k tvorbe a funkcionalite multijadrových procesorov. Výrobcovia ako Intel (napr. 2-jadrový Intel Core 2 Duo) alebo AMD (napr. 2-jadrový AMD Athlon64 X2) sa vydali cestou vysokovýkonných jadier, ktorých funkčnosť je rovnaká a sú vhodné na urýchlenie výpočtov, ktoré sú paralelizovateľné, ale zároveň sú výpočtovo náročné. Príkladom je výpočet *raytracingu* (pre konkrétny bod je to výpočtovo náročný proces, hoci pre rôzne body je výpočet možné paralelizovať).

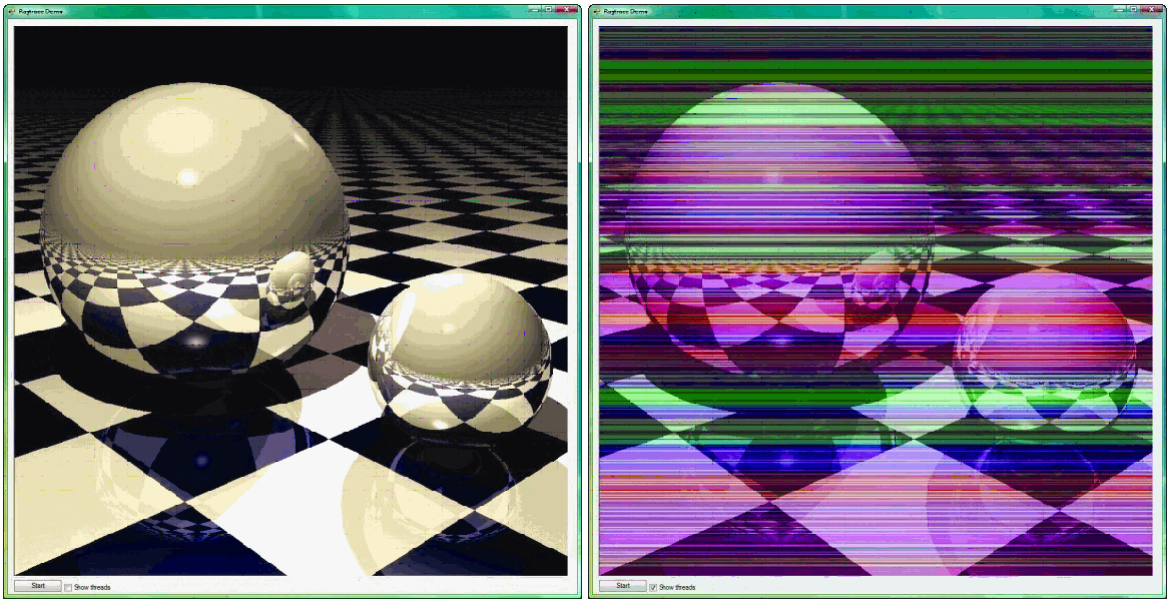
Firma Sun podobne vyrába procesor (napr. 8-jadrový UltraSPARC T2) s rovnakou funkčnosťou jednotlivých jadier. V tomto konkrétnom príklade navyše každé jadro podporuje spracovanie štyroch *threadov* – vlákien. Tento typ je oveľa vhodnejší na paralelné spracovanie dát. Ich náročnosť by však nemala byť príliš vysoká, keďže zvyčajne tieto jadrá nepatria k výkonnostnej špičke. Klasickým príkladom je spracovanie požiadaviek webového servera.

Samostatnou kapitolou sú procesory do hracích konzol (napr. procesor Cell od firmy IBM). Jadrá tohto procesora sú špecializované na určité činnosti a zvyčajne nie sú vhodné na paralelné spracovanie všeobecných výpočtov.

Na využitie multijadrových alebo multiprocesorových počítačov je potrebné program koncipovať tak, aby bol natívne *multithreadový* (viacvláknový). Pri veľkom počte vlákien je využitie viacerých jadier automatické, bez akejkoľvek zmeny programu. Je len na plánovači operačného systému, aby úlohy rovnomerne rozdeľoval na jednotlivé jadrá a zabezpečil ich efektívne využitie. V súčasnosti existuje viacero knižníc aj priamo od

výrobcov procesorov na podporu *multithreadového* programovania, napr. Intel Thread Profiler a pod.

Ďalším významným faktorom pri implementovaní multiprocesorového riešenia je jeho cena. Hoci ceny počítačových komponentov prudko klesajú, bežným používateľom je multiprocesorová pracovná stanica ešte stále nedostupná. Využitie grafického procesora na paralelné výpočty je v tomto smere výhodnejšie, keďže za zlomok ceny je možné získať grafickú kartu s veľakrát oveľa vyšším výpočtovým výkonom.



Obrázok 6. Rozdelenie renderovania raytracingom vláknam [16]

2.1.4 Výpočty na GPU

Využiť GPU na iné výpočty ako priamo súvisiace so spracovaním grafiky bolo donedávna nemožné. Zmena nastala až implementovaním *programmable pipeline*. Programovateľnosť *vertex*, *geometry* a *fragment shaderov* umožňuje prístup k dátam tak, ako si používateľ, v tomto prípade programátor, želá. Z princípu práce GPU je najvhodnejším miestom na výpočty na GPU využitie *fragment shadera* – napr. prístup ku textúre alebo *framebufferu* ako k dvojrozmernému poľu dát. Oproti výpočtom na CPU však tento postup nie je zďaleka priamočiary. CPU je priamo určené na všeobecné výpočty, naproti tomu pri použití GPU je potrebné vykonať vhodné transformácie dát do formátov, ktoré sú natívne pre GPU. Ako vhodný spôsob sa ukázalo spomenuté využitie textúr na uloženie dát. Keďže textúry môžu byť jedno (1D), dvoj (2D) a trojrozmerné (3D) a existuje množstvo formátov na uloženie dát v nich, transformácie nemusia byť triviálne.

Pri renderovaní obrazu sú jednotlivé *fragmenty* spracovávané nezávisle od seba, paralelne (pri novších architektúrach grafickej karty) a bez kontroly poradia ich spracovania. Nemožnosť ovplyvniť poradie spracovania fragmentov je jeden z dôležitých momentov pri prechode k data-parallel programovaciemu prístupu. Hlbšie sa využitím grafického procesora na výpočty zaoberá kapitola č. 3.

2.1.5 Dátovo-paralelný počítač

Už príchod prvého procesora umožňujúceho paralelné spracovanie dát predznamenal potrebu zmeny v myslení programátorov. Klasický spôsob uvažovania, kedy sú všetky dáta spracovávané sekvenčne, bolo potrebné zmeniť na uvažovanie o paralelnom spracovaní dát. V prípade CPU táto zmena nebola natoľko veľká ani potrebná, keďže tok programu a zároveň použitie SIMD inštrukcií bolo plne pod kontrolou programátora (nezahŕňajúc *multithreadové* programovanie).

Zmena nastáva pri programovaní využitím GPU. V tomto prípade sa na vstup posunú dáta, ktoré je potrebné spracovať a program, ktorý určuje ich spracovanie. Nasleduje proces spracovania dát, avšak tento nie je vôbec ovplyvňovaný programátorom. Tento proces má plne pod kontrolou grafická karta a jediné, na čo sa môžeme spoľahnúť, je, že po ukončení práce GPU boli všetky dáta zo vstupu spracované daným programom. Tento spôsob práce je diametrálne odlišný od pôvodnej koncepcie CPU a jeho vhodným využitím a kombinovaním práce CPU a GPU je možné dosiahnuť ďaleko lepšie výsledky, čo sa týka rýchlosti spracovania. Táto paradigma sa tiež nazýva Data-parallel Computer.

2.2 Príklady aplikácií

Od počiatkov počítačového veku sa popri vytváraní užitočného softvéru vytvorilo aj kvantum softvéru, ktorého hlavnou, hoci nie jedinou úlohou, bolo odpútanie a rozptýlenie používateľov počítačov, teda hier. Ako sa vyvíjal hardvér, tomu sa prispôsoboval aj vývoj hier. Z pôvodne textových alebo jednoduchých grafických hier sa do dnešnej doby vyvinuli hry, ktoré svojím spracovaním a kvalitou zobrazovania môžu súperiť s filmami. Kvalita zobrazovania je v súčasných počítačoch úzko spätá s komponentom, ktorý má na starosti spracovanie obrazových dát, teda s grafickou kartou. Od dôb začiatkov IT sa tvorba hier natoľko rozšírila, že v súčasnosti sa hovorí o hernom priemysle ako o samostatnej časti IT. Tento má takú silu, že spätne ovplyvňuje vývoj grafických kariet a iných komponentov priamo vplývajúcich na možnosti hier.

Možno je využitie grafických kariet pri hraní hier jeho najznámejšou aplikáciu, zďaleka nie však jedinou. Zo spôsobu práce grafického procesora vyplýva, že najlepšie výsledky je možné dosiahnuť pri spracovaní dát, ktoré na sebe nezávisia a pri ktorých sa najlepšie využije paralelizmus GPU. Touto vlastnosťou sa vyznačujú obrázky alebo videá, čo je vlastne aj pôvodný objekt spracovávaný pomocou GPU. Iným využitím je akcelerovaný desktop, napr. Aero pre Windows Vista alebo Compiz pre prostredie Linux-u. Ďalšie oblasti sú spracovávanie audio signálu, výpočet FFT alebo *raytracing*. Inými príkladmi sú napr. predpoveď počasia, modelovanie molekúl, simulácia neurónových sietí alebo kryptografia. Jednou z najnovších aplikácií je aj vizualizácia nálezísk ropy a plynu [18].

2.3 Príbuzné aplikácie

Zadanie tejto práce, Simulácia tekutín pomocou grafického procesora v reálnom čase, patrí do veľkej skupiny prác s témou fyzikálnych simulácií využitím grafického procesora. Mnohé z nich stavajú na prelomovej práci J. Stam-a, Stable Fluids [20]. Práca An

improved study of real-time fluid simulation on GPU [21] pridáva do simulácie prekážky a optimalizuje spôsob výpočtov na GPU. V práci Real-Time Cloud Simulation and Rendering [25] sa modifikácia algoritmu používa pri simulácii oblakov. Práca Real-Time Simulation and Rendering of 3D Fluids [26] obsahuje simuláciu vody a dymu ako tekutín s rôznymi vlastnosťami. Výsledky tejto práce boli použité ako interaktívnom demo pri prezentácii kariet NVIDIA GeForce 8800. Snímky obrazoviek príbuzných aplikácií sa nachádzajú v Prílohe A. Táto práca sa na rozdiel od vyššie uvedených zameriava viac na porovnanie výkonnosti CPU a GPU na všeobecné výpočty, ako na samotnú simuláciu tekutín v reálnom čase.

3 Použitie GPU na všeobecné výpočty

Využitie GPU na všeobecné výpočty sa označuje pod skratkou *GPGPU – General-Purpose computation on GPU* [9]. Konkrétne spôsoby a implementácie sa môžu líšiť použitou technológiou alebo spôsobom výpočtov. Jedno však majú všetky implementácie rovnaké a tým je skutočnosť, že všetky operácie je potrebné prispôsobiť architektúre grafického subsystému a grafickej karty samotnej. Táto práca sa zameriava na využitie grafického systému OpenGL.

3.1 Špecifiká programovania grafického procesora

Každý vrchol, ktorý vstúpi do spracovania grafického procesora, je najprv spracovávaný *vertex shaderom*, ktorý, pokiaľ je použitá *fixed pipeline*, vykoná transformáciu zo súradníc objektu do súradníc kamery, uskutoční orezanie hranolom pohľadu, príp. ďalšie definované operácie a pomocou projekčnej matice transformuje bod do súradníc okna. Tieto operácie sú na účely použitia GPU na všeobecné výpočty zbytočné. Je potrebné dosiahnuť to, aby *fragment shader* spracoval všetky alebo časť bodov obrazu a využitím textúr ako úložiska dát vykonal potrebné operácie. To sa zabezpečí tak, že *modelview* matica (kombinovaná matica modelu a pohľadu) sa nastaví na jednotkovú, projekčná matica (určuje spôsob zobrazenia – projekciu) na ortogonálnu a na obrazovku sa vykreslí jeden obdĺžnik alebo štvorec potrebných rozmerov. Tým je zabezpečené, aby každý pixel odpovedal jednému fragmentu textúry a nedochádzalo ku interpoláciám.

Spracovanie *fragmentu* a teda aj výstupu je plne v réžii *fragment shadera*. Na správne fungovanie programu treba zabezpečiť vhodné nastavenie *Z-buffera*, *stencil-buffera*, miešania (*blending*), testovania priehľadnosti (*alfa-masking*) a pod.

Oproti predchádzajúcim verziám grafických procesorov, najnovšie verzie podporujú vetvenie aj cykly. Týmto sa *fragment shadery* svojou komplexnosťou vyrovnali programom pre CPU.

3.1.1 Príklad výpočtu

Na preukázanie špecifík programovania GPU bude použitá ukážka sčítania N polí A_1 až A_N o M prvkoch, kde výsledok je posledné pole obsahujúce súčet prvkov predchádzajúcich polí:

$$\forall j \in \{1...M\} : A_N[j] = \sum_{i=1}^{N-1} A_i[j] \quad (1)$$

Je potrebných N-1 textúr na uloženie obsahu polí A_1 až A_{N-1}, hodnota výsledku bude uložená vo *framebufferi*. Rozmer textúr a *framebuffera* musí byť dostatočne veľký na poňatie M prvkov, uvažujúc štvorcové textúry, tak $ceil\left(\sqrt{M}\right)$. Keďže sú textúry v pamäti reprezentované jednorozmerným poľom, nemusia byť v tomto prípade realizované žiadne

transformácie, stačí obsah polí skopírovať do textúr. Čo je však potrebné zachovať, je rovnaký typ dát v textúre (*byte*, *float* a pod.), ako je typ položiek v poliach. Po inicializovaní grafického systému, nastavení vhodného *fragment shadera* a vykreslení obdĺžnika o rozmeroch textúry je vo *framebufferi* výsledok sčítania. Následne stačí skopírovať obsah *framebuffera* do textúry v pamäti. Kód (GLSL) *fragment shadera* pre 3 textúry:

```
// parametre nastavené pred renderovaním
uniform sampler2D texture1;
uniform sampler2D texture2;

void main(void)
{
    float o1 = texture2D(texture1, gl_TexCoord[0].st).x;
    float o2 = texture2D(texture2, gl_TexCoord[0].st).x;

    gl_FragColor.x = o1 + o2;
}
```

Na uloženie výpočtov je možné namiesto *framebuffera* použiť aj textúru. Ako výstupnú textúru sa odporúča použitie novej textúry, nie niektorej z existujúcich, použitých na vstup dát. Takto bude pracovná sada textúr pozostávať iba z read-only a write-only textúr. Pri použití read-write textúr by prístup ku nim musel byť synchronizovaný, čo by nevyhnutne viedlo k spomaleniu spracovania.

3.2 Programovacie jazyky pre OpenGL

Existuje viacero jazykov na definíciu jednotlivých *shaderov* (teda programov). Postupom času sa vyvíjali, podobne ako klasické programovacie jazyky, od najnižších po relatívne komplexné. Najnižším je ARB - OpenGL Assembly Language, vyššími sú jazyk od spoločnosti NVIDIA – Cg (C for Graphics), jazyk so syntaxou podobnou programovaciemu jazyku C – GLSL (OpenGL Shading Language) alebo najnovšie NVIDIA CUDA (Compute Unified Device Architecture). Okrem nich existuje ešte mnoho ďalších. Tie sú buď pre grafický systém MS DirectX, napr. HLSL (High Level Shading Language) alebo sú nezávislé od grafického subsystému, napr. BrookGPU. Dôvodom rozšírenia OpenGL ako prostredia pre všeobecné výpočty je jeho platformová nezávislosť.

3.2.1 ARB - OpenGL Assembly Language

Jazyk veľmi podobný assembleru pre CPU je najnižším a prvým z existujúcich implementácií programovania na GPU pre OpenGL. Je podporovaný priamo v implementácii OpenGL, na jeho funkčnosť nie sú potrebné žiadne ďalšie knižnice. Zápis programu pozostáva zo zápisu inštrukcií, každej na jednom riadku.

Príklad *vertex programu*:

```
!!ARBvp1.0
TEMP vertexClip;
DP4 vertexClip.x, state.matrix.mvp.row[0], vertex.position;
DP4 vertexClip.y, state.matrix.mvp.row[1], vertex.position;
DP4 vertexClip.z, state.matrix.mvp.row[2], vertex.position;
DP4 vertexClip.w, state.matrix.mvp.row[3], vertex.position;
MOV result.position, vertexClip;
MOV result.color, vertex.color;
MOV result.texcoord[0], vertex.texcoord;
END
```

Príklad *fragment programu*:

```
!!ARBfp1.0
TEMP color;
MUL color, fragment.texcoord[0].y, 2.0;
ADD color, 1.0, -color;
ABS color, color;
ADD result.color, 1.0, -color;
MOV result.color.a, 1.0;
END
```

Výhody	**Nevýhody**
• Dostupný priamo v OpenGL API	• Ťažko čitateľný • Zápis programu v inštrukciách

3.2.2 Cg

Firma NVIDIA, ako veľký hráč na poli výrobcov grafických kariet, spoločne s firmou Microsoft, vytvorila sadu nástrojov na programovanie GPU [6]. Tento takisto špecifikuje vlastný jazyk. Na využitie tejto technológie je potrebné použiť špeciálne knižnice dodávané spoločnosťou NVIDIA.

Príklad *vertex shadera*:

```
// vstupný vrchol
struct VertIn {
    float4 pos   : POSITION;
    float4 color : COLOR0;
};

// výstupný vrchol
struct VertOut {
    float4 pos   : POSITION;
    float4 color : COLOR0;
};

// vstupný bod vertex shadera
VertOut main(VertIn IN, uniform float4x4 modelViewProj) {
    VertOut OUT;
    OUT.pos     = mul(modelViewProj, IN.pos); // vypočítat koordináty
výstupného vrchola
    OUT.color   = IN.color; // skopíruj vstupnú farbu do výstupnej
    OUT.color.b = 1.0f; // modrá zložka farby = 1.0f
    return OUT;
}
```

Výhody	Nevýhody
• Vysokoúrovňový jazyk	• Proprietárna technológia • Knižnica od NVIDIA-e

3.2.3 GLSL

Vyšší programovací jazyk vytvorený komunitou OpenGL. Jeho celý názov je OpenGL Shading Language [12]. Zápisom a konštrukciami sa podobá na jazyk C. Technológia DirectX pozná podobný jazyk, známy pod skratkou HLSL (High Level Shading Language). Na označenie kódu sa používa pojem *shader*, namiesto pojmu *program* pri assembleri pre OpenGL. Tým sa zdôrazňuje fakt, že ide o vysokoúrovňový jazyk.

Príklad *vertex shadera* simulujúceho činnosť *fixed pipeline* využitím vstavaných funkcií jazyka (konkrétne ftransform()):

```
void main(void)
{
    gl_Position = ftransform();
}
```

Príklad *fragment shadera* simulujúceho činnosť *fixed pipeline* (nastavenie červenej ako farby fragmentu):

```
void main(void)
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Výhody	Nevýhody
• Dostupný priamo v OpenGL API • Vysokoúrovňový jazyk	• Viazaný obmedzeniami programovania GPU

3.2.4 NVIDIA CUDA

Ďalší z radu technológií firmy NVIDIA, NVIDIA CUDA, celým názvom Compute Unified Device Architecture [7], ponúka API umožňujúce paralelné výpočty, čím odbúrava potrebu znalosti vytvárania *fragment shaderov*. Programovanie abstrahuje od programovania a približuje ho klasickému používaniu funkcií knižnice v jazyku C. Obmedzením technológie CUDA je skutočnosť, že je podporovaná iba hardvérom poslednej, 8. generácie grafických kariet spoločnosti NVIDIA, GeForce.

Výňatok z programu využívajúceho knižnicu CUDA. Definícia funkcie pre paralelné spracovanie:

```
// definícia funkcie na pre paralelné spracovanie
void histogram64CPU(unsigned int *h_Result, unsigned int *h_Data, int
dataN)
{
    int i;
    unsigned int data4;

    for(i = 0; i < BIN_COUNT; i++)
        h_Result[i] = 0;

    for (i = 0; i < dataN; i++){
        data4 = h_Data[i];
        h_Result[(data4 >>  2) & 0x3F]++;
        h_Result[(data4 >> 10) & 0x3F]++;
        h_Result[(data4 >> 18) & 0x3F]++;
        h_Result[(data4 >> 26) & 0x3F]++;
    }
}
```

Výňatok z programu využívajúceho knižnicu CUDA. Príprava a paralelné spracovanie dát:

```
// alokácia pamäte
cudaMalloc((void **)&d_Data, DATA_SIZE);
// skopírovanie obsahuje polí
cudaMemcpy(d_Data, h_Data, DATA_SIZE, cudaMemcpyHostToDevice);
...
// paralelné spracovanie dát pomocou funkcie
histogram64GPU<<<BLOCK_N, THREAD_N>>>(d_Result, (unsigned int
*)d_Data, DATA_N / 4);
...
// skopírovanie výsledkov spracovania
cudaMemcpy(h_ResultGPU, d_Result, RESULT_SIZE,
cudaMemcpyDeviceToHost);
```

Výhody	Nevýhody
• Vysoko-úrovňový jazyk • Nie je viazaný obmedzeniami priameho programovania GPU • Technologicky dokonalejší	• Proprietárna technológia • Knižnica od NVIDIA-e • Iba pre grafické karty NVIDIA GeForce 8x00 a lepšie

3.2.5 Zhodnotenie jazykov

Všetky z opisovaných jazykov majú svoje výhody a nevýhody. Záleží na cieli projektu, ktorý jazyk je pre daný účel najvhodnejší. Napríklad nevýhodou assemblera pre OpenGL je zápis programu pomocou inštrukcií. Hoci je dostupný priamo pomocou OpenGL API, z dôvodu nízkoúrovňovosti jazyka vyplýva nízka efektivita produkcie kódu a teda nevhodnosť na vytváranie komplexných programov. Tento nedostatok je odstránený pri ďalších jazykoch. Na druhej strane CUDA, ktorá abstrahuje od klasického programovania pre grafické procesory, je proprietárna knižnica a je dostupná iba pre najnovší hardvér. Kompromisom medzi nimi je GLSL. Síce je obmedzený programovaním pre GPU, ale je to vysokoúrovňový jazyk a navyše je dostupný priamo cez OpenGL API.

3.3 Riešenie lineárnej algebry na GPU

Použitý jazyk na programovanie GPU ovplyvňuje detaily implementácie programu. Princíp práce zostáva pri každom z jazykov rovnaký. Hlavným rozdielom oproti programovaniu CPU je zmena paradigmy na paralelné programovanie. To znamená, že je potrebné navrhnúť výpočet pre jednu bunku informácie, ktorý bude prevedený nad celou množinou dát.

Ďalším rozdielom je odlišnosť inštrukčného súboru procesorov. Pokiaľ CPU obsahuje rozsiahlu množinu bez špecializácie jednotlivých inštrukcií (snáď s výnimkou SSE a pod., ale aj tie sú relatívne všeobecné), GPU je špecializované na prácu s vektormi, vektorovými a všeobecne matematickými funkciami. Táto skutočnosť vyplýva

z pôvodného určenia grafického procesora, t.j. spracovanie obrazových dát typu RGBA alebo podobných a z typu výpočtov, napr. výpočet osvetlenia, viditeľnosti a pod.. Dá sa však s výhodou využiť aj pri programovaní negrafických výpočtov.

Azda najväčší rozdiel pri programovaní CPU a GPU je v prístupe k dátam a práci s nimi. Pred spustením výpočtov na grafickom procesore je potrebné alokovať všetku potrebnú pamäť, naplniť ju dátami a sprístupniť vhodnou formou. *Fragment shader* síce umožňuje definovanie premenných, ktoré je možné inicializovať pred spustením výpočtov, ich veľkosť a počet je však obmedzený a ako miesto uloženia dát sú teda nevhodné. Jediným vhodným spôsobom (použitím klasických jazykov ako GLSL, Cg a pod., nie CUDA), je použitie textúr. Pri najnovších grafických kartách je možné nastaviť vhodný formát textúry, napr. na 32-bitový typ float (podľa normy IEEE 754-1985 [22]) a textúru používať ako klasické dvojrozmerné pole. Takýmto spôsobom sa textúry pred spracovaním inicializujú dátami z hlavnej pamäte RAM a pomocou operácie vzorkovania sa pristupuje k ich obsahu. Výstupom *fragment shaderov* sú *fragment elementy*, ktoré sú rasterizované na výstupe. Štandardným výstupom je *framebuffer*. OpenGL však podporuje rozšírene, ktorým je možné zmeniť výstup *fragment shadera* a previazať ho s textúrou. Jedná sa o rozšírenie EXT_framebuffer_object [28]. Týmto sa eliminuje potreba kopírovania z *framebuffera* do textúr a pri ďalších výpočtoch stačí zmeniť vstupnú textúru za výstupnú a naopak. Tento spôsob optimalizujúci prácu s textúrami je známy pod menom The Ping-Pong technique.

4 Simulácia tekutín

Vhodným príkladom na porovnanie výkonnosti CPU a GPU je fyzikálna simulácia nestlačiteľných tekutín. Základom sú Navier-Stokesove diferenciálne rovnice opisujúce správanie sa tekutín v čase. Rovnice vychádzajú zo základných zákonov fyziky, ako je zachovanie hmotnosti, rýchlosti a energie a využijú sa na výpočet zmeny rýchlosti a tlaku tekutiny. Z ich diferenciálneho charakteru vyplýva potreba numerického riešenia. To zároveň zaručuje, že tento príklad bude dostatočne výpočtovo náročný na testovanie výkonnosti CPU a GPU. Rozšírenosť fyzikálnych simulácií nie len tekutín využitím GPU demonštruje Príloha A. Tá obsahuje obrázky príkladov aplikácií.

4.1 Navier-Stokes diferenciálne rovnice

Rovnica opisujúca správanie sa nestlačiteľnej tekutiny pozostáva z dvoch častí. Prvá zaručuje (2) zachovanie hmotnosti a druhá zachovanie hybnosti (3) [11]:

$$\nabla \cdot u = 0 \qquad (2)$$

$$\frac{\partial u}{\partial t} = v\nabla^2 u - (u \cdot \nabla)u - \frac{1}{\rho}\nabla p + f \qquad (3)$$

kde u je vektorové pole rýchlosti, p skalárne pole tlaku, t čas, v kinematická viskozita, ρ hustota kvapaliny, f externá sila pôsobiaca na kvapalinu (napr. gravitácia) a ∇ diferenčný operátor:

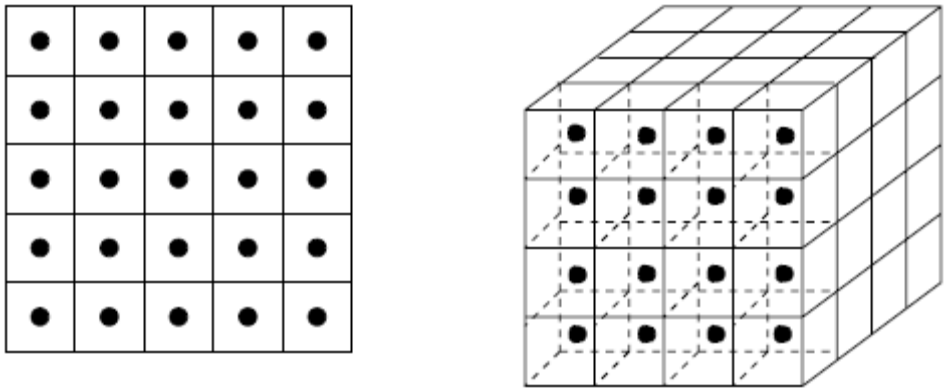
$$\left[\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\right]^T \qquad (4)$$

v prípade 2D simulácie alebo

$$\left[\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right]^T \qquad (5)$$

v prípade 3D simulácie.

Tieto rovnice opisujú zmenu rýchlosti kvapaliny v určitom delta okolí daného bodu. Na počítačovú simuláciu je preto potrebné pristúpiť k diskretizácii priestoru simulácie. Ten je rozdelený na rovnako veľké časti a vytvorí tak mriežku s bunkami, pre ktoré platí, že rýchlosť kvapaliny v celom ich objeme je rovnaká. Pre každú bunku je potrebné vypočítať nové hodnoty tlaku a rýchlosti v nej pre každý krok simulácie. Preto počet buniek priamo úmerne ovplyvňuje rýchlosť simulácie.



Obrázok 7. Mriežka rozdeľujúca priestor simulácie [20]

4.2 Metódy numerického riešenia diferenciálnych rovníc

Prezentované diferenciálne rovnice majú formu:

$$\frac{\partial x}{\partial t} = f(x,t) \qquad (6)$$

Na ich riešenie je možné použiť známe metódy numerických výpočtov, ako Eulerova metóda, Verletova metóda a pod [5]. Stabilita a presnosť riešenia sú určené rádom použitej metódy. Najjednoduchšia z nich je Eulerova metóda. Vzorec na výpočet hodnoty v čase $+\Delta t$ je:

$$x_{n+1} = x_n + f(x_n, t_n)\Delta t \qquad (7)$$

Chyba výpočtu pomocou expanzie Taylorovho radu:

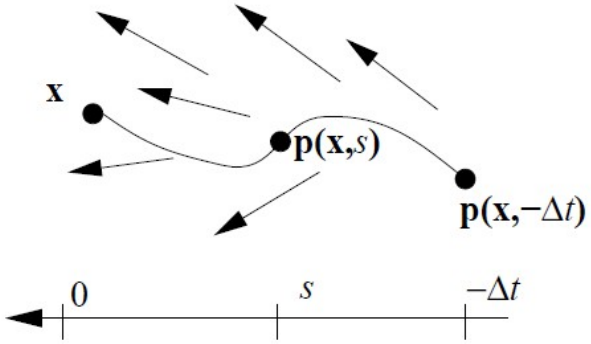
$$\frac{1}{2}(\Delta t)^2 f''(x_n, t_n) + O\left((\Delta t)^3\right) \qquad (8)$$

Pre malé hodnoty kroku Δt je dominantná chyba na krok proporcionálna Δt^2. Na riešenie za čas t je počet potrebných krokov proporcionálny $1/\Delta t$, teda sumárna chyba za konštantný čas bude proporcionálna Δt. Preto sa Eulerova metóda nazýva metódou prvého rádu.

Ostatné metódy, ako Verletova alebo metóda Runge–Kutta, sú oproti Eulerovej metóde zložitejšie, prinášajú však väčšiu presnosť aproximácie. Existujúce práce zaoberajúce sa simuláciou tekutín však nevyužívajú ani jednu z uvedených metód. Hlavným dôvodom je nepresnosť výpočtov, vyúsťujúca do nestability simulácie. V roku 1999 v práci Stable Fluids [20] Jos Stam prezentoval novú metódu, ktorá tento nedostatok odstraňuje. Táto práca sa stala základom mnohých iných zaoberajúcich sa simuláciou tekutín v reálnom čase.

4.3 Metóda charakteristík

Klasické numerické metódy riešenia diferenciálnych rovníc prinášajú chybu, ktorá v čase simulácie nakoniec vyúsťuje v jej nestabilitu. Riešenie tohto problému prináša tzv. metóda charakteristík. Zakladá sa na fakte, že v každom kroku simulácie je smer pohybu častíc kvapaliny určený vektorovým poľom opisujúcim kvapalinu. Na získanie rýchlosti v bode x v novom čase $t + \Delta t$, spätne sledujeme pohyb bodu x pomocou vektorového poľa v čase t po dobu Δt. Takto sa definuje cesta pohybu korešpondujúca prúdnici v danej časti vektorového poľa. Nová rýchlosť v bode x je nastavená na rýchlosť v takom bode, ktorý bol bodom x pred časom Δt, teda v čase t.



Obrázok 8. Zistenie rýchlosti bodu v čase –dt [20]

Na výpočet nových rýchlostí je potrebné zostaviť sústavu lineárnych rovníc opisujúcu vzťah rýchlostí v čase. Rozmery matice týchto rovníc zodpovedajú rozmerom mriežky simulácie. Na získanie riešenia sústavy lineárnych rovníc je potrebné použiť jednu z metód numerického riešenia. Vhodnosť konkrétnej metódy musí byť zvážená s ohľadom na schopnosti grafického procesora, pre ktorý je algoritmus implementovaný. Rozšírený opis metódy charakteristík aj s kompletným matematickým modelom je uvedený v práci a prílohe práce [20].

4.3.1 Metóda riešenia sústavy lineárnych rovníc

Zo známych numerických metód riešenia sústavy lineárnych rovníc je jednou aj Jacobiho iteračná metóda. Táto je špeciálne vhodná pri GPU implementácii, pretože v jednej iterácii nemodifikuje obsah poľa, nad ktorým pracuje. Vzorec na riešenie Jacobiho metódou [24]:

$$x_i^{(k)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(k-1)}}{a_{ii}} \qquad (9)$$

Výňatok z kódu na riešenie sústavy lineárnych rovníc:

```
for ( k = 0 ; k < iter ; k++ ) {
      for ( i = 1 ; i <= N ; i++ ) {
            for ( j = 1 ; j <= N ; j++ ) {
                  t[IX(i,j)] = (x0[IX(i,j)] + a * (x[IX(i-1,j)] +
x[IX(i+1,j)] + x[IX(i,j-1)] + x[IX(i,j+1)])) / (1 + 4 * a);
            }
      }
      set_bnd ( N, b, t, o );
      SWAP(x, t);
}
```

Pri výpočte sa používajú dve polia, pole *x* a *t*. Jedno sa používa ako zdrojové pole a druhé ako pole na uloženie výpočtu – analógia read-only a write-only textúr. Po každej iterácii sa pole vymenia a výpočet pokračuje – obsah polí nie je potrebné kopírovať. Funkcia *set_bnd* je potrebná na interakciu tekutiny s prekážkami. Viac v kapitole 4.3.3.

4.3.2 Časová zložitosť metódy

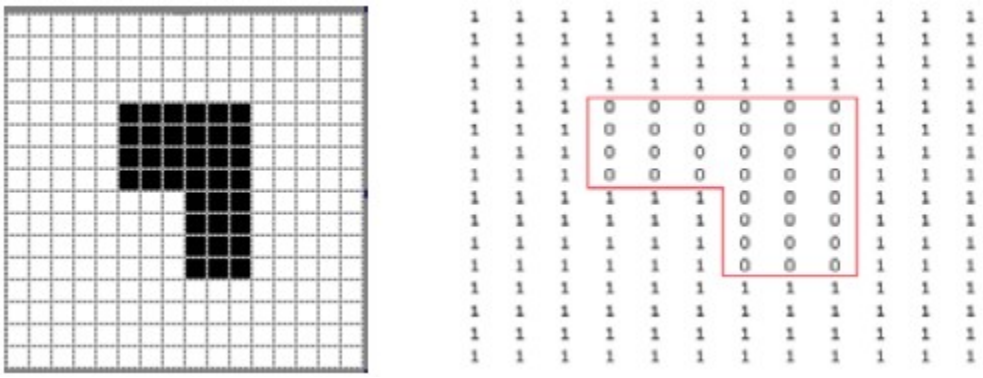
Použitím metódy charakteristík je v každom kroku simulácie zostavená sústava lineárnych rovníc zložená zo súčasných hodnôt rýchlostí a hodnôt rýchlostí v čase $t + \Delta t$. Tieto sú v podobe neznámych.

Pre každú bunku simulovaného priestoru je potrebné vypočítať nové hodnoty ako pre rýchlosť, tak aj pre tlak (implementované v tejto práci). Numerické riešenie lineárnych rovníc sa deje v iteráciách. Ich počet je buď pevne daný, alebo v prípade požiadavky dosiahnutia určitej presnosti premenlivý. Pre potreby vizuálnej prezentácie simulácie je výhodnejšie použiť konštantný počet iterácií. Dôvodom je priorita simulácie v reálnom čase a stabilita rýchlosti pred presnosťou výpočtov.

V prípade dvojrozmerného priestoru je potrebné uskutočniť *(d + 1) * m * n * I* výpočtov (neuvažujem konkrétne operácie), kde *m* a *n* sú rozmery mriežky, *d* je počet dimenzií rýchlosti a *I* počet iterácií. Pri dvojrozmernom priestore a skutočnosti, že I je pre jednotlivé simulácie konštantné, je asymptotická zložitosť O(m * n). V každom kroku je potrebné výpočet sústavy lineárnych rovníc realizovať viackrát, keďže však ide o konštantný počet (v rámci jedného kroku), asymptotická zložitosť zostáva rovnaká.

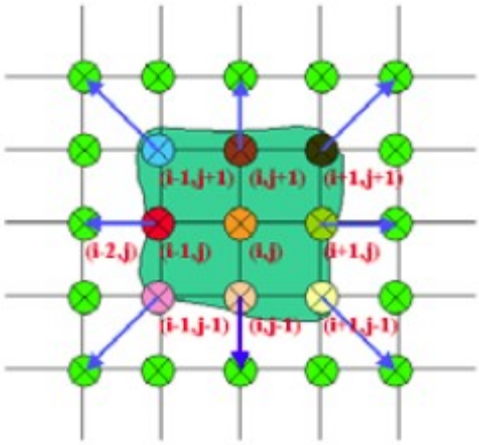
4.3.3 Interakcia s prekážkami

Metóda charakteristík, ako je prezentovaná v práci [20] pôvodne neuvažuje s prekážkami, ale len s ohraničením simulovaného priestoru zvonku. Bolo vyvinutých mnoho spôsobov vloženia prekážok do simulácie a interakcie kvapaliny s nimi. Jedna z nich je uvedená v práci An Improved Study of Real-Time Fluid Simulation on GPU [21]. Na definíciu prekážok je zavedená ďalšia mriežka.



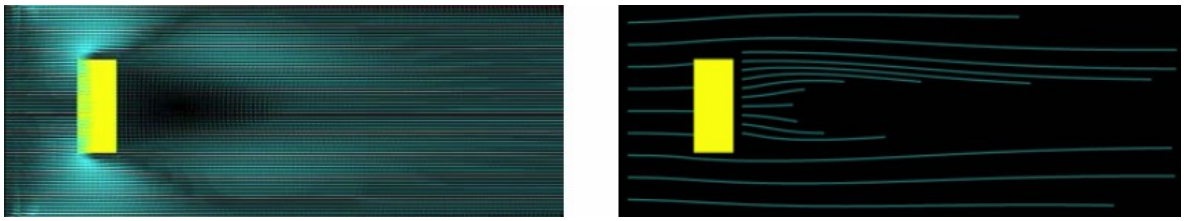
Obrázok 9. Definícia prekážok pomocou mriežky [21]

V tomto prípade je definíciu umiestnenia prekážok možné prenechať na používateľa. Pole s prekážkami je ešte pred spustením simulácie spracované a výsledkom je nové pole definujúce buď objem kvapaliny, objem prekážky alebo styčnú plochu.



Obrázok 10. Pole definujúce plochu rozhrania [21]

Výpočet hodnôt rýchlostí na rozhraní prekážka – kvapalina je potom kontrolovaný v závislosti na type rozhrania. Všeobecne tangenciálna zložka rýchlosti na rozhraní nemôže smerovať do vnútra prekážky. Príklad nekĺzavého rozhrania je na Obrázok 11. Príklad nekĺzavého rozhrania, vektorové pole rýchlostí a prúdnice [21]. V tomto prípade je normálová zložka rýchlosti veľmi malá.



Obrázok 11. Príklad nekĺzavého rozhrania, vektorové pole rýchlostí a prúdnice [21]

4.4 Technologické obmedzenia GPU implementácie

Pre stabilitu simulácie je potrebné, aby prebiehajúce výpočty boli čo najpresnejšie. Definíciou formátu reprezentácie a aritmetiky čísiel s pohyblivou rádovou čiarkou sa zaoberá štandard IEEE 754-1985 [22]. Pri súčasných CPU je konformita so štandardom úplná a sú dostupné všetky formáty: jednoduchá presnosť (32 bitov), dvojitá presnosť (64 bitov) a dvojitá rozšírená presnosť (80 bitov). V niektorých implementáciách jazyka C je dostupná dokonca štvoritá presnosť (128 bitov).

Naproti tomu grafické procesory prinášajú rozšírenú presnosť (64 bitov pri internom spracovaní oproti klasickým 32 bitom na pixel) a pohyblivú rádovú čiarku až s nástupom technológií ako HDR (High dynamic range) a pod. Pre tie je vyššia presnosť výpočtov nevyhnutná. Dovtedy bolo pri podobných výpočtoch potrebné používať buď pevnú rádovú čiarku alebo prevod medzi pohyblivou rádovou čiarkou a celými číslami. Až posledná generácia grafických kariet prináša podporu premenných s pohyblivou rádovou čiarkou s jednoduchou presnosťou (32 bitov) vyhovujúcu štandardu IEEE 754-1985. Preto je tento typ ako jediný vhodný na relevantné porovnanie výkonností procesorov.

5 Ciele práce

Cieľom práce je porovnať výkon GPU implementácie simulácie tekutín s výkonom CPU implementácie. Na to práca vyžaduje implementovanie referenčného CPU výpočtu. Referenčný výpočet by mal byť ľahko transformovateľný na použitie s GPU, aby bolo porovnanie výpočtovej sily čo najvernejšie. Algoritmus je postavený na Navier-Stokes rovniciach, ktoré opisujú správanie sa nestlačiteľnej tekutiny v čase. Požiadavka na beh simulácie v reálnom čase je kritická. Simulácia bude prebiehať v obmedzenom 2D priestore s prekážkami. Program musí umožňovať spustenie simulácie na CPU a GPU (nemusí naraz) a merať rýchlosť simulácie.

Aplikácia implementujúca simuláciu by mala vizualizovať pohyb tekutiny, jej turbulencie, prípadne jej ostatné vlastnosti. Ďalšou požiadavkou na aplikáciu je možnosť používateľa interaktívne pôsobiť na tekutinu, napríklad pridávaním sily. Na získanie presnejších výsledkov je potreba možnosti zmeny rozmerov simulačnej mriežky.

Body špecifikácie potrebné na porovnanie výkonnosti GPU a CPU implementácie:

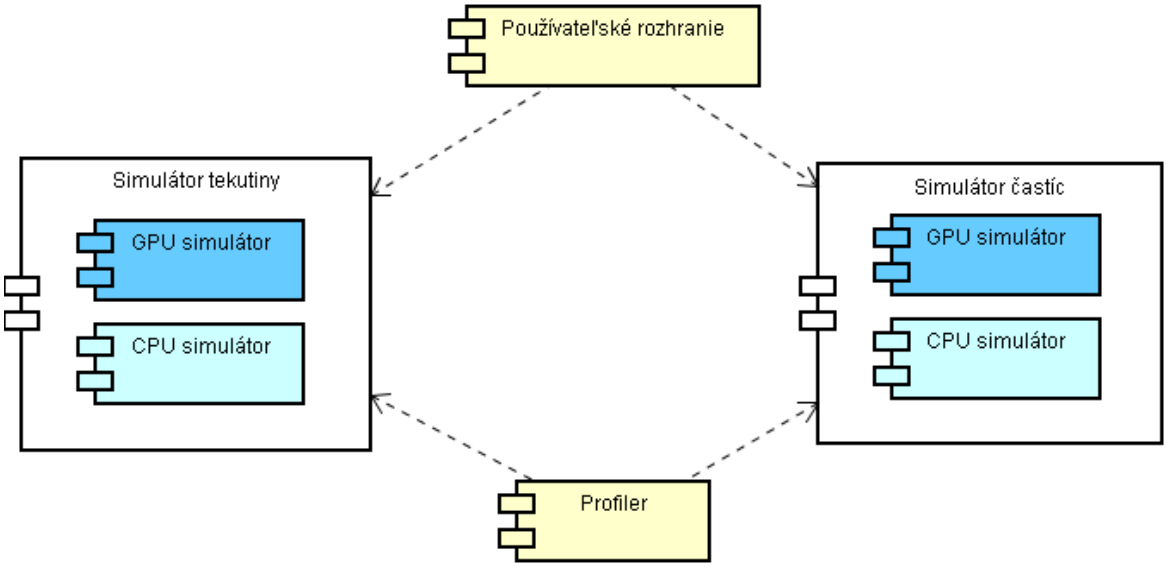
- Simulácia na základe Navier-Stokes rovníc
- Referenčný výpočet na CPU
- Ekvivalentný výpočet na GPU s použitím vybraného jazyka
- Interakcia tekutiny s prekážkami
- Zaznamenávanie rýchlosti simulácie

Ďalšie rozširujúce body špecifikácie:

- Vizualizácia vektorového poľa rýchlosti a skalárneho poľa tlaku
- Jednoduchá vizualizácia tekutiny pomocou častíc unášaných tekutinou
- Pôsobenie sily na tekutinu – interaktívne pridávanie síl alebo tlaku
- Zmena veľkosti simulačnej mriežky

6 Návrh a implementácia

Na základe špecifikácie je možné identifikovať hlavné komponenty systému: používateľské rozhranie vo forme okna aplikácie, simulátor tekutiny, simulátor častíc a profiler na vytváranie benchmarkov, meranie rýchlosti aplikácie. Simulátor tekutiny a simulátor častíc je navyše možné rozdeliť na CPU a GPU implementáciu.



Obrázok 12. Diagram komponentov systému

Simulácia tekutiny využije metódu charakteristík opísanú v kapitole 4.3. Použije sa upravená podoba Navier-Stokes rovníc, ako je uvedené v práci Real-Time Fluid Dynamics for Games [19]:

$$\frac{\partial u}{\partial t} = \nu \nabla^2 u - (u \cdot \nabla) u + f \quad (10)$$

pre rýchlosť a

$$\frac{\partial p}{\partial t} = k \nabla^2 p - (u \cdot \nabla) p + S \quad (11)$$

pre tlak, kde *k* je difúznosť kvapaliny a S zmena tlaku v kvapaline.

Referenčná implementácia na riešenie sústavy lineárnych rovníc používa Gauss-Seidelovu iteračnú metódu [23]. Táto pri výpočte modifikuje pole, nad ktorým práve pracuje. Vzhľadom na spôsob práce GPU priblížený v kapitole 3.1.1, konkrétne použitie iba read-only a write-only textúr, je výhodnejšie použiť Jacobiho iteračnú metódu [24]. Tá nemodifikuje obsah polí.

Algoritmus bude rozšírený o podporu interakcie tekutiny s prekážkami. Prekážky budú statické a budú definované pred spustením simulácie. Poloha prekážok bude načítaná z externého súboru. Dobrý základ na spôsob implementácie interakcie s prekážkami poskytuje práca [21]. Program umožní definovanie síl pôsobiacich v tekutine takisto

pomocou externého súboru a ďalej bude program podporovať zmenu veľkosti mriežky simulácie.

Implementovaný algoritmus pre CPU bude jednovláknový. Úprava na podporu viacerých vlákien nie je z hľadiska náročnosti implementácie výhodná a z hľadiska porovnania výkonnosti procesorov ani potrebná.

Simulátor častíc obsahuje množinu častíc – bodov, ktoré sa pohybujú v priestore simulácie. Každú časticu charakterizuje iba jej poloha. Rýchlosť, ktorou sa má častica pohybovať, je určená miestom, kde sa aktuálne nachádza. Keďže poloha častíc je určená s väčšou presnosťou oproti presnosti simulácie určenej rozmermi mriežky, rýchlosť častice je interpolovaná z rýchlostí v najbližšom okolí častice (lineárna interpolácia zo 4 okolitých buniek simulácie). Počet častíc bude možné meniť pomocou parametra príkazového riadka.

Profiler je komponent systému, ktorého úlohou je merať čas trvania jednotlivých výpočtov. Tie sú CPU alebo GPU simulácia tekutiny a častíc. Pri GPU simulácii sa bude navyše zaznamenávať aj čas potrebný na presun dát z a do hlavnej pamäte do a z pamäte grafického adaptéra. Na čo najpresnejšie merania bude použitý časovač s vysokým rozlíšením – konkrétna implementácia bude závislá na platforme. Profiler bude poskytovať 2 druhy výstupov, prvý, použitý pri interaktívnej simulácii, bude vypisovať v čitateľnej forme všetky merané údaje. Druhý, použitý pri benchmarkovom behu, na záver simulácie vypíše všetky dáta v CSV formáte (comma separated values).

Príklad výstupu profilera pri interaktívnom behu programu:

```
NSE CPU/GPU solver (by Ondrej Hirjak, 2008)
=== Simulation settings ==========================
Jacobi solver iterations = 20
Particles count = 100000
Grid size = 64x64
Using GPU: yes
Benchmark mode: no
==================================================
```

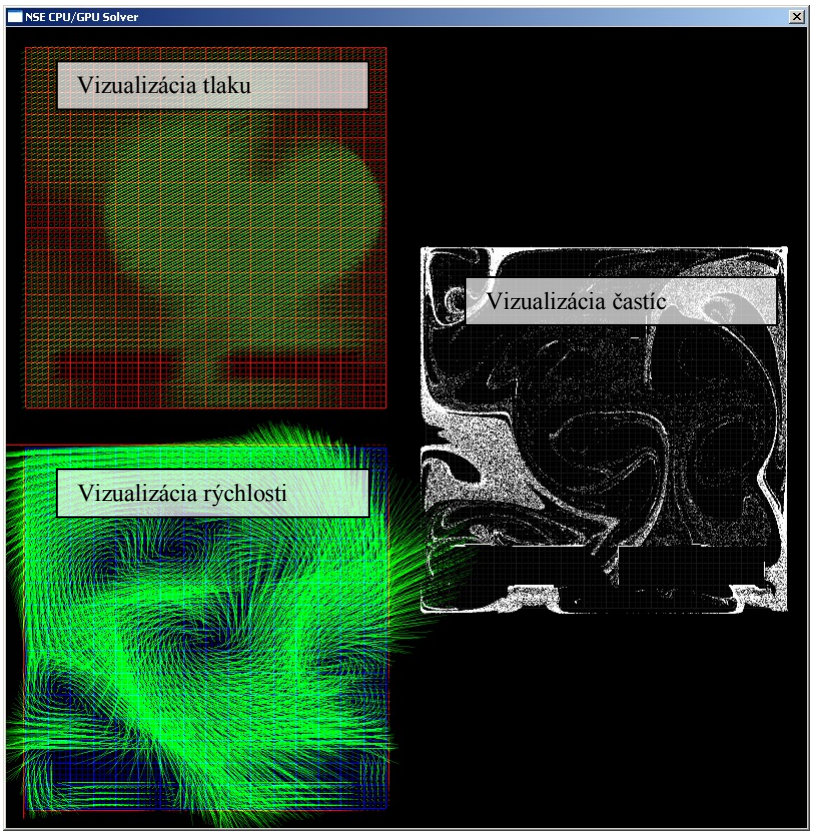
Tieto informácie sa vypíšu pri štarte programu a informujú o parametroch simulácie, konkrétne: počet iterácií, počet častíc, rozmer simulačnej mriežky, použitie GPU alebo CPU na simuláciu a informácii o tom, že program bol spustený v interaktívnom móde. Nasledujúci výpis sa opakuje v pravidelných intervaloch a informuje o aktuálnom stave simulácie. Zobrazuje aktuálnu rýchlosť simulácie v obrázkoch za sekundu (fps) a podrobný zoznam jednotlivých profilovaných činností. Každá položka obsahuje celkový počet milisekúnd strávený v danej časti programu, počet spustení časti programu, minimálny a priemerný čas vykonávania daného kódu.

```
fps = 216.46
=== Profiler report ========================================
Name             |   Sum (ms) | Samples | Min (ms) | Avg. (ms)
-----------------|------------|---------|----------|----------
        Renderer |     568.56 |     599 |     0.78 |     0.95
  NSE solver GPU |    1478.42 |     600 |     2.29 |     2.46
NSE transfer M2G |     165.38 |     600 |     0.26 |     0.28
NSE transfer G2M |     528.73 |     600 |     0.73 |     0.88
   Particles GPU |     891.07 |     600 |     1.42 |     1.49
Par transfer M2G |     238.23 |     600 |     0.36 |     0.40
Par transfer G2M |     593.66 |     600 |     0.83 |     0.99
      Frame time |    3291.62 |     599 |     4.53 |     5.50
============================================================
```

Príklad výstupu profilera v benchmarkingov móde programu – tento výpis sa zobrazí iba raz počas behu programu, a to po skončení simulácie:

```
64;20;100000;2,25;2,34;0,24;0,25;0,78;0,85;1,43;1,48;0,38;0,41;0,83;0,99;3,70;3,82;
GPU
```

V poradí: rozmer mriežky, počet iterácií, počet častíc, simulácia tekutiny (minimum), simulácia tekutiny (priemer), prenos dát tekutiny RAM -> GPU (minimum), prenos dát tekutiny RAM -> GPU (priemer), prenos dát tekutiny GPU -> RAM (minimum), prenos dát tekutiny GPU -> RAM (priemer), simulácia častíc (minimum), simulácia častíc (priemer), prenos dát častíc RAM -> GPU (minimum), prenos dát častíc RAM -> GPU (priemer), prenos dát častíc GPU -> RAM (minimum), prenos dát častíc GPU -> RAM (priemer), čas celého jedného kroku simulácie (minimum), čas celého jedného kroku simulácie (priemer), typ simulácie. Tieto údaje budú v časti testovania spracované a vyhodnotené.



Obrázok 13. Grafické rozhranie programu

6.1 Implementácia

V rámci diplomovej práce bola implementovaná CPU aj GPU implementácia simulácie tekutín. Simulácia prebieha v 2D priestore, v ktorom sú staticky umiestnené prekážky a umiestnené preddefinované sily. Prekážky aj sily je možné definovať pomocou externých súborov. Program vizualizuje priebeh simulácie a priebežne podáva správu o dĺžke trvania jednotlivých úloh.

Grafický subsystém potrebný na vizualizáciu aj samotný výpočet simulácie je OpenGL. Dôvodom je multiplatformovosť riešenia. Jazykom na implementáciu algoritmu pre GPU je GLSL. Ten je dostupný v každej implementácii OpenGL vyhovujúcej definícii *programmable pipeline* a jeho podobnosť s jazykom C podporí názornosť rozdielov v CPU a GPU implementácii. Program využíva knižnicu GLEW [29] na prístup k rozšíreniam OpenGL. Grafické rozhranie s vizualizáciou tlaku, rýchlosti a pohybu častíc je na Obrázok 13. Grafické rozhranie programu.

V ľavej-hornej časti okna programu je skalárne pole tlaku a v ľavej-dolnej časti okna je vektorové pole rýchlosti. V pravej časti je vizualizovaný samotný priestor simulácie. Na zobrazenie sa používa množina častíc, ktoré sú unášané tekutinou, ale samotnú simuláciu neovplyvňujú. Ich pohybom sa vytvárajú vzory, ktoré umožňujú sledovať prúdnice. Program podporuje interaktívne pridávanie sily a tlaku pomocou tlačidla myši.

Program do konzoly vypisuje rýchlosť simulácie a veľkosť aktuálneho kroku. Okrem času potrebného na výpočet simulácie tekutiny meria aj čas potrebný na výpočet pohybu častíc a renderovanie grafického subsystému. Ukazuje sa, že vzhľadom na množstvo času

venovaného simulácii častíc, je aj pre tento výpočet vhodné uvažovať nad využitím GPU. Fakt, že častice sa pri svojom pohybe navzájom neovplyvňujú a teda aj výpočet ich pohybu je nezávislý, túto domnienku len potvrdzuje. Simulácia častíc obsahuje detekciu kolízií s prekážkami.

Zmena veľkosti simulačnej mriežky a počtu iterácií pri riešení sústavy lineárnych rovníc je možná pomocou parametrov príkazového. Veľkosť kroku simulácie Δt je prispôsobená rýchlosti simulácie, aby celková rýchlosť zodpovedala reálnemu času.

Podrobnejšie informácie o implementácii programu sa nachádzajú v Prílohe B.

6.2 Systémové požiadavky

CPU implementácia algoritmu nie je obmedzená žiadnou verziu procesora. Explicitne nepoužíva žiadne špeciálne SIMD inštrukcie (výsledný kód programu závisí od použitého kompilátora a parametrov kompilácie). Hlavným faktorom obmedzujúcim rýchlosť simulácie je rýchlosť procesora. Počet jadier v prípade *singlethreadovej* implementácie takisto nehrá žiadnu podstatnú rolu (pri zaťažení počítača iba touto aplikáciou, inak je rozdiel znateľný).

Naproti tomu má GPU implementácia striktnejšie obmedzenia. Od grafickej karty požaduje podporu *programmable pipeline* a podporu *fragment shaderov*. V závislosti od konkrétnej implementácie je minimálna potrebná grafická karta NVIDIA GeForce FX alebo ATI Radeon 9x00 (alebo iná podporujúca GLSL, teda OpenGL 2). V prípade GPU implementácie má na rýchlosť simulácie vplyv počet *fragment* alebo *stream procesorov* (najnovšia generácia grafických kariet s univerzálnymi procesormi) a ich rýchlosť. V oboch prípadoch sa dá spoliehať na fakt, že úzkym hrdlom bude procesor a nie napríklad rýchlosť pamäte alebo zbernice. Dôvodom je množstvo a náročnosť výpočtov oproti objemu presúvaných dát.

Keďže použitým grafickým subsystémom je OpenGL, program nie je viazaný na žiadnu konkrétnu softvérovú platformu. Jeho použiteľnosť závisí iba od úrovne konkrétnej implementácie OpenGL.

7 Experimenty

V rámci činnosti na práci bol implementovaný program na simuláciu tekutiny v reálnom čase. Jedna z vlastností programu je schopnosť zaznamenávať rýchlosť simulácie aj jej jednotlivých častíc. Skutočnosť, že program podporuje zmenu parametrov simulácie, umožňuje vytvoriť sadu experimentov = testov = benchmarkových behov programu. Výsledky týchto behov budú použité na zhodnotenie rozdielu v rýchlosti CPU a GPU simulácie a rýchlosti simulácie na rôznom type hardvéru.

7.1 Výstupy programu

Počas behu simulácie profiler zaznamenáva dĺžku behu jednotlivých častí programu (v milisekundách, vo výstupe vypisuje minimálnu a priemernú dĺžku – na účely testovania je relevantná priemerná dĺžka). V prípade CPU simulácie to je:

- Simulácia tekutiny
- Simulácia častíc
- Celková dĺžka behu jednej iterácie (frame time)

Dĺžka behu jednej iterácie je doba medzi prekresleniami obrazovky pri behu programu. Inverzná hodnota dĺžky behu jednej iterácie je rovná počtu snímok za sekundu (fps = frames per second). Za beh v reálnom čase sa považuje beh programu s minimálnou rýchlosťou 24 snímok za sekundu. To je dosiahnuté vtedy, keď dĺžka behu jednej iterácie klesne pod 41,67 milisekúnd.

Časti programu pri GPU simulácii:

- Simulácia tekutiny
- Transfer dát z RAM na grafickú kartu (simulácia tekutiny)
- Transfer dát z grafickej karty do RAM (simulácia tekutiny)
- Simulácia častíc
- Transfer dát z RAM na grafickú kartu (simulácia častíc)
- Transfer dát z grafickej karty do RAM (simulácia častíc)
- Celková dĺžka behu jednej iterácie (frame time)

Novými položkami oproti simulácii na CPU je doba transferu dát medzi RAM a grafickou kartou. To platí ako pre simuláciu tekutiny, tak aj pre simuláciu častíc. Táto doba nie je z hľadiska dĺžky trvania simulácie zanedbateľná a je vhodné ju samostatne merať.

7.2 Návrh experimentov

Poskytované výstupy programu dovoľujú vytvorenie viacerých typov testov. Najpodstatnejším z nich je testovanie dĺžky vykonávania jednej iterácie, teda zistenie schopnosti behu programu v reálnom čase. Ďalším typom je meranie dĺžky behu

jednotlivých simulácií. Na základe meraní je možné porovnať výkonnosť jednotlivých procesorov a určiť zrýchlenie GPU simulácie oproti jej CPU implementácii. Z hľadiska efektívnosti GPU implementácie je zaujímavé zhodnotiť množstvo času potrebné na transfer dát medzi grafickou kartou a pamäťou RAM oproti samotným výpočtom na GPU.

Na zistenie hladiny real-time-mového behu simulácie je potrebné zistiť vhodné nastavenia parametrov simulácie. Najdôležitejšími parametrami, ktoré ovplyvňujú rýchlosť simulácie, sú: počet iterácii pri riešení Navier-Stokes rovníc, rozmer mriežky simulácie a počet simulovaných častíc. Ostatné parametre ako pridanie súboru s prekážkami alebo externými silami nemajú na rýchlosť simulácie znateľný vplyv. Na vyhodnotenie veľkosti vplyvu zmeny daného parametra na rýchlosť simulácie je potrebné benchmarking spustiť viacnásobne vždy s inou hodnotou jedného parametra z vhodne zvoleného intervalu. Experimentálne boli zistené hranice, ktoré svojim rozsahom pokryjú väčšinu hardvéru a teda testovanie bude možné púšťať automatizovane. Pri testovaní sa bude meniť iba jeden parameter, ostatné budú počas behov programu rovnaké.

Štandardné parametre simulácie	
Počet iterácií	20
Rozmery mriežky (? x ?)	64
Počet častíc (v tisícoch)	100

Premenlivé parametre simulácie	
GPU simulácia	
Počet iterácií	10, 20, 30, 40, 50, 60, 70, 80, 90, 100
Rozmery mriežky (? x ?)	16, 32, 64, 128, 192, 256, 320, 384, 448, 512
Počet častíc (v tisícoch)	100, 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000
CPU simulácia	
Počet iterácií	10, 20, 30, 40, 50, 60, 70, 80, 90, 100
Rozmery mriežky (? x ?)	16, 32, 64, 96, 128, 160, 192, 256
Počet častíc (v tisícoch)	50, 100, 200, 300, 400, 500

Rozdiely v rozsahoch testovaných hodnôt medzi CPU a GPU sú z dôvodu relatívne nízkej rýchlosti CPU implementácie. Preto sú intervaly menšie a rozdiel hodnôt jemnejší.

Podmienka rýchlosti behu simulácie	
Dĺžka trvania jedného kroku	max. 41,67 ms (milisekúnd)

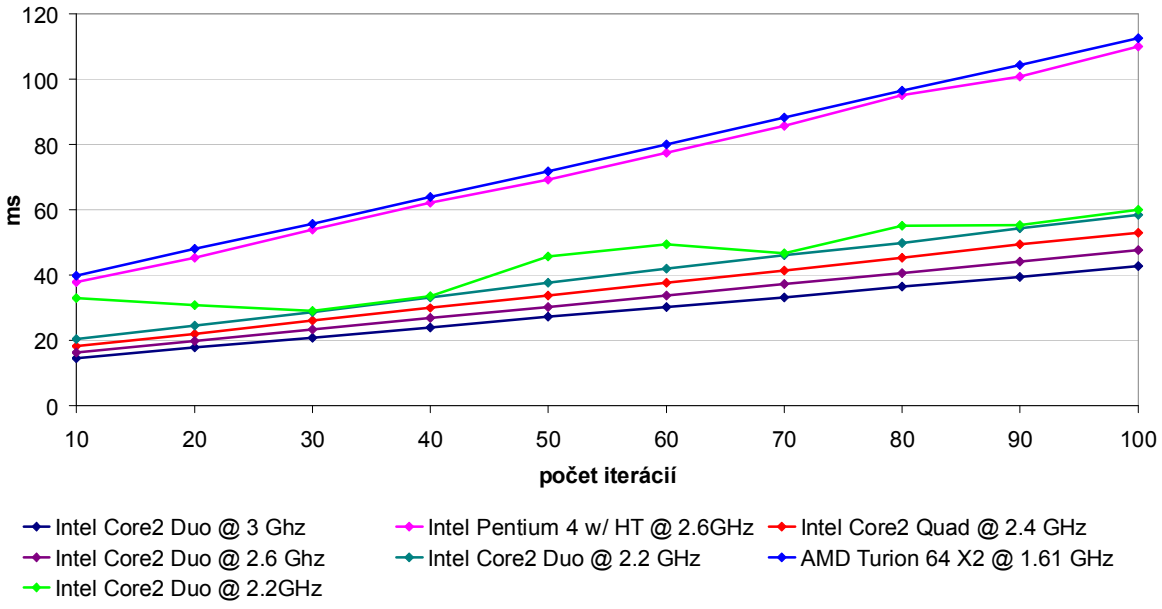
7.2.1 Spôsob testovania

Na získanie všetkých potrebných výsledkov je potrebné program mnohonásobne pustiť, navyše s rôznymi parametrami. Aby bolo testovanie automatické a s čo najmenšou interakciou používateľa, bol vytvorený skript, ktorý program spustí a výsledky aj s hlavičkou zapíše do súboru. Tento je vo vhodnom formáte (CSV – comma separated values) a po drobných úpravách ho je možné otvoriť v tabuľkovom editore a výsledky spracovať.

7.3 Získané výsledky

Vytvorená množina testov bola spustená na rozličnom hardvéri v rôznych konfiguráciách. Kompletný zoznam hardvéru použitého na testovanie je v Prílohe D. Táto kapitola zhŕňa výsledky a prezentuje ich vo forme grafov. Pre každý typ testu (test počtu iterácií, test rozmerov mriežky, test počtu častíc) bol vytvorený graf samostatne pre CPU a GPU implementáciu. Na základe týchto grafov boli vynesené grafy spájajúce obe implementácie, s tým, že v týchto grafoch sú uvedené najlepšie aj najhoršie výsledky z oboch implementácií. Prvým testom bol test počtu iterácií.

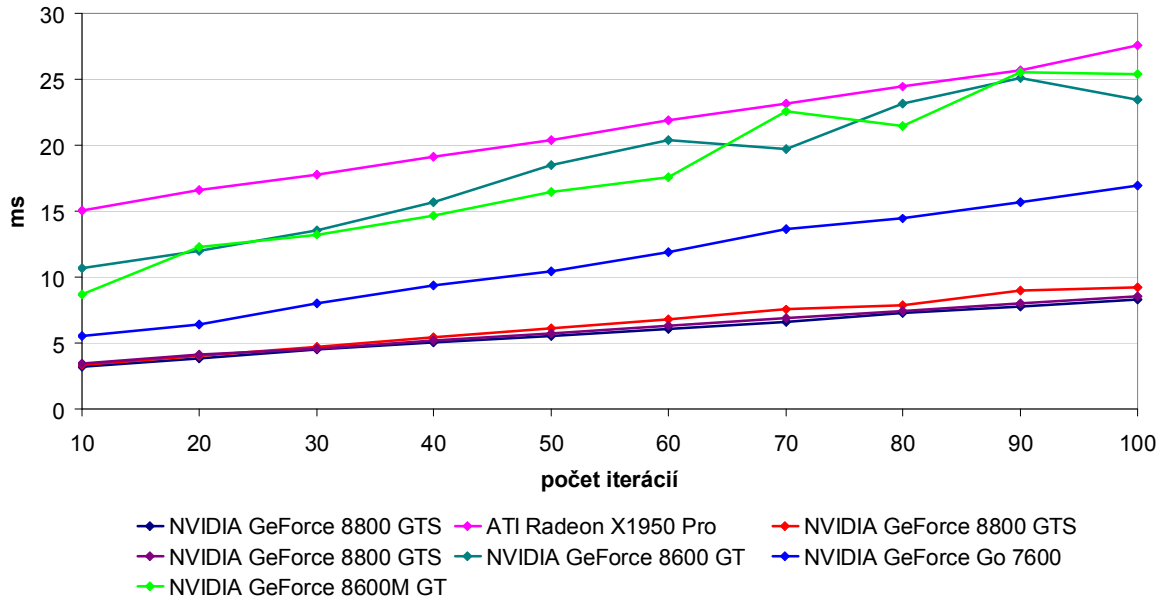
7.3.1 Počet iterácií



Obrázok 14. Test počtu iterácií (CPU implementácia)

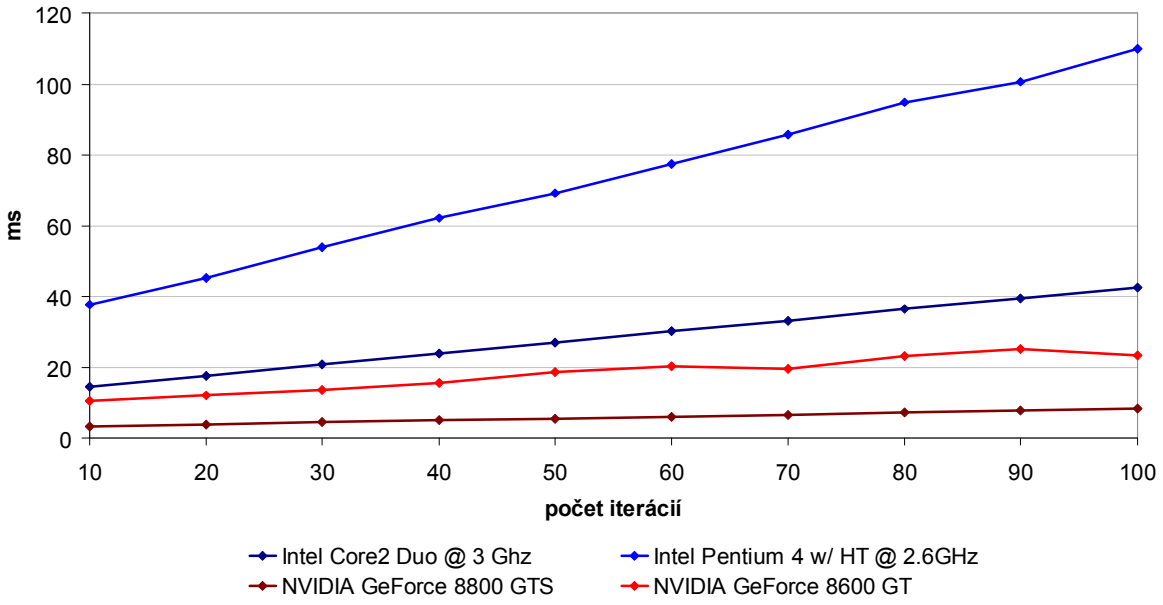
Dĺžka kroku simulácie so zvýšením počtu iterácií zreteľne stúpa. Na dosiahnutie behu simulácie v reálnom čase je pri pomalších procesoroch možné iba pri minimálnom počte iterácií. Pri novších procesoroch je možné zvýšiť počet iterácií až na 60 – 80. V praktickom meradle však zlepšenie realistickosti simulácie pri zvýšení počtu iterácií nie je natoľko výrazné, aby stálo za nepomerne vyšší nárast výpočtovej zložitosti. Nejednoznačný nárast dĺžky jedného kroku pri niektorých z testov môže byť spôsobený

momentálnym zaťažením testovaného počítača, keďže sa jednalo o testy pri normálnej prevádzke systému.



Obrázok 15. Test počtu iterácií (GPU implementácia)

GPU implementácia so zvyšovaním počtu iterácií škáluje veľmi dobre, pri každej z testovaných kariet pri danom rozsahu testov simulácia spĺňa požiadavku na beh v reálnom čase. Nasledujúci graf porovnáva najrýchlejšie a najpomalšie CPU a GPU.

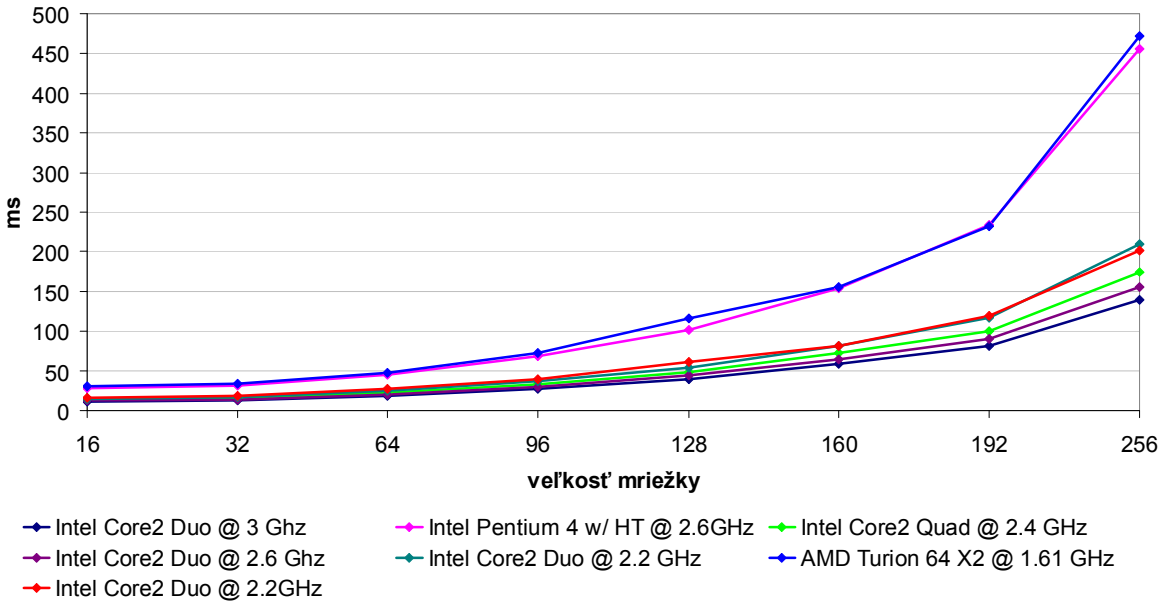


Obrázok 16. Porovnanie testu počtu iterácií pri CPU a GPU implementácii

Ako je zrejmé z grafu, aj najpomalšia grafická karta z testu prekoná najrýchlejší CPU. Zrýchlenie pri porovnaní najpomalšieho procesora a najrýchlejšej grafickej karty je ešte markantnejšie. Presnejšie výsledky a porovnania sú uvedené v kapitole 7.4.

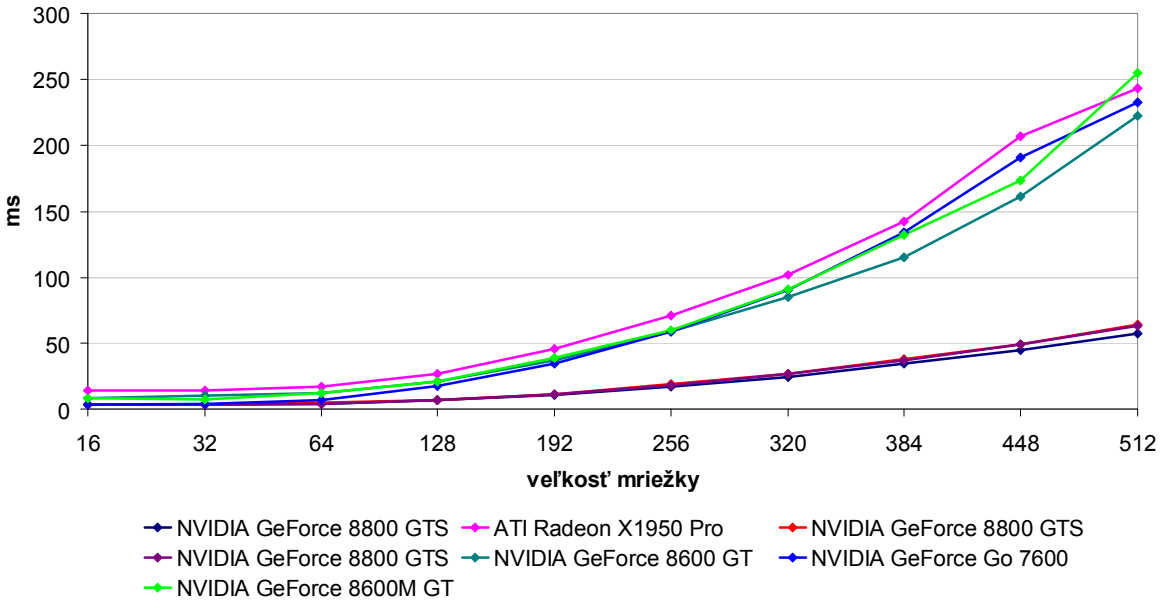
7.3.2 Veľkosť mriežky

Ďalším typom testu je porovnanie výkonnosti simulácie pri zmene veľkosti 2D mriežky simulácie.



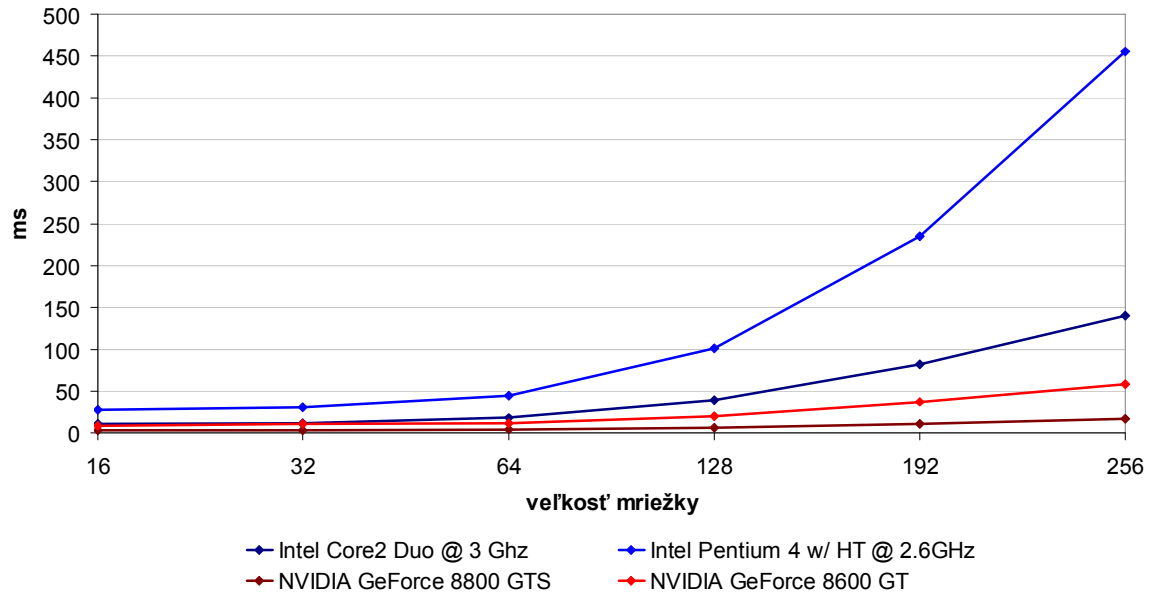
Obrázok 17. Test veľkosti mriežky simulácie (CPU implementácia)

Pod zmenou veľkosti mriežky sa myslí zmena oboch jej rozmerov (nakoľko sa jedná o štvorcovú mriežka, jej hrany sú rovnako dlhé). Ako z grafu jasne vidno, zmena veľkosti mriežky rapídne predlžuje dobu behu jedného kroku simulácie, pozorujeme kvadratický nárast výpočtovej náročnosti. Pri pomalších procesoroch je možné interaktívnu rýchlosť simulácie zachovať iba pri jej minimálnych rozmeroch. Novšie procesory v závislosti od taktovacej frekvencie umožňujú beh pri rozmeroch mriežky v rozmedzí 64 – 128.



Obrázok 18. Test veľkosti mriežky simulácie (GPU implementácia)

Škálovanie GPU implementácie je vcelku dobré. Konkrétne výsledky sa líšia v závislosti na generácii hardvéru alebo jeho konkrétnej špecifikácii. Napr. pri grafických kartách 8. generácie NVIDIA GeForce, konkrétne jej najvyššej triedy 8800, sa doba simulácie zvyšuje veľmi pozvoľne a ešte pri rozmeroch mriežky 320x320 je simulácia plynulá. Oproti tomu pri staršej generácii kariet alebo pri výbehových modeloch poslednej generácie sa maximum veľkosti mriežky pri zachovaní behu v reálnom čase pohybuje pri rozmeroch okolo 128x128 až 192x192.

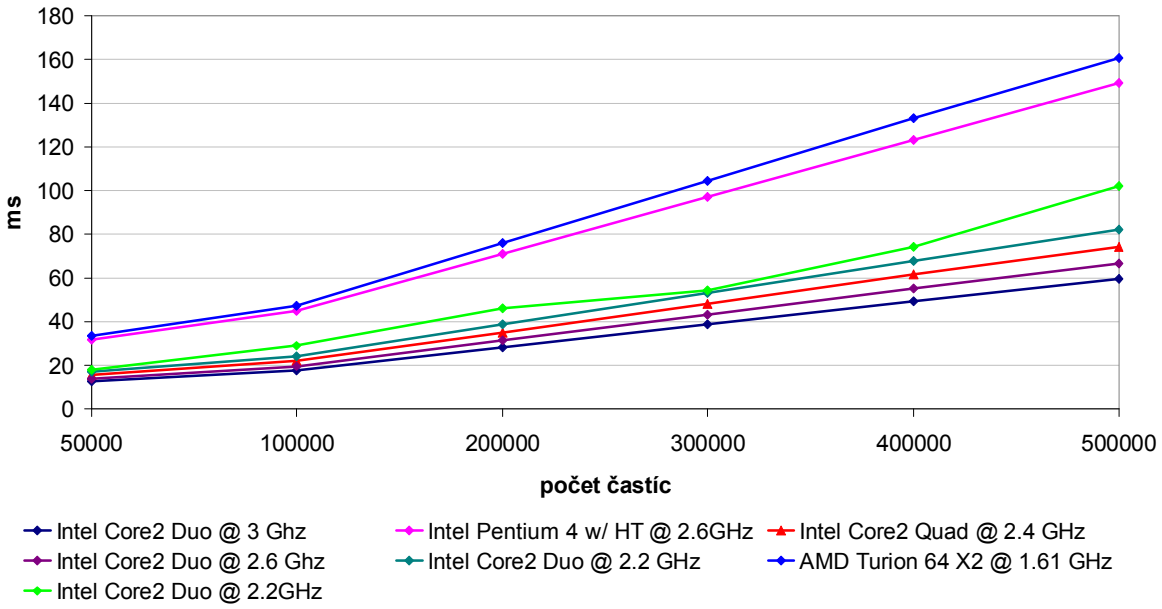


Obrázok 19. Porovnanie testu veľkosti mriežky pri CPU a GPU implementácii

Keďže testy CPU a GPU implementácie prebiehali pri rôznych rozsahoch rozmerov mriežky, graf porovnania implementácií obsahuje iba spoločné hodnoty veľkostí. Ako pri teste počtu iterácií, aj pri tomto teste je najrýchlejší procesor pri nízkych položkách relatívne porovnateľný s najpomalšou grafickou kartou.

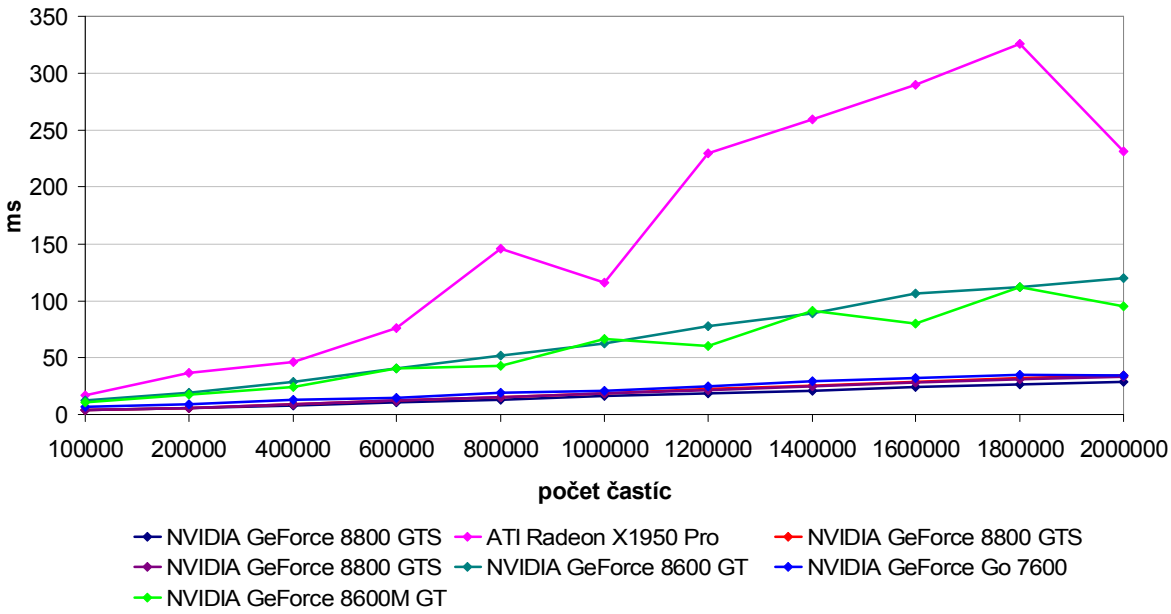
7.3.3 Počet častíc

Tento test oproti predchádzajúcim dvom nesúvisí s testovaním výpočtovej náročnosti simulácie tekutín, ale simulácie častíc. Pri návrhu riešenia sa došlo ku záveru, že aj táto simulácia je vhodná na GPU implementáciu a teda bola následne zaradená aj medzi navrhnuté testy.



Obrázok 20. Test počtu častíc (CPU implementácia)

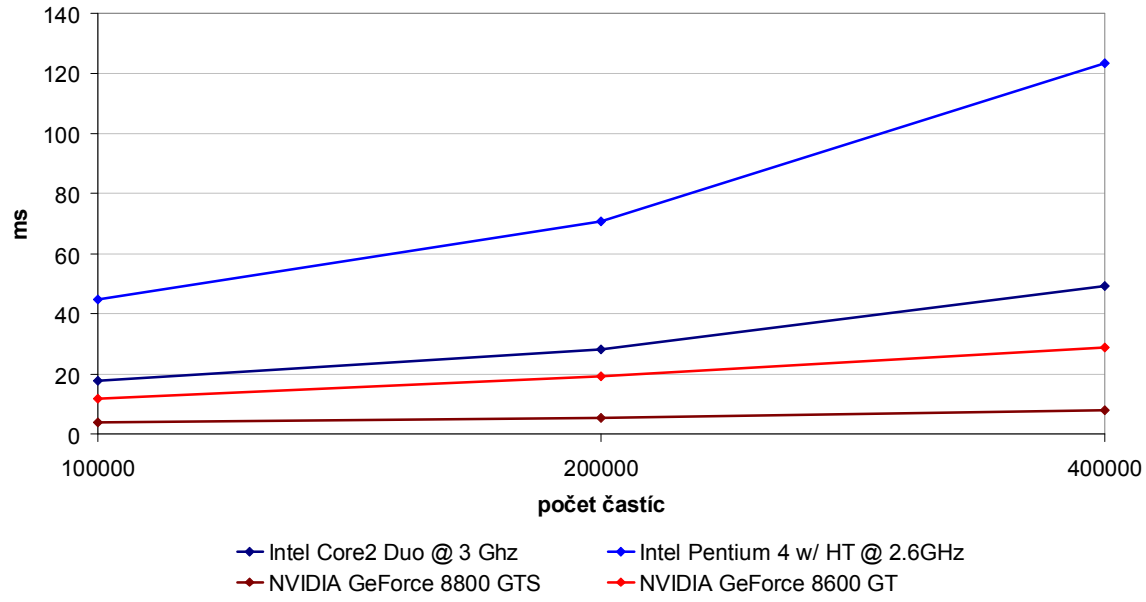
Pri vytváraní testu sa experimentálne overilo, že výpočtová náročnosť pri CPU implementácii je relatívne vysoká a nie je dôvod testovať simuláciu pri vysokom počte častíc. Iba najrýchlejšie procesory udržali rýchlosť simulácie pod 42 milisekúnd aj pri počte častíc 200000. Inak už pri viac minimálnej hodnote 50000 výbehové procesory neposkytovali dostatok výkonu na real-time simuláciu.



Obrázok 21. Test počtu častíc (GPU implementácia)

Oproti CPU implementácii GPU implementácia dosahovala pri testovaní kariet výborné výsledky. Výnimkou sú snáď staršie generácie kariet alebo výbehové modely súčasnej generácie. Najvýkonnejšie grafické procesory aj pri maximálnom testovanom počte častíc 2 milióny disponovali dostatočným výkonom. V porovnaní oproti simulácii

tekutín je to zrejme aj vďaka faktu, že *fragment shader* na výpočet simulácie častíc je podstatne jednoduchší.



Obrázok 22. Porovnanie testu počtu častíc pri CPU a GPU implementácii

Porovnanie CPU a GPU už klasicky ukazuje relatívne podobnú rýchlosť simulácie najrýchlejšieho testovaného procesora a najpomalšej testovanej grafickej karty.

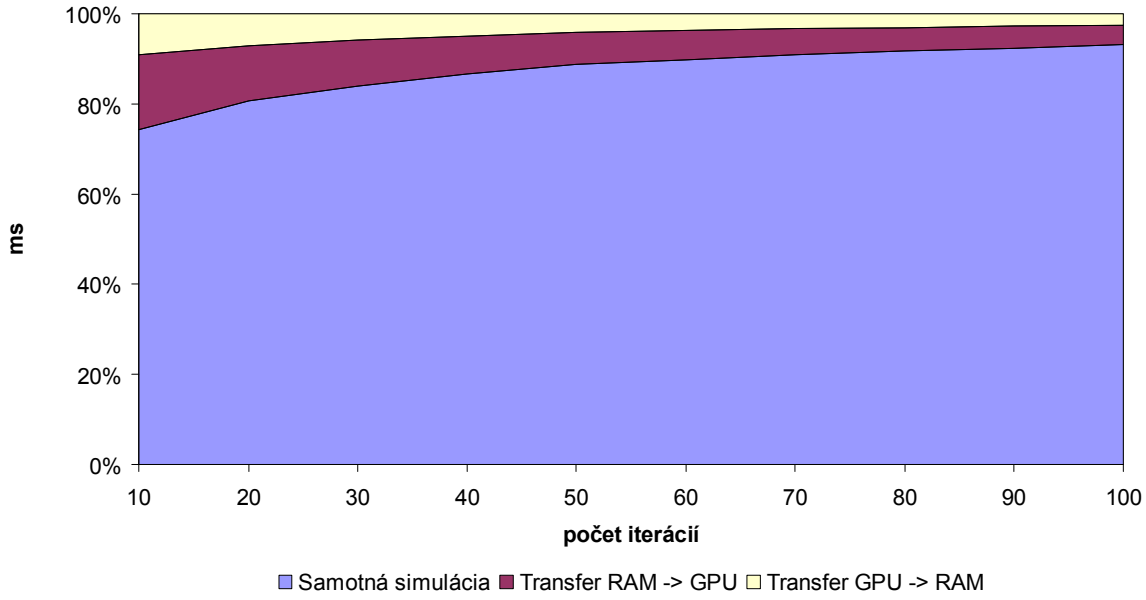
7.3.4 Transfer dát

Zásadným rozdielom medzi a CPU a GPU implementáciou je spôsob prístupu k dátam. Pokiaľ pri klasickom programovaní sú všetky dáta dostupné v hlavnej pamäti RAM, pri programovaní využitím GPU je všetky potrebné dáta nutné skopírovať do pamäte grafickej karty. Pomer medzi množstvom presúvaných dát a objemom výpočtom je preto rozhodujúci pri zisťovaní, či brzdou v škálovaní bude priepustnosť dátovej zbernice alebo výkonnosť grafického procesora.

Pre všetky 3 typy testov boli uskutočnené merania dĺžky prenosu dát v smere na a v smere z grafickej karty a dĺžky samotného výpočtu na GPU. Merania boli uskutočnené iba na jednej karte a to NVIDIA GeForce 8800 GTS. Táto je dostatočne výkonná a disponuje širokou 320-bitovou pamäťovou zbernicou. Navyše sa jedná o novú generáciou kariet, t.j. o kartu s prúdovými procesormi (konkrétne obsahuje 96 prúdových procesorov). Z testovaných kariet vyšla ako jedna z najrýchlejších.

Test počtu iterácií

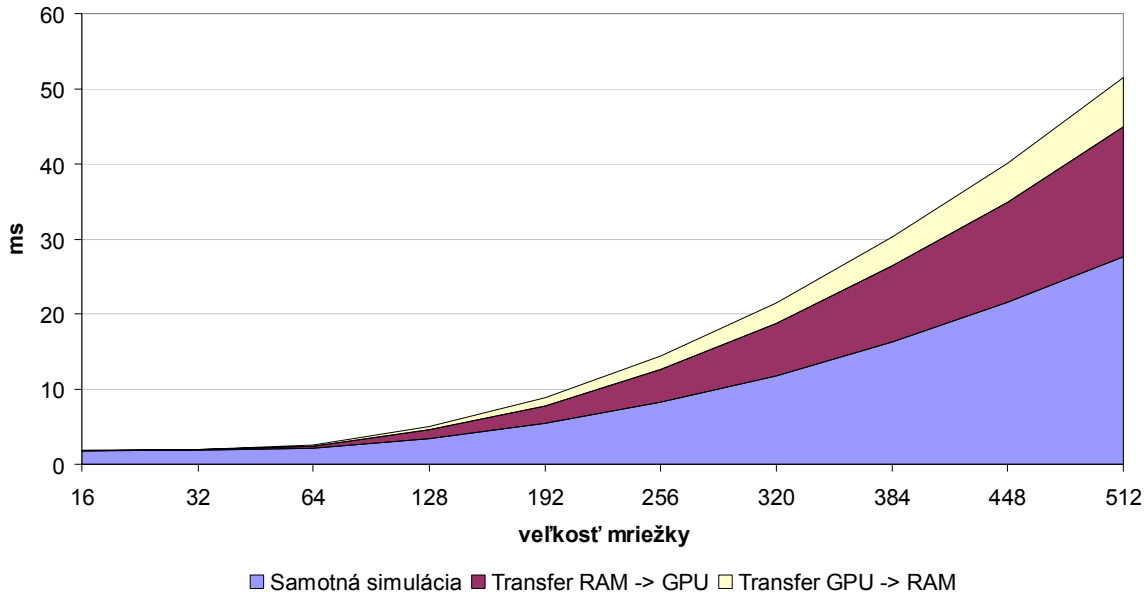
Pri zvyšovaní počtu iterácií sa zvyšovala iba doba výpočtu samotnej simulácie, čo je logické, keďže množstvo prenášaných dát zostávalo rovnaké. Doba prenosu dát v oboch smeroch je minimálna.



Obrázok 23. Pomer dôb transferu dát a samotného výpočtu (test počtu iterácií)

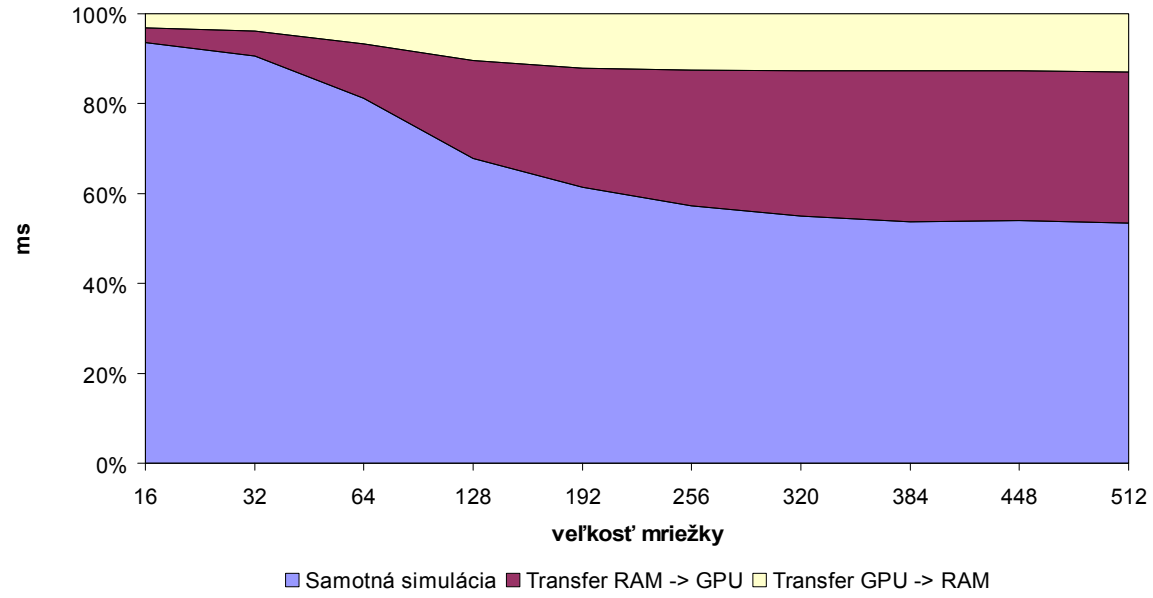
Dĺžka simulácie výrazne dominuje oproti transferu dát. Pri tomto type testu sa ukazuje, že transfer dát nie je hlavným činiteľom spomalenia simulácie.

Test veľkosti mriežky



Obrázok 24. Transfer dát pri teste veľkosti mriežky simulácie

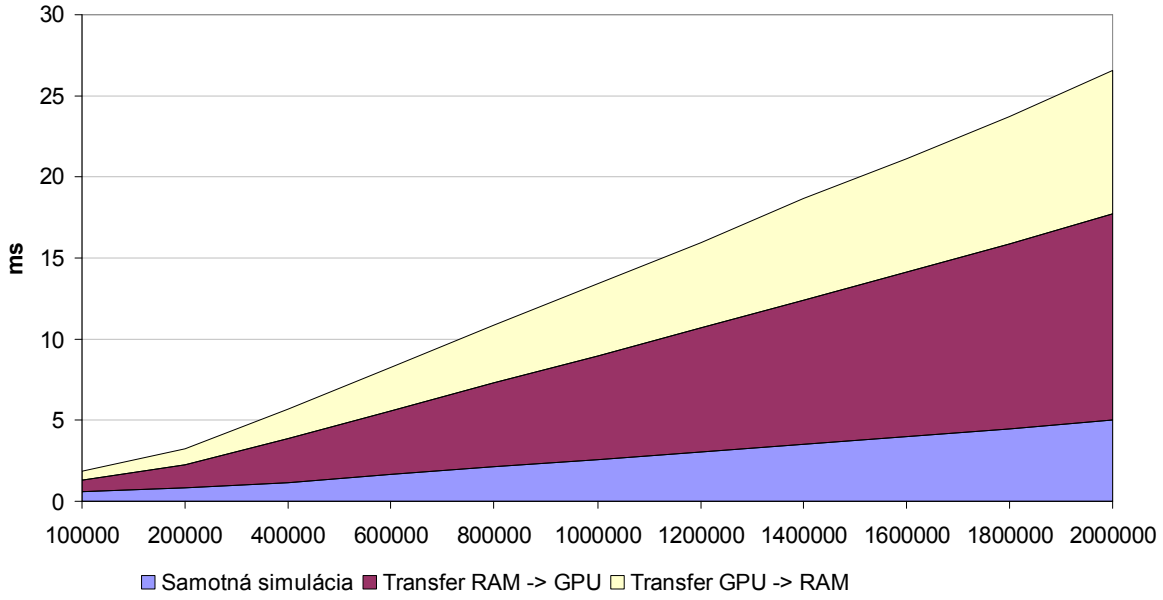
V tomto teste v porovnaní s predchádzajúcim so zväčšovaním rozmeru mriežky ide ruka v ruke aj zväčšovanie objemu prenášaných dát. Takisto stúpa aj čistý čas výpočtu. Nasledujúci graf pomerne zobrazuje množstvo času potrebného na jednotlivé fázy simulácie.



Obrázok 25. Pomer dôb transferu dát a samotného výpočtu (test veľkosti mriežky simulácie)

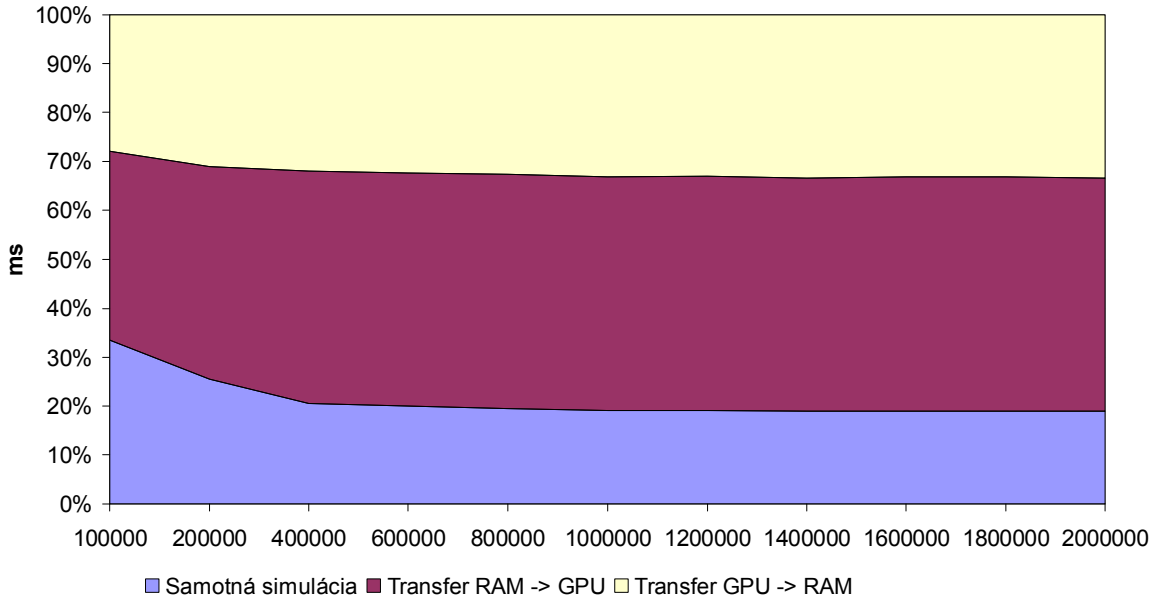
Pri teste zväčšovanie veľkosti mriežky simulácie do popredia ako činiteľ ovplyvňujúci dĺžku simulácie aj transfer dát medzi pamäťou grafickej karty a hlavnou pamäťou RAM. Čistý čas výpočtu však ešte stále dominuje.

Test počtu častíc



Obrázok 26. Transfer dát pri teste počtu častíc

Posledný z testov je najvýraznejšie ovplyvnený dobou transferu dát. Ako je spomenuté v kapitole 7.3.3, *fragment shader* na výpočet simulácie častíc je oveľa jednoduchší ako *fragment shader-e* použité na simuláciu tekutín. Zároveň veľké množstvo dát (častíc) je dôvodom, prečo je dĺžka samotného výpočtu pohybu častíc v porovnaní s časom prenosu dát nízka.



Obrázok 27. Pomer dôb transferu dát a samotného výpočtu (test počtu častíc)

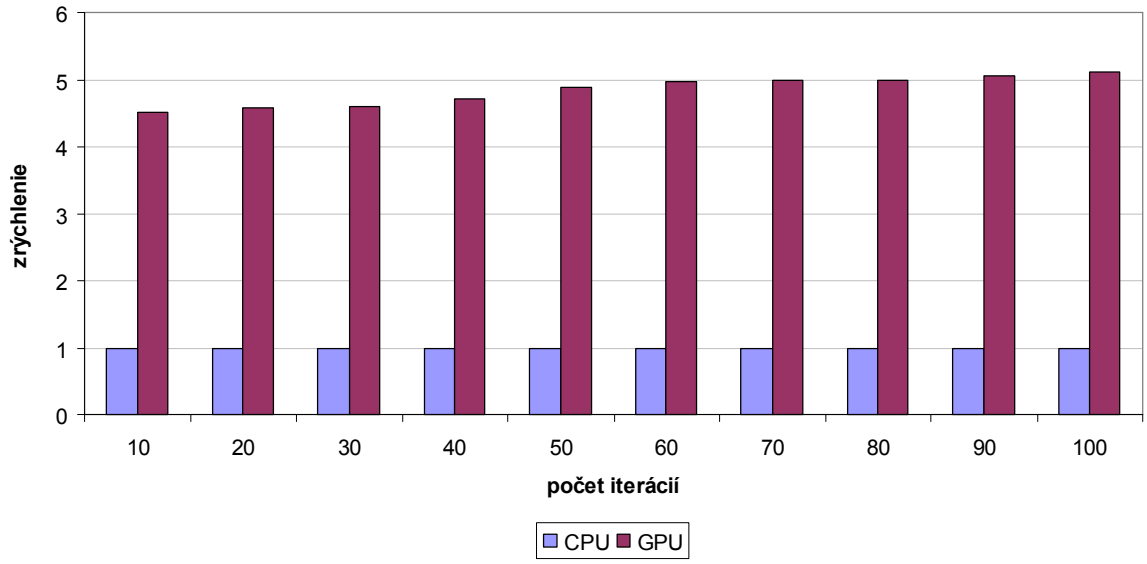
Pri tomto type testu je pomer dĺžky prenosu dát a výpočtu na GPU od určitého množstva častíc rovnaký. Takýto prípad, ako je tento, kedy nárast množstva spracovávaných dát vyvolá pomerne rovnaké zvýšenie zaťaženia pamäťovej zbernice a výpočtovej jednotky, dáva dobré predpoklady, aby nedošlo k situácii, kedy jedna z menovaných častí grafického subsystému bude vytvárať úzke hrdlo, ale obe jednotky budú využité naplno. To je však ideálna situácia a jej podmienkou je, aby dĺžka prenosu dát pamäťovou zbernicou a dĺžka ich spracovania grafickým procesorom bola rovnaká. To sa však v praxi dosahuje už ťažšie.

7.4 Zhrnutie a vyhodnotenie výsledkov

Kapitola vyhodnocuje výsledky testov a vyčísľuje zrýchlenie použitia grafických procesorov oproti klasickým procesorom CPU. Z grafov jednoznačne vyplýva, že použitie GPU je výhodnejšie, tabuľky poskytujú presné údaje.

Na výpočet zrýchlenia bol vybraný najrýchlejší CPU z testu – Intel Core2 Duo taktovaný na frekvencii 3 Ghz a najrýchlejší GPU – NVIDIA GeForce 8800 GTS.

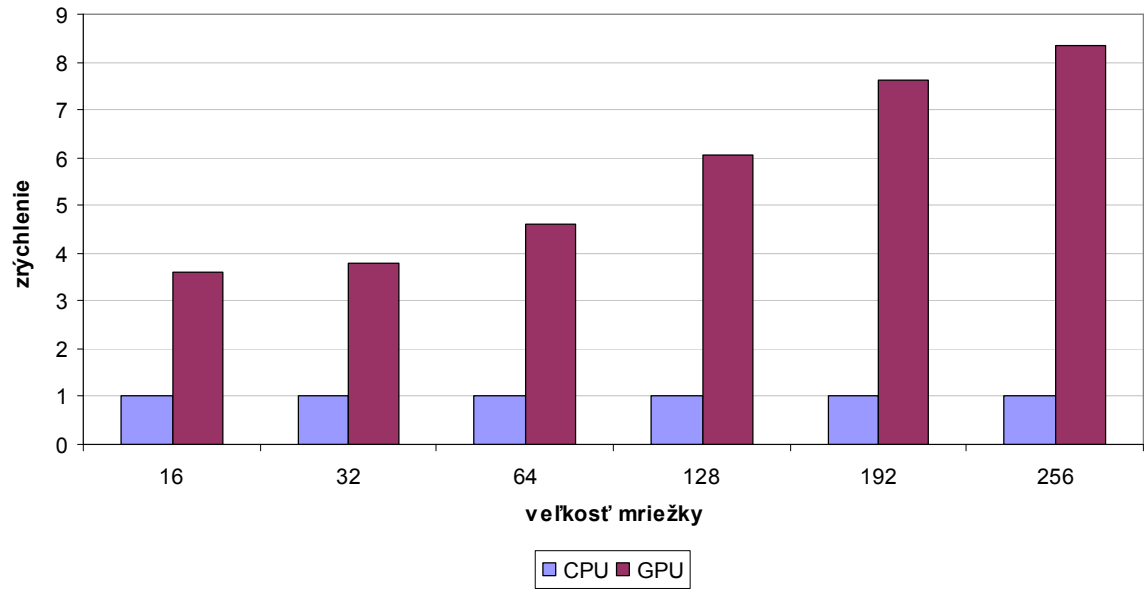
Test počtu iterácií										
Počet iterácií	10	20	30	40	50	60	70	80	90	100
Zrýchlenie	4,52	4,57	4,60	4,72	4,88	4,96	5,00	5,00	5,06	5,11



Obrázok 28. Zrýchlenie pri zvyšovaní počtu iterácií

Tento test sa vyznačuje zachovaním rovnakého množstva spracovávaných dát. Mení sa iba počet iterácií spracovania, de facto počet opakovaní cyklu. Pri zvyšovaní počtu iterácií pomer rýchlosti GPU implementácie oproti rýchlosti CPU implementácie stúpal iba pozvoľne. Faktor zrýchlenia sa pohybuje okolo hodnoty 5.

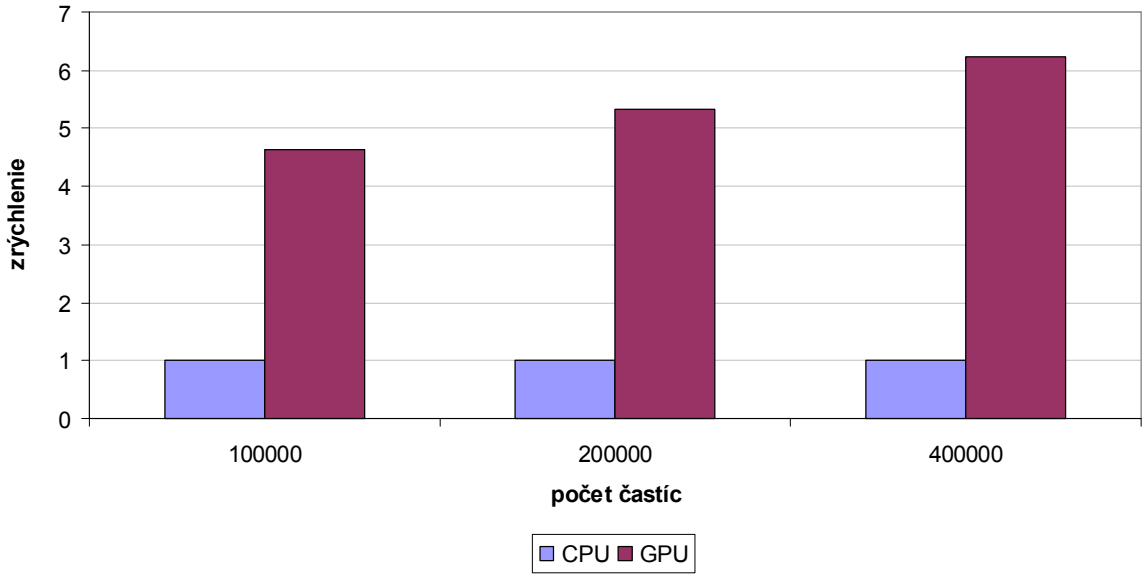
Test veľkosti mriežky						
Veľkosť mriežky	16	32	64	128	192	256
Zrýchlenie	3,59	3,81	4,63	6,05	7,63	8,35



Obrázok 29. Zrýchlenie pri zväčšovaní rozmerov mriežky

Zmena veľkosti mriežky výrazne ovplyvňuje množstvo spracovávaných dát. S týmto faktom sa lepšie, ako vyplýva z tabuľky, vysporadúva GPU. Faktor zrýchlenia stúpa od hodnoty 3,5 až k 8-násobnému zlepšeniu výkonu.

Test počtu častíc			
Počet častíc	100000	200000	400000
Zrýchlenie	4,64	5,34	6,24



Obrázok 30. Zrýchlenie pri zvyšovaní počtu častíc

Pri teste počtu častíc, podobne ako pri teste veľkosti mriežky, je benchmarking založený na zväčšovaní objemu spracovávaných dát. Zrýchlenie pri tomto teste je v rozmedzí 4,6 až 6,2.

Na základe výsledkov testov a porovnaní zrýchlenia pri jednotlivých typoch sa ukazuje, že zväčšenie množstva spracovávaných dát (zväčšenie mriežky simulácie, zvýšenie počtu častíc) škáluje lepšie ako zintenzívnenie výpočtovej náročnosti (zväčšenie počtu iterácií). Pri všetkých prípadoch je však zrýchlenie GPU simulácie oproti CPU implementácii výrazné.

8 Zhodnotenie a budúca práca

Pozadie problému a získané informácie o ňom sú uvedené v časti analýzy. Tá sa skladá z dvoch častí. Prvou je analýza problémovej oblasti, ako je paralelná programovacia paradigma a analýza programovania konkrétneho programovania grafického hardvéru. Druhou časťou je simulácia tekutín pomocou Navier-Stokes rovníc, ich numerického riešenia a diskusie jednotlivých metód. Keďže výstupom projektu je aj implementácia simulácie tekutín, v dokumente je uvedená špecifikácia programu, vypracovaný návrh a uvedený opis implementácie.

Nosnú časť dokumentu tvorí kapitola testovania. V nej sú porovnávané výsledky ako CPU a GPU implementácie medzi sebou, tak aj navzájom. Na základe získaných výsledkov je možné konštatovať, že použitie grafickej karty na výpočty výrazne urýchľuje celú simuláciu. Faktor zrýchlenia sa v testoch pohybujú v intervale od 3,5 do 8,3. Vzhľadom na trend vývoja grafických čipov sa však dá predpokladať ďalšie zväčšovanie výkonnostnej medzery medzi CPU a GPU. V globálnom meradle je zrýchlenie výpočtov značné a určite stojí za dodatočné úsilie potrebné na prepis programu do formy umožňujúcej paralelné spracovanie.

Ako sa počas testovania ukázalo, nezanedbateľným faktorom obmedzujúcim rýchlosť simulácie je potreba transferu dát z a na GPU. Tento nedostatok je možné odstrániť použitím napríklad rozšírenia OpenGL ARB_pixel_buffer_object [30]. To umožňuje prenos dát na grafickú kartou použitím DMA (Direct Memory Access), teda prenos bude prebiehať na pozadí a nebude pri ňom potrebné zaťažovať procesor. Takýto scenár je možné uskutočniť v prípade, že pokiaľ sú dáta prenášané, procesor vykonáva inú užitočnú činnosť, inak je zrýchlenie minimálne, keďže by sa muselo čakať na dokončenie prenosu. To by bolo možné pri využití simulácie tekutín vo väčšom projekte, súčasná implementácia by veľký prínos nezaznamenala.

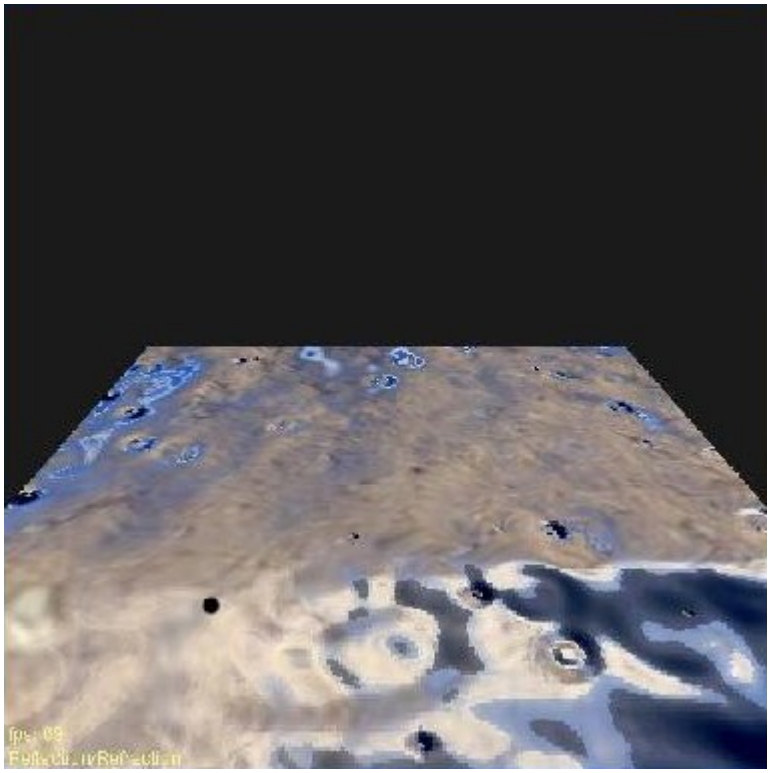
Ďalším možným zlepšením je odstránenie závislosti na grafickom programovacom jazyku, t.j. GLSL, a použitie novšej technológie, CUDA. Tá poskytuje programovacie rozhranie (API) v jazyku C a približuje programovanie použitím GPU klasickému programovanie CPU. Okrem tejto hlavnej výhody táto technológia prežíva v súčasnej dobe rozmach, je multiplatformová a existuje pre ňu dostatok nástrojov pre vývoj.

Použitá literatúra

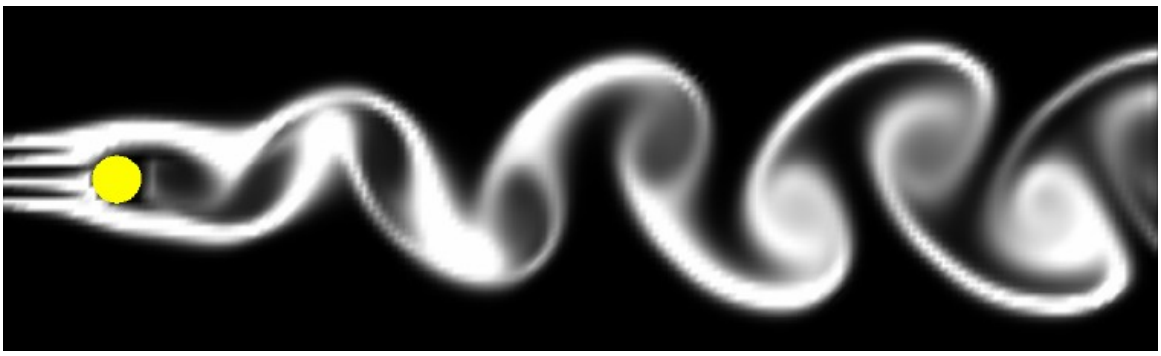
1. WEISKOPF, D., HOPF, M., ERTL, T., 2001: *Hardware-accelerated visualization of time-varying 2D and 3D vector fields by texture advection via programmable per-pixel operations*. Workshop on Vision, Modeling, and Visualization VMV, s. 439-446
2. HARRIS, M., COOMBE, G., SCHEUERMANN, T., LASTRA, A. 2002: *Physically-based visual simulation on graphics hardware*. SIGGRAPH / Eurographics Workshop on Graphics Hardware (Sep), s. 109-118
3. OpenGL - *The Industry Standard for High Performance Graphics* http://www.opengl.org/ (10. 12. 2007)
4. The Khronos Group, 2007: *OpenGL ES 2.X and the OpenGL ES Shading Language for programmable hardware* http://www.khronos.org/opengles/2_X/ (10. 12. 2007)
5. DAWKINS, P., 2007: *Differential Equations* (Math 3401) http://tutorial.math.lamar.edu/classes/de/de.aspx (10. 12. 2007)
6. NVIDIA, 2007: *Cg* http://developer.nvidia.com/page/cg_main.html (10. 12. 2007)
7. NVIDIA 2007: *NVIDIA CUDA* http://developer.nvidia.com/object/cuda.html (10. 12. 2007)
8. GÖDDEKE, D., 2005: *GPGPU Tutorials* http://www.mathematik.uni-dortmund.de/%7Egoeddeke/gpgpu/tutorial.html (10. 12. 2007)
9. GPGPU: *General-Purpose Computation Using Graphics Hardware* http://www.gpgpu.org/ (10. 12. 2007)
10. ATI Developer: *Technical Publications* http://ati.amd.com/developer/techreports.html (10. 12. 2007)
11. Wolfram MathWorld: *Navier-Stokes Equations* http://mathworld.wolfram.com/Navier-StokesEquations.html (10. 12. 2007)
12. SHREINER, D., WOO, M., NEIDER, J., DAVIS, T: *OpenGL - Průvodce programátora*, 2006, s. 523-556
13. AMD ATI Radeon HD 2000 Series Architecture: *Hardware Secrets* http://www.hardwaresecrets.com/article/448/2 (10. 12. 2007)
14. GeForce 8 Series Architecture: *Hardware Secrets* http://www.hardwaresecrets.com/article/398/2 (10. 12. 2007)
15. TOMMESANI, S.: *Programming Models* http://www.tommesani.com/ProgrammingModels.html (10. 12. 2007)
16. MSDN Magazine: *Parallel Performance: Optimize Managed Code For Multi-Core Machines*, October 2007 http://msdn.microsoft.com/msdnmag/issues/07/10/Futures/default.aspx (10. 12. 2007)

17. Karsten Noe's Projects: *GPU water simulation* http://projects.n-o-e.dk/?page=show&name=GPU%20water%20simulation (10. 12. 2007)
18. Mercury Computer Systems and NVIDIA to Provide Oil Industry with Data Computation on the Fly http://www.industrial-embedded.com/news/db/?9313 (10. 12. 2007)
19. STAM, J.: *Real-Time Fluid Dynamics for Games*, Proceedings of the Game Developer Conference, March 2003 http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf (10. 12. 2007)
20. STAM, J.: *Stable Fluids*, In SIGGRAPH 99 Conference Proceedings, Annual Conference Series, August 1999, s. 121-128 http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/ns.pdf (10. 12. 2007)
21. ENHUA, W., YOUQUAN, L., XUEHUI, L.: *An improved study of real-time fluid simulation on GPU*, Research Articles, Computer Animation and Virtual Worlds, Volume 15, Issue 3-4 (July 2004), s. 139 – 146 http://www.ce.chalmers.se/~uffe/xjobb/Readings/GPU-Fluids/An%20Improved%20Study%20of%20Real-Time%20Fluid%20Simulation%20on%20GPU.pdf (10. 12. 2007)
22. IEEE 754: *Standard for Binary Floating-Point Arithmetic* http://grouper.ieee.org/groups/754/ (10. 12. 2007)
23. Wolfram MathWorld: *Gauss-Seidel Method* http://mathworld.wolfram.com/Gauss-SeidelMethod.html (10. 12. 2007)
24. Wolfram MathWorld: *Jacobi Method* http://mathworld.wolfram.com/JacobiMethod.html (10. 12. 2007)
25. MARK J. H.: *Real-Time Cloud Simulation and Rendering,* University of North Carolina Technical Report #TR03-040. 2003 http://www.markmark.net/dissertation/ (10. 12. 2007)
26. CRANE, K., LLAMAS, I., TARIQ, S.: *Real-Time Simulation and Rendering of 3D Fluids*. GPU Gems 3 by Hubert Nguyen, Chapter 30, s. 633-673 http://www.cs.caltech.edu/~keenan/project_fluid.html (10. 12. 2007)
27. BUCHANAN, J. L., TURNER, P. R., 1992: *Numerical Methods and Analysis*, s. 659-713
28. OpenGL Extension Registry: *EXT_framebuffer_object* http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt (15. 4. 2008)
29. GLEW: *The OpenGL Extension Wrangler Library* http://glew.sourceforge.net/ (23. 4. 2008)
30. OpenGL Extension Registry: *ARB_pixel_buffer_object* http://www.opengl.org/registry/specs/ARB/pixel_buffer_object.txt (26. 4. 2008)

Príloha A: Príklady existujúcich aplikácií

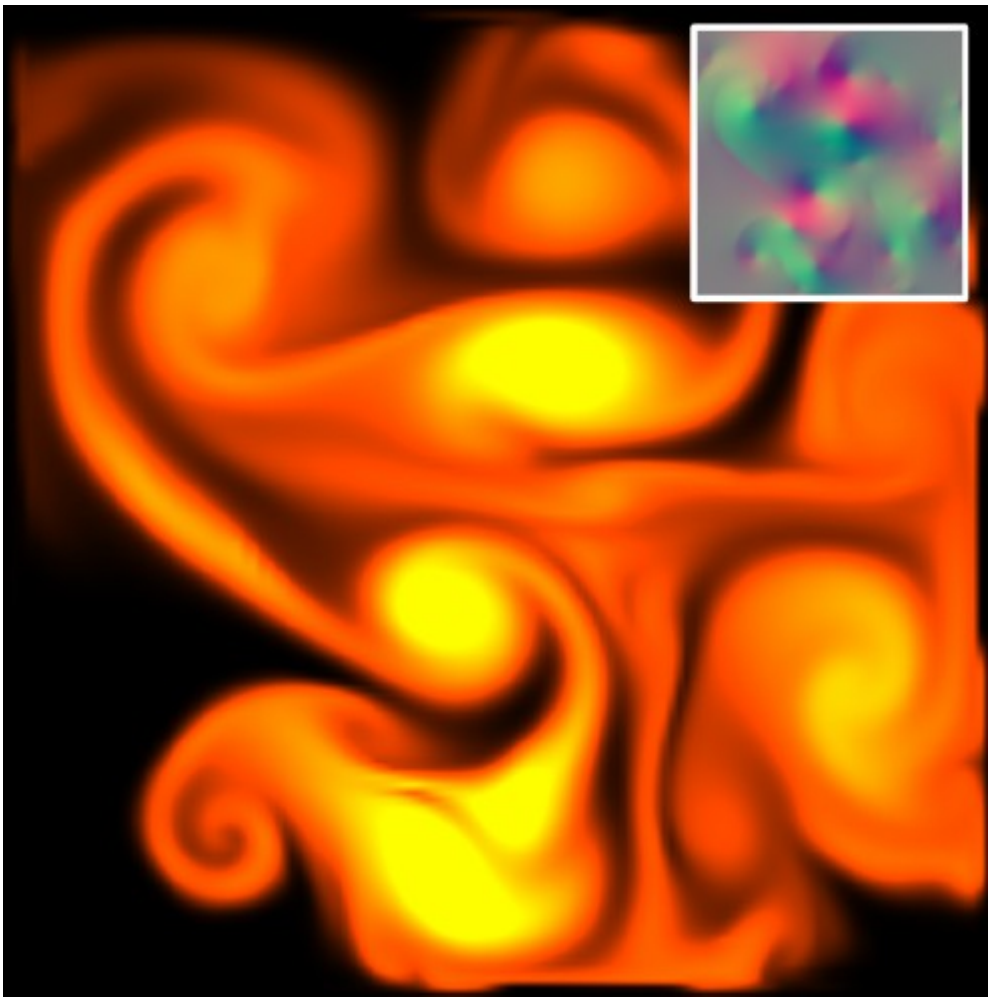


Simulácia vody reprezentovanej výškovou mapou na GPU [17]

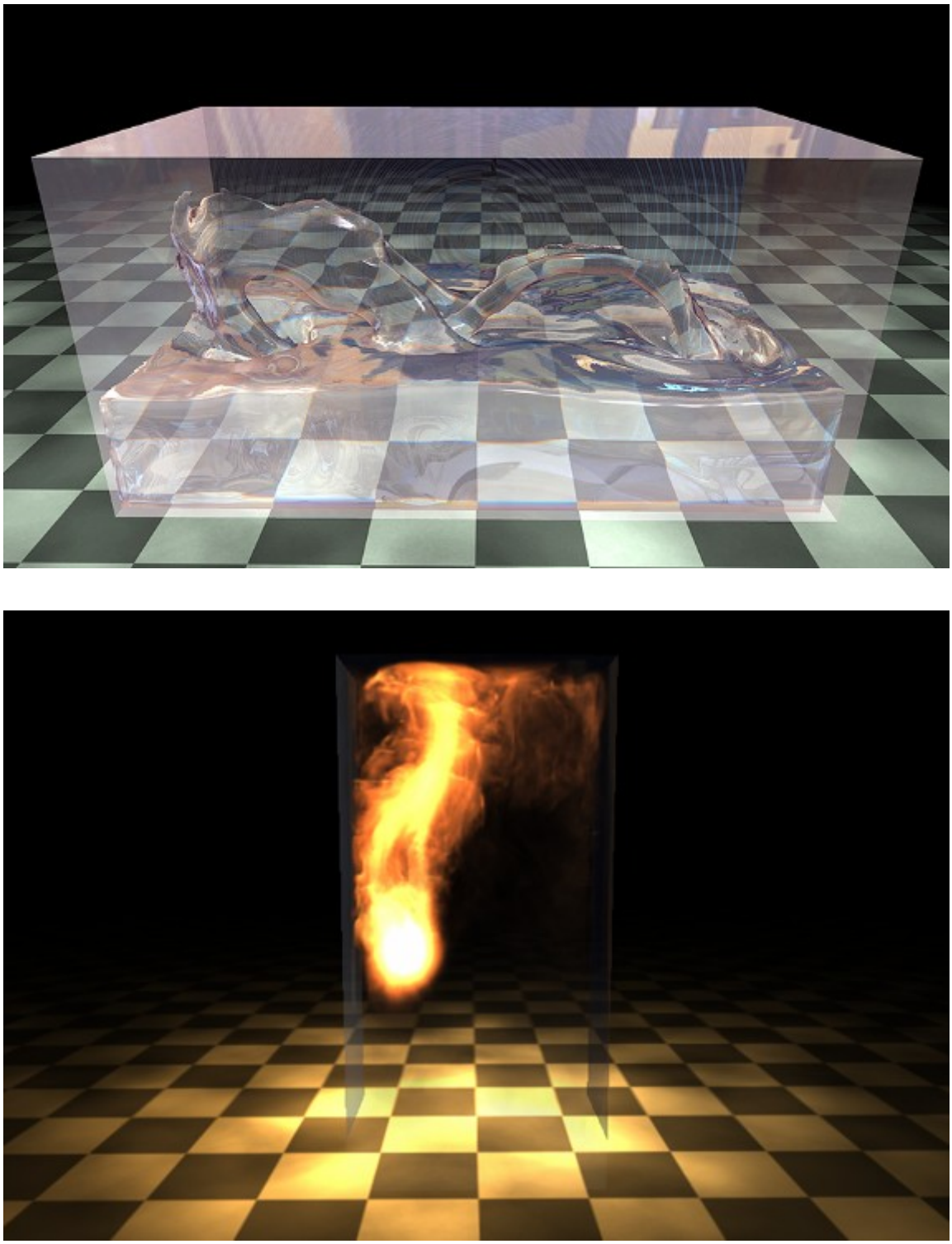


Obtekanie prekážky tekutinou [20]

Obtekanie prekážky tekutinou [21]



Simulácia tekutiny pomocou GPU [25]



Simulácia tekutiny a ohňa pomocou GPU [26]

Príloha B: Technická dokumentácia

Implementácia systému

Z dôvodu požiadavky na beh systému v reálnom čase bol za implementačný jazyk zvolený jazyk C++. Tento umožňuje spojiť výhody vysokej rýchlosti behu programu podobnú s jazykom C a zároveň možnosť využitia objektovej-orientovanej paradigmy vývoja softvéru.

Na základe diagramu komponentov systému (Obrázok 12. Diagram komponentov systému) bol navrhnutý a vytvorený diagram tried. Ten obsahuje nasledovné triedy:

- `Singleton<T>` – šablónová trieda implementujúca návrhový vzor Singleton.
- `Profiler` – trieda zaznamenávajúca rýchlosť behu simulácie a jej jednotlivých častí, zároveň umožňuje výpis štatistických informácií.
- `Particles` – abstraktná trieda definujúca rozhranie na prácu s časticovým systémom.
- `Particles`CPU, Particles GPU – konkrétne implementácie časticovej simulácie.
- `NSESolver` – abstraktná trieda definujúca rozhranie na prácu so simuláciou tekutiny.
- `NSESolverCPU, NSESolverGPU` – konkrétne implementácie simulátora.

Ku týmto triedam prislúchajú súbory so zodpovedajúcim názvom. Systémové funkcie a funkcie na prácu s grafickým subsystémom sú implementované štrukturálnou paradigmou. Zo spôsobu práce s nimi zapuzdrenie do tried nie je potrebné. Diagram tried systému je na samostatnej strane na konci tejto prílohy.

Súbory `gl.h` a `gl.cpp` obsahujú funkcie na prácu s grafickým rozhraním systému a prácu s OpenGL. Súbory `common.h` a `common.cpp` definujú všeobecne používané funkcie (konkrétne funkcie na prácu s časom, vypisovaním logov a generáciu náhodných čísiel). Súbor `main.cpp` obsahuje funkcie na spracovanie parametrov príkazového riadka, vykresľovanie základného grafického rozhrania, spúšťanie simulácie s potrebnými parametrami a reakciu na vstupy používateľa.

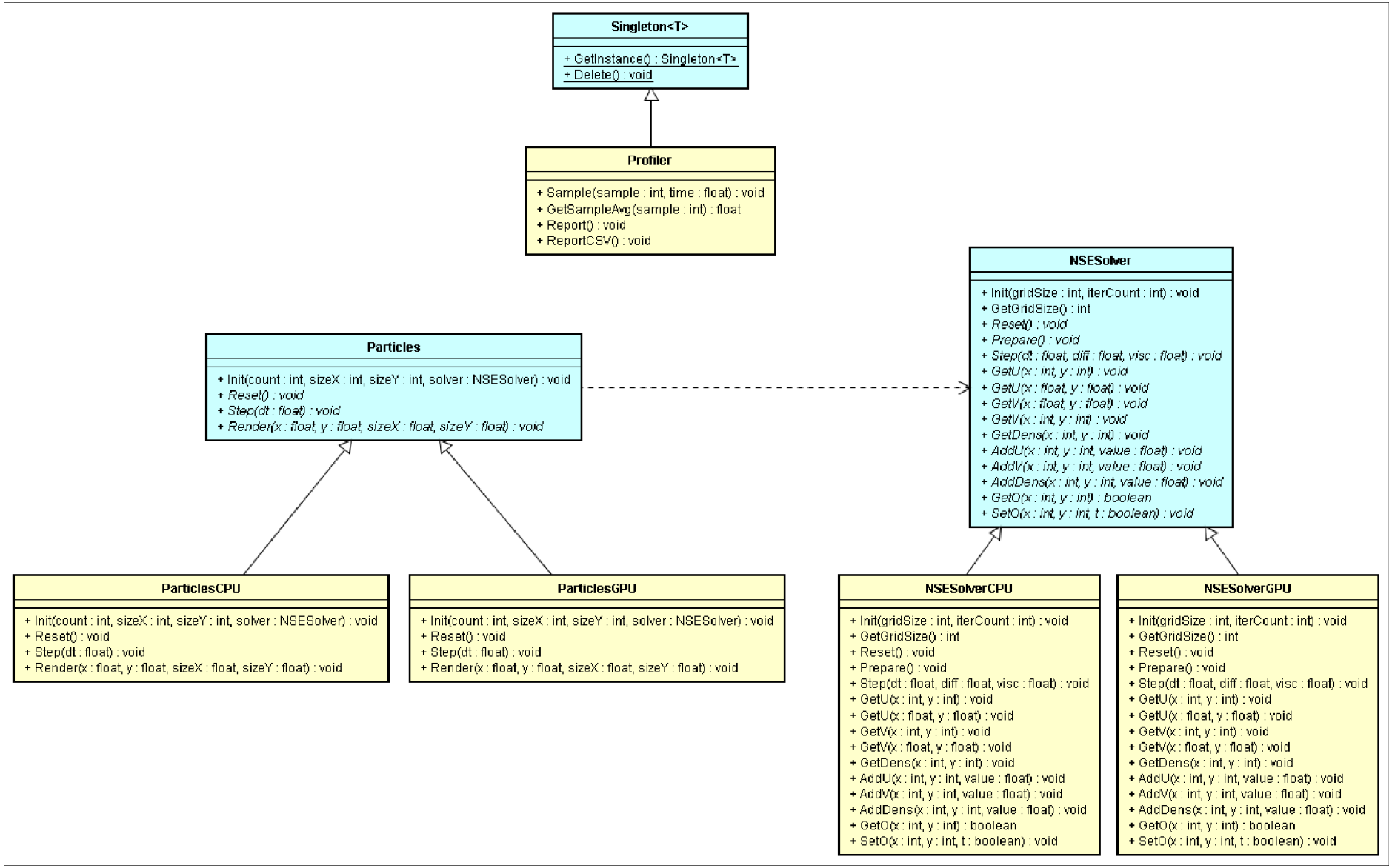
GPU implementácia

Grafický subsystém použitý na GPU implementáciu riešenia je OpenGL. Konkrétnym implementačný jazyk je GLSL [12]. Ako simulácia tekutín, tak aj simulácia častíc, obsahuje *fragment programy* použité na výpočty. Výpočty prebiehali pri nasledovných nastaveniach OpenGL:

- typ textúry – `GL_TEXTURE_RECTANGLE_ARB` – obdĺžniková textúra
- formát textúry – `GL_RGBA32F_ARB` – 32-bitov pre každú položku RGBA textúry
- filtrovanie - `GL_NEAREST` – žiadne filtrovanie

Týmto sa zabezpečila možnosť použitia ľubovoľných rozmerov pre mriežku a dostatočná presnosť výpočtov. Výpis *fragment programu* na simuláciu pohybu a kolízií tekutín:

```
// Fragment program for particles simulation
const char *particles_shader_source = \
      // Enable rect textures
      "#extension GL_ARB_texture_rectangle : enable\n" \
      // Textures definition
      "uniform sampler2DRect textureVertices;" \
      "uniform sampler2DRect textureValues;" \
      // Time step
      "uniform float dt;" \
      // Grid size
      "uniform int size;" \
      // Tests collision with obstacles
      "bool isO(const in vec2 pos) { " \
      "  return (texture2DRect(textureValues, (pos.xy * float(size) +
vec2(1.0, 1.0))).a != 0.0);" \
      "}" \
      // Get particle speed
      "vec2 getUV(const in vec2 pos) { " \
      /* XXX: Using the same interpolation as in CPU implementation */
      "  float i0 = float(int(pos.x * float(size) + 1.0));" \
      "  float j0 = float(int(pos.y * float(size) + 1.0));" \
      "  float i1 = i0 + 1.0;" \
      "  float j1 = j0 + 1.0;" \
      "  float s1 = (pos.x * float(size) + 1.0) - i0; "\
      "  float s0 = 1.0 - s1; "\
      "  float t1 = (pos.y * float(size) + 1.0) - j0; "\
      "  float t0 = 1.0 - t1; "\
      "  return s0 * (t0 * texture2DRect(textureValues, vec2(i0,
j0)).rg + t1 * texture2DRect(textureValues, vec2(i0, j1)).rg) + s1 *
(t0 * texture2DRect(textureValues, vec2(i1, j0)).rg + t1 *
texture2DRect(textureValues, vec2(i1, j1)).rg); " \
      "}" \
      // Compute new position
      "void computeNewPos(const in vec2 pos, out vec2 newPos) {" \
      "  vec2 speed = getUV(pos.xy);" \
      "  newPos = pos.xy + speed * dt; " \
      "  if (!isO(newPos))" \
      "    return;" \
      /* XXX: ATI Technologies Inc M22 [Mobility Radeon X300] -
available number of texture instructions exceeded */
      "  newPos = pos.xy + speed * dt * 0.5; " \
      "  if (!isO(newPos))" \
      "    return;" \
      "  newPos = pos.xy + speed * dt * 0.25; " \
      "  if (!isO(newPos))" \
      "    return;" \
      "  newPos = pos;" \
      "}" \
      // Main program entry
      "void main(void) { " \
      "  vec4 vertex = texture2DRect(textureVertices,
gl_TexCoord[0].st);" \
      "  vec2 pos1, pos2; " \
      "  computeNewPos(vertex.rg, pos1); " \
      "  computeNewPos(vertex.ba, pos2); " \
      "  gl_FragColor = clamp(vec4(pos1, pos2), vec4(0.0, 0.0, 0.0,
0.0), vec4(1.0, 1.0, 1.0, 1.0));" \
      "}";
```


Singleton<T>
+ GetInstance() : Singleton<T>
+ Delete() : void
Profiler
+ Sample(sample : int, time : float) : void
+ GetSampleAvg(sample : int) : float
+ Report() : void
+ ReportCSV() : void
NSESolver
+ Init(gridSize : int, iterCount : int) : void
+ GetGridSize() : int
+ Reset() : void
+ Prepare() : void
+ Step(dt : float, diff : float, visc : float) : void
+ GetU(x : int, y : int) : void
+ GetU(x : float, y : float) : void
+ GetV(x : float, y : float) : void
+ GetV(x : int, y : int) : void
+ GetDens(x : int, y : int) : void
+ AddU(x : int, y : int, value : float) : void
+ AddV(x : int, y : int, value : float) : void
+ AddDens(x : int, y : int, value : float) : void
+ GetO(x : int, y : int) : boolean
+ SetO(x : int, y : int, t : boolean) : void
Particles
+ Init(count : int, sizeX : int, sizeY : int, solver : NSESolver) : void
+ Reset() : void
+ Step(dt : float) : void
+ Render(x : float, y : float, sizeX : float, sizeY : float) : void
ParticlesCPU
+ Init(count : int, sizeX : int, sizeY : int, solver : NSESolver) : void
+ Reset() : void
+ Step(dt : float) : void
+ Render(x : float, y : float, sizeX : float, sizeY : float) : void
ParticlesGPU
+ Init(count : int, sizeX : int, sizeY : int, solver : NSESolver) : void
+ Reset() : void
+ Step(dt : float) : void
+ Render(x : float, y : float, sizeX : float, sizeY : float) : void
NSESolverCPU
+ Init(gridSize : int, iterCount : int) : void
+ GetGridSize() : int
+ Reset() : void
+ Prepare() : void
+ Step(dt : float, diff : float, visc : float) : void
+ GetU(x : int, y : int) : void
+ GetU(x : float, y : float) : void
+ GetV(x : int, y : int) : void
+ GetV(x : float, y : float) : void
+ GetDens(x : int, y : int) : void
+ AddU(x : int, y : int, value : float) : void
+ AddV(x : int, y : int, value : float) : void
+ AddDens(x : int, y : int, value : float) : void
+ GetO(x : int, y : int) : boolean
+ SetO(x : int, y : int, t : boolean) : void
NSESolverGPU
+ Init(gridSize : int, iterCount : int) : void
+ GetGridSize() : int
+ Reset() : void
+ Prepare() : void
+ Step(dt : float, diff : float, visc : float) : void
+ GetU(x : int, y : int) : void
+ GetU(x : float, y : float) : void
+ GetV(x : int, y : int) : void
+ GetV(x : float, y : float) : void
+ GetDens(x : int, y : int) : void
+ AddU(x : int, y : int, value : float) : void
+ AddV(x : int, y : int, value : float) : void
+ AddDens(x : int, y : int, value : float) : void
+ GetO(x : int, y : int) : boolean
+ SetO(x : int, y : int, t : boolean) : void

Príloha C: Používateľská príručka

Vytvorený program je multiplatformový a funguje v operačných systémoch Windows, Linux a MACOS. Predpokladom úspešného portovania na ďalšie platformy je podpora OpenGL, verzia min. 2. Podmienkou je samozrejme podporovaný grafický hardvér (testovanie prebiehalo na grafických kartách fy NVIDIA a AMD/ATI poslednej alebo predposlednej generácie).

Inštalácia

Program využíva štandardné knižnice OpenGL (závisí na operačnom systéme) a knižnicu GLEW [29]. Linux-ová implementácia navyše používa knižnicu `librt.so`. Všetky potrebné knižnice (pre systémy Windows a Linux) sú umiestnené na dátovom nosiči, ktorý je prílohou tejto práce, v rovnakom adresári ako samotný program. Pre ostatné systémy je ich potrebné nainštalovať (konkrétny spôsob závisí od operačného systému).

Parametre a ovládanie programu

Program podporuje zmenu parametrov simulácie pomocou argumentov príkazového riadka. Dostupné parametre sú:

- *-b* – pustenie simulácie v benchmarkingovom móde, t.j. bez okna aplikácia a len na určitý čas
- *-c* – použitie CPU namiesto GPU (štandardne)
- *-d* – nastaví hodnotu difúzie
- *-f* – načíta súbor s definíciou síl a tlaku
- *-h* – vypíše informácie o používaní programu
- *-i* – pri benchmarkingovom behu zobrazí okno programu
- *-l* – nastaví dĺžku benchmarku (v milisekundách)
- *-n* – nastaví počet iterácií pri simulácii tekutín
- *-o* – načíta súbor s definíciou prekážok
- *-p* – nastaví počet častíc
- *-s* – nastaví rozmer mriežky pre simuláciu tekutín
- *-v* – nastaví hodnotu viskozity

Počas behu simulácie je možné použitím myšky pridávať do simulácie silu a zvyšovať hustotu / tlak v tekutine. Stlačením tlačidla myši v časti okna s vizualizáciou častíc (viď Obrázok 13. Grafické rozhranie programu) a jej pohybom sa aplikuje pridanie sily a tlak do simulácie. Beh simulácie je možné ovplyvniť nasledujúcimi klávesami (nie v benchmarkingovom móde):

- *Esc* – ukončí program
- *p* – pozastaví simuláciu (najprv simuláciu tekutiny a potom aj simuláciu častíc)
- *r* – obnoví pôvodnú pozíciu častíc pred spustením simulácie

- *s* – (de)aktivuje zdroje síl a tlaku

Formát súboru definujúceho sily:

Súbor obsahuje zoznam síl, každú na jednom riadku. Ten obsahuje (v poradí): X pozíciu sily, Y pozíciu silu, X smer vektoru sily, Y smer vektoru sily, pridanie hustoty za jednotku času. Pozícia sily je v percentách – kvôli premenlivým rozmerom mriežky. Príklad súboru:

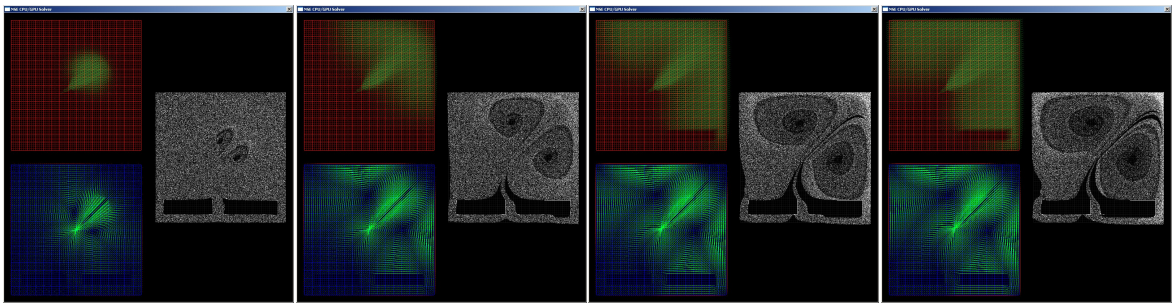
```
0.50 0.50 20.0 20.0 5.0
```

Formát súboru definujúceho prekážky:

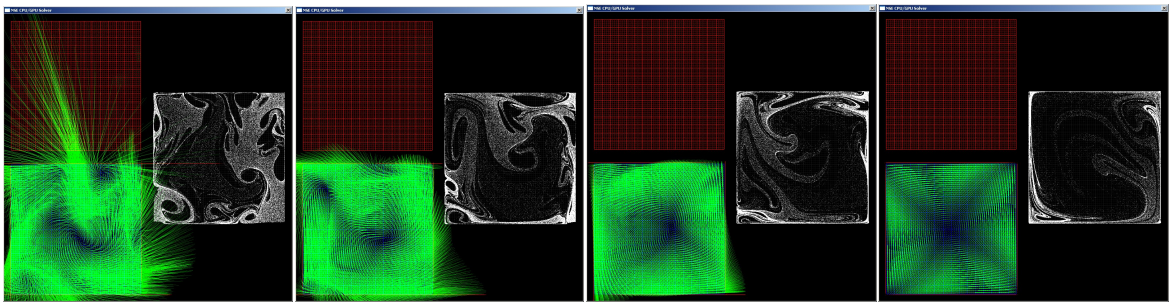
Súbor obsahuje zoznam prekážok, každú na jednom riadku. Ten obsahuje (v poradí): X min poloha prekážky, Y min poloha prekážky, X max poloha prekážky, Y max poloha prekážky. Poloha prekážky je v percentách – kvôli premenlivým rozmerom mriežky. Príklad súboru:

```
0.08 0.08 0.45 0.18
0.55 0.08 0.94 0.18
```

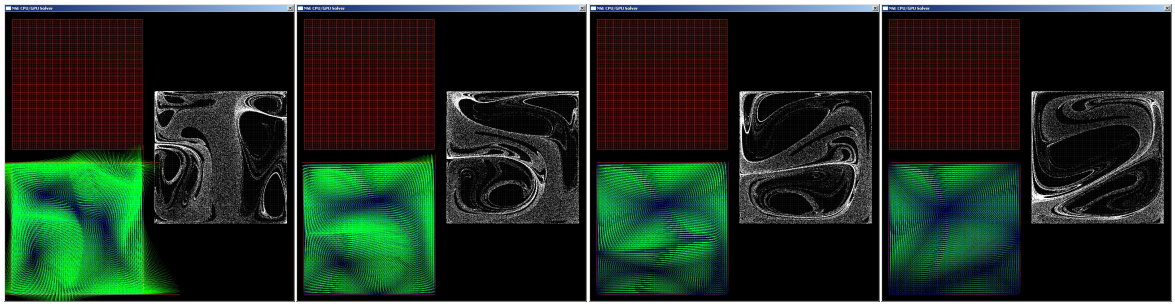
Príklady priebehu simulácie



Simulácia s konštantnými silami a prekážkami



Jeden dominantný vír



Viacero postupne zanikajúcich vírov

Príloha D: Zoznam použitého hardvéru

Dôležitým faktorom pri testovaní bolo použitie rôzneho hardvéru. Takto sa získalo dostatočné množstvo dát na ich spracovanie a vyvodenie záverov. Zoznam použitého hardvéru je v nasledujúcej tabuľke:

Použitý hardvér		
CPU	**GPU**	**iné**
Intel Core2 Duo @ 3 Ghz	NVIDIA GeForce 8800 GTS	2 GB RAM (WinXP Pro 32bit)
Intel Pentium 4 HT @ 2.6GHz	ATI Radeon X1950 Pro	1.5GB RAM(WinXP Pro 32bit)
Intel Core2 Quad @ 2.4 GHz	NVIDIA GeForce 8800 GTS	4 GB RAM (WinXP 64bit)
Intel Core2 Duo @ 2.6 Ghz	NVIDIA GeForce 8800 GTS	2 GB RAM (WinXP Pro 32bit)
Intel Core2 Duo @ 2.2 GHz	NVIDIA GeForce 8600 GT	1 GB RAM (Win Vista Home)
AMD Turion 64 X2@ 1.6 GHz	NVIDIA GeForce Go 7600	1 GB RAM (WinXP 32-bit)
Intel Core2 Duo @ 2.2GHz	NVIDIA GeForce 8600M GT	4 GB RAM (Win Vista 32bit)

Za spoluprácu pri poskytnutí hardvéru na vykonanie testov ďakujem (v abecednom poradí): Stanislav Bočinec, Peter Drahoš, Ivan Kišac, Pavol Lesňák, Michal Okresa, Jozef Pánik, Michal Pohančeník.

Príloha E: Obsah elektronického média

Elektronické médium je CD/DVD – nosič obsahujúci dokumentáciu totožnú s tlačenou formou, zdrojové súbory programu a použité pramene v elektronickej forme. Je priložené ku tlačenej forme dokumentácie. Štruktúra nosiča je nasledovná:

- */bin* – obsahuje spustiteľnú verziu programu s knižnicami a skriptami na spustenie testovania
    - */data* – obsahuje príklady súborov definujúcich prekážky a sily
- */doc* – dokumentácia k práci
    - */paper* – obsahuje samotný dokument
    - */sources* – použité pramene v elektronickej forme
- */src* – zdrojové súbory
    - */libs* – knižnice použité pri vytvorení programu
    - */solver* – zdrojové súbory programu
    - */work* – dokument vo formáte MS Word 2003, súbor s UML diagramami vo formáte JUDE a výsledky testovania vo formáte MS Excel 2003

Ostatné súbory nachádzajúce sa na CD/DVD nosiči mimo uvedených adresárov nie sú predmetom diplomovej práce.