VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

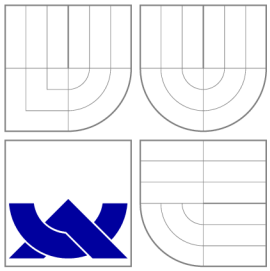
SIMULACE TEKUTIN V REÁLNÍM ČASE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

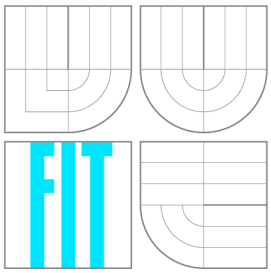
AUTOR PRÁCE
AUTHOR

Bc. MATÚŠ FEDORKO

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SIMULACE TEKUTIN V REÁLNÍM ČASE

REAL-TIME FLUID SIMULATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MATÚŠ FEDORKO

VEDOUCÍ PRÁCE
SUPERVISOR

prof. Dr. Ing. PAVEL ZEMČÍK

BRNO 2015

Abstrakt

Přimárním cílem této práce je simulace tekutin v reálním čase běžící na moderním programovatelným grafickým hardvéru. Práce začíná vysvětlením základních principů simulace tekutin se zaměřením na metodu Smoothed particle hydrodynamics. Následujúcí diskuze pak přináší stručný úvod do OpenCL jako i popis současného grafického hardvéru se zaměřením na odlišnosti při programování těchto specifických čipů ve srovnání s tradičními procesory. Poslední dvě kapitole této práce pak popisují návrh a implementaci problému.

Abstract

The primary concern of this work is real-time fluid simulation on modern programmable graphics hardware. It starts by introducing fundamental fluid simulation principles with focus on Smoothed particle hydrodynamics technique. The following discussion then provides a brief introduction to OpenCL as well as contemporary GPU hardware and outlines their programming specifics in comparison with CPUs. Finally, the last two chapters of this work, detail the problem analysis and its implementation.

Klíčová slova

simulace tekutin, smoothed particle hydrodynamics, uniformní mřížka, ray casting, OpenCL, GPU, OpenGL, Screen Space Fluid Rendering with Curvature Flow

Keywords

fluid simulation, smoothed particle hydrodynamics, uniform grid, ray casting OpenCL, GPU, OpenGL, Screen Space Fluid Rendering with Curvature Flow

Citace

Matúš Fedorko: Simulace tekutin v reálním čase, diplomová práce, Brno, FIT VUT v Brně, 2015

Simulace tekutin v reálním čase

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Dr. Ing. Pavla Zemčíka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Matúš Fedorko
29. května 2015

Poděkování

Rád bych zde poděkoval vedoucímu práce prof. Dr. Ing. Pavlovi Zemčíkovi za odborné vedení a pomoc při její tvorbě a také svým rodičům za vytrvalou podporu a trpělivost.

Obsah

Kapitola 1

Úvod

S neustále rastúcou úrovňou dnešných technológií a stále dokonalejšou schopnosťou modelovať okolitú realitu prostredníctvom počítača rastú aj nároky jej konzumentov. To čo stačilo, keď prišli prvé počítačové hry dnes zaujme snáď už len nadšencov. Je preto logické, že vývojári sa nepretržite snažia prichádzať s novými postupmi a nápadmi ako prehĺbiť virtuálne zážitky užívateľov, pričom jedným z týchto smerov sú aj stále náročnejšie a sofistikovanejšie fyzikálne simulácie javov bežne známych z reálneho sveta.

Jedným z takýchto fenoménov je i komplexné správanie tekutín, ktoré bez toho aby sme si to nejako zvlášť uvedomovali tvorí neoddeliteľnú súčasť nášho každodenného života. Vedci sa preto už pred desiatkami rokov snažili o exaktné popísanie ich princípov, čo napokon viedlo k vytvoreniu všeobecne uznávaného matematického modelu v podobe Navier-Stokes rovníc. Algoritmy, ktoré ich riešia sú však dobre známe pre svoju vysokú výpočtovú náročnosť a až v poslednej dobe s vývojom v oblasti moderného počítačového hardvéru začína byť realistické uvažovať o pridaní týchto postupov do hier a ďalších interaktívnych aplikácií.

Táto práca sa tak bude zaoberať možnosťami real-time simulácie a vizualizácie tekutín na dnešných počítačoch so zameraním na využitie moderných grafických procesorov k dosiahnutiu rozumného výkonu a presvedčivého zobrazenia.

V prvej kapitole budú ukázané základné princípy simulácie a vizualizácie tekutín spolu s popisom tých najdôležitejších algoritmov. Ku koncu potom ešte táto kapitola venuje určitý priestor krátkemu predstaveniu existujúcich softvérových riešení. V druhej kapitole budú uvedené informácie o typickej architektúre dnešných grafických procesorov a dostupných aplikačných rozhraniach, ktoré sa dajú k ich programovaniu využiť. Kapitoly o návrhu a implementácii zas predstavia zvolené riešenie a spôsob jeho realizácie a v závere budú potom zosumarizované získané poznatky a načrtnuté možné rozšírenia práce do budúcnosti.

Kapitola 2

Techniky počítačovej simulácie tekutín

Kvapaliny a plyny sú všade okolo nás, či už v podobe dažďa, vody v riekach a jazerách, alebo dymu z komínov. Vedci sa preto výskumu ich chovania venujú už dlhé desaťročia, čoho výsledkom sú takzvané Navier-Stokes rovnice, ktoré tieto javy presne opisujú. Jedná sa o nelineárne parciálne diferenciálne rovnice, ktoré je veľmi ťažké vyriešiť. Vo všeobecnosti sa riešia numerickými metódami, pretože analytické riešenie je možné nájsť len v najjednoduchších prípadoch. Našťastie pri programovaní počítačových hier, alebo vytváraní filmových efektov, resp. v počítačovej grafike ako takej, nezáleží väčšinou až toľko na úplnej numerickej presnosti, ale stačí, keď simulácia vyzerá dostatočne vierohodne.

Časom sa tak vyformovali dva základné postupy riešenia týchto rovníc:

- Eulerovská simulácia
- Lagrangeovská simulácia

Tie budú v tejto kapitole stručne predstavené spolu s metódami vizualizácie ich výstupov. V závere sa potom diskusia presunie na popis existujúcich softvérových riešení, ktoré tieto algoritmy implementujú.

2.1 Simulácia tekutín

Eulerovská simulácia

Táto metóda vychádza z predstavy fixného objemu, cez ktorý tečie nejaká tekutina. Tá je rozdelená do mriežky a v každom bode tejto mriežky má priradené vlastnosti ako rýchlosť, hustota, teplota alebo tlak. Pozície jednotlivých bodov, v ktorých tekutinu sledujeme sa tak nikdy nemenia.

Matematicky je jej stav v každom časovom okamihu modelovaný vektorovým rýchlostným poľom, čo je funkcia priradzujúca každému bodu v priestore vektor rýchlosti. Toto pole následne spôsobuje pohyb dymu, čiastočiek prachu, lístia, alebo iných objektov, ktoré sa v ňom nachádzajú.

Navier-Stokes rovnice sú presným popisom vývoja tohto rýchlostného poľa v čase. Za predpokladu, že poznáme jeho počiatočný stav a sily, ktoré na tekutinu pôsobia, umožňujú nám tieto rovnice zistiť jeho presnú zmenu počas nekonečne malého časového úseku. Navier-Stokes rovnice majú nasledujúci tvar:

$$\mathbf{u} \cdot \nabla = 0 \tag{2.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{2.2}$$

kde $\mathbf{u}$ predstavuje vektor rýchlosti, t je čas, ρ hustota, p tlak, ν viskozita a $\mathbf{f}$ reprezentuje vektor externých síl pôsobiacich na kvapalinu. Symbol ∇ je v rovniciach uvedený v 3 rôznych významoch, ktoré sú zhrnuté v nasledujúcej tabulke:

Operátor	**Názov**	**Definícia**
∇	Gradient	$\nabla p = (\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y})$
$\nabla \cdot$	Divergencia	$\nabla \cdot \mathbf{u} = \frac{\partial p}{\partial x} + \frac{\partial p}{\partial y}$
∇^2	Laplaceov Operátor	$\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2}$

Tabulka 2.1: Operátory vektorového kalkulu použité v Navier-Stokes rovniciach.

Prvá z rovníc hovorí, že súčet rýchlosti v danom bode a v jeho okolí musí byť nulový, inými slovami pokiaľ je hustota konštantná, tak hmota musí byť zachovaná.

Druhá z rovníc má 4 členy, ktoré majú nasledujúci význam:

- Prvý člen je advekcia. Tá vyjadruje myšlienku, že rýchlosť tekutiny spôsobuje transport objektov, ale aj fyzikálnych veličín, ako napr. už spomínaná hustota, alebo tlak, pozdĺž jej toku. Súčasne sa však prenáša aj rýchlosť samotná, čomu hovoríme samoadvekcia. Táto vlastnosť spôsobuje dobre známe vírivé správanie kvapalín a dymu a zároveň je aj príčinou prečo sú Navier-Stokes rovnice nelineárne.
- Druhý člen hovorí, že rýchlosť toku tekutiny sa mení podľa gradientu tlaku. V skutočnosti však tento člen spôsobuje divergenciu a preto sa v prípade simulácie s konštantnou hustotou vynecháva.
- Tretí člen je difúzny člen predstavujúci fakt, že rýchlosť má tendenciu difundovať pozdĺž gradientu rýchlosti. Konštanta ν reprezentuje viskozitu tekutiny, teda mieru toho ako veľmi sa tekutina rozplýva do okolia.
- Štvrtý člen zahŕňa externé sily pôsobiace na tekutinu. Tie môžu byť buď lokálne, pôsobiace len na určitú oblasť, alebo globálne pôsobiace na celú tekutinu. ako napr. gravitácia.

Oproti Lagrangeovmu prístupu sú častice nahradené hustotou tekutiny v bode mriežky, čo je spojitá funkcia, ktorá pre každý bod v priestore určuje koľko častíc sa tam aktuálne nachádza. Vývoj poľa hustôt opisuje rovnica:

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S \tag{2.3}$$

Táto rovnica je jednoduchšia ako rovnica 2.2 pretože je lineárna. Vyjadruje fakt, že zmeny v hustote kvapaliny v jednom kroku simulácie sú spôsobené troma príčinami vyjadrenými troma členmi pravej strany tejto rovnice:

- Prvý člen hovorí, že hustota by mala nasledovať (kopírovať) pole rýchlostí.
- Druhý člen hovorí, že hustota môže určitou mierou difundovať (rozplývať sa) do okolia.
- Tretí člen hovorí, že hustota rastie vďaka zdrojom hustoty.

Príkladom algoritmu, ktorý je čiastočne založený na tomto prístupe je i slávna metóda Josa Stama publikovaná pod názvom Stable Fluids [24].

Lagrangeovská simulácia

Táto metóda modeluje kvapalinu ako veľké množstvo častíc, ktoré sa môžu voľne pohybovať. Každá z nich má okrem rýchlosti, pozície a hmotnosti priradenú aj hustotu, tlak a silu, prípadne farbu, alebo ďalšie užitočné vlastnosti. Oproti Eulerovmu prístupu, tak simulácia nie je obmedzená na fixne danú doménu, ale tekutina sa môže ľubovoľne síriť do okolia.

Za predpokladu, že počet častíc zostáva počas simulácie konštantný a ani ich hmotnosť sa nijako nemení, tak táto formulácia umožňuje výrazne zjednodušiť používané výpočty. Prvá z Navier-Stokes rovníc, rovnica zachovania hmoty/kontinuity, je totižto v takomto prípade triviálne splnená a netreba ju vôbec počítať.

Pre pohyb častíc tekutiny v Lagrangeovom modeli je možné z Navier-Stokes rovníc odvodiť vzťah:

$$\frac{d\mathbf{v}_i}{dt} = g - \frac{1}{\rho_i}\nabla p + \frac{\mu}{\rho_i}\nabla^2 v \tag{2.4}$$

ktorý hovorí, že na zrýchlenie častice majú vplyv 3 rôzne druhy pôsobenia. Prvou skupinou vplyvov sú externé sily ako je gravitácia, vietor, alebo pôsobenie užívateľa v rámci virtuálneho sveta simulácie. Druhou príčinou sú zmeny tlakových síl vnútri kvapaliny a tretím faktorom je viskózne pôsobenie, ktoré v podstate určuje ako rýchlo kvapalina tečie.

Smoothed particle hydrodynamics

Smoothed particle hydrodynamics je metóda simulácie tekutín založená na Lagrangeovom časticovom prístupe. Prvý krát ju predstavili jej autori R. A. Gingold, J. J. Monaghan a L. B. Lucy v roku 1977 k riešeniu problémov v astrofyzike. Postupne si však našla uplatnenie aj v mnohých ďalších odvetviach výskumu vrátane balistiky, vulkanológie, oceánografie a už spomínanej simulácie tekutín.

Všetky častice majú v SPH definovanú priestorovú vzdialenosť h, obyčajne nazývanú aj vyhladzovací polomer. Tá určuje veľkosť okolia v rámci ktorého budú pomocou zvolenej kernel funkcie vlastnosti týchto častíc priemerované spolu s odpovedajúcimi vlastnosťami susedných častíc. V praxi to potom znamená, že určitá vlastnosť častice ako je napríklad hustota, alebo tlak, sa určí ako súčet príspevkov odpovedajúcich vlastností všetkých jej susedov v definovanom okolí h, pričom tie sú ešte váhované na základe ich vzdialenosti od aktuálne počítanej častice.

Pre ľubovoľnú vlastnosť SPH častice umiestnenej v bode $\mathbf{r}$ tak platí nasledujúca všeobecná rovnica:

$$A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(|\mathbf{r} - \mathbf{r}_j|, h) \tag{2.5}$$

kde m_j predstavuje hmotnosť častice j, A_j niektorú z jej vlastností, ρ_j hustotu a W označuje už spomínanú kernel funkciu nazývanú aj vyhladzovacie jadro, ktoré by ale malo splňovať niekoľko nutných podmienok.

Prvou z nich je takzvaná normalizačná podmienka, čo znamená že pre W musí platiť nasledujúca rovnica:

$$\int_r W(\mathbf{r}, h)dr = 1 \tag{2.6}$$

Druhou je, že vo vzdialenosti väčšej alebo rovnej ako h musí mať jadro nulovú hodnotu. Okrem toho by však malo byť aj pozitívne a párne, čo znamená, že rovnosť $W(\mathbf{r}, h) = W(-\mathbf{r}, h)$ by mala byť pravdivá.

K výpočtu jednotlivých členov rovnice 2.4 zavádza metóda SPH niekoľko aproximácií. Prvou z nich je rovnica výpočtu hustoty. Tá je závislá len od hmotností jednotlivých častíc v jej okolí a má nasledujúci tvar:

$$\rho_i = \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{2.7}$$

Hodnota gradientu tlaku je aproximovaná rovnicou:

$$\frac{\nabla p_i}{\rho_i} \approx \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla W \left(\mathbf{r}_i - \mathbf{r}_j, h \right) \tag{2.8}$$

Pri určovaní tlaku p_i a p_j potrebného v tomto vzťahu je možné vychádzať zo zákona ideálneho plynu, na základe ktorého sa dá dopracovať k vzorcu:

$$p = k \left(\rho - \rho_0 \right) \tag{2.9}$$

v ktorom ρ predstavuje hustotu aktuálnej častice spočítanú podľa 2.7, ρ_0 predstavuje kľudovú hustotu tekutiny a k je konštanta plynovej tuhosti pre túto tekutinu. Tlak potom v kvapaline ovplyvňuje veľkosť sily akou sú od seba navzájom jednotlivé častice odpudzované.

Posledný, viskózny člen rovnice **??** určuje mieru toho ako veľmi budú mať jednotlivé častice tendenciu pohybovať sa rovnakým smerom ako jedna súdržná masa. Príkladom materiálu s vysokou viskozitou môže byť bahno, alebo med, na druhej strane, opačným príkladom nízko viskózneho materiálu je voda. Viskozita je teda inak povedané miera odolnosti voči toku a v SPH je definovaná pomocou nasledujúcej aproximácie:

$$\frac{\mu}{\rho_i} \nabla^2 v_i \approx \frac{\mu}{\rho_i} \sum_j m_j \left(\frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \right) \nabla^2 W \left(\mathbf{r}_i - \mathbf{r}_j, h \right) \tag{2.10}$$

V rovniciach všetkých uvedených aproximácií 2.7 2.8 2.10 predstavujú členy $\mathbf{r}_i$ a $\mathbf{r}_j$ pozície aktuálne spracovávanej i-tej a jej susednej j-tej častice. Člen m_j určuje hmotnosť j-tej častice. Podobne aj ρ_j, ρ_i, v_j, v_i a p_j a p_i sú postupne hustota, rýchlosť a tlak j-tej a i-tej častice v simulácii. Zvyšné dva členy μ a h sú koeficient viskozity, ktorý čím bude vyšší tým bude vyššia aj viskozita a naopak a veľkosť vyhlaďovacieho polomeru kernel funkcií.

Praxou sa osvedčilo niekoľko bežne používaných vyhladzovacích jadier. V prípade hustoty 2.7 sa používa jadro polynómu 6. stupňa:

$$W(\mathbf{r}_i - \mathbf{r}_j, h) = \frac{315}{64\pi h^9} \left(h^2 - |\mathbf{r}_i - \mathbf{r}_j|^2 \right)^3 \tag{2.11}$$

Pri výpočte gradientu tlaku 2.8 je to tzv. spiky jadro, ktoré dostalo názov podľa svojho špicatého grafu. Jeho rovnica je definovaná následovne:

$$W(\mathbf{r}_i - \mathbf{r}_j, h) = \frac{15}{\pi h^6} \begin{cases} (h - |\mathbf{r}_i - \mathbf{r}_j|)^3 & 0 \leq |\mathbf{r}_i - \mathbf{r}_j| \leq h \\ 0 & |\mathbf{r}_i - \mathbf{r}_j| > h \end{cases} \tag{2.12}$$

a jeho gradient má podobu:

$$\nabla W(\mathbf{r}_i - \mathbf{r}_j, h) = \frac{-45}{\pi h^6} (h - |\mathbf{r}_i - \mathbf{r}_j|)^2 \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|} \tag{2.13}$$

Posledným je vyhladzovacie jadro používané pri výpočte viskozity:

$$W(\mathbf{r}_i - \mathbf{r}_j, h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{|\mathbf{r}_i - \mathbf{r}_j|^3}{2h^3} + \frac{|\mathbf{r}_i - \mathbf{r}_j|^2}{h^2} + \frac{h}{2|\mathbf{r}_i - \mathbf{r}_j|} - 1 & 0 \leq |\mathbf{r}_i - \mathbf{r}_j| \leq h \\ 0 & |\mathbf{r}_i - \mathbf{r}_j| > h \end{cases} \tag{2.14}$$

ktorého laplacián potrebný v rovnici 2.10 má tvar:

$$\nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) = \frac{45}{\pi h^6} (h - |\mathbf{r}_i - \mathbf{r}_j|) \tag{2.15}$$

Dosadením vyhladzovacích jadier do aproximovaných rovníc dostaneme vzťahy:

$$\rho_i \approx \sum_j m_j \frac{315}{64\pi h^9} \left(h^2 - |\mathbf{r}_i - \mathbf{r}_j|^2\right)^3 \tag{2.16}$$

$$\frac{\nabla p_i}{\rho_i} \approx \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right) \frac{-45}{\pi h^6} (h - |\mathbf{r}_i - \mathbf{r}_j|)^2 \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|} \tag{2.17}$$

$$\frac{\mu}{\rho_i} \nabla^2 v_i \approx \frac{\mu}{\rho_i} \sum_j m_j \left(\frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j}\right) \frac{45}{\pi h^6} (h - |\mathbf{r}_i - \mathbf{r}_j|) \tag{2.18}$$

ktoré je už možno priamo použiť k výpočtu simulácie.

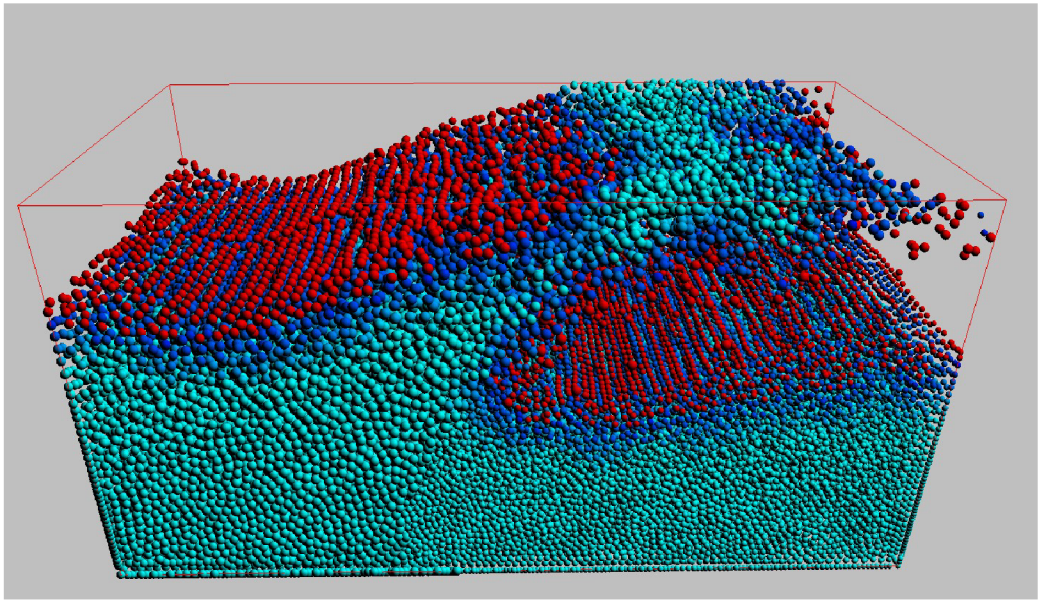
Naivná implementácia SPH algoritmu má časovú zložitosť $O(n^2)$ pretože je potrebné pri výpočte každej častice skontrolovať, ktoré ďalšie častice s ňou ešte susedia. Za predpokladu konečného dosahu väčšiny používaných vyhladzovacích jadier je však možné tento naivný algoritmus až rádovo zlepšiť použitím pokročilých vyhľadávacích štruktúr, ktoré budú podrobnejšie popísané v ďalších častiach tejto práce 5.1.

2.2 Zobrazovanie tekutín

Point Sprites

Táto metóda neposkytuje žiadne realistické zobrazenie, ale pre svoju jednoduchosť a rýchlosť je prakticky štandardom pri prvotnej vizualizácii ľubovoľného časticového systému a taktiež sú na nej založené aj niektoré pokročilejšie zobrazovacie metódy.

Pod pojmom point sprite sa väčšinou chápe štvorcová ploška umiestnená stredom v nejakom konkrétnom bode, na ktorej je nanesená textúra. V prípade vizualizácie výstupu zo Smoothed Particle Hydrodynamics by touto textúrou bol obrázok guličky, ktorý je však možné veľmi jednoducho spočítať aj analyticky pomocou rovnice jednotkovej gule (gule s polomerom 1).



Obrázek 2.1: Vizualizácia metódou point sprites. Obrázok je prebraný z [27]

Pri realizácii tohto algoritmu v OpenGL je potrebné najprv povoliť vykresľovanie bodov ako point spritov príkazom `glEnable(GL_POINT_SPRITE)`, prípadne ešte špecifikovať veľkosť point spritu príkazom `glPointSize`. Následne sa jednotlivé častice vykreslia ako body pomocou primitíva `GL_POINTS`. Vo vertex shaderi sa bežným spôsobom ztransformujú do clip space a OpenGL potom na tejto ztransformovanej pozícii vyrasterizuje štvorec, ktorého súradnice sú vo fragment shaderi dostupné prostredníctvom vstavanej premennej `gl_PointCoord`.

Keďže sme uvažovali jednotkovú guľu s polomerom 1, tak pre všetky body, ktoré spĺňajú nerovnosť $x^2 + y^2 <= 1$ môžme spočítať osvetľovací model a všetky ostatné body môžme zahodiť použitím príkazu `discard`. Chýbajúcu z-ovú súradnicu je možné jednoducho dopočítať zo vzťahu $z = \sqrt{1 - (x^2 + y^2)}$.

Avšak kvôli tomu, že výsledná guličkа je v skutočnosti pre OpenGL len plochý 2D obrázok, treba ešte ručne dopočítať hĺbku každého fragmentu, ktorú by inak OpenGL počítalo za programátora automaticky.Tento fakt môže mať mierne negatívny dopad na výkon, keďže znemožnuje OpenGL optimalizáciu pomocou early depth testu, avšak v rámci vizualizácie časticových systémov je táto metóda stále mnohonásobne rýchlejšia ako kreslenie množstva malých častíc pomocou polygónov a inštancovania.

Ďalším problémom, ktorý je pri tejto metóde potrebné riešiť je zmena veľkosti častice pri priblížení a oddialení kamery. Pri práci s polygónmi je táto funkcionalita vďaka transformáciám vo vertex shaderi automatická, ale pri point spritoch ju musí programátor pridať ručne. K tomuto účelu je vo vertex shaderi vstavaná GLSL premenná `gl_PointSize`, do ktorej môže programátor zapísať výslednú veľkosť bodu v pixeloch. V niektorých implementáciách OpenGL je ešte nutné povoliť aj atribút `GL_PROGRAM_POINT_SIZE`.

Takto vyrasterizované guličky však trpia ešte jedným nedostatkom a tým je nefunkčné perspektívne skreslenie. V normálnom prípade by pri použití perspektívnej projekcie mali mať vyrasterizované guličky mierne pretiahnutý tvar. V prípade vizualizácií časticových systémov je však tento postup väčšinou dostatočný, pretože výsledné vyrasterizované častice sú často pomerne malé na to, aby bol tento efekt viditeľný.

Nepríjemnejším problémom je skôr fakt, že pri point spritoch je viditeľnosť jednotlivých častíc určovaná len na základe jedného bodu, ktorým je ich pozícia. Potom sa môže stať, že hoci by štvrtina častice bola ešte viditeľná, tak OpenGL ju už zahodí, pretože jej stredový bod je mimo pohľadového telesa. Toto potom môže spôsobovať hlavne pri väčších point spritoch nepríjemné preblikávanie.

Riešením je kresliť častice ako obdĺžniky zložené z dvoch trojuholníkov. Tie možu byť generované buď geometry shaderom, alebo vo vertex shaderi za pomoci vstavaných premenných `gl_InstanceID`, prípadne `gl_VertexID`. Ak sa v tomto prípade ocitne niektorý z rohov odĺžniku mimo podhľadové teleso, tak OpenGL ho oreže namiesto toho, aby ho hneď celý zahodilo.

Ray Casting/Ray Tracing

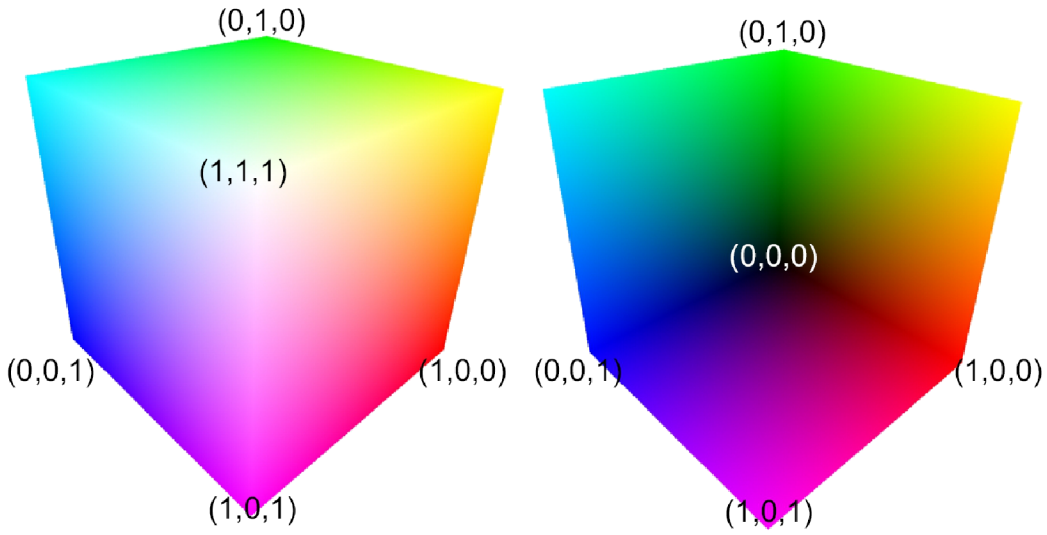
Ray Tracing je dobre známa technika renderovania vysoko kvalitných realistických obrázkov. Jej základom je vrhanie lúčov cez jednotlivé pixely obrazovej roviny a hľadanie ich priesečníku s objektami v scéne. Lúče vrhnuté priamo z kamery cez obrazovú rovinu sa nazývajú primárne lúče a po dopade na objekt sa môžu ďalej rekurzívne odrážať a lámať podľa príslušných fyzikálnych zákonov, prípadne je možné využiť lúč vrhnutý z povrchu objektu smerom k zdroju svetla na určenie miery jeho zatienenia. Tento rekurzívny výpočet skončí v okamihu keď lúč narazí na objekt, na ktorom už k žiadnemu lomu ani odrazu nedochádza, prípadne ak dosiahne stanovenú maximálnu úroveň zanorenia.

Ray Casting je v podstate rovnaký algoritmus, ale s tým rozdielom, že do scény sa vrhajú len primárne lúče a teda nie je rekurzívny. Jeho prvým krokom je opäť výpočet priesečníku lúča vrhnutého z kamery cez pixel v obrazovej rovine s objektami v scéne. Aby bolo možné tento priesečník efektívne počítať je potrebné najprv extrahovať povrch z častíc simulácie. Pokiaľ je ale pri simulácii používaná nejaká akceleračná štruktúra typu uniformnej mriežky, je možné s výhodou túto štruktúru využiť a aplikovať na renderovanie techniky dobre známej z priameho vykresľovania volumetrických dát. Tými sú napríklad Volume Ray Casting, alebo Texture based volume rendering, ktorá bude popísaná v nasledujúcej stati.

Pri realizácii Volume Ray Castingu v OpenGL je možné efektívne využiť podporu rasterizačného hardvéru a 3D textúr. Samotný proces vykresľovania potom prebieha tak, že OpenGL sa pošle geometria obalového telesa okolo volumetrických dát spolu so súradnicami priesečníkov lúča s týmto telesom v jeho vrcholoch. OpenGL už ďalej automaticky zinterpoluje zadané priesečníky počas rasterizácie a ray casting sa tak v rámci fragment shaderu zredukuje len na jednoduchý Ray Marching. Ten pomocou vyrasterizovaných vstupných a výstupných priesečníkov už iba vzorkuje dáta v 3D textúre podľa zvoleného kroku a akumuluje farbu jednotlivých voxelov po ceste. Výpočet končí v momente, keď sa alfa zložka nasýti, alebo keď sa doiteruje k výstupnému priesečníku.

Tento prístup funguje dobre pri renderovaní javov ako hmla, dym, alebo vodná para, ale už nie je veľmi vhodný pre vizualizáciu kvapalín. Pri kvapalinách je totižto dôležité vedieť, kde sa nachádza povrch, pretože väčšina opticky zaujímavých fyzikálnych fenoménov sa odohráva práve tam. Volume rendering algoritmus je preto nutné modifikovať tak, že osvetľovací model a všetky optické vlastnosti sa budú vyhodnocovať až v okamihu keď lúč počas Ray Marchingu narazí na povrch kvapaliny. Tento prístup je použitý napríklad aj v článku 30. Real-Time Simulation and Rendering of 3D Fluids z knihy GPU Gems 3 [10].

Jedným z problémov tejto techniky je ale fakt, že kamera sa musí vždy pozerať na kvapalinu zvonka, nemôže sa nachádzať vnútri obalového telesa tekutiny, pretože inak by HW rasterizér orezal jeho predné steny a vo fragment shaderi by potom pri konštrukcii lúča



Obrázek 2.2: Vyrasterizované vstupné a výstupné priesečníky lúča s obalovým telesom tekutiny. Obrázok je prebraný z [26]

chýbali súradnice vstupných bodov. Riešenie tohto problému je opäť ukázané v kapitole 30. z knihy GPU Gems 3 [10].

Pri výpočte osvetlenia je nevyhnutne dôležité poznať aj normálu k počítanému povrchu. Tú je možné zistiť pomocou derivácií, typicky metódou centrálnych diferencií, ktorá funguje tak, že pre aktuálne počítaný voxel zistí rozdiel jeho hodnoty oproti susedným voxelom v x-ovom, y-ovom a z-ovom smere.

Texture based volume rendering

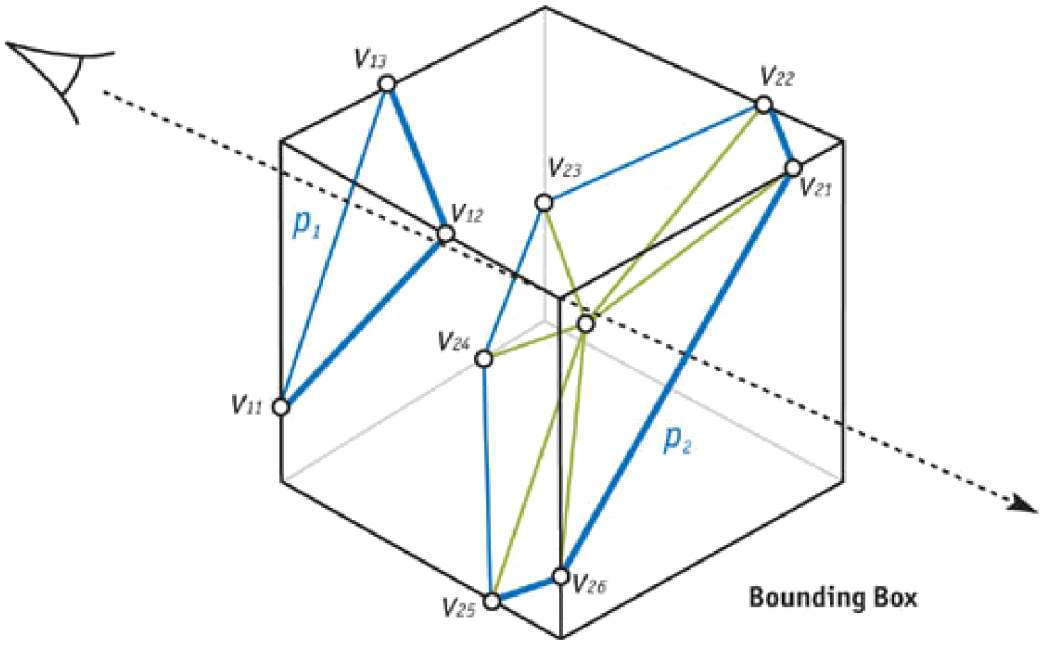
Texture based volume rendering je jedna z techník používaných napr. v medicíne pri vizualizácii volumetrických dát. Pokiaľ je ale pri simulácii používaná akceleračná štruktúra podobná uniformnej mriežke, je možné túto metódu adaptovať aj na vizualizáciu tekutín.

Metóda je založená na vykresľovaní série pomocných polygónov, ktorými sú väčšinou obdĺžniky o veľkosti zhodnej s aktuálnou veľkosťou viewportu a natočenými kolmo na smer, ktorým sa pozerá kamera do scény. Tieto pomocné polygóny, v odbornej literatúre nazývané aj proxy geometria, sú následne využívané na vzorkovanie volumetrických dát, ktoré sú ideálne uložené v 3D textúre, prípade v poli 2D textúr.

Komplikácie nastávajú až pri rotácii scény. Pri malých uhloch je možné rotovať aj s pomocnými polygónmi, avšak tento prístup úplne zlyháva pri otoční scény o 90 stupňov, pretože v tomto prípade sa scéna dostane do polohy, keď je kamera natočená presne na hrany týchto pomocných obdĺžnikov. Lepším prístupom je preto rotovať textúrovacie súradnice, ktoré sa do shaderov predávajú spolu s proxy geometriou.

Počet pomocných polygónov je voliteľný, ale vo všeobecnosti platí, že čím ich je viac, tým lepšie výsledky, samozrejme za cenu vyšších výpočtových nárokov, je možné dosiahnuť a naopak.

V prípade ak je scéna natočená k pozorovateľovi takým spôsobom, že ten sa pozerá priamo cez jeden z jej rohov, tak napríklad niekoľko prvých obdĺžnikov proxy geometrie pokrýva len časť 3D textúry s volumetrickými dátami a ostatné pixely týchto proxy polygónov ležia mimo. Jednou z možných optimalizácií je preto orezanie pomocnej proxy geometrie, čím sa zníži množstvo pixelov, ktoré treba vyrasterizovať.



Obrázek 2.3: Vykresľovanie pomocou orezanej proxy geometrie. Obrázok je prebraný z [18]

Technika založená na podobnom princípe je použitá napr. na vizualizáciu dymu v demo programe Teapot k aplikácii CodeXL od spoločnosti AMD.

Marching Cubes

Marching cubes je dobre známa metóda extrakcie trojuholníkovej siete (izoplochy) z volumetrických dát. Hoci sa jedná už o staršiu metódu publikovanú v roku 1987, s menšími obmenami sa používa dodnes.

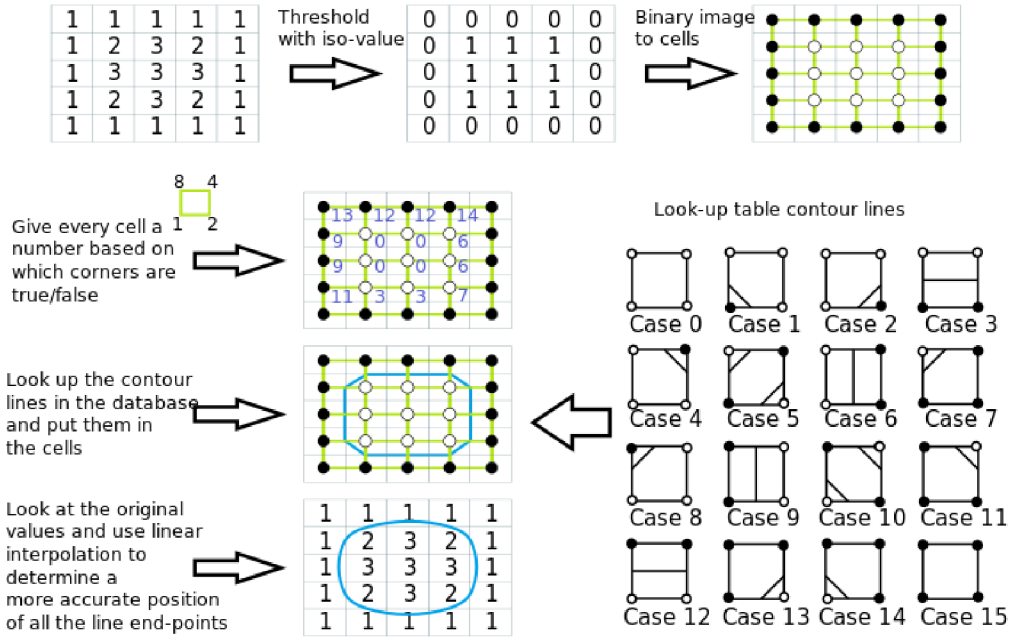
Princíp jej 2D verzie, niekedy nazývanej aj marching squares, je veľmi dobre ilustrovaný nasledujúcim obrázkom 2.4 z wikipédie.

Zo skalárneho poľa hodnôt, ktoré v prípade tekutiny reprezentuje počet častíc v danej bunke uniformnej mriežky sa najprv stanoví, ktoré bunky ešte obsahujú tekutinu a ktoré už nie. To je možné určiť na základe nejakej zvolenej prahovej hodnoty, napr. bunka ktorá patrí kvapaline musí obsahovať aspoň 5 častíc.

Následne algoritmus prechádza týmto skalárnym poľom po jednotlivých bunkách a pre každý blok o veľkosti 2x2 (na obrázku sú vzniknuté bloky vyznačené zelenou farbou) spočíta index do lookup tabuľky predpočítaných konfigurácií, ktoré sú navrhnuté presne tak, aby hrany určené susednými blokmi spolu vždy vytvorili súvislý povrch. Vypočítané indexy sú v prípade marching squares 4-bitové čísla, kde každý bit odpovedá jednému rohu v rámci bloku. Pokiaľ je skalárna hodnota v bunke odpovedajúcej tomuto rohu vyššia ako prah, tak príslušný bit indexu sa nastaví na jedničku, inak zostane na nule.

Za účelom dosiahnutia presnejších výsledkov je ešte možné predpočítané priesečníky hrán so stenami bloku posúvať pre každý blok individuálne na základe množstva častíc v každej bunke (veľkosti príslušnej hodnoty v skalárnom poli). Toto je možné vďaka tomu, že susedné bloky zdieľajú tie isté rohové bunky a vygenerovaný povrch tak zostane súvislý.

Problém nastáva akurát pri konfiguráciách označených v obrázku ako "Case 10" a "Case 5". V tomto prípade nie je jasné, či sa jedná 2 oddelené hrany 2 rôznych objektov, tak ako



Obrázek 2.4: Princíp algoritmu marching cubes v 2D. Obrázok je prebraný z [18]

je to tam naznačené, alebo sa jedná o jednu veľmi tenkú plošku, ktorá je natiahnutá medzi oboma rohmi.

Jedným z možných riešení tohto problému je vypočítanie priemernej hodnoty zo 4 buniek v bloku a pokiaľ je tento priemer vyšší ako prahová hodnota, tak zameniť konfiguráciu 10 za konfiguráciu 5, resp. naopak.

Všetky princípy uvádzané doteraz je možné rozšíriť aj do 3D, s tým rozdielom, že v 3D majú jednotlivé bloky veľkosť 8 buniek (2x2x2), na reprezentáciu vypočítaných indexov je treba 8 bitov a množstvo konfigurácií, ktoré je nutné predpočítať narastie na 256.

Z uvedeného taktiež vyplýva, že rozlíšenie skalárneho poľa bude mať veľký vplyv na kvalitu výsledkov a zároveň aj na výpočtové nároky, čím bude rozlíšenie jemnejšie, tým bude výsledná povrchová reprezentácia presnejšia a jej vypočet pomalší a naopak.

Alternatívou k marching cubes je marching tetrahedra. Táto metóda funguje prakticky na rovnakom princípe ako marching cubes, ale s tým rozdielom, že namiesto kociek používa na vzorkovanie volumetrických dát tetraédre (najmenší 3D objekt). Jej výhodou je to, že nemá žiadne nejednoznačné konfigurácie a zároveň počet predpočítaných konfigurácií je výrazne nižší, vďaka tomu, že tetraéder má len 4 rôzne vrcholy.

Point Splatting

Táto skupina metód je založená na vykresľovaní častíc simulácie viditeľných z pohľadu kamery ako malých rozmazaných fliačkov. Jednou z metód patriacich do tejto skupiny je aj metóda Screen Space Fluid Rendering with Curvature Flow prezentovaná v článku [21], ktorá bude bližšie popísaná v kapitolách zaoberajúcich sa realizáciou programu.

2.3 Existujúci softvér

Existuje obrovské množstvo rôznych pluginov, knižníc a nástrojov, ktoré nejakým spôsobom riešia simuláciu tekutín. Táto podkapitola sa preto zameria len na vybranú skupinu niekoľkých najpopulárnejších a najzaujímavejších z nich a nemožno ju tak považovať za vyčerpávajúci prehľad všetkých dostupných možností.

GameWorks

GameWorks predstavuje middleware od firmy NVidia, teda súbor predpripravených algoritmov a knižníc, ktoré môžu vývojári použiť na pridanie pokročilých vizuálnych efektov a fyzikálnych simulácií do svojich hier. Tými sú napríklad simulácia vlasov, kožušiny, soft body a rigid body systémov, simulácia kvapalín a plynov, simulácia látky a mnohé ďalšie.

Má štyri hlavné moduly, medzi ktoré patrí `VisualFX`, `PhysX`, `Core SDK` a `OptiX`. Prvý z nich, VisualFX, je určený najmä na renderovanie pokročilých vizuálnych efektov ako je napríklad depth of field, ambient occlusion pomocou algoritmu HBAO+, global illumination, alebo realistické renderovanie kože a očí. Súčasťou VisualFX sú však aj simulácia oceánu WaveWorks a už spomínaná simulácia vlasov v podobe knižnice HairWorks. Druhý zo zoznamu je modul PhysX, ktorý predstavuje riešenie pre modelovanie rôznych časticových systémov, tekutín, látky a ďalších fyzikálnych fenoménov, ale môže byť využitý aj k realistickej simulácii deštrukcií. Modul Core SDK poskytuje API umožňujúce lepšie prepojenie s NVidia produktami, napríklad s aplikáciou GeForce Experience. Posledný zo zoznamu, modul OptiX, predstavuje programovateľný framework pre vykresľovanie pomocou algoritmu sledovania lúča. Ten zas môže byť využitý k presnému výpočtu ambient occlusion efektu, alebo k zapečeniu svetla do textúr počas vývoja hry.

Veľmi zaujímavou je aj nová verzia technológie FleX, ktorá je vyvíjaná ako súčasť modulu PhysX. Tá vďaka jednotnej časticovej reprezentácii umožňuje spolu kombinovať rôzne fyzikálne simulácie, ktoré boli tradične počítané oddelene špecializovanými algoritmami a používané najmä v oblasti offline programov na vytváranie vysoko kvalitných vizuálnych efektov. FleX tak plne podporuje two-way coupling, čo znamená, že častice z rôznych simulácií na seba vzájomne pôsobia a je potom jednoduché pomocou neho vytvoriť simuláciu, v ktorej napríklad rigid body objekty plávajú na hladine vody, alebo interagujú s látkou, v ktorej sú nasypané a to všetko v reálnom čase. Táto technológia plne podporuje GPU akceleráciu prostredníctvom CUDA a je integrovaná aj v rámci herného enginu Unreal Engine.

GameWorks je však stále primárne proprietárne riešenie dodávané bez zdrojového kódu aj keď od roku 2015 boli niektoré jeho časti uvoľnené ako open source a zprístupnené registrovaným vývojárom v zdrojovej podobe v rámci repozitáru na github [8].

Táto dlhodobá uzavretosť ale znemožňovala konkurencii optimalizovať hry pre svoj vlastný hardvér, čo AMD viedlo k úvahám o spustení konkurenčného projektu OpenWorks, ktorý by bol od začiatku plne open source [23]. V súčasnosti je však tento projekt stále len v rámci úvah a AMD sa ho ešte nepodarilo spustiť.

Fluids

Fluids je interaktívny program vytvorený R. C. Hoetzleinom za účelom výskumu efektívnych metód implementácie časticových systémov o veľmi veľkých rozmeroch ako je napríklad simulácia oceánu.

Jeho staršia verzia Fluids v.2 dokázala zvládnuť maximálne 65536 častíc a nemala ešte plnú podporu GPU, konkrétne posledný integračný krok jej SPH algoritmu prebiehal na CPU. Plná podpora GPU bola pridaná až vo verzii 3, ktorá zároveň aj výrazne navýšila maximálny počet simulovaných častíc. Táto verzia tak dokáže efektívne modelovať až jednotky miliónov častíc. Autor uvádza, že program je schopný dosiahnuť framerate okolo 4 snímkov za sekundu pri simulácii približne jedného milióna častíc na dnes už pomerne starej a málo výkonnej karte NVidia GeForce GTX 460M.

K implementácii používa platformu CUDA, z čoho vyplýva, že nie je prenositeľná na iný ako NVidia hardvér. Program je však celý open source a vývojári si ho tak môžu ľubovoľne upraviť a pridať podporu aj pre ďalšie zariadenia.

Fluids v.3 je ale viac-menej len technologickým demom a okrem rýchlej simulácie množstva častíc nepodporuje žiadnu ďalšiu sofistikovanejšiu funkcionalitu akou je napríklad detekcia kolízií so zložitejším okolím, alebo interakcia s ostatnými časticovými systémami. Program dokonca okrem jednoduchej Point Sprite metódy neimplementuje ani žiadnu pokročilejšiu vizualizáciu.

RealFlow

RealFlow predstavuje profesionálny softvér od madridského vývojára Next Limit Technologies určený primárne k ultra realistickej offline simulácii obrovskej masy kvapaliny spolu s doplňujúcimi efektami ako sú rozprsknuté kvapky vody vygenerované dopadajúcimi vlnami, alebo pena a hmla, ktoré pri veľkorozmerných simuláciách tieto javy často sprevádzajú. Okrem toho však umožňuje počítať aj kolíziu s komplexným virtuálnym okolím a obohatiť tak simuláciu kvapaliny o interakciu s rigid a soft body systémami, prípadne modelovať popraskanie 3D objektov a mnohé ďalšie efekty. Program je teda určený v prvom rade k použitiu vo filmovom priemysle a reklame, k vytváraniu trailerov k hrám a rôznym kinematických prestrihom, ktoré sa v nich často vyskytujú.

K modelovaniu malých a stredne veľkých simulácií využíva RealFlow predovšetkým algoritmus Smoothed Particle Hydrodynamics, ktorý je schopný verne zachytiť všetky spletité detaily v pohybe kvapaliny. Pri veľmi veľkých rozmeroch však prestáva byť táto metóda efektívna, pretože na pokrytie celej simulácie vyžaduje obrovské množstvo častíc, čím sa automaticky rapídne zvyšujú aj nároky na výpočtový výkon a čas. RealFlow tento problém rieši hybridnou simuláciou, ktorá kombinuje eulerovský prístup spolu s Lagrangeovou metódu, čo umožňuje simulovať jadro kvapaliny, kde nedochádza k toľkým vírivým pohybom pomocou relatívne hrubej fixnej mriežky a povrch spolu s ostatnými časťami tekutiny, kde je naopak potrebné zachovať čo najväčší detail modeluje časticami. Vývojár Next Limit Technologies tento algoritmus nazýva HYBRIDO.

K zrýchleniu výpočtov je podporované aj paralelné spracovanie na viacerých počítačoch naraz. Užívateľ môže buď označiť nezávislé časti scény, ktoré sa budú spracovávať simultánne, alebo nechať paralelne simulovať viaceré verzie tej istej scény ale s alternatívnymi nastaveniami. Prítomná je i GPU akcelerácia pomocou OpenCL. Tú však vývojári programu stále považujú len za experimentálnu, čo je dané najmä obrovskými pamäťovými nárokmi takéhoto typu ultra realistických aplikácií, ktoré ďaleko presahujú kapacitu väčšiny dnešných grafických kariet.

Výsledky je možné vizualizovať extrakciou povrchu z častíc kvapaliny a následným exportom tohto povrchu do niektorého zo štandardných súborových formátov podporovaných populárnymi 3D aplikáciami. Druhou, rýchlejšou možnosťou, je využitie vstavaného render kitu v kombinácii s realistickým rendererom ako je napríklad Maxwell render.

Ďalšou z mnohých vymožeností RealFlow je plná kontrola a schopnosť prispôsobiť program vlastným požiadavkám pomocou rozšírení a pluginov v jazykoch Python a C++.

Blender 3D

Blender je populárny open source 3D grafický program umožňujúci v prvom rade vytvárať polygonálne modely a vizualizovať ich pomocou interného vysoko kvalitného realistického rendereru nazývaného Cycles. Okrem toho však obsahuje aj množstvo ďalších nástrojov uľahčujúcich každodennú prácu dizajnérov pri vytváraní digitálneho obsahu. Sú to napríklad funkcie určené na skladanie (compositing) scén, editáciu videa, alebo možnosti rôznych časticových, soft body a iných simulácií.

K modelovaniu tekutín existuje v Blenderi niekoľko alternatív. Jednou z nich je použitie vstavaného simulátoru založeného na Lattice-Boltzmann metóde. Tá predstavuje pomerne novú techniku, ktorá namiesto riešenia Navier-Stokes rovníc počíta diskrétnu Boltzmannovu rovnicu. Nepodarilo sa presne zistiť, či tento vstavaný simulátor podporuje aj GPU akceleráciu. Bolo však vykonaných niekoľko drobných experimentov a vo všetkých prípadoch bol vyťažovaný len procesor.

Druhou možnosťou je spočítať správanie tekutiny pomocou niektorého z externých nástrojov tretích strán a Blender použiť len k definovaniu vstupnej scény a k finálnej vizualizácii. Takýmto nástrojom môže byť napríklad voľne dostupný simulačný balík DualSPHysics, ktorý umožňuje modelovať tekutinu algoritmom SPH, pričom dokáže plne využiť akceleráciu grafickým hardvérom, alebo nástroj FumeFX schopný generovať veľmi pekné a realisticky vyzerajúce efekty dymu a výbuchov.

Práca so simuláciou potom v Blenderi prebieha v dvoch krokoch. V prvom sú najprv jednotlivé snímky uložené do cache súboru na disk, z ktorého sú následne v druhom kroku načítané a vizualizované pomocou konvenčného rendereru akým je už v úvode spomínaný Cycles.

Houdini

Houdini je ďalším zo skupiny profesionálnych programov určených na tvorbu modelov, ich renderovanie, kompozitovanie a animovanie, či tvorbu špeciálnych efektov prostredníctvom rôznych rigid body, soft body a časticových simulácií. Oproti konkurencii sa však odlišuje svojim procedurálnym prístupom k riešeniu týchto úloh, čo prakticky znamená, že väčšina činností je v ňom vyjadrená grafom vzájomne prepojených uzlov. To má potom veľké praktické výhody najmä v modulárnosti a znovu použiteľnosti už raz vytvoreného obsahu. Je vyvíjaný torontskou spoločnosťou Side Effects Software a obľubu si získal najmä v oblasti filmovej produkcie, kde bol okrem iných použitý napríklad aj pri tvorbe animovaných Disney rozprávok Chicken little a Frozen.

Program vyžaduje 64-bitový operačný systém, inak je ale plne multiplatformný a podporuje jak operačné systémy Windows a MAC OS X, tak i niekoľko najpopulárnejších linuxových distribúcií včítane Ubuntu, Fedora a Red Hat Enterprise. Okrem toho je program pre niektoré činnosti schopný čiastočne využiť aj GPU akceleráciu prostredníctvom OpenCL a paralelné spracovanie na viacerých nezávislých počítačoch zároveň.

Simulátor tekutín integrovaný v Houdini je založený na metóde FLIP, čo je skratka z anglického názvu Fluid-implicit particle. Tá predstavuje hybridný algoritmus schopný vďaka kombinácii mriežkových a časticových prístupov presne a v pomerne krátkom čase simulovať i rýchlo sa pohybujúce, či veľké masy kvapaliny. Okrem toho Houdini medzi svoje možnosti zahŕňa aj špecializované simulátori plynov a oceánu nazvané Pyro FX, resp. Ocean

FX spolu s množstvom predpripravených materiálov pomocou, ktorých je možné vytvoriť presvedčivé autenticky vyzerajúce simulácie dymu, ohňa, mora, alebo vĺn.

Pre nekomerčné vzdelávacie účely je k dispozícii zdarma obmedzená verzia Houdini Apprentice, pre profesionálne použitie sa však už jedná o platený a pomerne drahý softvér.

Ostané

Okrem vyššie spomenutých riešení existuje ešte ohromné množstvo ďalších viac alebo menej známych programov, ktoré ich užívateľom umožňujú simulovať tekutiny.

Za všetky stačí spomenúť napríklad 3D programy Maya a 3ds Max od spoločnosti Autodesk, Cinema 4D od nemeckej spoločnosti MAXON Computer GmbH, alebo už zmienený nástroj FumeFX určený k detailnej simulácii rôznych plynov vrátane dymu a ohňa. Všetky tieto programy sú ale podobne ako vyššie spomenuté aplikácie Blender, Houdini, alebo RealFlow určené predovšetkým k ultra realistickým offline výpočtom používaným vo filmovej produkcii.

Spomedzi programov, ktoré sú založené na algoritme Smoothed Particle Hydrodynamics je možné spomenúť napríklad projekt GPUSPH. Ten na akceleráciu výpočtov využíva platformu CUDA a bol prvým, ktorému sa SPH algoritmus podarilo implementovať tak, aby bežal kompletne celý na GPU. Ďalšími alternatívami sú aj open source framework pysph napísaný v jazyku Python, alebo program Fluidx, ktorý rovnako ako GPUSPH využíva CUDA akceleráciu.

Kapitola 3

GPGPU

Pre reálne real-time nasadenie simulácie kvapalín v hrách, grafických editoroch alebo iných interaktívnych programoch nemusí stačiť výkon ponúkaný dnešnými procesormi a treba využiť schopnosti masívneho paralelného spracovania v súčasných moderných grafických kartách. V minulosti boli grafické procesory určené výlučne k akcelerácii vykresľovania, avšak s postupným pridávaním funkcionality a príchodom programovateľných častí do vykresľovacieho reťazca sa postupne začala objavovať myšlienka využitia týchto procesorov aj k negrafickým účelom. Skutočný rozvoj `GPGPU` programovania, teda programovania všeobecných výpočtov na grafických kartách, však prišiel až so vznikom unifikovanej shader architektúry a príchodom aplikačného rozhrania `CUDA` v roku 2006 [5].

Táto kapitola preto prináša prehľad stále rastúceho počtu rôznych vysoko aj nízko úrovňových rozhraní pre programovanie na grafických kartách, predstavuje pohľad na typickú logickú architektúru grafického procesora a jeho dôležité aspekty, na ktoré treba pri programovaní týchto špecifických výpočtových strojov myslieť. V záverečných odsekoch sa potom táto kapitola zameriava na bližšie predstavenie štandardu `OpenCL`, ktorý bol v tejto práci využitý.

3.1 API pre paralelné programovanie na GPU

V prvopočiatkoch `GPGPU` bolo jedinou možnosťou ako využiť ohromný potenciál grafických akcelerátorov na všeobecné výpočty ich namapovanie na grafické primitíva, s ktorými vedel pracovať vykresľovací reťazec v `OpenGL`, alebo v `DirectX`. V dnešnej dobe však už jestvuje veľké množstvo najrôznejších rozhraní od tých úplne nízkoúrovňových v podobe `OpenCL` a `CUDA` až po tie vysokoúrovňové v podobe pragma príkazov štandardu `OpenACC`. Táto podkapitola preto prináša prehľad dostupných technológií a sumarizuje ich výhody a nevýhody.

CUDA

Priekopníckym rozhraním, ktoré odštartovalo dnešnú masovú popularitu akcelerovania výpočtov na grafických procesoroch je rozhranie `CUDA` [6] od spoločnosti `NVidia`. Jedná sa o sériu rozšírení jazykov `C` a `C++` o konštrukcie uľahčujúce paralelné programovanie. Konkrétne, kód určený pre vykonanie na GPU je písaný do oddelených funkcií označených kľúčovým slovom `__global__` a nazývaných kernelom. Taktiež na spúšťanie týchto funkcií zavádza `CUDA` špeciálnu syntax pomocou, ktorej je možné určiť parametre paralelného vykonania ako sú počet simultánnych vlákien, ich rozdelenie do skupín, prípadne množstvo zdieľanej pamäte, ktoré môžu využiť. Toto rozhranie, označované aj ako `Runtime API`, je

veľmi pohodlné na použitie a vyžaduje od programátora len minimálnu námahu. Na druhej strane je však programátor závislý od prekladača podporovaného kompilátorom `nvcc`, ktorý predspracováva programy napísané pomocou týchto rozšírení. Pokiaľ sa chce programátor vyhnúť používaniu neštandardných konštrukcií vo svojich programoch, tak môže využiť tzv. `Driver API`, ktoré mu sprístupní nízkoúrovňové rozhranie `CUDA` v štandardnom jazyku `C/C++`. Nevýhodou tohto prístupu je ale nutnosť písania väčšieho množstva kódu na dosiahnutie rovnakého výsledku, nutnosť ručnej inicializácie `CUDA` a nutnosť ručného nahrania predkompilovaných `CUDA` programov na grafickú kartu. Okrem samotného rozhrania pre prístup ku grafickému hardvéru a kompilátoru pre preklad kernelov poskytuje `CUDA` aj množstvo ďalších podporných nástrojov v podobe profilerov `NVprof` a `NVidia Visual Profiler`, debugeru `CUDA-GDB`, pamäťového analyzátoru `CUDA-MEMCHECK`, alebo integrovaného vývojového prostredia `NVidia Nsight`.Samozrejmosťou je aj veľmi obsiahla dokumentácia dodávaná priamo pri inštalácii `CUDA SDK` alebo dostupná z internetu a množstvo veľmi dobre spracovaných ukážkových príkladov. Ďalšie uľahčenie programovania umožňuje niekoľko vysoko optimalizovaných knižníc zameraných na časté matematické operácie. Za všetky stačí spomenúť knižnicu `cuFFT` implementujúcu rýchlu fourierovu transformáciu, knižnicu `cuRand` zaoberajúcu sa generovaním náhodných čísel, alebo knižnicu `cuBLAS` implementujúcu množstvo funkcií pre lineárnu algebru. Iné jazyky ako `C/C++` a `Fortran` nie sú `NVidia`-ou oficiálne podporované, ale existuje množstvo implementácií, tzv. bindings, ktoré umožňujú využívať `CUDA`-u aj z iných jazykov, napríklad z `Python`-onu, `Java`-y, `Matlab`-u, `Ruby`, atď.

OpenACC

Najväčšou nevýhodou `CUDA` je to, že funguje len na grafických procesoroch od firmy `NVidia`, a preto v posledných rokoch dochádzalo k viacerým snahám o vytvorenie jednotného otvoreného multiplatformného riešenia, ktoré by bolo schopné využiť aj akcelerátory iných spoločností, prípadne uľahčiť heterogénne programovanie. Jednou z týchto snáh je štandard `OpenACC` [3] spoločne vyvinutý firmami `Cray`, `NVidia`, `CAPS` a `PGI`. `OpenACC` je postavený na rovnakej filozofii pragma príkazov ako štandard `OpenMP`. Tie mu umožňujú označiť miesta v kóde `C/C++` aplikácie, ktoré môžu byť paralelizované a na základe takto anotovaného zdrojového kódu potom kompilátor sám rozhodne ako označený `C++` kód namapuje na dostupný akcelerátor. Pôvodný kód algoritmu v `C++` jazyku sa tak zmení len minimálne pri zachovaní dostatočnej kompatibility naprieč rôznymi kompilátormi a behovými prostrediami. V prípade, že používaný kompilátor uvedenú `OpenACC` direktívu nepozná, tak ju bude jednoducho ignorovať a vytvorí klasický sekvenčný program. Tento systém veľmi uľahčuje aj údržbu kódu aplikácie, pretože nie je treba osobitne aktualizovať verziu napísanú v čistom `C++` a verziu napísanú pomocou `CUDA`, alebo iného paralelného API. Taktiež pokiaľ si vývojár nie je istý, či je jeho algoritmus vhodný pre urýchlenie pomocou GPU, alebo nie, tak môže využiť `OpenACC` pragma direktívy k experimentovaniu a rýchlemu prototypovaniu. Problémom s týmto štandardom je v súčasnosti len jeho slabá podpora v populárnych mainstreamových prekladačoch ako je `GCC`, `Intel` kompilátor, alebo kompilátor dodávaný spolu s `Microsoft Visual Studio`-m. Jediné prekladače, ktoré v súčasnosti tento štandard podporujú sú komerčné kompilátory od `PGI`, `CAPS/HMPP` a `Cray`. Podpora v `GCC` sa pripravuje, ale ešte nie je hotová.

OpenGL Compute Shaders a DirectCompute

Pre grafické aplikácie, ktoré už GPU využívajú pomocou `OpenGL`, alebo `DirectX` môže byť výhodnejšie namiesto pridávania nových závislostí rozšírenie stávajúcich rozhraní o novú funkcionalitu. Z tohto dôvodu boli do štandardu `OpenGL` pridané `Compute Shader`-i [1] a do `DirectX` bol pridaný `DirectCompute` [4], čo sú v podstate tiež compute shader-i využívajúce `HLSL`. `Compute shader`-i v `OpenGL` fungujú na veľmi podobných princípoch ako `OpenCL` alebo `CUDA`, ale s niekoľkými obmedzeniami. Jedným z obmedzení je napríklad to, že počet vlákien v pracovnej skupine, čo zodpovedá jednému thread block-u v `CUDA`, alebo jednej work group v `OpenCL`, musí byť v čase kompilácie shaderu konštanta. Toto sa dá obísť jedine použitím rozšírenia `ARB_compute_variable_group_size`, alebo využitím preprocesoru pri kompilácii shaderu. Na druhej strane oproti `OpenCL C`, alebo `CUDA C` je však možné využiť bohatšie možnosti jazyka `GLSL`, ktorý sa na programovnie compute shader-ov používa. Ďalšou výhodou použitia `Compute Shader`-ov z `OpenGL` namiesto možností `OpenGL` interoperability ponúkaných v `CUDA` a v `OpenCL` je absencia potreby synchronizácie medzi `OpenGL` a `OpenCL`, resp. `OpenGL` a `CUDA`, čo môže priniesť mierne zlepšenie výkonu.

Podobne ako `OpenGL Compute Shader`-i aj `DirectCompute` cieli na podporu programovania všeobecných výpočtov na GPU v rámci grafického rozhrania `Direct3D`. `Direct Compute` bol pôvodne určený pre `DirectX 11`, ale neskôr bol backportovaný aj do `DirectX 10`. Podobne ako compute shader-i v `OpenGL` nepodporuje niektoré pokročilé funkcie známe z `CUDA 5.5`, alebo z `OpenCL 2.0` ako je napríklad dynamický paralelizmus.

C++ AMP

Z pohľadu jednoduchej použiteľnosti je však oveľa zaujímavejšia technológia `C++ AMP` (`C++` Accelerated Massive Parallelism) [7], ktorá je nad `DirectX` postavená. Jedná sa o otvorený štandard vyvinutý firmou `Microsoft`, ktorý pridáva len minimum nových rozšírení do jazyka `C++` v podobe kľúčových slov `restrict(amp)` a `tile_static`, inak ide hlavne o knižnicu kompatibilnú so štandardnou šablónovou knižnicou `C++`. Kľúčové slovo `restrict(amp)` sa uvádza za signatúrou funkcie a prikazuje prekladaču, aby skontroloval, či daná funkcia náhodou nevyužíva konštrukcie, ktoré nie sú na GPU podporované (napríklad výnimky, alebo RTTI). Druhé kľúčové slovo `tile_static` umožňuje `C++` programu pracovať so zdieľanou pamäťou. O vygenerovanie `DirectX` príkazov a kódu spustiteľného na GPU sa už stará prekladač jazyka `C++`. Použitie `DirectX` je však pre programátora `C++ AMP` aplikácií len implementačný detail a s týmto faktom nie je nijako inak konfrontovaný. To teoreticky umožňuje veľmi jednoduchú prenositeľnosť programov na iné platformy a nezávislosť od konkrétneho hardvéru. V skutočnosti ale zatiaľ okrem implementácie v `Microsoft Visual Studio`, ktorá navyše vyžaduje minimálne `Windows 7` alebo `Windows Server 2008 R2`, neexistuje žiadna ďalšia poriadna implementácia. Výhodou oproti väčšine ostatných doteraz spomenutých riešení je softvérová emulácia, ktorá sa použije v prípade, že na cieľovom počítači nie je dostupný žiadny vhodný hardvérový akcelerátor. To zabezpečuje, že program bude vždy schopný fungovať. Nevýhodou tejto automatickej emulácie môže byť nečakane veľké spomalenie kritického kódu, pretože softvérový emulátor je primárne určený k ladeniu programu počas vývoja. Tento problém je čiastočne vyriešený na `Windows 8`, kde existujú dva softvérové emulátory. Prvý z nich je stále pomalý ladiaci emulátor, ale druhý už využíva `WARP` (Windows Advanced Rasterization Platform), čo je veľmi výkonný softvérový rasterizér optimalizovaný pomocou `SSE` inštrukcií a viacvláknového paralelného spracovania.

Ostatné

Z ďalších API pre programovanie na GPU je možné spomenúť ešte rozhranie `Metal`, ktoré bolo predstavené spoločnosťou `Apple` a jedná sa o nízko úrovňové rozhranie dostupné na `iOS` od verzie 8, alebo štandard `OpenMP`, ktorý je v súčasnosti používaný na paralelizáciu kódu pomocou vlákien procesoru, ale do najnovšej verzie plánuje pridať podporu aj pre GPU akcelerátory.

3.2 Hardvérový pohľad na GPU

Pre jednoduchšie pochopenie programovacích modelov zavedených v jednotlivých API (najmä v `CUDA` a v `OpenCL`), a to ako sa mapujú na hardvérové prostriedky daného grafického čipu je dobré poznať logický pohľad na jeho architektúru. Táto podkapitola preto prináša stručný prehľad najdôležitejších konceptov využívaných v hardvéri dnešných GPU, pričom niektoré údaje môžu byť špecifické pre procesory od firmy `NVidia`. Základné princípy by však mali byť rovnaké aj v prípade procesorov od firmy `AMD`.

Logické členenie GPU

Ako už bolo aj v úvode k tejto kapitole naznačované v architektúrach pôvodných grafických procesorov boli jednotky spracúvajúce vertex programy a jednotky spracúvajúce pixel programy oddelené kusy hardvéru. Toto delenie malo nevýhodu v tom, že niektoré aplikácie využívajú veľké množstvo detailnej geometrie s veľmi malými trojuholníkmi, zatiaľ čo iné aplikácie využívajú len malé množstvo geometrie a kreslia veľmi veľké polygóny. Pri fixnom rozdelení hardvéru je potom veľmi náročné navrhnúť čip, ktorý by bol v oboch prípadoch efektívne využitý a preto vznikla unifikovaná shader architekúra, ktorá je dnes už základom prakticky každého moderného GPU od každého výrobcu.

Z pohľadu programátora sú teda výpočtové jednotky na GPU združené do blokov, ktoré sa nazývajú Streaming Multiprocesor (na obrázku 3.1 označené skratkou SMM) a tie sú ďalej zložené zo skalárnych streaming procesorov, ktoré navzájom bežia paralelne. Samotné streaming procesory však už spracovávajú jednotlivé inštrukcie násobenia a sčítania sekvenčne. Čítanie a zápis do pamäte majú na starosti LDST (Load/Store) jednotky a delenie, sínus, kosínus, odmocninu, a podobné zložité funkcie majú na starosť špeciálne jednotky nazývané SFU (Special Function Unit). LDST a SFU jednotiek je však oproti streaming procesorom výrazne menej a pri programovaní, tak treba s ohľadom na čo najlepší výkon dbať na to, aby počet týchto operácií bol v správnom pomere k násobeniu a sčítaniu. Okrem toho je ešte súčasťou Streaming multiprocesoru súbor registrov, ktorého veľkosť je rôzna v závislosti od konkrétnej architektúry, rýchla zdieľaná pamäť pomocou, ktorej môžu jednotlivé vlákna medzi sebou komunikovať, textúrovacie jednotky, textúrovacia a L1 cache, inštrukčný bufer a plánovač vlákien.

Plánovanie vlákien

Vlákna sú plánované po skupinkách nazývaných v terminológii `NVidie` warp a v terminológii `AMD` wavefront, pričom všetky vlákna v rámci jedného warp-u/wavefront-u vykonávajú v tom istom okamihu tú istú inštrukciu, ale nad rôznymi dátami. Vlákna z rozdielnych warpov už ale môžu vykonávať iné inštrukcie. Problém nastáva pri vetvení, kde môže dôjsť k divergencii, čo je situácia, keď rôzne vlákna v rámci toho istého warpu-u/waverfont-u postupujú inými vetvami. V hardvéri sa táto situácia rieši tak, že každému vláknu sa nastaví

Obrázek 3.1: Architektúra Maxwell v high-end grafickej karte GeForce GTX 980. Obrázok je prebraný z [9]

exekučná maska a potom všetky vykonajú najprv if vetvu, invertuje sa maska a následne všetky vlákna vykonajú aj else vetvu, ale výsledky sa použijú len z tých vlákien, ktoré sú exekučnou maskou povolené. V konečnom dôsledku sa tak warp vykoná akoby dvakrát.

Na to aby hardvér dokázal efektívne vyťažiť veľké množstvo výpočtových jednotiek, ktoré má k dispozícii potrebuje aj veľké množstvo aktívnych vlákien, medzi ktorými môže prepínať, aby tak bol schopný vykryť latenciu prístupov do pamäte. S týmto súvisí ďalší veľmi dôležitý pojem v kontexte grafických procesorov a to je occupancy. Tá vyjadruje pomer medzi počtom aktívnych warp-ov/wavefront-ov a maximálnym počtom warp-ov/wavefront-ov, ktoré môžu bežať na jednom streaming multiprocesore. Snahou by malo byť spustiť čo najviac vlákien na každom multiprocesore.To však závisí na množstve spotrebovaných zdrojov na jedno vlákno, najmä na počte spotrebovaných registrov a počte využitých bytov zdieľanej pamäte.

Pamäťová hierarchia

Ďalším veľmi dôležitým aspektom programovania na grafických kartách je ich pamäťová štruktúra. Moderné grafické procesory majú niekoľko typov pamätí využiteľných pre pro-

gramátora. Prvou a najväčšou z nich je globálna pamäť, ktorá sa nachádza mimo čipu. Jedná sa väčšinou o pamäť typu GDDR, ktorá má obvykle veľmi vysokú priepustnosť, ale aj veľmi vysokú latenciu.

Globálna pamäť sa číta a zapisuje po blokoch o veľkosti najčastejšie 32, 64, alebo 128 bytov, ktoré sa nazývajú transakcie. Jednotlivé transakcie musia byť zarovnané, čo znamená, že počiatočná adresa každého takéhoto bloku musí byť násobkom 32, 64, alebo 128 bytov, prípadne iného čísla, pokiaľ konkrétna architektúra používa transakcie inej veľkosti. Hardvér potom združuje prístupy do globálnej pamäte z jednotlivých vlákien warp-u/wafront-y do transakcií, čiže namiesto toho, aby sekvenčne pre každé vlákno vydal požiadavku na načítanie jedného prvku z globálnej pamäte, tak vydá požiadavku na načítanie niekoľkých za sebou idúcich prvkov. Tomuto združovaniu sa v terminológii `NVidie` hovorí coalescing.

Úspešnosť coalescing-u závisí na veľkosti prvku (počte jeho bytov) a poradí v akom jednotlivé vlákna prvky čítajú. Ak po sebe idúce vlákna čítajú po sebe idúce prvky, tak dochádza k najlepšiemu coalescing-u a zároveň aj k najlepšej priepustnosti, v opačnom prípade môže byť čítanie rozbité do niekoľkých transakcií a priepustnosť klesá. Presné pravidlá, kedy ešte dochádza ku coalescing-u a kedy už nie, však závisia od konkrétneho GPU.

Napríklad pri starších kartách od firmy `NVidia` boli požiadavky na coalescing veľmi striktné a pokiaľ vlákna nepristupovali do pamäte sekvenčne na zarovnané adresy a s jednotkovým rozostupom, tak sa prvok z každého vlákna preniesol v samostatnej transakcii a priepustnosť okamžite dramaticky klesla. Novšie karty od `NVidie` už požiadavky na coalescing zmiernili a zaviedli cache, ktoré pomáhajú združovať do transakcií aj náhodne permutované a do určitej miery nezarovnané prístupy, avšak stále platí, že najlepšiu priepustnosť je možné dosiahnuť len dodržiavaním spomenutých pravidiel.

Ďalším veľmi užitočným typom pamäte je zdieľaná pamäť, ktorá by sa dala prirovnať k rýchlej vyrovnávacej pamäti cache používanej na procesoroch, ale s tým rozdielom, že zdieľaná pamäť je spravovaná ručne programátorom. Fyzicky je táto pamäť rozdelená na niekoľko častí, ktoré sa nazývajú banky.

Banky sú usporiadané tak, aby sa po sebe nasledujúce adresy mapovali na po sebe nasledujúce banky. Celkový počet a šírka bank-ov závisí od konkrétneho výrobcu a od konkrétneho modelu karty, ale napríklad na novších GPU od firmy `NVidia` je 32 bankov širokých 32-bitov. V tomto prípade sa potom adresa číslo 0 mapuje na bank číslo 0, adresa číslo 1 na bank číslo 1, a tak podobne až po adresu číslo 31, ktorá sa mapuje na bank číslo 31. Ďalej sa už číslovanie bankov periodicky opakuje, teda adresa číslo 32 sa mapuje na bank číslo 0, adresa číslo 33 sa mapuje na bank číslo 1, atď. Toto usporiadanie je znázornené aj na obrázku 3.2.

Obrázek 3.2: Mapovanie adries na banky. Obrázok je prebraný z [11]

Dôležité je, že do bankov môžu vlákna pristupovať paralelne. Ak však viaceré vlákna z warpu pristupujú do toho istého banku na rôzne adresy, tak dochádza k serializácii a teda k podstatnému zníženiu priepustnosti. Túto situáciu označujeme ako bank konflikt. V prípade, že do jedného banku pristupuje n vlákien hovoríme o n-cestnom konflikte. Naopak za predpokladu, že vlákna pristupujú v rámci jedného banku na tú istú adresu, umožňujú niektoré GPU broadcastovať hodnotu z tejto adresy všetkým zainteresovaným vláknam a

vyhnúť sa tak konfliktu. Z praktického pohľadu k najväčším bank konfliktom dochádza napríklad pri prechádzaní matice po stĺpcoch.

Okrem už spomenutých pamäťových oblastí je na grafickom procesore aj špecializovaná pamäť konštánt a textúrovacia pamäť.

Pamäť konštánt je väčšinou malá oblasť vyhradená v globálnej pamäti, ktorá je však na čipe veľmi efektívne cache-ovaná a umožňuje tak vláknam rýchly prístup ku globálnym premenným, ktoré sa počas výpočtu nemenia.

Textúrovacia pamäť je v skutočnosti, podobne ako konštantná pamäť, súčasťou hlavnej pamäte, ale o čítanie z nej sa starajú textúrovacie jednotky. Tie umožňujú jej vysoko efektívne adresovanie vrátane kontroly rozsahu zadaných indexov implementované priamo v hardvéri, lineárne, bilineárne, alebo trilineárne filtrovanie a konverziu texelov už priamo pri čítaní. Odhliadnuc od toho je pamäť adresovaná týmito jednotkami aj cache-ovaná v špeciálnej priestorovej cache, ktorá okrem požadovaného prvku cache-uje i jeho okolie.

3.3 OpenCL

V tejto podkapitole budú postupne predstavené základné princípy `OpenCL` a jeho vlastnosti, ktoré budú ďalej rozvinuté v nadväzujúcich pasážach. Kompletný výklad je však ponechaný na oficiálnu dokumentáciu [20], prípadne odbornú literatúru [22].

Základné vlastnosti

`OpenCL`, čiže Open Computing Language, je otvorený priemyslový štandard na podporu paralelného programovania v heterogénnom prostredí. Jeho vznik a ďalší vývoj má na starosti konzorcium Khronos, ktoré združuje popredných svetových výrobcov hardvéru a softvéru ako je `Apple`, `AMD`, `Intel`, `NVidia`, `IBM` a mnohí iní. Toto isté konzorcium má na svedomí aj ďalšie populárne štandardy, napríklad `OpenGL`, `WebGL`, alebo `COLLADA`.

Samotné `OpenCL`, tak nepredstavuje žiadnu konkrétnu knižnicu funkcií, ale skôr sa jedná o definíciu spoločných vlastností a súbor požiadaviek na konformnú implementáciu. Vlastné skompilované knižnice sú potom poskytované vývojárom ako súčasť rôznych SDK (Software Developement Kit) výrobcov.

Typický `OpenCL` program je rozdelený na dve časti, a to na hostiteľskú časť a časť, ktorá beží na vybranom zariadení. Takýmto zariadením je väčšinou GPU, ale môže sa jednať aj o CPU, či iné špecializované procesory. Vďaka dômyselnému návrhu samotného štandardu nie je dokonca vylúčené ani simultánne spracovanie výpočtov na obidvoch typoch čipov naraz, hoci zvyčajne v roli hostiteľa pôsobí CPU a v roli výpočtového zariadenia GPU.

Úlohou hostiteľa je riadenie výpočtov, alokácia pamäte, správa získaných zdrojov a všetka nevyhnutná inicializácia. V prevažnej väčšine prípadov je táto časť programu napísaná v `C/C++`, pre ktoré je `OpenCL` primárne určené. Z ostatných jazykov, býva rozhranie štandardu dostupné prostredníctvom tzv. bindings, čiže tenkej medzivrstvy, ktorá prekladá volania príslušných funkcií na ich ekvivalenty v konkrétnom jazyku. Bindings však už nie sú súčasťou `OpenCL` štandardu spravovaného združením `Khronos`.

Výpočtové zariadenia sa tak starajú len o čo najrýchlejšie spracovanie zadaných úloh, ktoré sú programované v jazyku `OpenCL C`. Ten je odvodený z jazyka C, konkrétnejšie z jeho normy `ISO C99` a rozširuje ho o dodatočné konštrukty vhodné najmä pre paralelné algoritmy. Napríklad `OpenCL C` zavádza nové vektorové dátové typy, ktoré umožňujú prirodzeným spôsobom využívať SIMD inštrukcie spracúvajúce niekoľko čísel naraz. Vo svete x86 procesorov sú tieto inštrukcie dobre známe pod skratkami `SSE` a `AVX`. Ďalším nemenej

významným vylepšením je aj rozsiahla štandardná knižnica matematických funkcií, ktorá výrazne uľahčuje programovanie.

Avšak ešte predtým ako môže zariadenie začať vykonávať `OpenCL` kód je nutné ho skompilovať. Toto sa na rozdiel od klasických jazykov typu `C` a `C++` deje za behu aplikácie. Dôvodom tohto rozhodnutia je najmä platformová nezávislosť, keďže inštrukčné sady GPU procesorov rôznych dodávateľov sú medzi sebou výrazne menej kompatibilné ako v prípade klasických procesorov. Navyše ani miera spätnej kompatibility v rámci modelových radov toho istého výrobcu nemusí byť úplne stopercentná. Ďalším dobrým dôvodom je fakt, že kompilátor na cieľovom stroji môže uskutočniť efektívnejšie optimalizácie, prípadne programátor má šancu prispôsobiť svoju implementáciu pre konkrétnu platformu (využitím preprocesoru, alebo výberom vhodnej verzie kódu tesne pred prekladom).

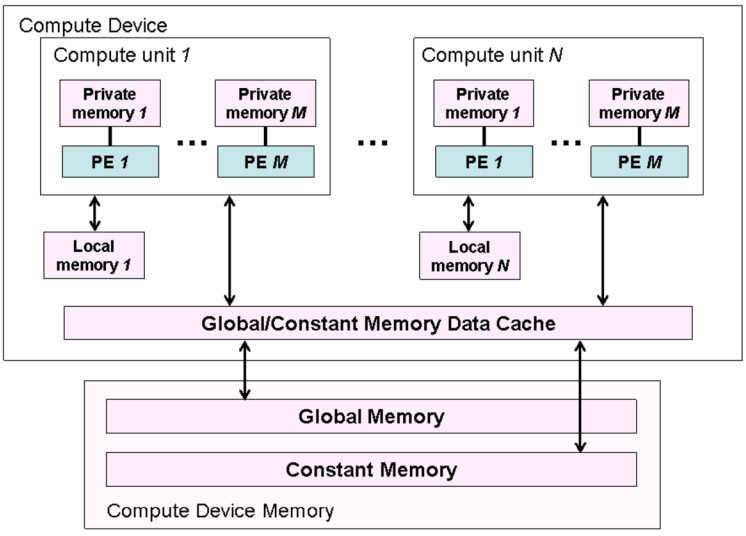
Pretože `OpenCL` štandard sa snaží obsiahnuť čo najširšie spektrum zariadení, tak definuje dva stupne splnenia jeho požiadaviek a to `Full Profile` a `Embedded profile`. Prvý je určený pre veľké stanice disponujúce množstvom výkonu, zatiaľ čo druhý je určený skôr pre malé vstavané prístroje a mobilné telefóny.

Hlavnými métami `OpenCL` je teda ponúknuť programový model pre dátový a úlohový paralelizmus, poskytnúť abstrakciu podporného hardvéru a vytvoriť jednotné API využívajúce rovnaké princípy ako už zavedený štandard `OpenGL`.

Abstrakcia hardvéru

OpenCL vo svojej špecifikácii definuje niekoľko rôznych modelov a abstrakcií uľahčujúcich konečnú platformovú nezávislosť.

Jednou z týchto abstrakcií je aj `OpenCL` zariadenie definované ako množina výpočtových jednotiek (compute units) ďalej rozdelených na procesné elementy (processing elements), v ktorých prebieha samotný výpočet. Procesné elementy sú zas definované ako SIMD (Single Instruction Multiple Data) alebo SPMD (Single Program Multiple Data) jednotky schopné sekvenčne spracovať vstupný prúd inštrukcií. Ich vzájomný beh však už ide plne paralelne.



Obrázek 3.3: Pamäťová hierarchia v `OpenCL`. Obrázok je prebraný z [20]

Druhým významným modelom zavedeným štandardom `OpenCL` je abstrakcia hierarchie pamätí ukázaná na obrázku 3.3. Podľa nej existujú štyri druhy pamätí, a to globálna, konštantná, lokálna a privátna. Najväčšou, ale zároveň aj najpomalšou z nich je globálna pamäť. Tá je prístupná na čítanie i zápis všetkým výpočtovým jednotkám a procesným elementom po celú dobu ich výpočtu. Zapisovať a čítať túto pamäť môže aj hostiteľ, ktorý prostredníctvom nej predáva zariadeniu vstupné dáta a získava výsledky.

Druhým typom pamäte je konštantná pamäť. Tá je väčšinou implementovaná ako súčasť globálnej pamäte, do ktorej zariadenie nemá právo zápisu, a preto býva veľmi často cacheovaná za účelom zvýšenia rýchlosti prístupu.

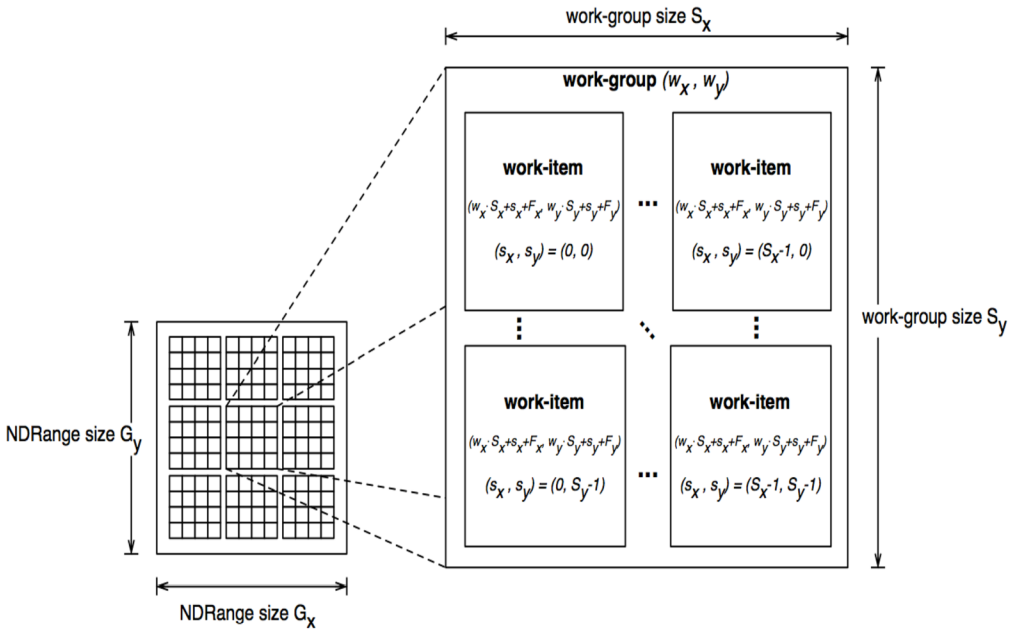
Lokálna pamäť býva typicky umiestňovaná priamo na zariadení a preto je prístup k nej rýchlejší ako v predošlých dvoch prípadoch. Využíva sa hlavne na zdieľanie dát medzi pracovnými jednotkami (work-items) v rámci pracovnej skupiny (work-group). Hostiteľ k nej nemá žiadny prístup a môže ju pre zariadenia len alokovať.

Posledným typom je privátna pamäť, ktorá je najrýchlejšia, ale zároveň aj najmenšia zo všetkých spomenutých. Využitie nachádza medzi pracovnými jednotkami, ktoré do nej ukladajú svoje lokálne premenné a medzivýpočty.

Exekučný model

Okrem už spomenutých konceptov definuje `OpenCL` štandard aj spôsob, akým má byť spracovanie paralelnej úlohy v zariadení rozdelené.

Toto rozdelenie je vyjadrené indexným priestorom nazývaným `NDRange` a znázorneným na obrázku 3.4. Jeho fungovanie sa dá prirovnať k sérii zanorených cyklov používaných v mnohých klasických programovacích jazykoch. Hĺbka zanorenia potom predstavuje počet dimenzií indexného priestoru a každá iterácia jednu pracovnú jednotku pomenovanú v terminológii `OpenCL` ako work-item. Tieto sú ďalej združované do pracovných skupín umožňujúcich ich synchronizáciu a zdieľanie medzivýsledkov.

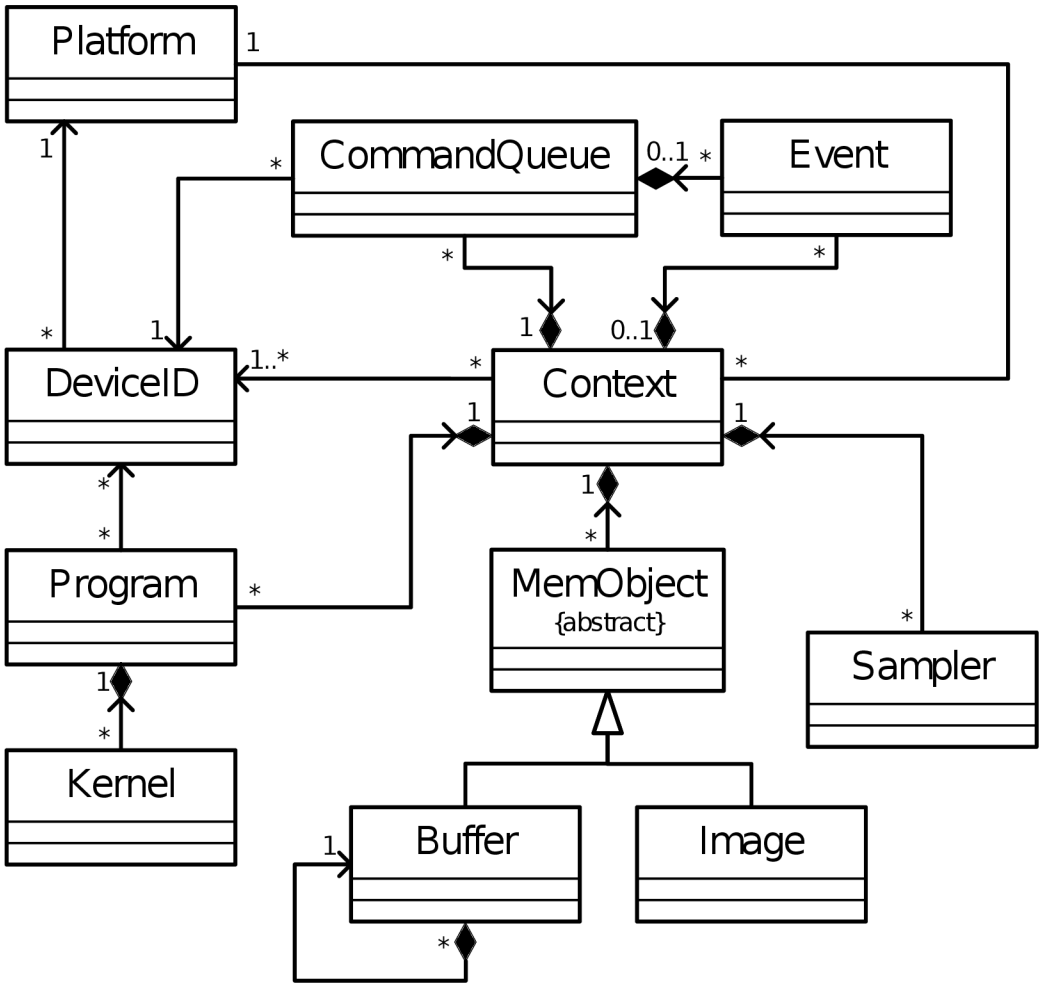


Obrázek 3.4: Indexný priestor `NDRange`. Obrázok je prebraný z [20]

Aby boli jednotlivé pracovné jednotky od seba navzájom jednoznačne odlíšiteľné definuje `OpenCL` niekoľko typov identifikátorov, ktorých návrh vychádza z prirovnania k zanoreným cyklom. Každý z nich je totiž n-ticou čísel odpovedajúcou aktuálnemu počtu dimenzií. Ak by teda programátor `NDRange` stanovil ako dvoj-dimenzionálny, bude každý identifikátor dvojicou indexov, ak by ho definoval ako trojdimenzionálny bude trojicou, atď. Identifikátory sú dvojakého druhu a to lokálne, ktoré odlišujú pracovnú jednotku len v rámci skupiny a globálne, ktoré ju odlišujú od všetkých ostatných definovaných pracovných jednotiek. Taktiež každá skupina má svoj vlastný skupinový identifikátor.

Hlavné prvky API

Aj napriek tomu, že špecifikácia `OpenCL` je určená pre jazyk `C`, ktorý je procedurálnym jazykom a sám o sebe neobsahuje priamu podporu objektovo orientovaného programovania je hostiteľské aplikačné rozhranie navrhnuté v duchu týchto princípov. Dokazuje to aj nižšie uvedený `UML` diagram tried znázorňujúci jednotlivé štruktúry `OpenCL` a ich vzájomné prepojenie.



Obrázek 3.5: `UML` diagram `OpenCL`. Obrázok je prebraný z [20]

Z obrázku vidno, že centrálnym objektom je kontext. Ten združuje všetky ostatné objekty a umožňuje medzi nimi komunikáciu a synchronizáciu. Súčasťou kontextu sú aj výpoč-

tové zariadenia, avšak nie je podmienkou, aby v ňom bolo zahrnuté úplne každé dostupné zariadenie. Navyše v rámci jedného kontextu sa môžu nachádzať len zariadenia z tej istej platformy, takže spravovať naraz grafickú kartu od `NVidie` a procesor od `Intel`-u je vylúčené. Riešením tohto problému môže byť vytvorenie osobitného kontextu pre každú platformu zvlášť. V kóde hostiteľa reprezentuje kontext štruktúra `cl_context`.

Ďalšími dvoma významnými abstrakciami sú platforma a zariadenie. Prvá menovaná identifikuje konkrétnu implementáciu `OpenCL` štandardu od konkrétneho dodávateľa. Jej úlohou je sprostredkovať informácie o podporovanej verzii, profile, rozšíreniach a dostupných zariadeniach. Druhá abstrakcia predstavuje skutočné GPU, alebo viacjadrový procesor prítomný na počítači. Pomocou nej sa v súčinnosti s platným identifikátorom platformy, alebo platným kontextom dajú získať informácie o zastupovanom zariadení a jeho schopnostiach. Platformu v kóde reprezentuje štruktúra `cl_platform` a zariadenie štruktúra `cl_device`.

Keďže sa `OpenCL` skladá z dvoch častí, kde jedna je napísaná v klasickom `C/C++` a druhá v `OpenCL C`, tak existuje objekt program, ktorý reprezentuje zdrojový kód tohto jazyka. Ten potom umožňuje spustiť kompiláciu, zistiť informácie o jej priebehu a získať výsledné preložené binárne súbory. Od úspechu kompilácie sa ďalej priamo odvíja schopnosť hostiteľa vyvárať kernel objekty a manipulovať s nimi. Kernely sú špeciálne funkcie `OpenCL C` kódu uvedené kľúčovým slovom `__kernel__` a označujú vstupný bod do programu. Toto je možné si predstaviť podobne ako funkciu `main` v mnohých iných programovacích jazykoch s tým rozdielom, že v jednom zdrojovom súbore môže byť hneď niekoľko kernelov. Kernel objekt je tak reprezentáciou kernel funkcie na strane hostiteľa, ktorý ho využíva k predávaniu parametrov a zahájeniu výpočtu.

Priamo na samotné spustenie kernelu sa ale využíva ešte jedna veľmi dôležitá štruktúra a tou je fronta príkazov, anglicky command queue. Každé zariadenie v kontexte musí mať svoju vlastnú frontu príkazov, prostredníctvom ktorej môže hostiteľ poveriť zariadenie vykonaním určitého kernelu, zapísať doňho alebo čítať z neho dáta a kopírovať dáta medzi jednotlivými zariadeniami. Existujú dva rozdielne režimy manipulácie s frontou. Prvý sa nazýva in-order, čiže synchrónne spracovanie príkazov v poradí v akom boli zadané, pričom zároveň platí, že vykonávanie nasledujúceho príkazu sa nezačne skôr ako budú spracované všetky predošlé úlohy. Druhý režim sa nazýva out-of-order, teda asynchrónne spracovanie príkazov mimo zadaného poradia.

Ako prostriedok prenosu vstupných a výstupných dát sa využívajú pamäťové objekty (Mem Objects), ktoré sú dvojakého typu. Buď sa jedná o tzv. buffer objects, čiže buferové objekty, alebo sú to tzv. image objects, tj. obrázkové objekty. Buferové objekty sa využívajú v prípade prenosu jedno-, dvoj- alebo troj-dimenzionálnych dát, ktoré nemajú štruktúru obrázkov. Naopak obrázkové objekty sú špeciálne určené pre obrazové dáta, čo dáva `OpenCL` možnosť využiť určité špecifické optimalizácie. Napríklad image object môže byť na GPU umiestnený v textúrovacej pamäti. Ďalším rozdielom je, že obrázkové objekty môžu byť alebo len na čítanie, alebo len na zápis oproti všeobecným buferovým objektom, ktoré môžu mať nastavené obe tieto prístupové práva naraz.

Sampler objekt taktiež úzko súvisí s obrázkami, jeho hlavným poslaním je totižto definovať spôsob akým budú obrázky pri čítaní v kerneli vzorkované. Napríklad určuje čo sa stane ak zadané súradnice budú mimo definované rozmery obrázku a či sa jedná o ich normalizovanú, alebo nenormalizovanú verziu.

Posledným objektom z diagramu je udalosť. Tá zapúzdruje stav príkazu vykonávaného zariadením a umožňuje tak programátorovi ich synchronizáciu.

Kapitola 4

Návrh

Prvá kapitola sa sústredila na stručné zhrnutie doterajších poznatkov o simulácii tekutín a postupoch vyvinutých k ich vizualizácii. V druhej kapitole sa dalo dočítať o súčasnom stave technológií určených na programovanie a akceleráciu náročných grafických a výpočtových úloh.

Ako vidno časom sa vyformovalo množstvo rôznych metód, objavili sa nové užitočné technológie a vzniklo veľa zaujímavých programov, ktoré sa snažia riešiť problém simulácie tekutín viac, alebo menej úspešne.

Táto kapitola, tak obsahuje zhodnotenie týchto postupov, ich výhody a nevýhody a nakoniec predstaví navrhnuté riešenie.

4.1 Zhodnotenie súčasného stavu

Zhodnotenie simulačných metód

V oblasti priemyslu, leteckého inžinierstva, dizajnu automobilov a všade tam kde sú potrebné presné a spoľahlivé výpočty sú skôr populárne metódy založené na Eulerovom princípe fixnej mriežky. To je dané tým, že tieto metódy obyčajne dávajú presnejšie výsledky. Na druhej strane ale bývajú väčšinou výpočtovo a hlavne pamäťovo náročnejšie ako časticové prístupy založené na Lagrangeovom princípe. Taktiež riešenie kolízií s polygonálnym okolím, resp. integrácia takéhoto mriežkového simulátoru do hry, alebo programu, ktorý pracuje prevažne s klasickou reprezentáciou modelov pomocou trojuholníkovej siete je obvykle zložitejšia v porovnaní s časticovými metódami.

Jedným z článkov, ktoré spôsobili prielom v používaní Eulerovho prístupu v hrách je práca Josa Stama s názvom Real-Time Fluid Dynamics for Games [25]. Tá prezentovala zjednodušené riešenie Navier Stokes rovníc, ktoré je dostatočne rýchle a vizuálne vierohodné na to, aby bolo použiteľné v reálnom čase v počítačových hrách. Jeho prednosťou je okrem iného aj implementačná jednoduchosť a stručnosť, čo demonštroval i samotný Jos Stam kódom v jazyku C, ktorý má len niečo málo cez 100 riadkov. Zároveň je toto riešenie veľmi dobre paralelizovateľné a aplikovateľné na dnešný grafický hardvér. Na druhej strane je však nevýhodou potreba pomerne veľkej priepustnosti pamäte a pri väčších simuláciách aj jej pomerne veľká spotreba, čo je ale možné čiastočne riešiť kompresiou textúr používaných pri simulácii. Nakoniec je ešte dôležité spomenúť, že metóda Josa Stama je patentovaná a to môže v prípade niektorých aplikácií predstavovať zbytočné komplikácie.

Výhoda časticových systémov založených na Lagrangeovom prístupe je vo všeobecnosti v neobmedzenej simulačnej doméne, čo znamená, že kvapalina sa môže v rámci virtuálneho

sveta ľubovoľne šíriť do všetkých smerov. Toto ale nemusí byť úplne pravda pri použití simulačnej mriežky s pevnou veľkosťou buniek, ktorá môže tekutinu v pohybe obmedzovať.

Ďalším nedostatkom časticového prístupu oproti Eulerovmu je neexistencia jasne definovaného povrchu, a tak väčšinou ešte predtým než môžu byť častice vizualizované musí byť z nich tento povrch extrahovaný. Problémom je, že takýto dodatočný krok môže byť výpočtovo pomerne náročný a vo výsledku tak učiniť simuláciu neschopnou bežať v reálnom čase. Na druhej strane ale toto nemusí byť nutne neprekonateľnou nevýhodou. Pokiaľ je na akceleráciu simulácie využívaná nejaká podporná štruktúra ako je trebárs uniformná mriežka, tento nedostatok je možné vyriešiť jej znovupoužitím k extrakcii povrchu, prípadne k priamemu vykresľovaniu pomocou algoritmov známych z priameho renderovania volumetrických dát. Navyše pokiaľ nezáleží až toľko na kvalite výsledných výstupov je možné použiť aj niektorú z point splatting metód a vyhnúť sa tak extrakcii povrchu úplne.

Skutočnou nevýhodou oproti metódam založeným na Eulerovom prístupe je ale fakt, že za účelom dosiahnutia čo najvernejších výstupov je potrebné veľké množstvo častíc, čo sa samozrejme negatívne odrazí na výkone celej simulácie.

Častice však umožňujú jednotnú reprezentáciu naprieč rôznymi simuláciami a spomenuté nevýhody sú tak aspoň čiastočne kompenzované možnosťou jednoduchého prepojenia s ostatnými časticovými systémami, ako je napríklad simulácia, látky, granulárnych materiálov, soft body a rigid body simulácií a podobne.

Vďaka tomuto faktu a vďaka lepšej kompatibilite s polygonálnou reprezentáciou objektov sú časticové metódy v hrách a real-time aplikáciach väčšinou populárnejšie ako eulerovské postupy.

Zhodnotenie vizualizačných metód

K vizualizácii časticovej simulácie tekutín bolo vyvinutých množstvo postupov, z ktorých niektoré boli bližšie popísané v úvodnej prvej kapitole.

Asi najjednoduchším z nich je metóda point spritov. Tá hoci neumožňuje verné realistické zobrazenie, vďaka svojej priamočiarosti a relatívne nízkym nárokom na výpočtový výkon býva pomerne obľúbená v rámci vizualizácie ľubovoľných časticových systémov.

Z opačného spektra metód sú zas ray tracing a ray casting, ktoré umožňujú dosiahnuť veľmi verné a kvalitné realistické výstupy, ale na druhej strane patria medzi najviac výpočtovo náročné metódy. Hlavne ray tracing je často-krát aj pre dnešné moderné a rýchle počítače príliš veľké sústo a nie je moc vhodný k nasadeniu do real-time aplikácií. Z tohto pohľadu je na tom lepšie volumetrický ray casting, s ktorým je možné dosiahnuť pomerne rozumný počet snímkov za sekundu a aj pomerne kvalitné zobrazenie. Problém tejto metódy ale spočíva hlavne v množstve spotrebovanej pamäte. Pokiaľ je totiž mriežka, ktorou sú častice simulácie vzorkované príliš hrubá, tak výsledný výstup môže byť rozmazaný a významne kaziť celkový dojem z vizualizácie. Riešenie pritom kvôli nedostatku pamäte, alebo iným hardvérovým obmedzeniam, napríklad na maximálne rozmery textúr, nemusí byť vôbec jednoduché.

Rovnakým neduhom trpí aj metóda Texture Base Volume rendering, ale na druhej strane by mala byť o čosi rýchlejšia ako volume ray casting, pretože jej implementácia vedie na trocha jednoduchší shader kód a je schopná lepšie využiť efektívnu neprogramovateľnú časť hardvéru grafickej karty. Druhou nevýhodou tejto metódy je jej vysoká náročnosť na priepustnosť pamäte, čo môže byť na dnešnom hardvéri, kde je typicky priepustnosť limitujúcim faktorom veľký problém. Preto je v prípade tejto metódy veľmi dôležité orezať proxy geometriu podľa hraničného telesa volumetrických dát a znížiť tak množstvo polygónov,

ktoré treba vyrasterizovať. Miernou výhodou oproti ray casting algoritmu by mohol byť aj fakt, že táto metóda nevyžaduje použitie for cyklu v rámci shader kódu a je teda aspoň teoreticky prenositeľná aj na starší hardvér, ktorý nepodporuje cyklenie.

Metóda marching cubes aj keď už pomerne stará, je stále používaná. Hlavne pre jej veľmi dobrú kompatibilitu so súčasným grafickým hardvérom, ktorý všetko vykresľuje ako polygóny. Vďaka tomu, že extrahuje povrch z častíc simulácie je veľmi jednoduché integrovať ju do klasického herného enginu s množstvom trojuholníkovej geometrie a využiť tak klasické renderovacie postupy dobre známe z počítačovej grafiky. Metóda je taktiež veľmi dobre paralelizovateľná a možno ju celú implementovať v grafickom hardvéri, čím sa navyše výrazne zníži réžia komunikácie medzi procesom a GPU. V minulosti bola metóda patentovaná a tak vznikali jej rôzne variácie, napríklad metóda marching tetrahedra, ktorá bola spomenutá aj v prvej kapitole. Dnes však už patent vypršal a metódu je možné ľubovoľne používať. Nevýhodou však je, že na vygenerovanie rozumne presnej povrchovej reprezentácie treba vysoké rozlíšenie pomocnej vzorkovacej mriežky, čo môže stáť nemalý výpočtový výkon.

Techniky z kategórie Point splatting v súčasnosti dosahujú asi najlepšieho výkonu a hlavne umožňujú ľahko zameniť rýchle spracovanie za kvalitný výstup a naopak. Taktiež bývajú obvykle veľmi jednoduché na implementáciu, porovnateľne s metódami point spritov. Dajú sa veľmi jednoducho paralelizovať a priamočiaro namapovať na dnešný grafický hardvér. Majú pomerne skromné pamäťové nároky a nevyžadujú extrakciu povrchu z častíc, čo ich robí vhodnými pre akýkoľvek typ časticovej simulácie bez ohľadu na to, či táto využíva nejakú akceleračnú štruktúru pomocou, ktorej je schopná rýchlo extrahovať povrch kvapaliny. Výsledné výstupy sú obvykle o niečo horšie ako v prípade ostatných metód, ale pozornou implementáciou sa aj napriek tomu dá dopracovať k relatívne zaujímavým výsledkom.

Zhodnotenie súčasných technológií a softvéru

V rámci technológií dostupných k akcelerácii paralelných algoritmov je situácia veľmi pestrá ako to dobre vidno aj z druhej kapitoly tejto práce. Dvoma najväčšími hráčmi sú však rozhranie, resp. platforma CUDA od spoločnosti NVidia a priemyselný štandard OpenCL spravovaný konzorciom Khronos.

CUDA je staršia a poznať to aj na množstve a vyspelosti dostupných knižníc a nástrojov. Existuje nad ňou hromada predprogramovaných algoritmov a utilít, ktoré už majú vychytané jednotlivé chyby a veľmi dobre odladený výkon pre všetky možné karty, ktoré ju podporujú. Nástroje od NVidie sú taktiež veľmi stabilné a prepracované.

Na druhej strane OpenCL začalo v poslednej dobe tento náskok doháňať a vzniklo množstvo zaujímavých projektov a knižníc, ktoré uľahčujú programovanie pomocou tohto štandardu. Sú nimi napríklad šablónové knižnice `Boost.Compute` a AMD `Bolt`, ktoré sa podobne ako knižnica `thrust` postavená nad CUDA technológiou snažia ponúknuť programátorovi zbierku wrapper objektov, kontajnerov a paralelných algoritmov kompatibilnú s rozhraním štandardnej knižnice jazyka C++. Obidve k tomu využívajú šablónové metaprogramovanie pomocou, ktorého z jednotlivých C++ konštrukcií budujú na pozadí OpenCL C kód, ktorý potom automaticky skompilujú a vykonajú na cieľovom zariadení. AMD `Bolt` však pritom využíva pomoc neštandardných rozšírení jazyka OpenCL C o konštrukcie známe zo C++ ako sú napríklad šablóny. V súčasnosti však tieto rozšírenia podporuje len AMD implementácia OpenCL štandardu, čo vo výsledku robí túto knižnicu v porovnaní s `Boost.Compute` neprenositeľnú na implementácie ostatných výrobcov.

Ďalšími zaujímavými projektami sú aj `ViennaCL` a `VexCL`, čo sú knižnice výrazových šablón, podobne ako `Eigen`, zamerané skôr na lineárnu algebru a vedecko-technické výpočty. Okrem toho bola aj v Qt istú dobu snaha vytvoriť wrapper nad OpenCL, ktorého api by bolo modelované v štýle Qt. Tento projekt však už nie je ďalej udržiavaný a okrem wrapper funkcionality, aj keď pomerne pohodlne použiteľnej, nikdy neposkytoval prakticky nič viac navyše.

Takmer každý HW výrobca poskytuje nejaké svoje nástroje na ladenie a profilovanie OpenCL programov, SDK s množstvom príkladov, dokumentáciu a tréningové videá. Avšak ku CUDA je aj napriek tomu dostupnej oveľa viac kvalitnej dokumentácie a aj hlbších informácií o fungovaní samotného hardvéru. Toto má sčasti príčinu i v tom, že OpenCL bolo od začiatku navrhnuté ako všeobecné rozhranie, ktoré by malo fungovať naprieč všetkými možnými akcelerátormi od všetkých možných výrobcov oproti CUDA, ktorá je šitá na mieru NVidia hardvéru. Preto je už z princípu zložitejšie nájsť informácie o tom ako sa abstraktný model definovaný OpenCL štandardom mapuje na konkrétny hardvér. Taktiež CUDA je jednoduchšia na použitie a je v nej možné rýchlejšie naprototypovať paralelný algoritmus, naproti tomu OpenCL programátora zdržuje nutnosťou explicitnej kompilácie kernelov, vytvaraním kontextu, atď.

CUDA je však proprietárna uzavretá platforma, ktorá funguje len s NVidia produktami, čo ju robí pre množstvo vývojárov, ktorí si nemôžu dovoliť ignorovať AMD zákazníkov nepoužiteľnou pre ich aplikácie. Ďalšou nevýhodou oproti OpenCL je nutnosť používať na kompiláciu svojich programov `nvcc`, čo môže byť v niektorých prípadoch na obtiaž a komplikovať vývoj v okamihu, keď vývojár chce pouziť kompilátor, ktorý nie je oficiálne podporovaný v CUDA SDK.

V súčasnosti sa zdá, že ostatné API a riešenia predstavené v druhej kapitole majú viac menej len okrajovú popularitu, čo sa ale do budúcna môže výrazne zmeniť, napríklad lepšou adaptáciou OpenACC, alebo najnovšieho štandardu OpenMP v populárnych C a C++ kompilátoroch. V porovnaní CUDA s OpenCL je však CUDA v súčasnosti pre väčšinu programátorov ešte stále obľúbenejšou platformou.

Aj keď dnes už existuje niekoľko štandardov pre všeobecné výpočty na rôznych akcelerátoroch, mnohé programy stále počítajú simuláciu tekutín na procesore, alebo nedokážu využiť viac ako jedno akceleračné zariadenie simultánne, prípadne využívajú len CUDA a nie sú tak prenositeľné na AMD a Intel karty.

Tie programy, ktoré GPU akceleráciu využívajú však umožňujú interaktívne v reálnom čase simulovať až jednotky miliónov častíc. Takýmto riešením je napríklad práca R. C. Hoetzleina, alebo séria algoritmov z NVidia GameWorks určená k real-time použitiu v hrách. Obidve tieto riešenia však využívajú CUDA. Určitá snaha pridať podporu pre simuláciu kvapalín pomocou OpenCL založenú na práci R. C. Hoetzleina prebieha aj v rámci knižnice Bullet Physics, avšak ešte nie je dostupná žiadna oficiálna stabilná implementácia.

4.2 Návrh riešenia

Za účelom širokého uplatnenia medzi rôznymi užívateľmi, by mala byť aplikácia pokiaľ možno čo najviac prenositeľná naprieč rôznymi výpočtovými zariadeniami a operačnými systémami. Na základe porovnania uvedeného v predošlej podkapitole by teda mala byť implementovaná v OpenCL s použitím OpenGL pre vykresľovanie. O správu okien, vytváranie OpenGL kontextu a ďalšie platformovo závislé činnosti by sa mal starať nejaký špecializovaný framework, najlepšie Qt alebo SDL.

Aplikácia by mala byť dostatočne flexibilná na to aby mohla byť v budúcnosti rozšírená o multi-gpu podporu, prípadne nové typy akcelerátorov a mala by byť schopná efektívne využiť aj rôzne heterogénne výpočtové prostredia. Taktiež komunikácia medzi vykresľovacou fázou a simuláciou by mala mať čo najnižšiu možnú réžiu a ideálne by mala prebiehať úplne bez pomoci procesoru. Toto všetko sú len ďalšie dôvody navyše k použitiu kombinácie OpenCL a OpenGL.

Keďže OpenCL je nechválne známe tým, že na rozbehnutie aj tých najjednoduchších programov je treba napísať veľké množstvo stále sa opakujúceho kódu, mala by byť použitá nejaká knižnica, ktorá písanie programu uľahčí. Takáto knižnica by však ideálne mala poskytovať okrem jednoduchého wrapperu nad OpenCL API aj určitú zbierku predpripravených paralelných algoritmov, nad ktorými by bolo možné vystavať celú aplikáciu. Na základe porovnania z predošlej podkapitoly sa tak ako najvhodnejšie riešenie ponúka knižnica `Boost.Compute`.

Z pohľadu programovacieho jazyka je pravdepodobne najlepšou voľbou C++, ktorý predstavuje rokmi overený štandard v oblasti počítačových hier, simulácií a väčšiny výpočtovo náročných aplikácií. Umožňuje písať vysoko-úrovňový objektovo orientovaný kód a zároveň si udržať plnú kontrolu nad optimalizáciou výkonovo kritických miest programu. Jazyk C++ je taktiež perfektne kompatibilný s OpenCL aj OpenGL a všetkými ostatnými technológiami vybranými k realizácii tejto práce.

Naivná implementácia metódy Smoothed Particle Hydrodynamics má kvadratickú časovú zložitosť, pretože je potrebné spočítať vzájomné pôsobenie každej častice na každú ďalšiu časticu v simulácii. Je preto jasné, že aby bolo možné dosiahnuť interaktívne riešenie schopné behu v reálnom čase je treba použiť nejakú pokročilejšiu metódu. Relatívne jednoduché riešenie, ktoré však dosahuje veľmi dobré výsledky je prezentované v CUDA SDK v rámci príkladu Particle Simulation using CUDA [14]. Simulácia by teda mala byť založená na metóde prezentovanej v tomto dokumente.

V pomere rýchlosti ku kvalite výsledkov vychádzajú najlepšie z porovnania uvedeného vyššie metódy založené princípe na Point Splattingu. Do tejto kategórie zapadá aj technika nazvaná Screen Space Fluid Rendering with Curvature Flow, ktorá bola opäť prezentovaná v rámci výskumu vedeného spoločnosťou NVidia. Ideálna vizualizačná metóda by tak mohla byť založená napríklad na tomto článku [21], ktorého autorom sa podarilo dosiahnuť pomerne presvedčivých výsledkov pri súčasnom zachovaní jednoduchej implementácie a rozumného výkonu.

Kapitola 5

Implementácia

V predošlých kapitolách boli stručne zhrnuté najdôležitejšie poznatky z oblasti simulácie tekutín. Boli predstavené súčasné technológie akcelerácie programov na grafických procesoroch a prezentovaný návrh riešenia.

Táto kapitola sa už bude zaoberať popisom samotnej realizácie. Jej prvá časť bude zameraná na popis simulácie, bude predstavená implementácia použitej uniformnej mriežky a ďalších optimalizácií aplikovaných za účelom dosiahnutia čo najlepšieho výkonu. Druhá časť bude orientovaná na popis implementácie vizualizačného algoritmu, ktorým je metóda Screen Space Fluid Rendering with Curvature Flow a posledná časť priblíži štruktúru a objektový návrh výsledného programu.

5.1 Simulácia

Uniformná mriežka

Uniformná mriežka je všeobecná paralelná dátová štruktúra často používaná k akcelerácii časticových systémov. Podľa práce Particle Simulation using CUDA [14], z ktorej táto implementácia vychádza existujú dva rôzne prístupy k realizácii paralelnej uniformnej mriežky na GPU.

Prvý z nich využíva atomické inštrukcie prostredníctvom, ktorých si najprv zistí počty častíc v jednotlivých bunkách mriežky a na základe týchto údajov už potom môže paralelne zo všetkých vlákien naraz poznačiť do každej bunky indexy častíc, ktoré sú v nej obsiahnuté. Bunky mriežky môžu mať buď pevne danú maximálnu kapacitu, alebo môžu byť variabilné. Táto druhá možnosť však mierne komplikuje výsledný algoritmus, pretože jednotlivé bunky mriežky v tomto prípade začínajú na rôznych pamäťových adresách a tie sa ešte navyše môžu v každom snímku meniť. Vo výsledku tak nie je možné poznačiť index častice do bunky mriežky bez hrozby konfliktov z viacerých simultánnych zápisov na to isté pamäťové miesto. Algoritmus musí teda prebiehať v dvoch fázach. V prvej si opäť zistí počty častíc v jednotlivých bunkách a v druhej použije algoritmus zvaný paralelný prefix scan pomocou, ktorého sčíta počty častíc v mriežke a zistí offset jej jednotlivých buniek v pamäti.

Táto metóda však nie je príliš efektívna na masívne paralelných zariadeniach akými sú dnešné grafické procesory. To je spôsobené najmä kvôli častým prístupom do pamäte pomocou atomických inštrukcií. Tie pri prístupe z viacerých vlákien na to isté pamäťové miesto spôsobujú serializáciu a veľmi výrazné spomalenie, ktoré je ešte výraznejšie pri použití atomických inštrukcií nad globálnou pamäťou. Okrem toho však niektoré najstaršie karty (napríklad v prípade NVidia karty s Compute capability 1.0) nepodporovali atomické

inštrukcie vôbec, a tak tento algoritmus by na nich ani nemohol byť implementovaný. Ďalej však tieto inštrukcie robia metódu aj vysoko dátovo závislou, čo znamená, že jej výkon sa môže výrazne meniť v závislosti na tom ako budú jednotlivé častice usporiadané v priestore. Pokiaľ budú častice rovnomerne rozprestreté po okolí, tak výkon bude najlepší pokiaľ ale naopak bude množstvo častíc v jednej bunke tak bude dochádzať k častým serializáciám pri atomickom zápise a výkon bude najhorší.

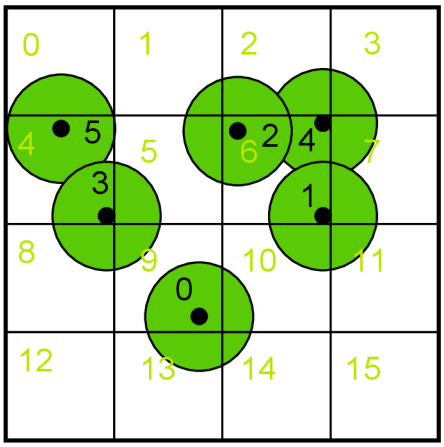
Okrem toho jednotlivé CUDA vlákna, resp. work itemy v OpenCL terminológii, spracovávajúce častice tekutiny pristupujú pri zápise indexov do buniek mriežky prakticky na náhodné miesta pamäte, čo predstavuje z pohľadu výkonu úplne najhorší možný spôsob práce s pamäťou. Prístupy do nej by mali byť ideálne združené a všetky vlákna z jedného warpu by mali pristupovať na za sebou idúce pamäťové adresy.

Druhý prístup, ktorý bol implementovaný aj v tejto práci využíva namiesto atomických inštrukcií radenie častíc.

Funguje tak, že na GPU sa naalokuje jeden kus súvislej pamäte pre uniformnú mriežku a druhý súvislý kus pre častice simulácie. Uniformná mriežka potom vyzerá tak, že obsahuje odkaz na začiatok a koniec zoradenej sekvencie častíc, ktoré do nej patria, prípade odkaz na začiatok a počet častíc v sekvencii, to už ale záleží na konkrétnej implementácii. V tejto práci obsahuje každá bunka mriežky odkaz na začiatok a koniec poľa častíc. Jednotlivé častice sa potom musia v každom snímku zoradiť na základe bunky do ktorej patria.

Príslušnosť častice ku konkrétnej bunke mriežky je možné veľmi jednoducho zistiť aj paralelne z jej aktuálnej pozície v simulácii. Túto pozíciu stačí vydeliť rozmerom jednej bunky, čím sa získa trojrozmerný index bunky v rámci uniformnej mriežky. Aby bolo radenie jednoduchšie je ešte možné tento index zlinearizovať do jednorozmerného priestoru a takto vypočítaný hash častice už možno priamo použiť ako kľúč do nejakej radiacej metódy.

Paralelné radiace algoritmy sú veľmi dobre popísaná a preskúmaná téma. Jedným z najefektívnejším sériových, ale aj paralelných radiacich algoritmov je radix sort.



Cell id	Count	Particle id
0	0	
1	0	
2	0	
3	0	
4	2	3, 5
5	0	
6	3	1, 2, 4
7	0	
8	0	
9	1	0
10	0	
11	0	
12	0	
13	0	
14	0	
15	0	

Obrázek 5.1: Ilustrácia princípu fungovania uniformnej mriežky. Obrázok naľavo znázorňuje uniformnú mriežku, tabuľka napravo zas pamäťové usporiadanie častíc a buniek mriežky. Zelené čísla predstavujú čísla týchto buniek, resp. ich hashe, zelené krúžky zas jednotlivé častice simulácie spolu s ich vyhladzovacím polomerom. Čierne čísla sú indexy jednotlivých častíc simulácie. Položka Count z tabuľky naľavo potom predstavuje počet častíc v jednotlivých bunkách a položka Particle Id ukazuje zoznam častíc prináležiacich konkrétnej bunke. Obrázok je prebraný z [14].

Ten funguje tak, že v prvom kroku sa vybuduje histogram zo vstupných čísel podľa ich poslednej najmenej významnej číslice. Na základe neho sa potom spočíta pre každú takto vzniknutú skupinu čísel offset od začiatku radeného poľa a v poslednej fáze sa už len preusporiadajú vstupné čísla podľa vybudovaného histogramu a offsetov. Celý tento proces sa následne opakuje d-krát. Pričom d predstavuje maximálny počet platných číslic v radených číslach. Pokiaľ majú napríklad vstupné čísla 32-bitov a chceme radiť podľa jednotlivých jedno-bitových číslic, tak vyššie opísaný postup sa bude opakovať 32-krát a histogram bude mať 2 rôzne položky. V praxi je potom typické radiť podľa 8-bitových číslic a používať histogram s 256 položkami.

V paralelnej verzii je radix sort veľmi podobný vyššie popísanému algoritmu, akurát namiesto sekvenčného sčítania offsetov sa použije paralelný prefix scan a na vytvorenie histogramu sa použije nejaká paralelná metóda, ako je napríklad technika privatizovaného lokálneho histogramu spolu s atomickými inštrukciami. Implementácia paralelného radix sortu však nie je úplne jednoduchá a keďže knižnica `Boost.Compute` poskytuje pomerne rýchlu metódu radenia polí podľa kľúča a hodnoty založenú na radix sorte, tak bola použitá metóda `sort_by_key` z tejto knižnice.

Ďalším krokom ešte predtým než sa začne počítať samotná simulácia je prezoradenie častíc v pamäti. Tento krok je v skutočnosti nepovinný a bolo by možné v rámci ostatných kernelov indexovať potrebné atribúty častíc aj nepriamo pomocou zoradených indexov, ale je dobré ho urobiť kvôli lepšej pamäťovej lokalite. Nezdružené prístupy do pamäte v ostatných fázach simulácie sú v konečnom dôsledku oveľa drahšie ako jeden preusporiadavací kernel navyše. Okrem toho netreba vždy prezoraďovať všetky pomocné polia. Napríklad v prípade implementácie v tejto práci stačí preusporiadať len pozíciu a rýchlosť častíc, ostatné atribúty sú vďaka tomu ako je algoritmus navrhnutý vždy pred akýmkoľvek čítaním kompletne prepísané novo vypočítanými hodnotami.

Poslednou fázou algoritmu je použitie uniformnej mriežky k vyhľadaniu najbližších susedov každej častice v kerneloch na výpočet tlaku a vzájomných síl. Jednotlivé bunky mriežky majú v implementácii v rámci tejto práce fixne danú veľkosť počas celej simulácie, ktorá je nastavená tak, aby bola rovná vyhladzovaciemu polomeru častíc SPH algoritmu. Z toho následne vyplýva, že na výpočet tlaku a vzájomných síl medzi časticami treba skontrolovať okrem aktuálnej bunky aj ďalších 26 susedných buniek. To je kvôli tomu, že častica sa v rámci bunky môže nachádzať mimo iného i v ktoromkoľvek z jej ôsmich rohov a môže tak teoreticky svojim vyhladzovacím polomerom zasahovať do ľubovoľnej susednej bunky. Rozmer bunky je možné zvoliť väčší i menší ako vyhladzovací polomer, tomu je však potom treba adekvátne prispôsobiť aj veľkosť prehľadávaného susedného okolia.

Ďalšou prekážkou, ktorú bolo treba vyriešiť je indexovanie mimo platný rozsah mriežky. To sa môže stať napríklad u okrajových buniek pri kontrolovaní susedov. Jedným zo spôsobov ako toto vyriešiť je použitie podmienky vnútri OpenCL C kódu, čo však z pohľadu výkonu nie je úplne najlepšie, pretože to pridáva množstvo inštrukcií navyše čo potom zbytočne zdržuje hlavný výpočet a zaberá cenné registre. V tejto práci bol teda použitý prístup zalamovania indexov pomocou modulo operácie. Tú je možné pre čísla, ktoré sú mocninou dvojky veľmi rýchlo spočítať pomocou bitového maskovania najbližším menším číslom ako je oná mocnina. Napríklad pre bunku číslo $(0, 0, 0)$, jej suseda pravého horného suseda s indexom $(-1, -1, -1)$ a mriežku o rozmeroch 8x8x8 buniek, by táto metóda vrátila výsledok $(7, 7, 7)$. Keďže častice v bunke $(7, 7, 7)$ sú príliš ďaleko od častíc v bunke $(0, 0, 0)$, tak výsledok zostane korektný. Tento prístup má však aj niekoľko obmedzení, konkrétne rozmery uniformnej mriežky musia byť mocninami čísla dva a musia byť väčšie ako 2x2x2, pretože inak by boli častice so susednej bunky po zalomení započítané 2-krát. Toto však nie

je až taký veľký problém, pretože mriežka tak malá ako 2x2x2 nemá príliš veľké praktické opodstatnenie.

Hoci bunky uniformnej mriežky implementovanej v tejto práci majú všetky rozmery rovnaké, samotná uniformná mriežka môže mať v každom smere iný počet buniek. To umožňuje definovať napríklad širokú ale nízku mriežku, ktorá bude schopná efektívnejšie pokryť simulovanú tekutinu.

Keďže jednotlivé bunky mriežky majú fixnú veľkosť odvodenú od vyhladzovacieho polomeru častíc, tak jej rozmery úzko súvisia aj s celkovou fyzickou veľkosťou simulačnej domény kvapaliny. Tá je aplikáciou automaticky dopočítaná z rozmerov mriežky a veľkosti buniek. Je možné definovať doménu aj priamo, ale program tomu opäť automaticky prispôsobí i mriežku.

V práci Particle Simulation using CUDA jej autor počíta hashe častíc v samostatnom kroku pred ich zoradením algoritmom radix sort. V tejto práci sú z dôvodu optimalizácie hashe počítané samostatne len raz počas inicializácie a potom je ich počítanie spojené spolu s integráciou v poslednom kroku simulácie, čím sa ušetrí jedno volanie kernelu na snímok.

5.2 Vykresľovanie

Metóda Screen Space Fluid Rendering with Curvature Flow implementovaná v tejto práci sa skladá z niekoľkých krokov:

- Generovanie hĺbkovej mapy scény
- Generovanie mapy hustoty scény
- Vyhladenie hĺbkovej mapy
- Výpočet normál z vyhladenej hĺbkovej mapy a skombinovanie so zvyškom scény

Generovanie hĺbkovej mapy scény

Prvý z nich je pomerne priamočiary a jednoduchý. Shader používaný k realizácii tejto fázy je viac menej rovnaký ako v prípade kreslenia pomocou Point Sprite metódy popísanej v prvej kapitole 2.2. Rozdiel je akurát v tom že nie je potrebné počítať žiadne osvetlenie a výstupom je len hĺbka pixelu v clip súradniciach. Aby nebolo treba ani normalizovať hodnoty ukladané do textúr sú vo všetkých fázach tohto algoritmu použité jedno-kanálové float textúry, ktoré podľa špecifikácie OpenGL nie sú automaticky orezávané do intervalu [0,1].

Generovanie mapy hustoty scény

Za normálnych okolností sa tenká vrstva vody javí ako priehľadná a s narastajúcim množstvom kvapaliny je svetlo, ktoré cez ňu prechádza stále viac a viac utlmované. Účelom druhého kroku je teda poskytnúť aproximáciu tohto javu tým, že v miestach, kde je veľa častíc sa bude výsledná farba javiť tmavšia a aj jej opacita bude vyššia a naopak v miestach, kde je častíc málo bude opacita nižšia. Shader, ktorý implementuje tento proces je nápadne podobný kódu z prvého kroku, avšak tento raz už nie je používaný žiaden bufer hĺbky, ale vygenerované pixely sú spolu sčítané pomocou aditívneho blendovania, ktoré je v OpenGL možné zapnúť príkazom `glEnable(GL_BLEND)` a nastavením blendovacieho módu na `glBlendFunc(GL_ONE, GL_ONE)`.

Obrázek 5.2: Výsledok prvej fázy - hĺbková mapa scény. Obrázok bol vygenerovaný vytvoreným programom.

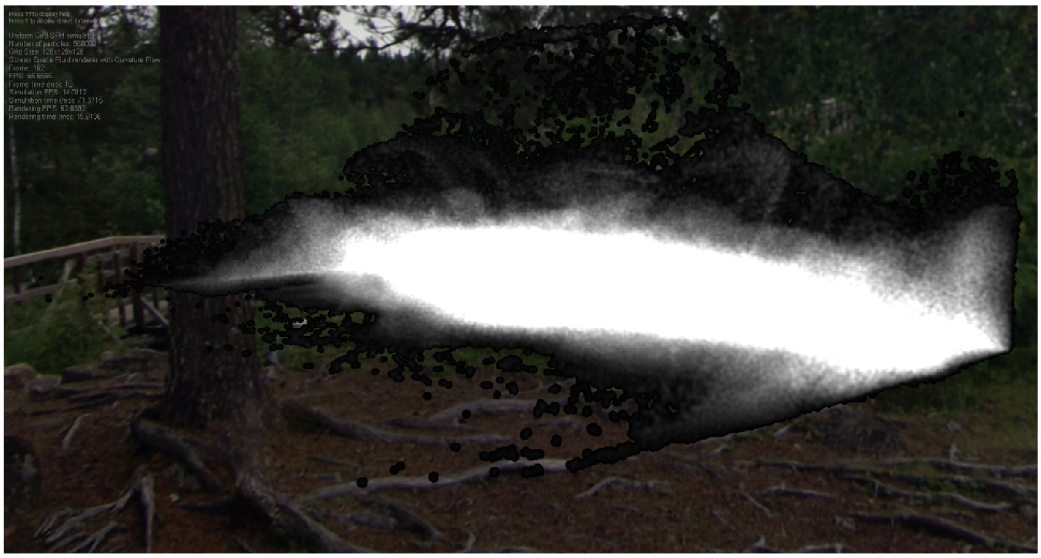
Podľa článku zmieňovaného v úvode kapitoly, na ktorom je táto implementácia založená by malo byť počas generovania mapy hustoty testovanie hĺbky zapnuté a výsledkom fragment shaderu by mala byť hodnota funkcie d udávajúca hĺbku častice v aktuálnom mieste. Následným sčítaním týchto hodnôt skrz aditívne blendovanie by potom mala vyjsť hľadaná aproximácia pre hustotu kvapaliny.

Podľa ďalšej prezentácie [15], ktorá sa tejto metóde venuje by zas mal byť test na hĺbku fragmentu vypnutý. Zapnuté by malo byť len aditívne blendovanie a častice by mali byť kreslené ako obyčajné koliečka alebo ako machule, ktoré postupne smerom k okrajom blednú. Experimentovaním bolo vizuálne najlepších výsledkov dosiahnuté pri vypnutom teste hĺbky a výpočte hustoty pomocou vzorca $1 - length(xy)$, kde xy sú 2D súradnice pixelu v rámci vyrasterizovaného point spritu a `length` je odpovedajúca funkcia jazyka GLSL.

Tento krok algoritmu môže byť pomerne náročný na výpočtový výkon, pretože vyžaduje spracovanie veľkého množstva fragmentov, ktoré musia byť spolu sčítané blendovaním. Keďže sa však jedná len o aproximáciu, tak v prípade potreby môže byť výstup tejto fázy podvzorkovaný bez výrazne viditeľného zhoršenia kvality výstupu. Implementácia uskutočnená ako súčasť tejto práce umožňuje nastaviť ľubovoľnú mieru podvzorkovania, ktorá je vo východzom stave nastavená na jednu šestnástinu z normálnej veľkosti okna, teda na každej strane je mapa hustoty podvzorkovaná 4-krát.

Vyhladenie hĺbkovej mapy

Tretí krok je gro samotného algoritmu a hlavne vďaka nemu vyzerá výsledok ako súvislá hladká kvapalina. Spôsobov ako hĺbku rozmazať je niekoľko. Tým najjednoduchším je asi klasické Gaussovské jadro. Problém tohto prístupu je ale v tom, že nezachováva ostré hrany. V prípade obrázku z bufferu hĺbky sú totižto ostré prechody väčšinou generované oddelenými objektami, ktoré sú v scéne od seba v skutočnosti pomerne vzdialené. Rozmazaním týchto hrán sa tak potom vytvorí falošná ilúzia akoby tieto separátne objekty boli spolu nejako prepojené.



Obrázek 5.3: Výsledok druhej fázy - mapa hustoty scény. Obrázok bol vygenerovaný vytvoreným programom.

Lepšou alternatívou je preto bilaterálny filter, ktorý je však neseparabilný, čo znamená že jeho výpočet nemožno rozdeliť a počítať v x-ovom a y-ovom smere zvlášť a je tak pomerne náročný na výkon. V prezentácii [15] je aj napriek tomu tento filter implementovaný ako separabilný, čo síce spôsobuje očividné artefakty, tie však po aplikácii osvetlenia a ostatných efektov nie sú až tak badateľné.

Ďalším spôsobom ako vyhladiť hĺbku častíc je metóda Curvature Flow, ktorá je použitá aj v tejto práci a prezentovaná článkom spomenutom v úvode. Jej implementácie je pomerne priamočiara, keďže v tomto článku sú už odvodené všetky potrebné výpočty, ktoré tak už len stačí prepísať do GLSL kódu.

V princípe sa metóda snaží dosiahnuť vyhladenie povrchu kvapaliny postupným priemerovaním jeho zakrivenia. K tomu potrebuje z hĺbky v depth buferi spočítať pôvodnú polohu 3D bodu vo view súradniciach, ktorú následne zderivuje v x-ovom a y-ovom smere, čím určí normálu k povrchu kvapalinu v tomto bode. Tú následne ešte znormalizuje a dosadí do vzorca na výpočet priemerného zakrivenia.

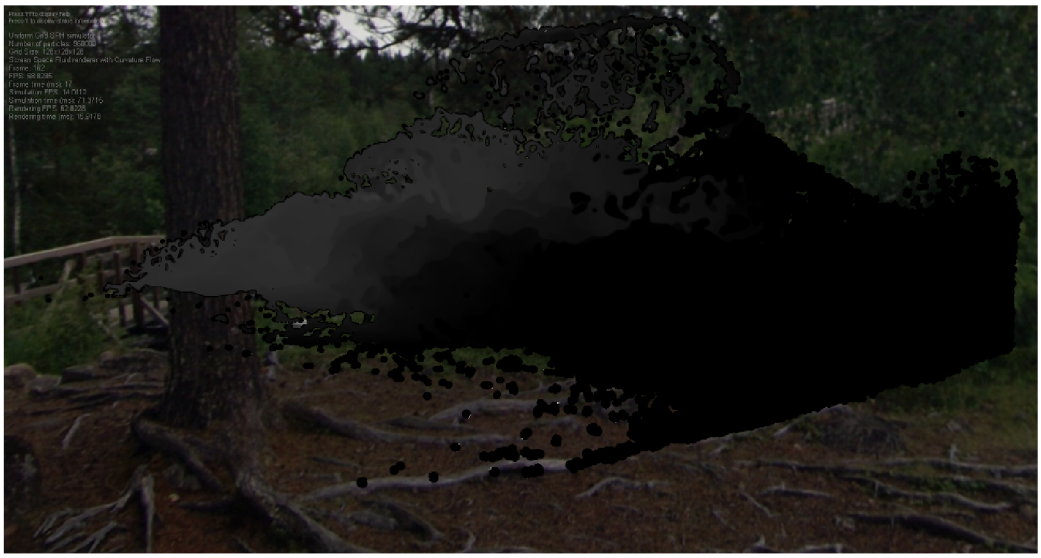
Tento krok prebieha dookola v niekoľkých iteráciách až kým nie je dosiahnuté požadované vyhladenie. Východzí počet vyhladzovacích iterácií je v tejto práci nastavený na číslo 49, avšak aplikácia umožňuje tento počet v ľubovoľnom okamihu zmeniť a prispôsobiť tak kvalitu výsledkov rýchlosti zobrazenia a naopak.

Výpočet normál z vyhladenej hĺbkovej mapy a skombinovanie so zvyškom scény

Účelom posledného kroku je z rozmazanej hĺbky vypočítanej v tretej fáze určiť normály k povrchu kvapaliny a spočítať osvetlenie spolu s ďalšími prípadnými efektami.

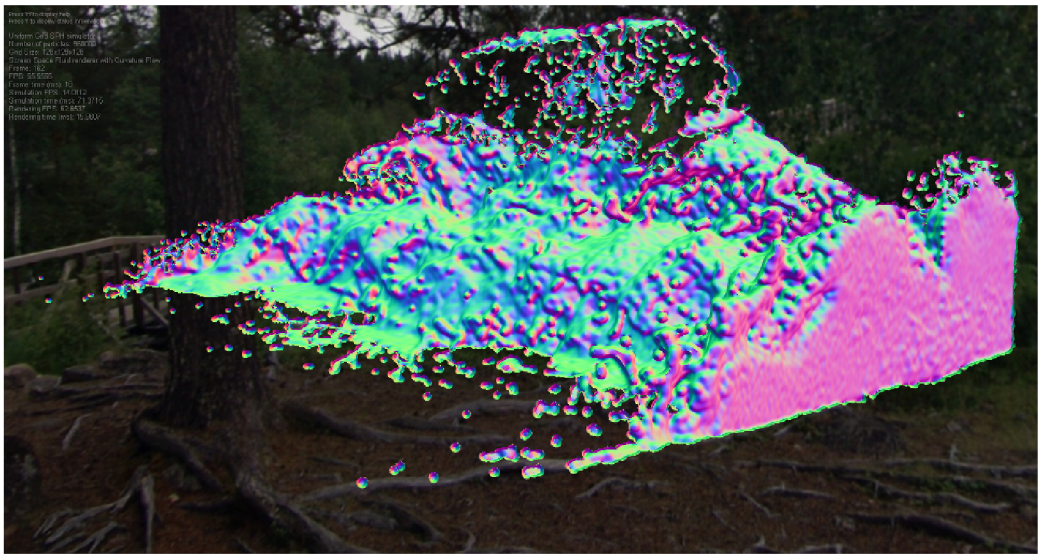
Počítanie normály funguje tak, že sa najprv určia parciálne derivácia hĺbky z depth bufferu v x-ovom aj y-ovom smere a z nich sa potom pomocou vektorového súčinu spočíta výsledná normála.

Odrazy sú počítané pomocou funkcie reflect z jazyka GLSL, ktorá k zadanému normálovému vektoru a pohľadovému vektoru smerom od kamery k miestu reflekcie určí, vektor



Obrázek 5.4: Výsledok tretej fázy - vyhladenie hĺbkovej mapy. Obrázok bol vygenerovaný vytvoreným programom.

ktorý bude orientovaný v smere odrazenému lúča svetla podľa zákonu odrazu. Tento odrazený lúč je potom použitý na adresovanie OpenGL cube map textúry z ktorej je vyčítaná hodnota odrazeného pixelu.

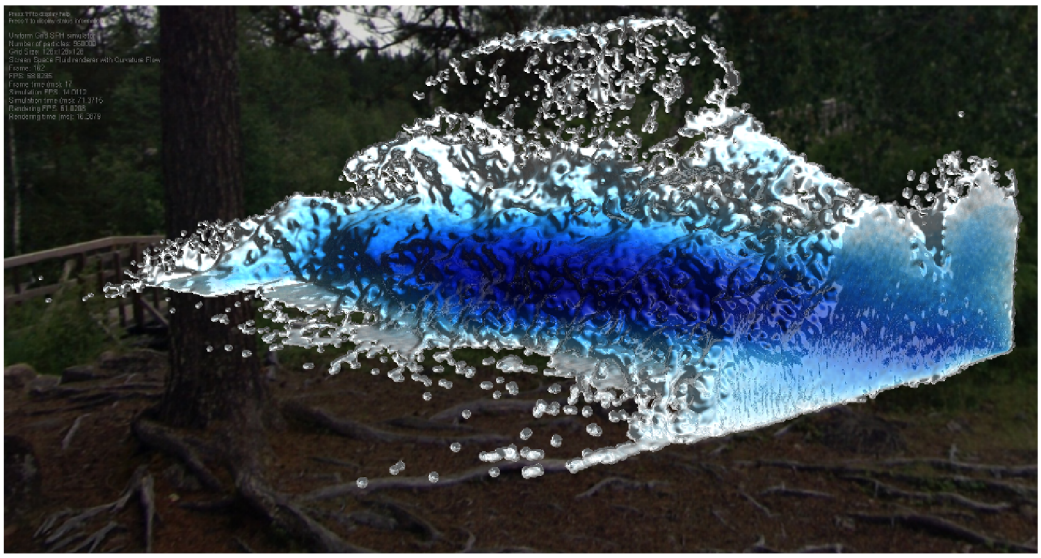


Obrázek 5.5: Výsledné normálové vektory vyhladeného povrchu kvapaliny. Obrázok bol vygenerovaný vytvoreným programom.

5.3 Štruktúra programu

Program je napísaný podľa najnovšieho štandardu jazyka C++11 a plne využíva jeho objektových vlastností.

Jadro programu tvoria dve rôzne hierachie objektov. Tá prvá je založená na triede



Obrázek 5.6: Finálny výsledok po spojení všetkých medzi krokov. Obrázok bol vygenerovaný vytvoreným programom.

`ParticleSystem` a týka sa implementácie simulácie, pričom tá druhá je založená na triede `BaseRenderer` a týka sa všetkého čo súvisí s vizualizáciou.

`ParticleSystem` je abstraktná trieda, ktorá má dve chránené čisto virtuálne metódy `reset_impl` a `update_impl`. Tá prvá sa zavolá v momente, keď treba simuláciu reštartovať a je určená k implementácii úloh akými sú napríklad alokácia a inicializácia pamäte, ale už neslúži ku kompilácii kernel programov, ktorá by mala prebehnúť len raz v konštruktore odvodenej triedy. Druhá metóda, `update_impl`, je určená na implementáciu samotného simulačného algoritmu a je zavolaná raz pre každý nový snímok. Okrem toho však `ParticleSystem` obsahuje aj niekoľko atribútov a metód spoločných pre všetky odvodené časticové systémy. Tými sú napríklad metódy používané ku komunikácii s renderovacím subsystémom, odkaz na OpenCL kontext, fronta OpenCL príkazov, alebo objekt typu `ocl::PerfStats` použiteľný k zberu štatistík o výkone.

Z triedy `ParticleSystem` je ďalej odvodená trieda `FluidParticleSystem`, ktorá slúži ako spoločný základ pre všetky časticové systémy, ktoré sa týkajú simulácie tekutín a z nej sú potom odvodené ešte tri konkrétne triedy `SPHNaive`, `SPHOptimized` a `SPHUniformGrid`, ktoré už implementujú samotný simulačný algoritmus.

Prvá z týchto konkrétnych tried je referenčná verzia, ktorá nepoužíva uniformnú mriežku a ani nie je veľmi optimalizovaná. Bola využívaná počas vývoja k verifikácii výsledkov z druhých dvoch tried. `SPHOptimized` tiež ešte nepoužíva uniformnú mriežku, ale je už lepšie optimalizovaná. Táto verzia bola používaná predovšetkým k testovaniu niektorých optimalizačných nápadov. Posledná menovaná trieda je už plne optimalizovaná verzia simulácie využívajúca uniformnú mriežku k rýchlemu výpočtu tlaku a vzájomných síl medzi časticami.

K správe parametrov časticovej simulácie tekutiny slúži trieda `SPHParams`, ktorá zapúzdruje štruktúru `tSimParams`, ktorá je zdieľaná medzi OpenCL C kódom a C++ kódom a definuje samotné parametre. Štruktúra `tSimParams` je obyčajná POD štruktúra jazyka C, ktorá je zdieľaná medzi OpenCL C a C++. Takého riešenie zaručuje lepšiu flexibilitu, udržovateľnosť do budúcnosti a v konečnom dôsledku aj pomáha znižovať množstvo chýb

a sprehľadňuje program.

Trieda `BaseRenderer` je podobne ako trieda `ParticleSystem` abstraktná a taktiež definuje dve chránené čisto virtuálne metódy `reset_impl` a `render_impl` plus jednu neabstraktnú chránenú virtuálnu metódu `resize_impl`. Tá posledná je zavolaná v okamžiku, keď sa zmenia rozmery okna do ktorého sa kreslí a je užitočná napríklad kvôli reinicializácii frame buffer objektu používaného napríkad v rámci odvodenej triedy. Prvá menovaná metóda slúži k rovnakému účelu ako v prípade `ParticleSystem` a to k inicializáciu potrebnej pamäte. Opäť však nie je určená na kompiláciu shader programov, ktorá by mala prebehnúť už v rámci konštruktoru odvodenej triedy. A nakoniec metóda `resize_impl`, tá je zavolaná raz pre každý nový snímok a je určená k vyrenderovaniu simulácie na obrazovku. Okrem spoločného rozhrania, ktoré musia všetky odvodené triedy povinne implementovať a množstva atribútov a spločných metód definuje trieda `BaseRenderer` aj niekoľko základných vykresľovacích krokov, ktoré by sa inak v každom odvodenom rendereri opakovali. Sú to napríklad vykresľovanie Skyboxu a vizuálnej indikácie domény kvapaliny.

Z `BaseRenderer` už ďalej odvodené dva konkrétne rendereri. Prvým z nich je trieda `PointSpriteRenderer`, ktorá sa stará o vykresľovanie pomocou Point Sprite metódy a tým druhým je trieda `CurvatureFlowRenderer`, ktorá implementuje metódu Screen Space Fluid Rendering with Curvature Flow.

Okrem toho implementácia do veľkej miery spolieha na menné priestory, ktoré používa na oddelenie jednotlivých logických celkov kódu.

Kapitola 6

Testovanie a vyhodnotenie

Predošlé kapitoly sa sústredili na popis implementovaných algoritmov, stručné zhrnutie súčasných teoretických poznatkov z oblasti simulácie tekutín, bolo uvedených niekoľko možných techník ich vizualizácie a predstavené technológie dostupné k ich implementácii.

Táto kapitola sa preto zameriava na popis dosiahnutých výsledkov. Na začiatku bude predstavená hardvérová a softvérová konfigurácia používaná pri vývoji a testovaní spolu s metodikou merania času stráveného vykonávaním simulácie a renderovania. Nasledovať bude zhodnotenie dosiahnutých výsledkov a posledná podkapitola sa zamerá na možné optimalizácie a vylepšenia do budúcnosti.

6.1 Testovanie

Testovací hardvér a softvér

Testovanie prebiehalo na dvoch rôznych strojoch. Prvým bol pomerne slabý, takmer 5 rokov starý laptop Acer Aspire TimelineX. Na tomto počítači je osadený procesor Intel Core i5 460M, čo je dvojjadrový procesor s hyperthreadingom, čiže technológiou ktorá umožňuje na jednom jadre simultánne spracovávať až dve vlákna. Je vybavený 8 GB operačnej pamäte, ktorá bola rozšírená z pôvodných 4GB a zároveň je na ňom nainštalovaný 64-bitový operačný systém Windows 7 schopný túto pamäť efektívne využiť. Z pohľadu grafického hardvéru je počítač vybavený integrovanou kartou Intel, ktorá však v rámci procesorovej architektúry Sandy Bridge ešte nepodporuje OpenCL a dedikovanou grafickou kartou ATI Mobility Radeon HD 5650, ktorá podporuje OpenCL vo verzii 1.2. Počas testov bola integrovaná karta zakázaná v Biose základnej dosky a používaná bola len dedikovaná karta.

Druhou testovacou konfiguráciou bol výkonný desktopový hráčsky počítač s procesorom Intel Core i5 4690K so základnou taktovacou frekvenciou 3500 MHZ. Tento procesor je už založený na novej architektúre Haswell, ktorá oproti Sandy Bridge obsahuje integrovaný grafický čip s podporou OpenCL 1.2. Pri testovaní však bola aj v tomto prípade integrovaná karta vypnutá v Biose základnej dosky. Ďalej tento počítač disponuje 8GB DDR3 pamäte a je na ňom nainštalovaný 64-bitový operačný systém Windows 8.1. O grafickú stránku sa stará jedna z najvýkonnejších kariet súčasnosti, SAPPHIRE R9 280X DUAL-X. Táto karta je postavená na grafickom čipe AMD Radeon R9 280X s architektúrou GCN (Graphics Core Next), disponuje 3GB rýchlej operačnej pamäte typu GDDR5 a teoretickým výkonom približne 3.5 tera flops.

V rámci testov boli všetky komponenty v obidvoch konfiguráciách taktované na základnej frekvencii určenej výrobcom.

Ako už bolo spomenuté vyššie testovanie prebiehalo na 64-bitových operačných systémoch Windows 7 a Windows 8.1 s nainštalovanými najnovšími verziami ovládačov. Na profilovanie kódu bol používaný program AMD CodeXL, ktorý okrem profilácie umožňuje aj ladiť OpenCL a OpenGL programy a staticky analyzovať výkon shader kódu pre rôzne modely grafických kariet od firmy AMD. K prvotnému približnému odhad zaťaženia systému bola využívaná utilita GPU-z od vývojára TechPowerUp. Veľkou výhodou tohoto programu je, že ho netreba vôbec inštalovať a okrem základného sumáru informácií o systéme a nainštalovanej grafickej karte poskytuje aj množstvo senzorov pomocou ktorých je jednoduché sledovať aktuálne percentuálne vyťaženie grafického čipu, aktuálnu teplotu, taktovaciu frekvenciu a množstvo spotrebovanej pamäte a zároveň všetky tieto informácie logovať do súboru.

Testovacia metodika

Generovanie dát pre priložené grafy prebiehalo automaticky spustením aplikácie z príkazového riadku a vykresľovaním do offscreen bufferu.

K meraniu času stráveného vykonávaním OpenCL kódu bola naprogramovaná pomocná trieda `Timer`. Tá funguje tak, že do zadanej fronty OpenCL príkazov vloží pred a za označený úsek programu špeciálne zarážky, tzv markery. Spracovanie markeru sa potom začne až v okamihu, keď sa dokončí vykonávanie všetkých ostatných príkazov pridaných do fronty pred ním. V OpenCL je možné marker pridať pomocou funkcie `clEnqueueMarker` a zároveň je možné nastaviť callback, ktorý sa zavolá pri jeho dokončení. Pomocou tohto callbacku sa dá zistiť časové razítko začiatku a konca spracovávania marker príkazu. Z týchto údajov je už potom jednoduché zistiť dĺžku trvania sledovaného úseku programu, stačí odčítať koncový čas počiatočného markeru od začiatočného času koncového markeru.

OpenCL je schopné merať dobu spracovania jednotlivých príkazov aj natívne, popísané riešenie je však flexibilnejšie v tom, že umožňuje merať dĺžku spracovania skupín niekoľkých príkazov naraz. Je tak možné presne zistiť napríklad dobu trvania celého jedného snímku simulácie. Navyše toto riešenie funguje aj asynchrónne a teda neblokuje a nespomaľuje beh aplikácie a umožňuje aj ľubovoľné vnorovanie sledovaných blokov kódu.

K meraniu množstva času stráveného renderovaním bol použitý OpenGL timer query. Timer query bol pôvodne len rozšírením OpenGL špecifikácie a súčasťou základnej špecifikácie sa stal až v OpenGL verzie 3.3. V záujme čo najlepšej prenositeľnosti výslednej aplikácie a pretože Qt od verzie 5 poskytuje sadu jednoduchých wrapperov nad týmito objektami bol timer query použitý prostredníctvom triedy `QOpenGLTimeMonitor`. Qt poskytuje ešte jednu podobnú triedu nazvanú `QOpenGLTimerQuery`. Táto druhá trieda má oproti `QOpenGLTimeMonitor` nízkoúrovňovejšie rozhranie a zdieľa niekoľko nedostatkov, ktorými trpia aj čisté OpenGL query objekty. Konkrétne, pokiaľ už nejaký timer query beží, teda bola zavolaná funkcia `glQueryBegin` s parametrom `GL_TIME_ELAPSED`, tak nie je možné vytvárať nové timer query objekty a ani volať ďalšie `glQueryBegin` príkazy pred ukončením tohto prvého query príkazom `glQueryEnd`. Trieda `QOpenGLTimeMonitor` rieši tento problém využitím timer query s parametrom `GL_TIMESTAMP` a ručným dopočítaním výsledných intervalov.

Za účelom profilovania aplikácie počas vývoja boli ešte naprogramované pomocné triedy `ogl::PerfStats` a `ocl::PerfStats`. Tie využívajú OpenCL udalosti a dáta z vyššie popísaných časovačov na zber štatistík o celkovom behu programu, ktoré potom zobrazia pred jeho ukončením vo forme textovej tabuľky.

Zaujímavá je pritom implementácia metódy `event` z triedy `ocl::PerfStats`. Tá vra-

cia dočasný proxy objekt, ktorý má definovaný operátor konverzie na `cl_event*`, čím je zabezpečené, že tento objekt sa dá predať ako parameter do ľubovoľnej OpenCL funkcie, ktorá pracuje s frontou príkazov a prostredníctvom svojho posledného parametru umožňuje priradiť k danému príkazu OpenCL udalosť. Po inicializácii proxy objektu takouto funkciou sa skrz jeho deštruktor nastaví callback na novovygenerovanú udalosť a keď potom príkaz, ktorý je s ňou spojený skončí, príslušný callback zaznamená dobu jeho trvania v štatistike vedenej odpovedajúcim `ocl::PerfStats` objektom. Tento spôsob je veľmi pohodlný na použitie a umožňuje získať pomerne dobrý odhad o výkone celého systému a jeho najkritickejších častiach.

Celkový čas na snímok je určený ako súčet času renderovania a simulácie odmeraný pomocou vyššie uvedených metód. Tento prístup nezohľadňuje réžiu CPU spracovania, ktorá by však v prípade tejto aplikácie by mala byť minimálna a možno ju teda zanedbať.

Počas testovania bol laptop z prvej konfigurácie celý čas pripojený na nabíjačku. Toto je veľmi dôležité, pretože v opačnom prípade sa automaticky k slovu dostane technológia AMD PowerPlay, ktorá v rámci šetrenia batérie zredukuje výkon grafickej karty zhruba na polovicu. Ďalšia dôležitá vec je, že počas testovania boli všetky nepotrebné programy vypnuté, aby sa vylúčila možnosť ich negatívneho vplyvu na merania.

Zhodnotenie výsledkov

Z pohľadu dnešných počítačových hier je za optimálnu hodnotu považovaných 30-60 snímkov za sekundu, čo umožňuje najlepšie synchronizovať generovanie obrazu s obnovovacou frekvenciou obrazovky. Všeobecne je však za hranicu real-time zobrazenia považovaná ešte aj hodnota 15 fps, pod touto hranicou však už dochádza k znateľnému trhaniu obrazu a nepríjemnému užívateľskému zážitku. V tejto práci je tak za real-time považovaná ľubovolná hodnota vyššia 15 snímkov za sekundu, čo znamená že súčet času spotrebovaného na simuláciu a vizualizáciu dohromady nesmie presiahnuť 66 milisekúnd.

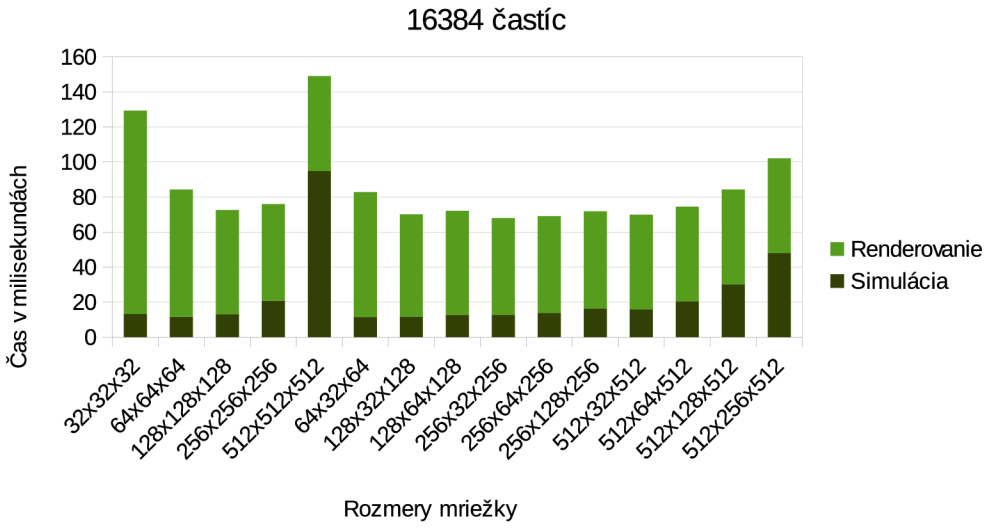
Pre každú testovanú konfiguráciu bolo vytvorených 5 grafov, ktoré ukazujú namerané časy na snímok v závislosti od veľkosti uniformnej mriežky a počtu častíc. V jednotlivých stĺpcoch je potom vždy tmavozelenou farbou naznačený podiel simulácie na celkovom čase na snímok a bledozelenou farbou je zobrazený podiel vizualizácie. Všetky namerané časy sú udávané v milisekundách ako priemer z 1000 simulovaných snímkov.

Podľa výsledkov nameraných na prvej konfigurácii, 5-ročnom latope, by sa mohlo zdať, že program vôbec nie je schopný behu v reálnom čase. Toto bolo bohužiaľ spôsobené nesprávnym nastavením parametrov vizualizácie počas testovania. Scéna bola v tomto prípade umiestnená príliš blízko kamery, čo spôsobilo vygenerovanie príliš veľkých point spritov a následne veľmi spomalilo celú vizualizáciu. Bohužiaľ táto chyba vyšla najavo až pri spracovávaní meraní a tie už pre svoju veľkú časovú náročnosť neboli opakované.

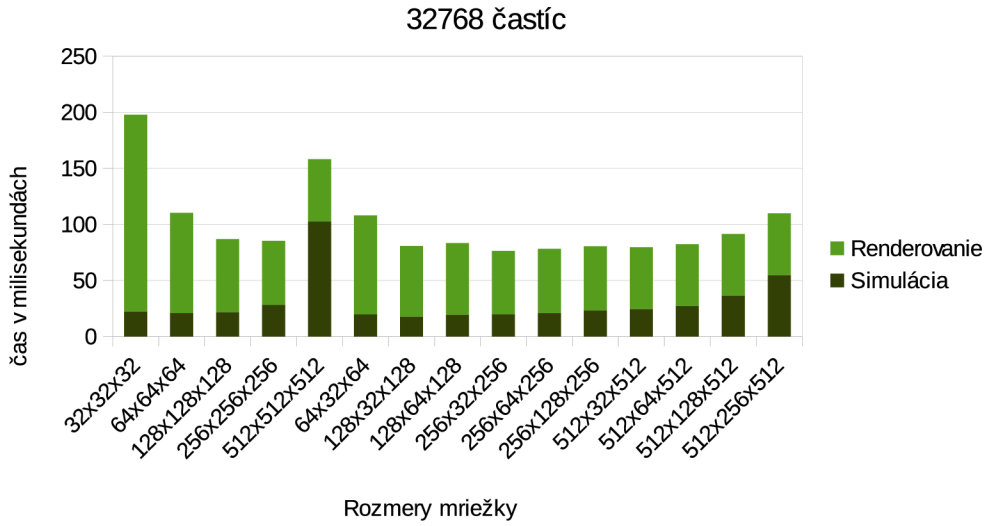
Z grafov a tabuliek priložených do prílohy je však vidieť, že program bol na tomto počítači schopný zvládnuť simuláciu v reálnom čase až do veľkosti 131072 častíc. Taktiež je možné si všimnúť, že s rastúcimi rozmermi mriežky má rastúcu tendenciu aj výkon programu a to vo väčšine prípadov až k hranici 256x256x256 buniek uniformnej mriežky. Od tohto bodu už potom nároky na udržovanie mriežky prevyšujú jej benefity, čo sa znova zmení s prechodom na mriežku o neuniformných rozmeroch.

To je spôsobené tým, že rozmery buniek uniformnej mriežky sú dané fixne a sú nastavované podľa vyhladzovacieho polomeru. Z toho automaticky vyplýva, že aby mohla uniformná mriežka efektívne pokryť celú simulačnú doménu, tak počet jej buniek na každej strane musí korešpondovať s nastavením tejto domény. Simulačná doména, ktorá je

plochá však umožňuje kvapaline, aby sa lepšie rozprestrela do okolia a tým sa vo výsledku aj automaticky zníži priemerný počet susedov každej častice.



Obrázek 6.1: Výsledky testovania s 16384 časticami na konfigurácii 1

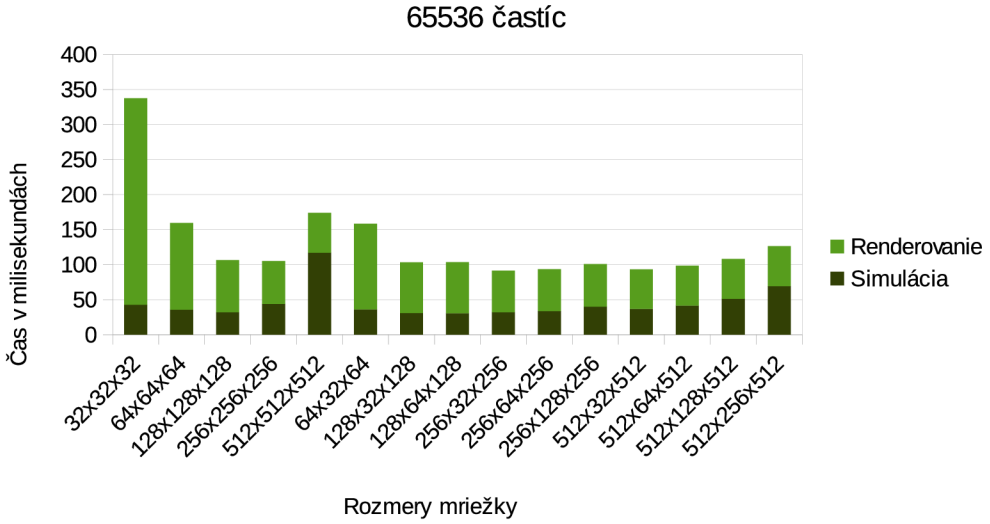


Obrázek 6.2: Výsledky testovania s 262144 časticami na konfigurácii 1

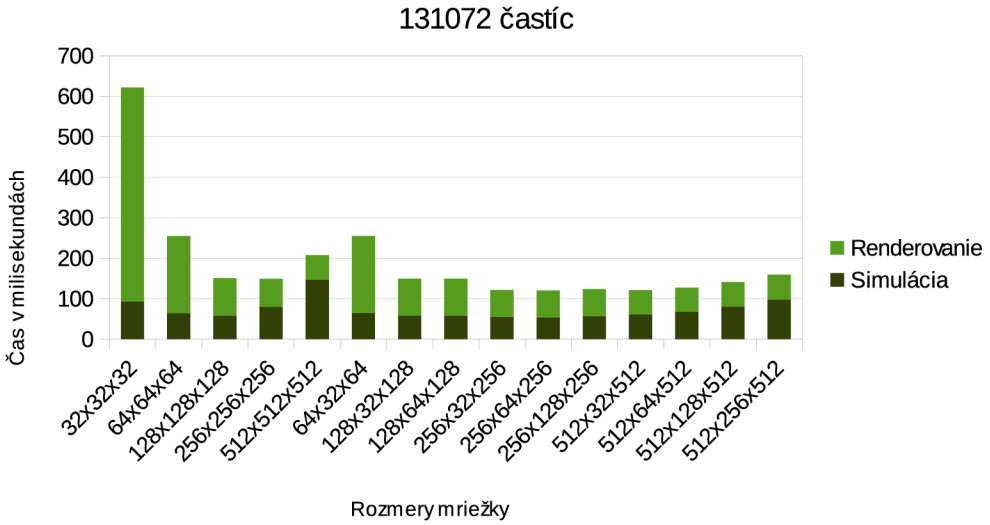
6.2 Vylepšenia do budúcnosti

Prácu je možné do budúcna rozšíriť a vylepšiť mnohými ďalšími smermi a pridať vlastnosti, ktoré by svojim rozsahom vydali na niekoľko ďalších diplomových prác.

K prvej skupine vylepšení patria nápady týkajúce sa optimalizácie výkonu. Merania a profilácia nástrojom AMD CodeXL ukázali, že program má ešte dostatok priestoru na



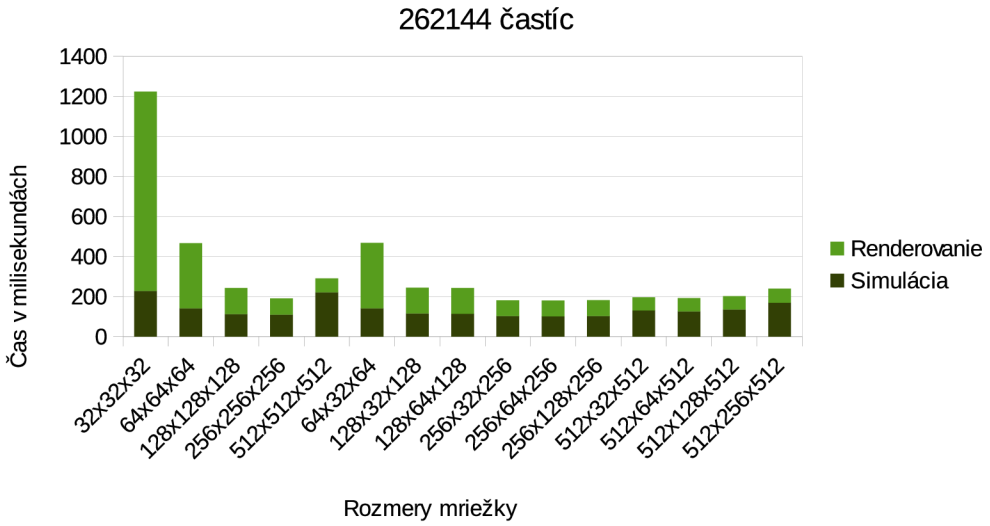
Obrázek 6.3: Výsledky testovania s 524288 časticami na konfigurácii 1



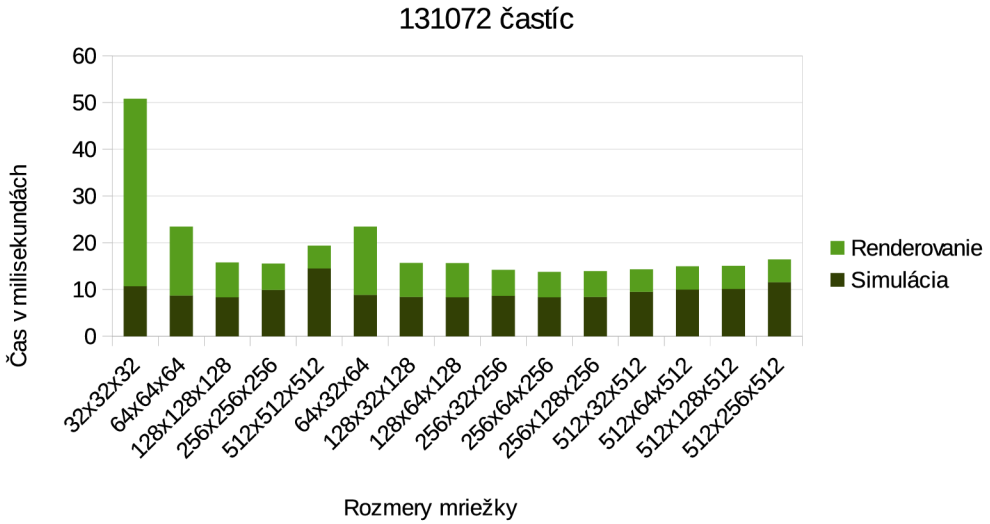
Obrázek 6.4: Výsledky testovania s 1048576 časticami na konfigurácii 1

zlepšenie výkonu. Jedným z limitujúcich faktorov je momentálne množstvo registrov spotrebovaných v kerneli na výpočet síl pôsobiacich medzi časticami, čo má za následok nedostatočnú utilizáciu jednotlivých výpočtových jadier grafického procesoru. Pokiaľ by sa nepodarilo preskladaním použitých jazykových konštrukcií prinútiť kompilátor k vygenerovaniu efektívnejšieho kódu, tak jedným zo spôsobov ako tento problém riešiť by mohlo byť využitie lokálnej pamäte. V súčasnosti program v rámci kernelu na výpočet tlaku a hustoty a ani v rámci kernelu na výpočet síl pôsobiacich medzi časticami nevyužíva žiadnu lokálnu pamäť, ktorá by sa tak dala použiť na presun niektorých premenných z registrov.

Okrem toho by stálo za zváženie aj celkovo lepšie využitie lokálnej pamäte, pretože v rámci spomínaných kernelov na výpočet tlaku a vzájomných síl je dostatok priestorovej



Obrázek 6.5: Výsledky testovania s 2097152 časticami na konfigurácii 1

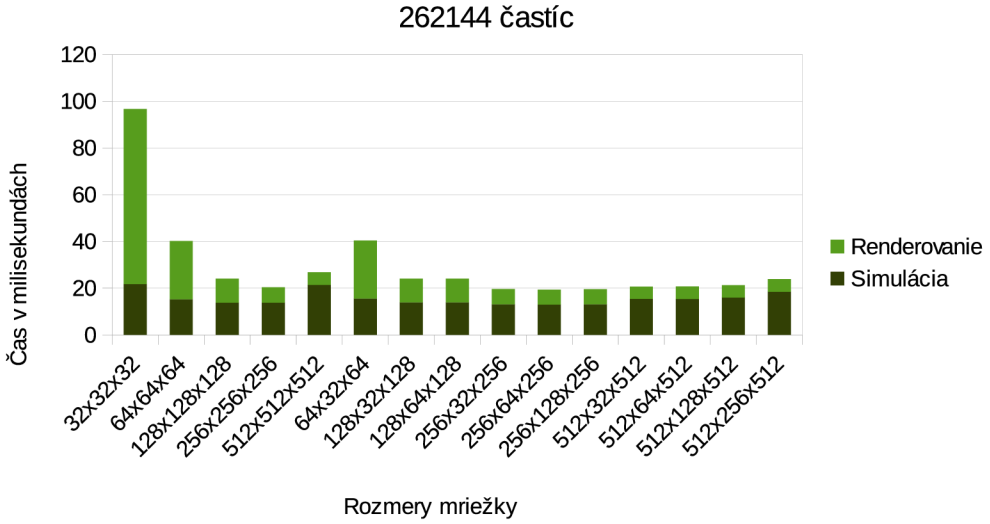


Obrázek 6.6: Výsledky testovania s 131072 časticami na konfigurácii 2

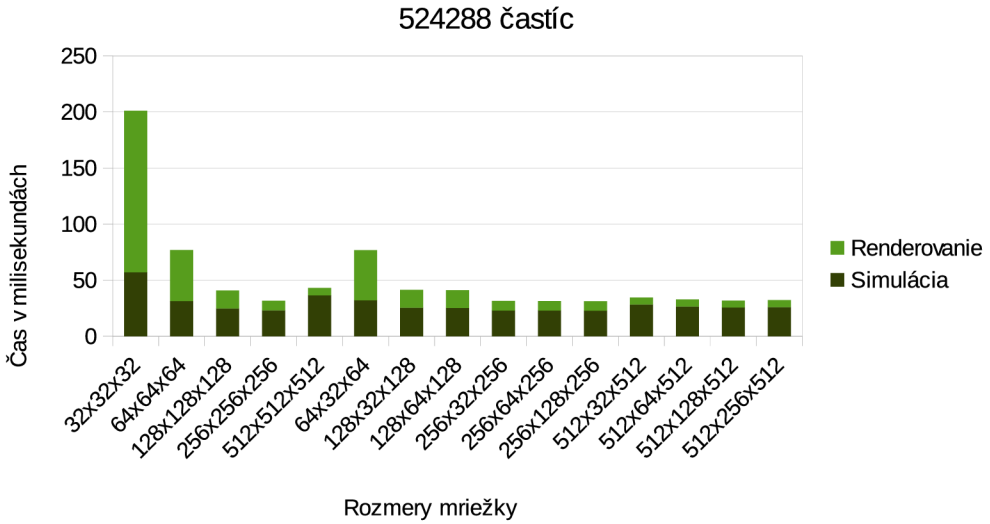
lokality. Toto však nebude jednoduchý problém, pretože množstvo susedných častíc a aj množstvo častíc v rámci jednotlivých buniek uniformnej mriežky sa môže prakticky ľubovoľne meniť.

Ďalším spôsobom ako optimalizovať pamäťové prístupy je zoradenie častíc do uniformnej mriežky podľa z-krivky, alebo podľa hilbertovej krivky. Takáto zmena by v priemere výrazne zlepšila lokalitu susedných častíc v pamäti a nemusela by byť ani veľmi náročná na výpočet. Jeden zo spôsobov ako ju možno realizovať je popísaný aj v článku [13].

Simulácia by sa dala taktiež značne urýchliť zapojením niekoľkých grafických kariet do simultánneho výpočtu. Je síce, že pravda, že mnoho ľudí dnes nemá doma počítač s dvoma, alebo troma výkonnými hráčskymi kartami, aby tak mohli využiť dodatočný



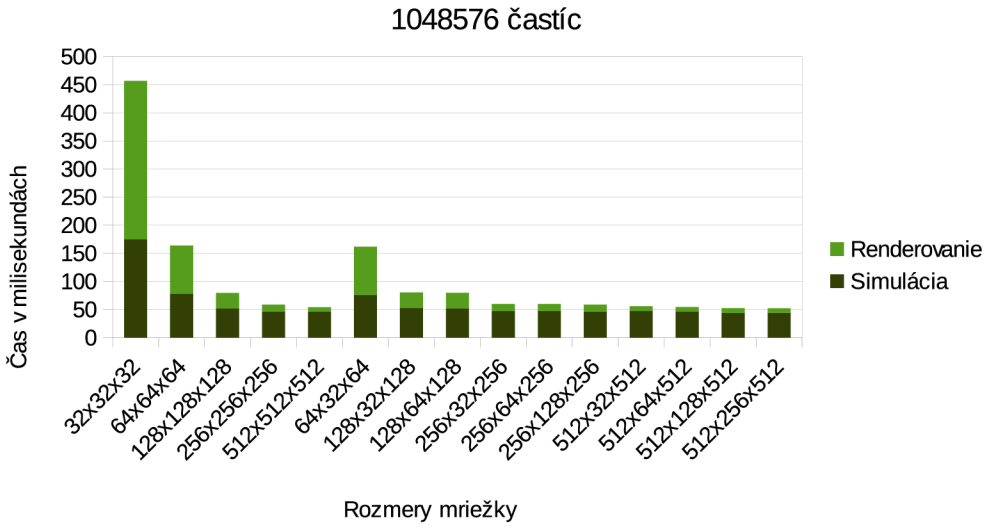
Obrázek 6.7: Výsledky testovania s 262144 časticami na konfigurácii 2



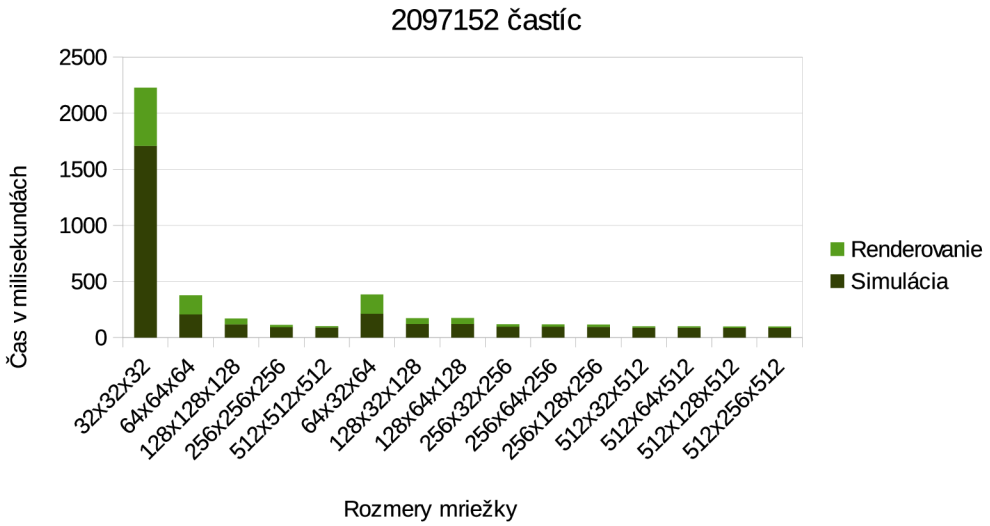
Obrázek 6.8: Výsledky testovania s 524288 časticami na konfigurácii 2

potenciál takéhoto riešenia, ale stále viac a viac ľudí má v dnešnej dobe popri dedikovanej karte aj procesor s integrovaným grafickým čipom. Tie sú čím ďalej výkonnejšie a novšie procesory s architektúrou Ivy Bridge, alebo APU procesory od firmy AMD umožňujú využiť ich dodatočný výpočtový potenciál skrz OpenCL. Tieto riešenia sa tak pomaly stávajú čoraz populárnejšie aj medzi hráčmi s obmedzeným rozpočtom a navyše v prípade APU čipov je dokonca možné využiť aj zapojenie do režimu spolupráce s výkonnejšou dedikovanou kartou nazývaného CrossFire. Z tohto pohľadu je tak škoda nevyužiť dodatočný výpočtový potenciál týchto zariadení, ktoré budú do každého novšieho počítača nainštalované tak, či tak. Takéto multi-gpu riešenie by sa mohlo inšpirovať napríklad článkom [19].

Súčasná implementácia využíva k radeniu častíc simulácie funkciu `sort_by_key` z kniž-



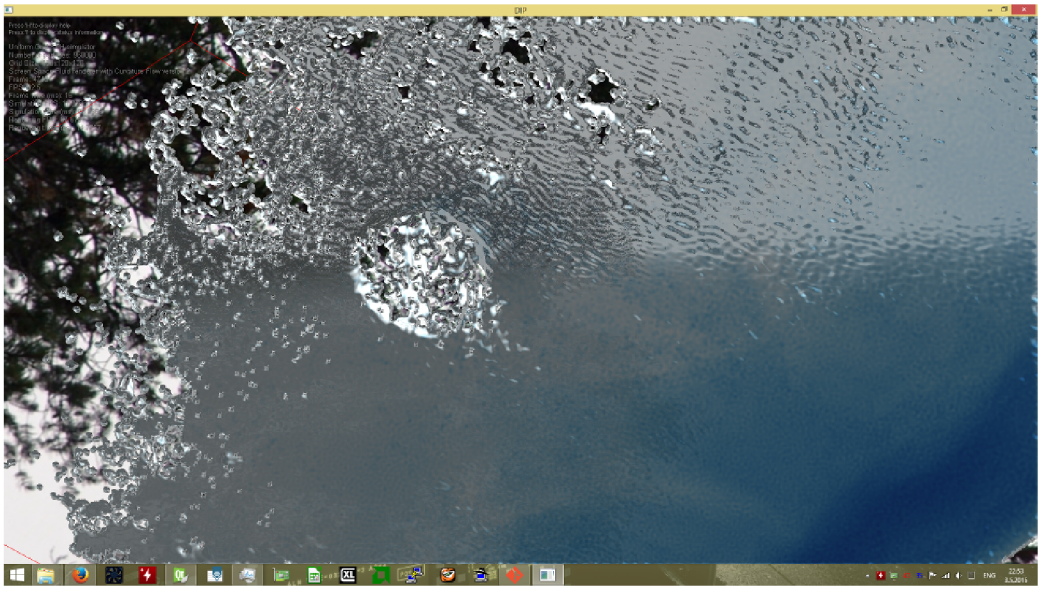
Obrázek 6.9: Výsledky testovania s 1048576 časticami na konfigurácii 2



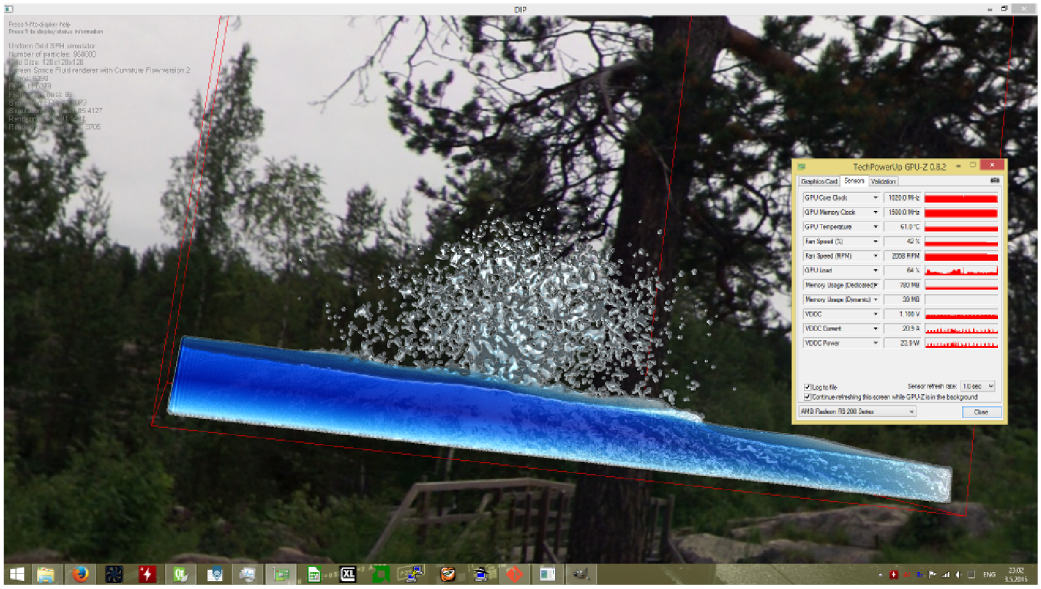
Obrázek 6.10: Výsledky testovania s 2097152 časticami na konfigurácii 2

nice `boost::compute`. V drvivej väčšine prípadov sú však efektívne rozmery uniformnej mriežky menšie alebo rovné ako 256^3 Z tohoto pohľadu je preto zbytočné používať pri radix sorte 32-bitové kľúče, stačilo by radiť podľa 24, prípadne iba 16-bitových kľúčov, čím by sa ušetrila jedna, alebo dve zbytočné iterácie radix sortu. Toto vylepšenie by vyžadovalo buď zmenu v knižnici `boost::compute`, ktorá v čase realizácie tejto práce nedovoľuje radiť 32-bitové čísla podľa 24-bitového, alebo menšieho kľúča. Druhou možnosťou by bolo napísať vlastnú paralelnú implementáciu radix sortu, čo by umožnilo aplikovať optimalizácie špecifické pre problém simulácie tekutín a dosiahnuť tak snáď ešte lepších výsledkov.

Zaujímavá je aj práca R. C. Hoetzleina, ktorý k radeniu častíc využíva counting sort. Ten funguje na podobnom princípe ako radix sort, ale s tým rozdielom, že histogram nebuduje na



Obrázek 6.11: Výsledky testovania s 262144 časticami na konfigurácii 2



Obrázek 6.12: Výsledky testovania s 262144 časticami na konfigurácii 2

jednotlivými d-bitovými číslicami, ale priamo nad celými číslami. To síce vyžaduje použitie atomických inštrukcií, ale na druhej strane sa výrazne zníži počet volaní kernelov na snímok. Autor tejto práce uvádza 5 až 10 násobné zrýchlenie oproti metóde s radix sortom [16].

Druhá skupina vylepšení sa týka kvality samotného simulačného algoritmu a grafického výstupu vizualizácie.

V súčasnosti aplikácia implementuje len detekciu kolízií častíc tekutiny s ľubovoľne natočenými rovinami v priestore, ale nepodporuje žiadne sofistikovanejšie interakcie s jej okolím. Bolo by zaujímavé pridať kolízie aj so zložitejšími polygonálnymi objektami a umožniť tak modelovať tok tekutiny v rámci rozmanitejšieho virtuálneho sveta a zlepšiť užívateľský zážitok zo simulácie. Taktiež by bolo veľmi zaujímavé pridať interakciu s objektami plávaj-

úcimi na hladine podobne ako v deme k technológii FleX od spoločnosti NVidia, prípadne prepojiť simuláciu tekutiny s ďalšími časticovými systémami ako je napríklad simulácia látky, alebo so simuláciou rigid body objektov.

Jedným z efektov, ktorý v súčasnej vizualizácii chýba je vykresľovanie kaustík a tieňov. Tento efekty by mohli hodne pridať na vizuálnej vernosti výstupu a opäť výrazne zlepšiť užívateľský zážitok. Prípadná implementácia by sa mohla inšpirovať napríklad prezentáciou Simona Greena Screen Space Fluid Rendering for Games [15]. Za zmienku určite stojí aj metóda Perspective Grid Raycasting [12], ktorá podľa jej autorov poskytuje pomerne kvalitné výstupy pri súčasnom zachovaní excelentného výpočtového výkonu.

Kapitola 7

Záver

Cieľom práce bolo pomocou dostupných technológií pre programovanie na GPU akcelerovať simuláciu tekutín so zameraním na vytvorenie riešenia, ktoré by bolo schopné pracovať interaktívne v reálnom čase.

V práci sa podarilo splniť všetky hlavné body zadania počnúc naštudovaním základných princípov až po konečnú implementáciu vybraného algoritmu. Tým bola simulačná metóda Smoothed Particle Hydrodynamics optimalizovaná použitím uniformnej mriežky spolu s vizualizáciou metódami Point Sprite a Screen Space Fluid Rendering with Curvature Flow. K urýchleniu výpočtov boli využité primárne otvorené štandardy OpenCL a OpenGL, čo vytvorenému programu zaručuje kompatibilitu naprieč rôznymi operačnými systémami i hardvérovými zariadeniami.

Posledná časť práce bola venovaná testovaniu a zhodnoteniu výsledkov implementácie, pričom boli stanovené optimálne parametre uniformnej mriežky a overená vhodnosť implementácie jak na výkonnom hráčskom počítači, tak aj na pomalom 5 rokov starom laptope.

Výsledný program je schopný na dostatočne výkonnom a modernom stroji interaktívne v reálnom čase simulovať až milión častíc, čo už je dostatočné množstvo na vytvorenie pomerne presvedčivej simulácie vody, alebo ohňa v počítačovej hre. Kód bol zároveň písaný s ohľadom na čo najlepšiu znovu použiteľnosť a kompatibilitu s ostatnými nástrojmi a platformami.

Práca mi ukázala nové zaujímavé možnosti využitia moderného grafického hardvéru a umožnila mi zdokonaliť sa jak v jeho programovaní, tak i lepšie pochopiť niektoré algoritmy a matematické postupy.

Množstvo nápadov k budúcim vylepšeniam práce bolo popísaných aj v samostatnej sekcii v rámci kapitoly o testovaní a vyhodnotení výsledkov. Tu boli rozdelené do dvoch skupín, na tie ktoré sa týkajú optimalizácie a tie, ktoré sa týkajú rozšírenia funkcionality. Za všetky tak stačí spomenúť napríklad pridanie podpory pre multi-gpu výpočty, alebo integráciu s ďalšími časticovými simuláciami, či implementáciu zložitejších detekcií kolízií kvapaliny s jej okolím.

Literatura

[1] Compute Shader. OpenGL Wiki [online]. 12.1.2015.
URL https://www.opengl.org/wiki/Compute_Shader

[2] Fast Fluid Dynamics Simulation on the GPU. *NVidia Developer Zone*, 2007.
URL http://http.developer.nvidia.com/GPUGems/gpugems_ch38.html

[3] The OpenACC ™ Application Programming Interface [online]. 2013.
URL http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf

[4] Compute Shader Overview. MICROSOFT. Microsoft Developer Network [online]. 2013 [cit. 2013-05-10].
URL http://msdn.microsoft.com/en-us/library/ff476331%28v=VS.85%29.aspx

[5] CUDA. NVIDIA. NVidia [online]. 2013 [cit. 2013-05-10].
URL http://www.nvidia.com/object/cuda_home_new.html

[6] CUDA C Programming Guide. © 2007-2014.
URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[7] C++ AMP (C++ Accelerated Massive Parallelism). Microsoft Developer Network [online]. © 2015.
URL http://msdn.microsoft.com/en-us/library/hh265137.aspx

[8] PhysX Source on GitHub. © 2015.
URL https://developer.nvidia.com/physx-source-github

[9] AnandTech: GeForce GTX 980 Block Diagram. 2015.
URL http://images.anandtech.com/doci/8526/GeForce_GTX_980_Block_Diagram_FINAL.png

[10] Crane, K.; Llamas, I.; Tariq, S.: Chapter 30. Real-Time Simulation and Rendering of 3D Fluids. In *GPU gems 3*, Upper Saddle River: Addison-Wesley, 2008, ISBN 978-0-321-51526-1.
URL http://http.developer.nvidia.com/GPUGems3/gpugems3_ch30.html

[11] Engel, W.: Memory banks. 2015.
URL http://diaryofagraphicsprogrammer.blogspot.cz/2014/03/compute-shader-optimizations-for-amd.html

[12] Fraedrich, R.; Auer, S.; Westermann, R.: Efficient high-quality volume rendering of SPH data. *Visualization and Computer Graphics, IEEE Transactions on*, ročník 16, č. 6, 2010: s. 1533–1540.

URL http://www.in.tum.de/fileadmin/user_upload/Lehrstuehle/Lehrstuhl_XV/Research/Publications/2010/Vis10EfficientSPHRendering.pdf

[13] Goswami, P.; Schlegel, P.; Solenthaler, B.; aj.: Interactive SPH simulation and rendering on the GPU. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Eurographics Association, 2010, s. 55–64.
URL https://graphics.ethz.ch/~sobarbar/papers/Sol10b/Sol10b.pdf

[14] Green, S.: Particle simulation using cuda. *NVIDIA whitepaper*, 2010.
URL
http://www.ecse.rpi.edu/homepages/wrf/wiki/ParallelComputingSpring2014/cuda-samples/samples/5_Simulations/particles/doc/particles.pdf

[15] Green, S.: Screen space fluid rendering for games. In *Proceedings for the Game Developers Conference*, 2010.
URL http://developer.download.nvidia.com/presentations/2010/gdc/Direct3D_Effects.pdf

[16] Hoetzlein, R. C.: Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids. 2014.
URL http://on-demand.gputechconf.com/gtc/2014/presentations/S4117-fast-fixed-radius-nearest-neighbor-gpu.pdf

[17] Horváth, Z.: ČÁSTICOVÉ SIMULACE V REÁLNÉM ČASE. 2012.

[18] Ikits, M.; Kniss, J.; Lefohn, A.; aj.: Chapter 39. Volume Rendering Techniques. In *GPU gems*, Boston: Addison-Wesley, vyd. 1. vydání, 2004, ISBN 0-321-22832-4.
URL http://http.developer.nvidia.com/GPUGems/gpugems_ch39.html

[19] Junior, J. R. d. S.; Joselli, M.; Zamith, M.; aj.: An architecture for real time fluid simulation using multiple GPUs. *SBC-Proceedings of SBGames*, 2012.
URL http://sbgames.org/sbgames2012/proceedings/papers/computacao/comp-full_12.pdf

[20] Khronos OpenCL Working Group and Aaftab Munshi: The OpenCL Specification Version: 1.1 Document Revision: 44 [online]. 2011 [cit. 2013-05-07].
URL http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[21] van der Laan, W. J.; Green, S.; Sainz, M.: Screen space fluid rendering with curvature flow. *Proceedings of the 2009 symposium on Interactive 3D graphics and games - I3D '09*, 2009.
URL http://www.cs.rug.nl/~roe/publications/fluidcurvature.pdf

[22] Scarpino, M.: *OpenCL in Action*. Manning Publications Co., 2012, ISBN 9781617290176.

[23] Shrout, R.: AMD Planning Open Source GameWorks Competitor, Mantle for Linux. June 19, 2014.
URL http://www.pcper.com/news/Graphics-Cards/AMD-Planning-Open-Source-GameWorks-Competitor-Mantle-Linux

[24] Stam, J.: Stable Fluids. *SIGGRAPH 99 Conference Proceedings*, 1999.
URL http://www.autodeskresearch.com/pdf/ns.pdf

[25] Stam, J.: Real-time fluid dynamics for games. In *Proceedings of the game developer conference*, ročník 18, 2003, str. 25.
URL http://www.intpowertechcorp.com/GDC03.pdf

[26] Voreen: Raycasting. 2015.
URL http://www.voreen.org/129-Ray-Casting.html

[27] Zhang, D.: GPU (CUDA) based Adaptive SPH Fluid Simulation. 2015.
URL http://www3.cs.stonybrook.edu/~dozhang/fluid/adaptive.jpg

Příloha A

Tabuľky s výsledkami meraní

Tabulka A.1: 16384 častíc - konfigurácia 1

Rozmery mriežky	Simulácia	Renderovanie	Celkový čas	Fps simulácie	Fps renderovania	Celkové Fps
32x32x32	13,255	116,000	129,255	75,442	8,620	7,736
64x64x64	11,656	72,602	84,259	85,788	13,773	11,868
128x128x128	13,116	59,498	72,614	76,242	16,807	13,771
256x256x256	20,735	55,231	75,967	48,227	18,105	13,163
512x512x512	94,826	54,220	149,046	10,545	18,443	6,709
64x32x64	11,419	71,365	82,784	87,572	14,012	12,079
128x32x128	11,715	58,429	70,144	85,357	17,114	14,256
128x64x128	12,668	59,449	72,117	78,934	16,821	13,866
256x32x256	12,763	55,215	67,978	78,348	18,111	14,710
256x64x256	13,869	55,182	69,051	72,101	18,121	14,481
256x128x256	16,365	55,489	71,855	61,103	18,021	13,916
512x32x512	15,784	54,112	69,897	63,352	18,479	14,306
512x64x512	20,440	54,072	74,513	48,921	18,493	13,420
512x128x512	30,068	54,149	84,217	33,257	18,467	11,874
512x256x512	47,979	54,111	102,091	20,842	18,480	9,795

Tabulka A.2: 32768 častíc - konfigurácia 1

Rozmery mriežky	Simulácia	Renderovanie	Celkový čas	Fps simulácie	Fps renderovania	Celkové Fps
32x32x32	22,296	175,541	197,837	44,849	5,696	5,054
64x64x64	20,982	89,330	110,312	47,659	11,194	9,065
128x128x128	21,859	65,145	87,004	45,747	15,350	11,493
256x256x256	28,194	57,191	85,385	35,468	17,485	11,711
512x512x512	102,718	55,401	158,119	9,735	18,049	6,324
64x32x64	19,762	88,315	108,078	50,600	11,323	9,252
128x32x128	17,728	63,055	80,783	56,406	15,859	12,378
128x64x128	19,549	63,728	83,277	51,151	15,691	12,008
256x32x256	19,829	56,512	76,341	50,430	17,695	13,098
256x64x256	21,103	57,155	78,258	47,385	17,496	12,778
256x128x256	23,303	57,152	80,456	42,912	17,496	12,429
512x32x512	24,403	55,196	79,599	40,977	18,117	12,562
512x64x512	27,298	55,048	82,347	36,631	18,165	12,143
512x128x512	36,374	55,136	91,511	27,491	18,136	10,927
512x256x512	54,823	55,138	109,962	18,240	18,136	9,094

Tabulka A.3: 65536 častíc - konfigurácia 1

Rozmery mriežky	Simulácia	Renderovanie	Celkový čas	Fps simulácie	Fps renderovania	Celkové Fps
32x32x32	42,797	294,568	337,365	23,365	3,394	2,964
64x64x64	35,794	123,835	159,629	27,937	8,075	6,264
128x128x128	32,126	74,368	106,495	31,126	13,446	9,390
256x256x256	43,983	61,097	105,081	22,735	16,367	9,516
512x512x512	116,923	57,130	174,053	8,552	17,503	5,745
64x32x64	36,119	122,163	158,282	27,686	8,185	6,317
128x32x128	30,897	72,466	103,363	32,365	13,799	9,674
128x64x128	30,742	72,906	103,649	32,527	13,716	9,647
256x32x256	31,982	59,460	91,442	31,267	16,817	10,935
256x64x256	33,562	59,823	93,385	29,795	16,715	10,708
256x128x256	39,900	60,975	100,876	25,062	16,400	9,913
512x32x512	36,806	56,298	93,104	27,169	17,762	10,740
512x64x512	41,627	56,956	98,583	24,022	17,557	10,143
512x128x512	51,105	57,041	108,147	19,567	17,531	9,246
512x256x512	69,289	57,031	126,321	14,432	17,534	7,916

Tabulka A.4: 131072 častíc - konfigurácia 1

Rozmery mriežky	Simulácia	Renderovanie	Celkový čas	Fps simulácie	Fps renderovania	Celkové Fps
32x32x32	92,814	528,584	621,398	10,774	1,891	1,609
64x64x64	63,997	190,781	254,778	15,625	5,241	3,924
128x128x128	57,941	92,909	150,850	17,258	10,763	6,629
256x256x256	79,974	69,043	149,017	12,504	14,483	6,710
512x512x512	146,707	61,143	207,850	6,816	16,354	4,811
64x32x64	65,069	190,084	255,153	15,368	5,260	3,919
128x32x128	58,218	91,225	149,444	17,176	10,961	6,691
128x64x128	57,840	91,820	149,660	17,288	10,890	6,681
256x32x256	55,865	65,639	121,505	17,900	15,234	8,230
256x64x256	54,117	66,001	120,118	18,478	15,151	8,325
256x128x256	56,835	66,936	123,772	17,594	14,939	8,079
512x32x512	61,791	59,362	121,153	16,183	16,845	8,253
512x64x512	67,835	59,683	127,519	14,741	16,754	7,841
512x128x512	80,195	61,250	141,445	12,469	16,326	7,069
512x256x512	97,968	61,318	159,287	10,207	16,308	6,277

Tabulka A.5: 262144 častíc - konfigurácia 1

Rozmery mriežky	Simulácia	Renderovanie	Celkový čas	Fps simulácie	Fps renderovania	Celkové Fps
32x32x32	228,380	995,879	1224,25	4,378	1,004	0,816
64x64x64	141,380	325,376	466,756	7,073	3,073	2,142
128x128x128	112,563	130,012	242,575	8,883	7,691	4,122
256x256x256	109,960	80,901	190,861	9,094	12,360	5,239
512x512x512	221,038	69,831	290,869	4,524	14,320	3,437
64x32x64	141,579	326,871	468,450	7,063	3,059	2,134
128x32x128	115,794	128,669	244,463	8,636	7,771	4,090
128x64x128	114,206	128,960	243,166	8,756	7,754	4,112
256x32x256	103,260	78,441	181,701	9,684	12,748	5,503
256x64x256	101,763	78,494	180,257	9,826	12,739	5,547
256x128x256	102,979	79,372	182,351	9,710	12,598	5,483
512x32x512	130,936	65,354	196,290	7,637	15,301	5,094
512x64x512	126,566	65,729	192,295	7,901	15,213	5,200
512x128x512	135,415	66,472	201,887	7,384	15,043	4,953
512x256x512	169,283	70,028	239,311	5,907	14,279	4,178

Tabulka A.6: 131072 častíc - konfigurácia 2

Rozmery mriežky	Simulácia	Renderovanie	Celkový čas	Fps simulácie	Fps renderovania	Celkové Fps
32x32x32	10,683	40,140	50,824	93,605	24,913	19,676
64x64x64	8,697	14,742	23,439	114,984	67,832	42,664
128x128x128	8,339	7,435	15,774	119,912	134,506	63,395
256x256x256	9,911	5,608	15,518	100,899	178,332	64,439
512x512x512	14,502	4,881	19,383	68,955	204,882	51,591
64x32x64	8,856	14,589	23,445	112,918	68,543	42,652
128x32x128	8,395	7,274	15,669	119,119	137,469	63,819
128x64x128	8,337	7,302	15,640	119,941	136,941	63,939
256x32x256	8,647	5,533	14,180	115,650	180,723	70,521
256x64x256	8,320	5,437	13,757	120,196	183,929	72,692
256x128x256	8,427	5,497	13,924	118,670	181,914	71,819
512x32x512	9,492	4,817	14,309	105,356	207,592	69,887
512x64x512	9,971	4,967	14,938	100,287	201,327	66,941
512x128x512	10,122	4,932	15,054	98,796	202,764	66,429
512x256x512	11,559	4,893	16,452	86,510	204,375	60,782

Tabulka A.7: 131072 častíc - konfigurácia 2

Rozmery mriežky	Simulácia	Renderovanie	Celkový čas	Fps simulácie	Fps renderovania	Celkové Fps
32x32x32	21,717	74,924	96,641	46,046	13,346	10,347
64x64x64	15,244	24,914	40,159	65,596	40,137	24,900
128x128x128	13,778	10,275	24,054	72,576	97,316	41,572
256x256x256	13,735	6,586	20,322	72,801	151,827	49,206
512x512x512	21,388	5,412	26,800	46,755	184,755	37,312
64x32x64	15,532	24,847	40,380	64,379	40,245	24,764
128x32x128	13,949	10,116	24,065	71,688	98,848	41,552
128x64x128	13,899	10,124	24,023	71,946	98,775	41,626
256x32x256	13,051	6,535	19,587	76,619	153,002	51,053
256x64x256	12,913	6,440	19,353	77,440	155,278	51,671
256x128x256	13,065	6,476	19,542	76,536	154,409	51,171
512x32x512	15,432	5,219	20,652	64,798	191,571	48,420
512x64x512	15,327	5,374	20,701	65,243	186,066	48,305
512x128x512	15,954	5,306	21,261	62,677	188,440	47,033
512x256x512	18,440	5,366	23,806	54,229	186,356	42,006

Tabulka A.8: 524288 častíc - konfigurácia 2

Rozmery mriežky	Simulácia	Renderovanie	Celkový čas	Fps simulácie	Fps renderovania	Celkové Fps
32x32x32	56,932	144,152	201,084	17,564	6,937	4,973
64x64x64	31,363	45,523	76,886	31,884	21,966	13,006
128x128x128	24,777	16,000	40,777	40,360	62,498	24,523
256x256x256	23,039	8,524	31,563	43,404	117,306	31,681
512x512x512	36,453	6,523	42,976	27,432	153,296	23,268
64x32x64	31,964	44,704	76,668	31,284	22,369	13,043
128x32x128	25,472	15,886	41,358	39,257	62,948	24,178
128x64x128	25,195	15,866	41,062	39,689	63,026	24,353
256x32x256	22,902	8,556	31,458	43,663	116,873	31,787
256x64x256	22,877	8,428	31,305	43,711	118,640	31,942
256x128x256	22,760	8,471	31,232	43,935	118,039	32,017
512x32x512	28,349	6,165	34,514	35,273	162,206	28,973
512x64x512	26,524	6,289	32,814	37,700	158,992	30,474
512x128x512	25,560	6,217	31,777	39,123	160,844	31,469
512x256x512	25,962	6,278	32,241	38,516	159,275	31,016

Tabulka A.9: 1048576 častíc - konfigurácia 2

Rozmery mriežky	Simulácia	Renderovanie	Celkový čas	Fps simulácie	Fps renderovania	Celkové Fps
32x32x32	174,855	281,758	456,613	5,719	3,549	2,190
64x64x64	78,203	85,489	163,692	12,787	11,697	6,109
128x128x128	51,827	27,522	79,349	19,294	36,334	12,602
256x256x256	46,178	12,429	58,607	21,655	80,452	17,062
512x512x512	45,998	8,094	54,092	21,740	123,539	18,486
64x32x64	75,989	85,871	161,861	13,159	11,645	6,178
128x32x128	52,571	27,818	80,389	19,021	35,9479	12,439
128x64x128	52,204	27,515	79,720	19,155	36,343	12,543
256x32x256	47,155	12,497	59,653	21,206	80,013	16,763
256x64x256	47,311	12,344	59,655	21,136	81,005	16,762
256x128x256	46,406	12,334	58,741	21,548	81,070	17,023
512x32x512	47,961	7,986	55,947	20,850	125,211	17,873
512x64x512	46,197	8,096	54,293	21,646	123,517	18,418
512x128x512	44,338	7,961	52,299	22,553	125,609	19,120
512x256x512	44,199	8,002	52,201	22,624	124,960	19,156

Tabulka A.10: 2097152 častíc - konfigurácia 2

Rozmery mriežky	Simulácia	Renderovanie	Celkový čas	Fps simulácie	Fps renderovania	Celkové Fps
32x32x32	1709,080	517,707	2226,78	0,585	1,931	0,449
64x64x64	206,780	169,533	376,313	4,836	5,898	2,657
128x128x128	117,371	51,893	169,264	8,519	19,270	5,907
256x256x256	92,378	20,186	112,565	10,825	49,537	8,883
512x512x512	88,744	11,558	100,303	11,268	86,517	9,969
64x32x64	212,370	169,853	382,223	4,708	5,887	2,616
128x32x128	119,783	51,735	171,518	8,348	19,329	5,830
128x64x128	120,645	52,692	173,337	8,288	18,977	5,769
256x32x256	96,317	20,607	116,924	10,382	48,526	8,552
256x64x256	96,223	20,331	116,554	10,392	49,185	8,579
256x128x256	94,681	20,160	114,842	10,561	49,600	8,707
512x32x512	87,121	11,582	98,703	11,478	86,337	10,131
512x64x512	87,030	11,640	98,670	11,490	85,909	10,134
512x128x512	86,239	11,492	97,731	11,595	87,015	10,232
512x256x512	86,218	11,477	97,695	11,598	87,130	10,235