VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

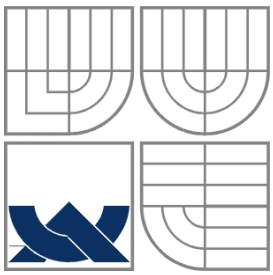
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SIMULACE TEKUTIN A PLYNŮ

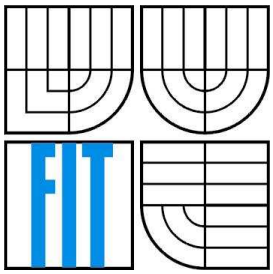
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JAN ZIVČÁK
AUTHOR

BRNO 2008



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 A PLYNŮ

FLUID AND GAS SIMULATION

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

JAN ZIVČÁK

VEDOUCÍ PRÁCE
SUPERVISOR

ING. RADOVAN JOŠTH

BRNO 2008

Abstrakt

Tato bakalářská práce zpracovává tématiku simulace tekutin a plynů na osobních počítačích. Práce porovnává různé přístupy s ohledem na proveditelnost simulace v realném čase. Pozornost je také věnována metodam pro zobrazovaní tekutiny - implicitním plochám a metodě marching cubes. Navíc se práce zaměřuje na moderní grafické adaptéry s ohledem na jejich využití při výpočtech simulace. Obzvláště se bude věnovat pozornost technoligii CUDA od společnosti NVIDIA. Vše je navíc doplněno popisem mé implementace simulace tekutin a plynů.

Klíčová slova

simulace, simulace tekutin, SPH, marching cubes, CUDA

Abstract

This bachelor´s thesis is focused on theme of fluid and gas simulation performed on personal computers. This branch of computer graphics and computer simulation is quite popular because it offers a lot of questions and a lot of solutions. Thesis is about to compare this solutions, focusing on those which can be real-time computed. One part of thesis is also about methods for fluid vizualization - isosurfaces and method marching cubes. The question of using modern graphics adapters to compute the simulation is also present. Especially NVIDIA CUDA technology will be analysed. And finally, there is an explanation of my implementation.

Keywords

simulation, fluid simulation, SPH, marching cubes, CUDA

Citace

Zivčák Jan: Simulace tekutin a plynů. Brno, 2008, bakalářská práce, FIT VUT v Brně.

Simulace tekutin a plynů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Radovana Joštha. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jan Zivčák
14. května 2008

Poděkování

Chtěl bych poděkovat svému vedoucímu za vypsání tohoto zajímavého zadání.

Obsah

Úvod

S tekutinami jsou lidé v kontaktu neustále. Proto každý člověk ví, jak se tekutina chová. Zná a dokáže popsat různé jevy, které v souvislosti s tekutinami vznikají. Bohužel fyzikální popis těchto jevů je obtížný. Tedy, přesněji, jeho aplikování není ve stoprocentním rozsahu možné.

Základní rovnice pro popis chování tekutin jsou Navier-Stokesovy rovnice (1.1), jejichž analytické řešení zatím není známé. Proto, pokud chceme simulovat chování tekutin, je třeba využít numerických řešení - tolik populárních v dnešní počítačové době. Lze využít jeden ze dvou přístupů: Eulerovu metodu (1.2.1) - ta je nejčastěji založena na pravidelné mřížce, nebo Lagrengeovu metodu(1.2.2) - ta je založena na částicích a jejich interakcích. Každá z těchto metod má své výhody a své nevýhody a každá je tedy vhodná v jiných případech.

Do nedávna se díky nedostatku výpočetního výkonu objevovaly spíše implementace Eulerovy metody. Ta je díky svým vlastnostem nevhodná pro real-time simulace. Na druhou stranu dokáže podat velice přesné výsledky, což může být důležité. Pro testování různých aerodynamický a hydrodynamických vlastností - například v leteckém či lodním průmyslu - je určitě přednější přenost simulace, než její rychlost. Pro filmový průmysl je asi také důležitá věrohodost simulace, aby oko diváka nepoznalo rozdíly oproti skutečnosti.

S postupně se zvětšujícím výkonem počítačů nastala možnost simulovat tekutiny real-time, pomocí částic - tedy pomocí Lagrangeovy metody. Tato metoda sice nedokáže podat přesnější výsledky než Eulerova metoda, ale její hlavní výhodou je rychlost. V dnešní době existuje spousta real-time simulátorů založených na této metodě. Jeden takový je i výsledkem mé práce.

Výše popsané metody řeší pohyb kapaliny, ale neřeší její vizualizaci. K té je možno přistoupit více způsoby. Od obyčejného zobrazení každé částice či buňky mřížky pomocí geometrických primitiv, přes metody typu marching cubes (3.2), marching tetrahedrons, nebo nové metody screen space meshes, až po metody typu raytracing. Opět každá má různe charakteristiky. Metoda raytracing, je velice přesná a podává vynikající výslekdy, ale kvalita je vyvážena velkou časovou náročností. Zato metoda marching cubes dokáže podat relativně kvalitní výstupy za relativné krátkou dobu.

Ještě je třeba zmínit možnost výpočtu simulace na grafických apdaptérech - této problematice se bude věnovat podstatná část práce. Díky svým chrakteristikám - nad spoustou částic (dat) je potřeba vykonat jeden algoritmus (SIMD - Single Instruction Multiple Data) - je simulace tekutin vhodná pro výpočty na grafický adaptérech. Ty se dají programovat různými způsoby. Já jsem si vybral novou technologii CUDA firmy NVIDIA (5.1).

1 Matematické řešení simulace tekutin

1.1 Navier-Stokesovy rovnice

Navier-Stokesovy rovnice popisují pohyb tekutin. Byly objeveny nezávisle na sobě Claude-Louis Navierem a George Gabriel Stokesem v letech 1827 a 1845. Oba z těchto fyziků a matematiků hledali matematický popis dynamiky tekutin.

Jedná se o diferenciální rovnice. Ty se narozdíl od algebraických rovnic nevěnují přímo proměnným, ale spíše jejich změnám. V kapalině dochází k neustálým změnám stavových veličin - rychlosti, tlaku, směru, teploty, atd. A na tyto změny je třeba reagovat. Ovšem je třeba zachovat určitá pravidla - zákony hmotnosti, hybnosti, energie...

Tyto diferenciální rovnice většinou přecházejí v nelineární parciální diferenciální rovnice, jejichž řešení je velice obtížné. Jejich analytické řešení je možné jen v několika specifických případech, jejich obecné řešení je zapsáno mezi Millenium Prize Problems s odměnou 1 000 000 USD pro řešitele.

Navier-Stokesovy rovnice popisují mnoho různých typů proudění. Pro naše potřeby je ale třeba znát pouze popis proudění Newtonovské nestlačitelné kapaliny. Ta je charakteristická tím, že tečné napětí v kapalině je přímo úměrné dynamické viskozitě a změně proudění (rychlosti) kapaliny:

$$\tau = -\eta \frac{du}{dt}$$

τ je tečné napětí v tekutině

η je dynamická viskozita

u je rychlost proudění kapaliny

Pro takovou kapalinu mají Navier-Stokesovy rovnice následující tvar:

$$\rho\left(\frac{\partial v}{\partial t} + v \cdot \nabla v\right) = -\nabla p + \mu\nabla^2 + f$$

$$\nabla \cdot v = 0$$

v je rychlost (vektor), ρ je hustota kapaliny,

p je tlak kapaliny, μ je kinematická viskozita, f reprezentuje externí síly (vektor),

∇ je operátor nabla

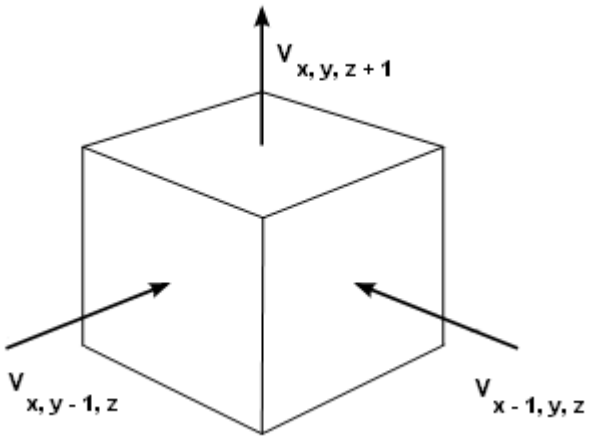
Ve zkratce lze říct, že tato rovnice popisuje zákon zachování energie. Kdy na jedné straně stojí zrychlení (místní zrychlení + konvektivní zrychlení) a toto zrychlení je v rovnováze s tlakovými, viskózními (třecími) a vnějšími silami.

1.2 Numerické řešení

Jak už bylo řečeno, analytické řešení proudění tekutin pomocí Navier-Stokesových rovnic je velice problematické, často i nemožné. Proto je potřeba přistoupit na numerické řešení. Existují v zásadě jen dvě možnosti - diskretizace prostoru (Eulerova metoda) a diskretizace kapaliny (Lagrangeova metoda).

1.2.1 Eulerova metoda

Dříve nejpoužívanější metoda pro numerické řešení simulace tekutin. Objevuje se již řadu let a díky tomu je o ní vypracováno spoustu dokumentů a prací. Je založena na dělení prostoru do mřížky. Poté je možné sledovat stav všech důležitých veličin v každé buňce mrížky s tím, že mezi buňkami existuje samozřejmě vazba. Rychlost kapaliny je nutné sledovat pro každou stěnu buňky, tedy spíše stačí sledovat stav jen pro polovinu stěn, protože každá stěna je stěnou pro dvě buňky - viz obrázek 1.1. Jak kapalina teče, je nutné sledovat přítok ze všech stěn a reagovat odtokem do příslušných stěn.



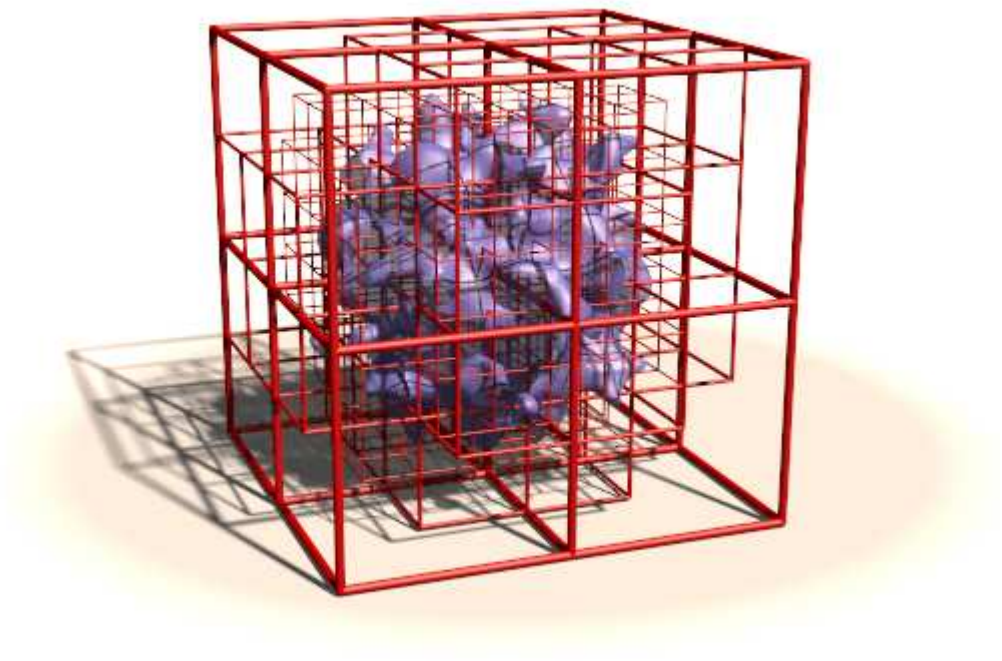
Obrázek 1.1 - Buňka mřížky

Hlavní výhodou této metody je, že se na ní velice snadno uplatňují Navier-Stokesovy zákony. Možná proto se zpočátku používala hlavně tato metoda. Další výhoda této metody je, že se jednoduše dodržuje zachovávání hustoty kapaliny v celé scéně. Za další výhodu by se dalo považovat jednoduché vyhodnocení mřížky pro metodu marching cubes. O té se dozvíme více v kapitole 3.2.

Pokud se dá říct, že se během simulace snadno dodržuje zachovávání hustoty, tak o zachovávní hmotností to už tak docela neplatí. V každé buňce totiž muže být kapalina, ale také nemusí. Nemluvě o problemách se zachováváním hmotnosti v nepravidelné mřížce. Další nevýhody této metody jsou spojené hlavně s existencí oné mřížky. Je zřejmé, že kapalina nemůže existovat mimo mřížku. Proto,

pokud chceme zobrazovat rozlehlou scénu ve velkých detailech, je nutné mít rozlehlou, hodně členěnou mřížku, což může být hodně paměťově náročné.

Nevýhodu týkající se náročnosti mřížky je možné částečně řešit. Jak už bylo řečeno, je možné provést nepravidelné dělení mřížky, a to tak, že se v zajímavých místech buňky více rozdělí. Jedna z možností je použít strukturu oktanový strom. Ta, jak je patrné z názvu, dělí každou buňku vždy na osm menších buněk. Viz následující obrázek:



Obrázek 1.2 - Dělění mřížky se strukturou oktanový strom (http://www.imagico.de)

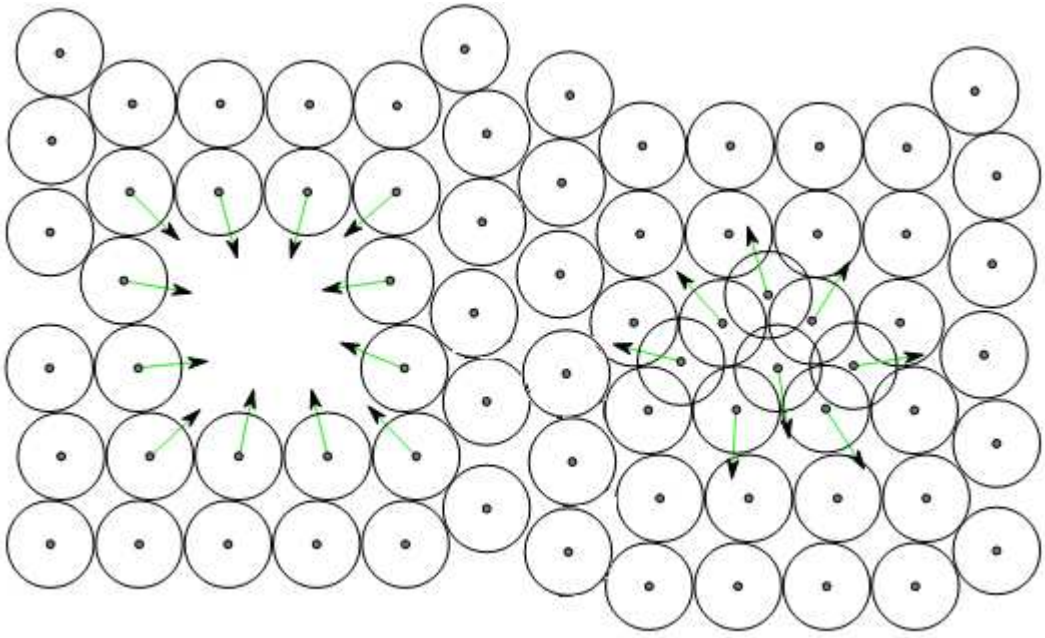
Další možné řešení patří mezi takzvaná hybridní řešení. To jsou řešení, která spojují Eulerovu metodu s Lagrangeovou metodou, popsanou v následující kapitole.

1.2.2 Lagrangeova metoda

Lagrangeova metoda narozdíl od Elerovy metody nedělí celý prostor, ale dělí pouze kapalinu na určité a stále množství částic (particles). Každá z těchto částic si o sobě nese informace. Minimálně potřebujeme znát o každé částici její polohu. Rychlost lze potom snadno vypočítat z derivace polohy. Dále je možné o každé částici zaznamenávvat například teplotu, hmotnost, atd. Částice mezi sebou typicky nebývají propojeny nějakou strukturou, ale vzájemně na sebe reagují. Síla reakce jedné částice na druhou je dána jejich vzdáleností - to je hlavní rys metody SPH popsané v kapitole 2.

Z toho, že každá částice má neměnitelnou hmotnost a že počet částic je konečný, lze usuzovat, že zachovávání hmotnosti u této metody není žádný problém.

Naopak problém se zachováním hustoty a nestlačitelnosti kapaliny je značný. Je třeba sledovat vzdálenosti všech částic a vhodně na ně reagovat. Pokud je například na jednom místě částic nadbytek, je potřeba částice od sebe oddělit. V tomto místě tedy vzniká na určitou dobu zvýšený tlak. Naopak, pokud je někde mezi částicemi díra, je třeba tuto díru zaplnit (tedy pokud neuvažujeme výskyt bublin). Toto místo by pak bylo místo podtlatku.



Obrázek 1.3 - Na levé straně je zobrazen příklad podtlaku vyvolaného nedostatkem částic. Na pravé straně je zobrazen příklad přetlaku vyvolaného nadbytkem částic.

Z výše napsaného tedy vyplívá, že metoda je vhodná pro real-time simulace. Hlavní důvod je ten, že metoda nepotřebuje žádnou mřížku. Sama od sebe, bez žádných vylepšení, se přizpůsobuje dané situaci. Je nutné ale dodat, že výsledky podané touto metodou nejsou tak kvalitní jako u Eulerovy metody. Je to nejspíš dáno tím, že se zde špatně dodržuje zachovávání hustoty a tedy i nestlačitelnosti. Následující kapitola se věnuje metodě SPH, která vychází z Lagrangeovy metody.

2 SPH

SPH je zkratka pro Smoothed Particle Hydrodynamics. Tato metoda byla vyvinuta v roce 1977 v NASA. Původně totiž sloužila pro simulaci pohybu kosmických plynů. Plyny jsou ovšem stlačitelné, takže pokud metodu aplikujeme na kapaliny, nemůžeme dosáhnout 100% nestlačitelnosti (té se ale těžko dosahuje u jakékoliv metody na Lagrangeově principu).

Jako u každé Lagrangeovy metody, dělí i tato metoda kapalinu či plyn na částice. Každá částice si nese svůj stav a každá částice určitým způsobem ovlivňuje stav okolních částic. To, zda je částice považována za okolní či nikoliv, je ovlivněno tzv. vyhlazovací vzdáleností (smoothnig lenght), obvykle označovanou jako *h*. Částice, jejichž vzdálenost je větší než *h*, se neovlivňují vůbec.

Pomocí tzv. jádra (kernel) je dána míra ovlivňění dvou okolních částic. Toto jádro může být pro každou veličinu jiné.

Pro každou veličinu, částici a její okolní částice musí platit následující rovnice:

$$A(r) = \sum_j A_j \frac{m_j}{\rho_j} W(|r - r_j|, h)$$

A je stav veličiny pro zkoumanou částici

$\sum_j$ značí sumu přes všechny okolní částice

A_j je stav veličiny částice *j*

m_j je hnotnost částice *j*

ρ_j je hustota částice *j*

$W(\ )$ je funkce jádra

r je poloha zkoumané částice

r_j je poloha částice *j*

h je vyhlazovací délka

Algoritmus se tedy musí snažit, aby předcházející rovnice platila.To znamená, že musí vhodně reagovat při nesplnění rovnice. Nejčastější reakcí je změna polohy (rychlosti).

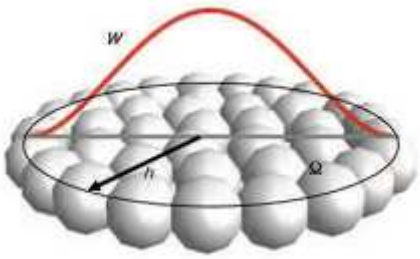
Pokud budeme uvažovat, že všechny částice mají stejnou hmotnost, je možné hmotnost z výpočtu odstranit.

Dále je nutné se zmínit o vyhlazovací délce. Tato veličina částečně koresponduje s délkou buňky u Eulerovy metody. Sice neovlivňuje paměťovou náročnost výpočtu, ale ovlivňuje časovou náročnost a ovlivňuje i výsledek samotný. Proto se objevují aplikace, kde vyhlazovací délka není konstatní, ale přizpůsobuje se situaci. To je také jedna z podobností s Eulerovou metodou, kde se objevují aplikace s nepravidelnou mřížkou.

V následující kapitole se přiblížíme pojem jádro.

2.1 SPH jádra

Připomeneme, že jádro je funkce se dvěma parametry:



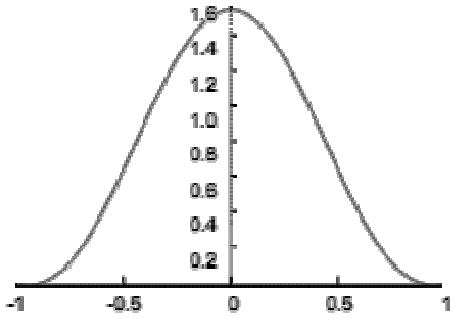
Obrázek 2.1 - Grafické znázornění SPH jádra ([2])

Na standratní jádro jsou kladeny určité požadavky (tyto požadavky jsou ale jen teoretické, v praxi být splněny nemusí):

- sudost jádra, to je spíše teoretický požadavek, protože je vždy kladné
- jádro by mělo být normalizované (plocha pod křivkou je rovna 1), opět spíše teoretický požadavek
- jádro by mělo být v celém definičním oboru kladné
- jádro by pro hodnoty mělo mít nulovou hodnotu - důležitý požadavek

Příklad teoretického jádra je Gaussovo jádro, které je dané vztahem:

$$W(r,h) = \frac{1}{(2\pi h^2)^{\frac{3}{2}}} e^{-\frac{|r|^2}{2h^2}}$$



Obrázek 2.2- Průběh Gaussova SPH jádra

Gaussovo jádro je teoretické, protože pro praxi je téměř nepoužitelné. Nejenže má nulovou hodnotu až v nekonečnu, ale hlavně je velmi výpočetně náročné. To je dáno tím, že se v něm objevují odmocniny. Proto je třeba použít nějakou aproximaci. Podle [2] je vhodná aproximace:

$$W(x,h) = \frac{315}{64\pi h^9}\begin{cases}(h^2 - x^2) & 0 \leq x \leq h \\ 0 & jinak\end{cases}$$

Toto jádro už splňuje všechny důležité podmínky. Nazývá se standartní a používá se u těch veličin, u kterých není nutné použít jádro jiné. Jednou s veličin, která by byla posuzována podle tohoto jádra, by mohla být například teplota.

2.1.1 Jádro pro tlakovou sílu

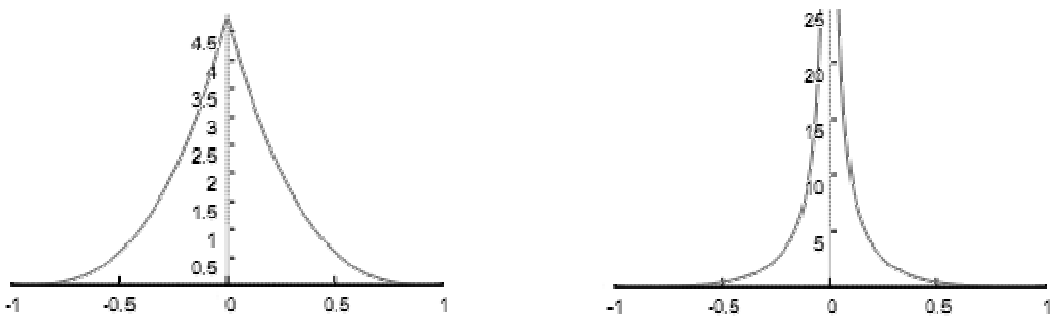
Pokud by bylo použito pro tlakovou sílu standartní jádro, docházelo by ke shlukovaní částic. To je dáno tím, že derivace standratního jádra v nule je nulová. Proto je třeba použít pro tlakové pole jádro s nenulovou derivací v nule. Příkladem takového "špičatého" jádra je:

$$W(x,h) = \frac{15}{\pi h^6}\begin{cases}(h - x)^3 & 0 \leq x \leq h \\ 0 & jinak\end{cases}$$

2.1.2 Jádro pro viskozitu

Viskozita, neboli prvek tření v kapalinách, má také své vlastní jádro. Čím více jsou kapaliny blíže, tím větší třecí síla na ně působí. Pro částice teměř u sebe, by měla být třecí síla co největší. Proto je pro viskozitu standartní jádro zcela nepoužitelné. Jako dobré jádro pro viskozitu se osvědčilo:

$$W(x,h) = \frac{15}{2\pi h^3}\begin{cases}-\frac{x^3}{2h^3} + \frac{x^2}{h^2} + \frac{h}{2x} - 1 & 0 \leq x \leq h \\ 0 & jinak\end{cases}$$



Obrázek 2.3 Vlevo je průběh SPH jádra pro tlakovou sílu. Vpravo je průběh SPH jádra pro viskózní sílu

3 Vizualizace

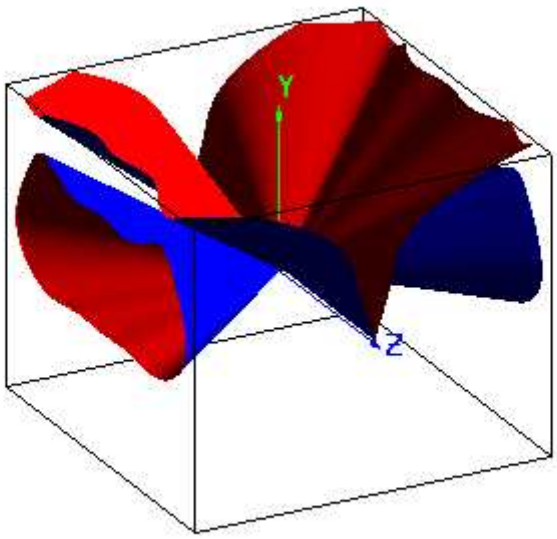
Simulace je velice důležitá, ale pokud bychom nedokázali data správně vizualizovat, byla by práce na simulaci samotné zbytečná. V této kapitole se hodlám věnovat principům, kterých se používá k vizualizaci kapalin.

3.1 Implicitní plochy a metaballs

Existuje několik způsobů, jak se v počítačove grafice reprezentují 3D objekty. Mezi nejznámější patří tradiční hraniční reprezentace. Jedná se vlastně o výčet vrcholů a hran, které je spojují. Tyto vrcholy a hrany představují hranici objektu. Bohužel pro potřeby vizualizace kapalin je tato metoda nevhodná, především kvůli své špatné prizpůsobivosti měnící se kapalině. Proto je potřeba použít jinou metodu.

3.1.1 Implicitní plocha

Mějmě funkci, která každému bodu v prostoru přiřadí určitou hodnotu - intenzitu. Potom všechny body se stejnou intenzitou vytvářejí implicitní plochu. To nám dává volnost v popisu plochy, protože pouhou změnou požadované intenzity lze změnit celou plochu - není vyjádřen povrch, ale intenzita.



Obrázek 3.1 - Příklad imlicitní plochy

3.1.2 Metaballs

Metaballs (někdy také bloby) jsou příkladem implictní plochy. Každý metaball je definován n-rozměrnou funkcí. Nejčastěji to bude asi 3-rozměrná funkce - *f(x,y,z)*. Poté je intenzita v jednom bodě v prostoru rovna výrazu

$$\sum_{i=0}^{n} f_i(x,y,z)$$

n je počet blobů

f_i je funkce blobu *i*

x, y, z jsou souřadnice bodu, jehož intenzitu hledáme

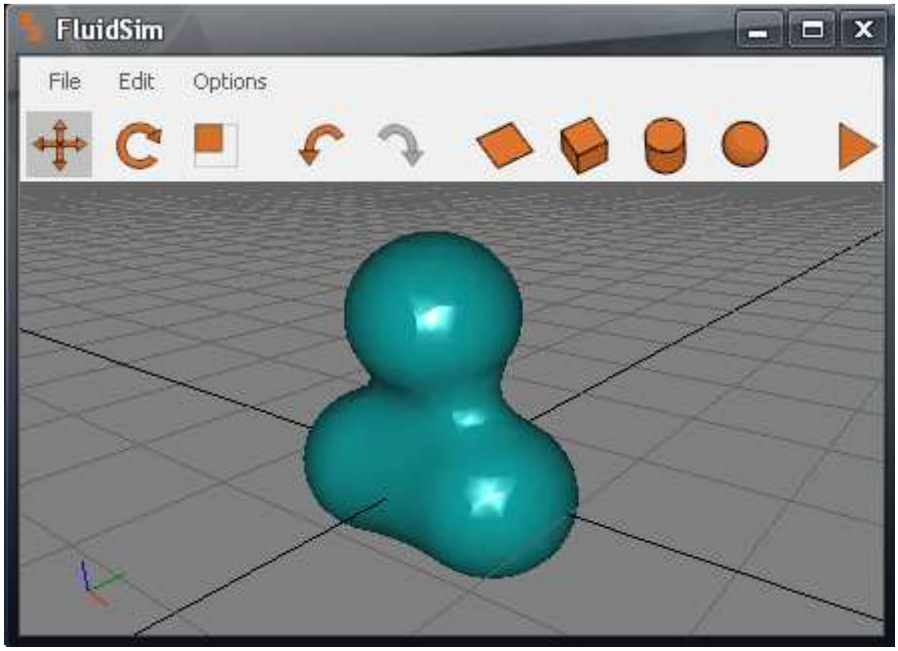
Jako funkci blobu lze zvolit pouhou vzdálenost od zkoumaného bodu:

$$f_i(x,y,z) = \sqrt{(x_i - x)^2 + (y_i - y)^2 + (z_i - z)^2}$$

Ovšem s použitím pouze této funkce nevypadají bloby příliš uspokojivě - nevypadají organicky. Proto je potřeba funkci rozšířit. Za ideální by se opět dalo považovat použití Gaussovy křivky, ta je ale opět pro svou obrovskou výpočetní náročnost nepoužitelná. Proto je vhodné zvolit nějakou aproximaci. Jako idealní se osvědčila:

$$F_i(r_i) = 1 - \frac{4}{9}\frac{r^6}{R^6} + \frac{17}{9}\frac{r^4}{R^4} - \frac{22}{9}\frac{r^2}{R^2}$$

R je vzdálenost, za kterou už blob nemá na dané místo vliv



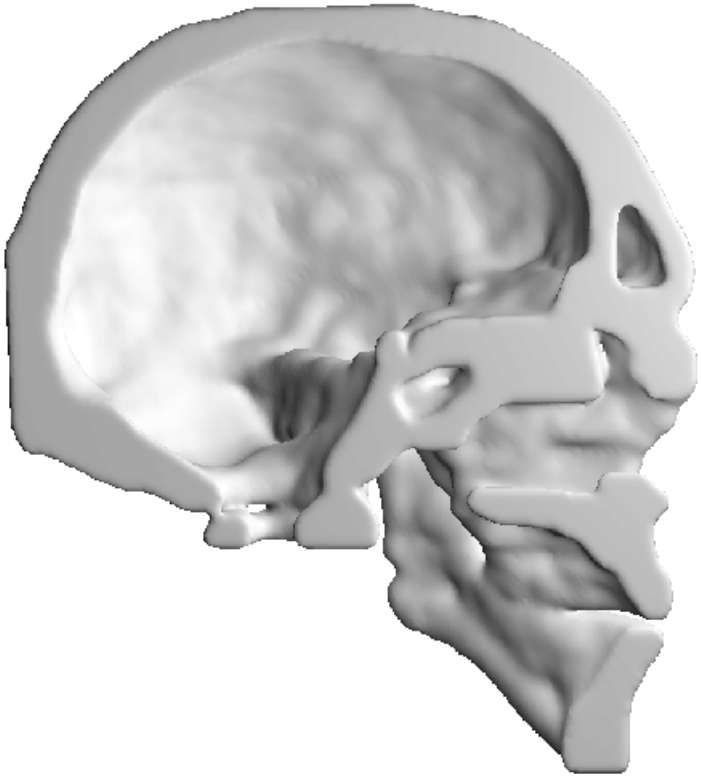
Obrázek 3.2 - Příklad tří metaballů. Obrázek je přímo z mé aplikace.

3.2 Metoda marching cubes

Reprezentovat objekt pomocí implicitních ploch je relativně snadné, intuitivní. Problém však nastává, pokud chceme implicitní plochu vizualizovat. Existuje několik možností, jak implicitní plochu vizualizovat. Jednou z nich je raytracing. Tato metoda je ale výpočetně velice náročná a v současných grafických kartách nemá hardwarovou podporu (avšak objevují se už určité pokusy v tomto směru a spousta výrobců, včetně firmy INTEL, se domnívá, že v budoucnu bude hardwarová akcelerace raytracingu v grafických adaptérech standartem). Moderní grafické adaptéry umí pracovat pouze s již zmiňovanou hraniční reprezentací. Proto je třeba nalézt způsob, jak převést implicitní plochu na hraniční reprezentaci.

Jednou z nejpoužívanějších metod je metoda marching cubes (český překlad "pochodující krychle" se příliš nepoužívá). Tato metoda byla poprvé představena na konferenci SIGGRAPH v roce 1987. Její alternativa ve 2D nese název marching squares.

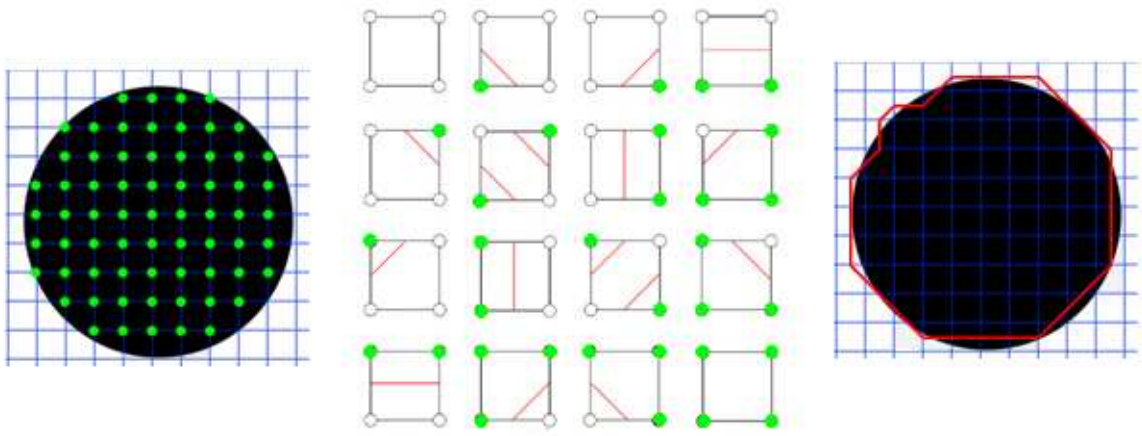
Tato metoda není určená pouze pro renderování implicitních ploch. Je použitelná všude tam, kde je možné získat 3D pole bodů, o kterých máme informaci, zda do zobrazovaného objektu patří či nikoliv. Velice často se tato metoda používá pro vizualizace medicínských dat.



Obrázek 3.3 - Řez lebkou zobrazený pomocí metody marching cubes (http://www.nlm.nih.gov)

Nejlépe se princip této metody vysvětluje na příkladu její 2D alternativy - metody marching squares.

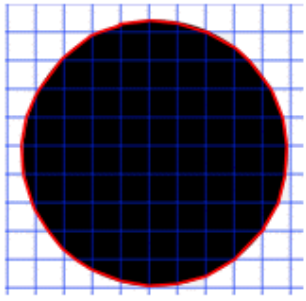
Máme kruh, jež je daný středem a poloměrem (kruhu by v 3D prostoru mohl odpovídat metaball). Nejprve je nutné rozdělit si plochu s kruhem na malé čtverce. Čím bude plocha detailněji rozdělena, tím kvalitnější výsledek dostaneme. Po té je třeba všem vrcholům všech čtverců přidělit hodnotu, jež udává jejich vzdálenost od středu kruhu. Potom vrcholy čtverců, jejichž hodnota je menší než poloměr, leží v kruhu, ostatní neleží. Následuje průchod přes všechny čtverce (proto marching squares). U každeho čtverce se vyhodnotí, které vrcholy leží v kruhu a které ne, a podle tabulky s jednotlivými případy - těch je samozřejmě $2^4 = 16$ - se na dané místo vloží hrany. Celý proces je zachycen na následujícím obrázku.



Obrázek 3.4 - Postup metody marching squares. Vlevo je kruh s ohodnocenou mřížkou. Uprostřed je tabulka 16-ti možných případů čtverce. Napravo je každému čterci přiřazen případ.

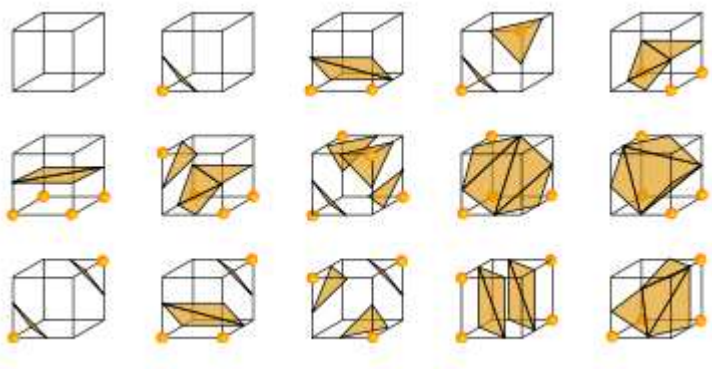
Některé případy jsou nejednoznačné. U metody marching squares to lze snadno řešit, u metody marching cubes mohou nastat mírné komplikace.

Výsledek ohraničení kruhu se může zdát poněkud hranatý a nepřesný. Lze to vyřešit zjemněním mřížky, což by ale bylo zbytečně výpočetně náročné. Lepší možnost je použít interpolaci vrcholů jednotlivých hran. To se vrchol hrany posune k tomu vrcholu čtverce, který je blíže hranici kruhu.



Obrázek 3.5 - Výsledek metody marching squares po použití interpolace.

Metoda marching cubes pracuje analogicky. V tabulce případů neleží informace o hranách, ale o trojúhelnících, které je potřeba na místo krychle vložit. Velikost tabulky případů je $2^8 = 256$. To se může zdát jako velké číslo, ale je třeba si uvědomit, že případy se opakují, jenom jsou různě natočené, zrcadlené, nebo jsou hodnoty ve vrcholech invertované. Ve výsledku pak zbývá pouze 15 jedinečných případů.



Obrázek 3.6 - 15 jedinečných případů metody marching cubes

4 Detekce kolizí

Kdyby kapalina během své simulace nedokázala reagovat (kolidovat) s okolním prosředí, postrádala by simulace význam. Vždyť právě to, jak budou částice reagovat na okolí, je to nejdůležitějsší, co nás na simulaci zajímá.

Způsobů řešení kolizí je nespočet. Kapalina může kolidovat s výškovou mapou (to je zdroj dat, který uvádí výškové rozdíly objektu, na který je výšková mapa aplikována), s různými grafickými primitivy, s implicitními plochami, atd.

V mé aplikaci částice reagují jednak na ohraničující kvádr (rozměry tohoto kvádru jsou shodny s velikostí mřížky pro metodu marching cubes). Detekce takové kolize není složitá, protože stěny kvádru jsou rovnoběžné s hlavními osami. Proto se dále tímto tipem kolize nebudeme zabývat.

Drůhý tip kolize vychází z koncepce, kterou jsem si stanovil před počátkem programování. Naplánoval jsem si, že částice budou schopny reagovat s jakýmkoliv objektem, který je ve scéně uložen. Objekty jsou ve scéně uloženy pomocí hraniční reprezentace - tedy seznamem trojúhelníků. Z toho všeho vychází jediná potřeba - umět detekovat kolizi částice s trojúhelníkem.

4.1 Detekce částice s trojúhelníkem

Detekce částice s trojúhelníkem se sestává ze dvou fází. Nejprve je potřeba detekovat, že vzdálenost částice od plochy dané vrcholy trojúhelníka klesla na nulovou (nebo spíše pod minimální povolenou) hodnotu. Po té je nutné detekovat, zda-li částice skutečně leží na trojúhelníku.

4.1.1 Vzdálenost částice od trojúhelníka

Pro vzdálenost částice od plochy dané vrcholy trojúhelníka platí vztah:

$$d = \frac{|n \cdot r| + d_t|n|}{|n|}$$

n je normálový vektor trojúhelníka

r je poloha částice

d_t je vzdálenost plochy od počátku současného systému

$n \cdot r$ značí skalární součin vektorů n a r

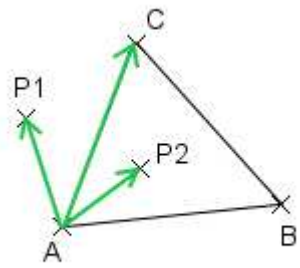
$|n|$ je velikost normálového vektoru plochy

Ze vzorce je patrné, že pokud normálový vektor bude normalizovaný, tak se už tak jednoduchý vzorec, ještě více zjednoduší. To, aby byly normálové vektory trojúhelníků normalizované, bude stejně potřeba v další fázi výpočtu detekce částice s trojúhelníkem. Navíc to vyžadují i obě nejpoužívanější grafické knihovny Direct3D i OpenGL pro správný výpočet osvětlení.

Ve výsledku nám tedy stačí pro každý trojúhelník informace o jeho normálovém vektoru a o vzdálenosti normálového vektoru od počátku.

4.1.2 Test polohy částice a trojúhelníka

Pokud je vzdálenost částice od plochy dané vrcholy trojúhelníka nulová (nebo spíše menší než minimální povolená hodnota), je možné přistoupit k dalšímu testu. A to k testu, zda-li částice leží v trojúhelníku.



Obrázek 4.1- Příklad dvou bodů a trojúhelníka.

Na předcházejícím obrázku jsou dva body. Bod *P1* v trojúhelníku neleží, ale bod *P2* ano. Pokud bychom vyhodnotili výraz $(\overrightarrow{AC} \times \overrightarrow{AP1}) \cdot (\overrightarrow{AC} \times \overrightarrow{AP2})$, kde $\times$ značí vektorový součet a $\cdot$ značí skalární součet, dostali bychom zápornou hodnotu. Záporná hodnota by vyšla proto, protože body *P1* a *P2* neleží ne stejné straně přímky $\overleftrightarrow{AC}$.

Pokud testujeme, zda částice leží v trojúhelníku, stačí pro každou přímku danou dvěma vrcholy trojúhelníka otestovat, zda-li zbývající vrchol a částice leží na stejné straně přímky. Test provedeme podle výše uvedeného vzorce.

5 GPGPU

GPGPU je zkratka pro General Purpose computation on GPUs, což v překladu znamená obecné výpočty na grafických procesorech. Jsou to tedy techniky, které programátorům umožňují programovat obecné, ne jen pouze s grafikou související algoritmy, jejichž výpočet nebude probíhat v centrální procesorové jednotce, ale právě v grafické kartě.

Dříve by něco podobného bývalo považováno za nesmyslenost, protože dříve byly výkony grafických procesororů v porovnání s výkony CPU mizivé. To ale neplatí v dnešní době, kdy grafické procesory dosahují výkonů srovnatelných s výkony CPU. Proto by bylo škoda, kdyby se grafické karty využívaly pouze pro vykreslování.

Ovšem ne každý algoritmus se hodí pro zpracování v GPU. Obecně lze říci, že vhodné jsou takové algoritmy, jež zpracovávají velké množství dat, která jsou v desetinné čárce, a zpracování jednotlivých dat může do jisté míry probíhat paralelně. Takové algoritmy jsou typické třeba ve zdravotnictví, ve finančnictví a ve vědeckém výzkumu vůbec. Další požadavek je dán na uspořádání dat - nelze používat dynamické datové struktury, nutné je použití polí a jiných datových struktur, kde jsou všechny prvky stejně dlouhé.

5.1 NVIDIA CUDA

CUDA je zkratka pro Compute Unified Device Architecture. Jedná se GPGPU technologii firmy NVIDIA. Je podporována pouze novými grafickými kartami této firmy - produkty GeForce 8. a vyšší série a produkty Tesla a Quadro.

Tato technologie umožňuje programátorům programovat GPU pomocí jazyka C s pár rozšířeními. Překlad probíhá tak, že pomocí dodávaného překladače je kód přeložen do "čistého" jazyka C a potom lze libovolným překladačem jazyka C přeložit tento mezivýsledek do strojového kódu.

5.1.1 GPU vs CPU

Během programování pomocí CUDA technologie je nutné rozlišovat dvojí prostředí. To se týka jak paměti tak funkcí. Funkce je nutné dělit na:

- funkce volané i spoušťené na CPU, identifikovány jsou klíčovým slovem *__host__*
- funkce volané z CPU, ale spouštěné na GPU, ty mají klíčové slovo *__global__*, nazývají se kernel funkce - funkce jádra
- funkce volané i spouštěné na GPU, identifikované klíčovým slovem *__device__*

Podle toho, kde je funkce spouštěna, v ní lze používat určitá rozšíření:

V CPU prostředí je možné pracovat s funkcemi pro zprávu GPU paměti (*cudaMalloc*, *cudaMemcpy*, *cudaFree*). Také zde lze získavat informace o zařízeních podporujících CUDA technologii, provádět navázání GPU paměti na grafické knihovny OpenGL a Direct3D a samozřejmě lze volat GPU funkce.

V GPU prostředí je možné používat rychlé, ale méně přesné funkce typu *_sin*, *_cosin*, různé zaokrouhlovací funkce, atd. Dále je možné používat funkci *__syncthreads*, která zajistí synchronizaci všech vláken. Je možné používat sdílenou paměť, globální paměť i paměť pro textury. Samozřejmě lze identifikovat vlákno a blok pomocí vestavěných proměnných *threadIdx* a *blockIdx*. Rozporuplnou možností jsou atomické funkce. Jejich přítomnost si technologie přímo žádá, ale bohužel jsou podporovány jen v novějších typech grafických karet (ty nesou označení *Compute capability 1.1*).

5.1.2 Způsob spouštění kernel funkcí

Po zavolání kernel funkce je tato funkce spuštěna pro každé vlákno (thread). Tato vlákna jsou organizována do bloků (block). Bloky jsou organizovány do mřížky (grid).

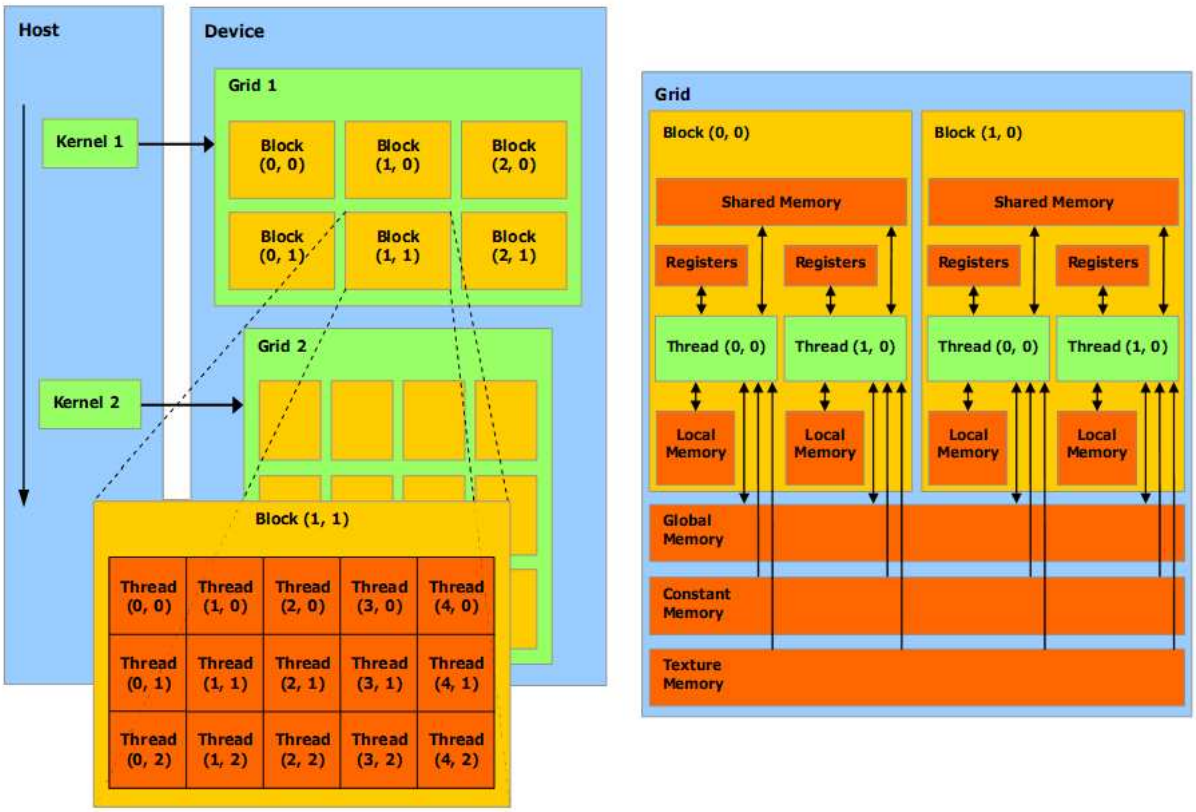
V bloku mohou být jádra organizována až do tří dimenzí. Již zmiňovaná vestavěná proměnná *threadIdx* je tedy struktura, která má položky *x, y, z*. Každý blok může mít 32 až 512 vláken. Vlákna v rámci bloku sdílí společné prostředky jako jsou registry a sdílená paměť. Registrů je pro každý blok k dispozici 8192, sdílené paměti 16 KB. Sdílených prostředků tedy není moc, proto je nutné dbát na řádné dělení vláken do bloků a na úsporné zacházení se sdílenou pamětí. Důvod proč se vyplatí pracovat se sdílenou pamětí je její rychlost. Přístup do sdílené paměti trvá někdy i pouhé 4 takty, kdežto přístup do globální paměti trvá zhruba 300 - 600 taktů.

Dále je třeba uvést, že již zmiňovaná funkce *__syncthreads* provede synchronizaci vláken pouze v rámci bloku.

Každý blok je spuštěn na jednom multiprocesoru. Jelikož mohou mít různá zařízení různý počet multiprocesorů, doporučuje se dělit vlákna tak, aby vždy bylo více bloků s vlákny. To na zařízeních s více multiprocsory umožňuje zpracovávat bloky paralelně. Pokud je multiprocesorů méně, musí se zpracovávat po částech sekvenčně. Doporučený počet bloků na vlákno je 92 - 256.

Bloky lze organizovat až do dvou rozměrů. Proměnná *blockIdx* je stejného typu jako proměnná *threadIdx*. Proměnná *blockIdx.z* je tedy za každých okolností rovna *1*.

Dělení vláken a bloků a celkový paměťový model představuje následující obrázek.



Obrázek 5.1- CUDA - na levé straně je organizace vláken do bloků. Na pravé straně je paměťový model.

5.1.3 Omezení technologie CUDA

Bohužel technologie CUDA má určitá omezení.

Jedno omezení jsem již zmínil - jedná se o atomické funkce. Nejenže nejsou podporovány ve všech zařízeních, ale navíc operují pouze s proměnnými typu integer.

Další omezení vyplívá částečně z předchozího a částečně je zřejmé z celého konceptu programování pro GPU. Pro GPU se těžko programují různé agregační a skenovací funkce - hledaní minim a maxim, řazení, atd. Naštěstí pro technologii CUDA existuje knihovna CUDPP - CUDA Data Parallel Primitives Library, která právě tento nedostatek řeší.

Další, podle mě vážný nedostatek je, že veškeré volání a navazovaní na kontext technologie CUDA je možné jen z jednoho CPU vlákna. Lépe řečeno - v CPU aplikaci může volat CUDA funkce jen jedno CPU vlákno, pokus jiných CPU vláken selže. Bohužel to, že se CUDA technologie takto chová, se mi nepodařilo najít v referenční příručce a stálo mě hodně usilí na toto omezení přijít.

6 Popis mé implementace

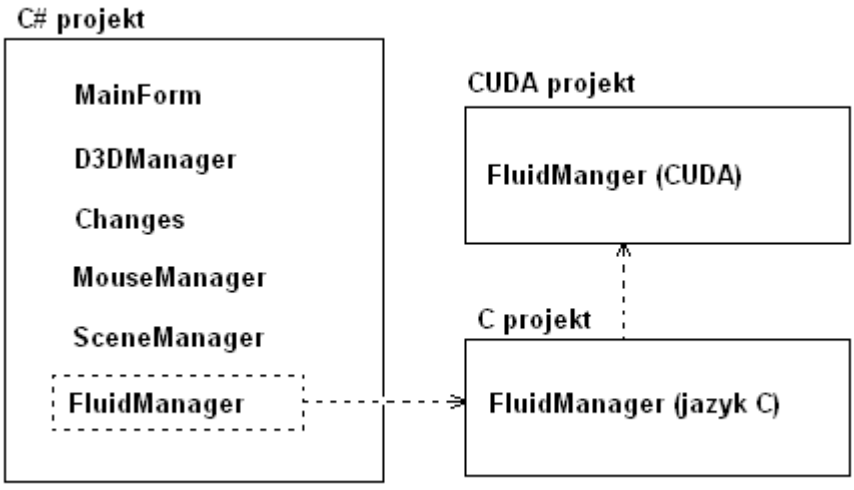
V této kapitole se pokusím shrnout postup, kterým jsem se ubíral při implementaci mé vlastní simulace tekutin.

Jako programovací jsem si zvolil jazyk C#. Měl jsem k tomu spoustu postranních důvodů jako jsou snadná tvorba uživatelského rozhraní, spousta naimplementovaných tříd a metod a vůbec celkově pohodlný styl programování. Hlavní důvod byl ovšem mnohem prostší - prostě jsem se chtěl tento jazyk naučit.

S volbou jazyka C# souvisí několik důsledků. C# je jazyk vyvíjený firmou Microsoft s čímž souvisí jeho úzká spojitost s operačním systémem Windows. Programy napsané v C# se spouštějí pod platformou .NET Framework, vyvíjenou rovněž firmou Microsoft. Programům běžícím pod .NET Frameworkem se říká managed. Jejich napojení na klasické - unmamanged programy může být problematické.

Program jsem vyvíjel v aplikaci Visual Studio 2005. Nejenže je to jediná schůdná možnost pro psaní C# programů, ale navíc i firma NVIDIA ve svém SDK pro technologii CUDA přímo přikládá projekty pro Visual Studio. Nakonec jsem tedy byl schopný překládat řešení, které se skládá z jedno C# projektu, z jednoho C projektu a z jednoho CUDA projektu najednou, přímo ve Visual Studiu.

Pojmem řešení myslím pojem solution z Visual Studia. V rámci jednoho řešení lze totiž uskupovat více projektů, z nichž každý může být v jiném jazyce. Na následujícím obrázku je rozložení důležitých tříd, projektů a vazby mezi nimi.



Obrázek 6.1 - Rozložení projektů v rámci řešení

Třída MainForm se stará o uživatelské rozhraní. Obstarává zobrazovaní oken, dialogů atd. Reakce na událostí způsobené myší předává třídě MouseManager. To proto, že reakce na tyto události jsou velice složité, tudíž se hodí mít pro ně vlastní třídu.

Třída D3DManager se kompletně stará o napojení na knihovnu Direct3D, která se stará o vykreslování. Knihovnu Direct3D jsem použil proto, protože narozdíl od knihovny OpenGL existuje i ve verzi pro managed kód. Třída D3DManager provádí transformace kamery, ukončuje i zakončuje vykreslování, umožňuje uložení právě vykreslované scény do bitmapy a mnoho dalšího.

Třída Changes se stará o ukládání změn. Umožňuje vracet provedené operace atd. Zkrátka umožňuje operace Undo a Redo známé z jiných programů.

Třída SceneManager se stará o scénu. Udržuje polohu všech objektů, umožňuje jejich vykreslování, transformace atd. Dále vykresluje úvodní logo, pomocnou mřížku či hraniční kvádr.

Nejdůležitější třídou z pohledu simulace kapalin je třída FluidManager. Ta je přítomna 3x, pokaždé v jiném projektu. V C# projektu má tato třída za úkol provést napojení na DLL, jež výsledkem překladu C projektu. Výsledkem překladu CUDA projektu je další DLL, na které se napojuje DLL z C projektu.

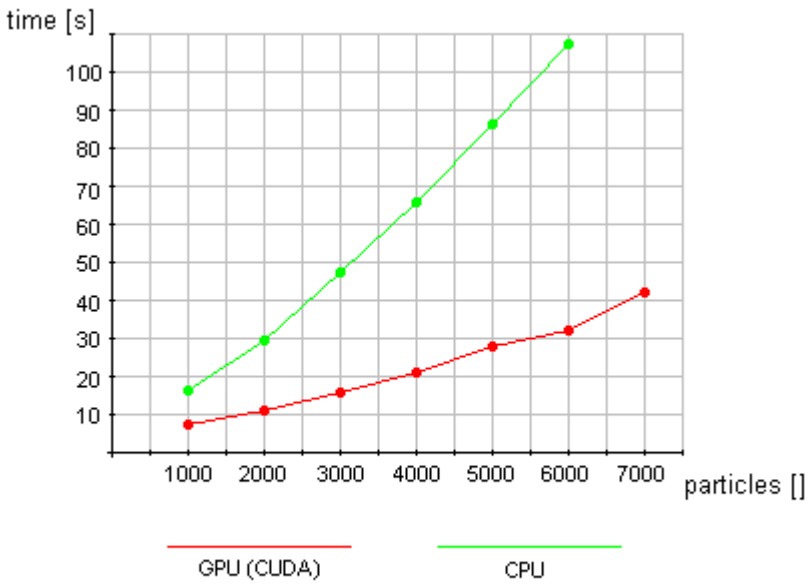
Třídy FluidManager v C projektu a FluidManager v CUDA projektu vykonávají stejnou funkci. Mají na starost celou simulaci a vizualizaci kapaliny. Vše je naprogramováno podle postupů a algoritmů, jež se objevily v této práci.

Navíc je třeba dodat, že všechny třídy umožňují zapsat svůj stav do XML podoby, takže je možné ukládat a načítat celou scénu i s nastavením. V následující kapitole se objevuje srovnání rychlostí výpočtů pomocí CPU a GPU programovaného technologií CUDA.

6.1 Srovnání CPU a GPU výpočtů

Jeden z cílů této práce bylo využití moderních grafických karet k počítaní určitých částí simulace tekutin a plynů. Snaha totiž byla, aby se dokázalo, že využití GPU k určitým typům výpočtů má smysl, protože dojde ke zrychlení. Snažil jsem se simulaci pomocí technologie CUDA naimplementovat co nejefektivněji. Avšak tuto technologii není lehké zvládnout za krátkou dobu, proto je mi jasné, že i má implementace má k optimální implementaci daleko.

Přesto jsem, jak je zřejmé z následujícího grafu, dosáhl oproti výpočtům na CPU určitého zrychlení.



Obrázek 6.2 - Graf porovnávající výkony GPU a CPU v mé aplikaci

Pro oba typy výpočtu jsem pustil stejnou scénu s narůstajícím počtem částic. Z grafu je patrné, že s přibývajícím počtem částic se časy obou výpočtů rozcházejí. Při počtu částic 1000 je GPU jen 2x rychlejší, ale při opravdu velkém počtu částic jsem naměřil i 15-ti násobné zrychlení. Ovšem při malých počtech částic - řádově desítky - je GPU implementace pomalejší. To je dáno tím, že se při každém kroku simulace určitá data musí přenést z operační paměti do paměti GPU a naopak.

Závěr

Simulace tekutin a plynů není snadné téma. Matematická teorie týkající se této problematiky je značně složitá. Simulace tekutin zasahuje do mnoha oblastí - počínaje fyzikou a matematikou, které jsou nutné pro zvládnutí teorie, přes simulaci samotnou až po prostorovou počítačovou grafiku. V mém projektu bylo navíc třeba zvládnout technologie GPGPU.

O teorii částicově založené simulace tekutin bylo napsáno mnoho článků, proto nebylo obtížné udělat si o této problematice určitý přehled. Přesto jsem během implementace narazil na nemálo problémů. Ty se nakonec s většími či menšími kompromisy podařilo vyřešit. Výsledkem je aplikace, která alespoň částečně dokáže simulovat pohyb tekutin. V této oblasti by se dalo mnohé vylepšovat, protože pohyb tekutin zahrnuje mnoho jevů, které zatím moje simulace není schopna zachytit - jedná se například o výření, změny teploty kapaliny ovlivňující její hustotu, lámání vln, spojení více kapalin s různou hustotou, atd. Do budoucna bych si také rád vyzkoušel druhý možný přístup k simulaci tekutin - Eulerovu metodu a porovnal výsledky dosažené použitím těchto dvou odlišných metod.

Problémy s vizualizací kapaliny jsem si myslím zvládl dobře. Pojmy jako jsou implicitní plochy, metaballs, marching cubes byly radost se učit. Navíc jsem se naučil zacházet s knihovnou Direct3D, která je alternativou ke knihovně OpenGL, se kterou jsem se setkal již dříve. To, že mě tato část projektu zaujala nejvíce, je jistě dáno tím, že mám počítačovou grafiku jako celek velice rád. V budoucnu by se do mé aplikace daly přidat efekty odrazů a lomů světla, které by jistě zvýšily realističnost výstupů.

Když jsem si zapisoval tento projekt, neměl jsem o technologii CUDA ani potuchy. Zpočátku jsem k ní byl velice skeptický, ale to jsem velice brzy překonal. Problémy typu paralelismus, přenos dat mezi CPU a GPU, synchronizace, přístupy do paměti a jiné, které programování GPU přinaší, jsou jistě výzvou pro každého programátora. Nejinak tomu bylo i v mém případě a velice mě těšilo, jak snadno jsem se s touto technolgií sžil. Přesto je hlavně v této časti mého projektu co vylepšovat. Jedná se hlavně o další optimalizace. Především v přistupovaní ke sdílené paměti je třeba mnohé dodělat. Jelikož ke sdílené paměti přistupují jednotlivá vlákna paralelně, může docházet ke konfliktům (bank conflicts). Tyto konfikty mají za následek zpomalení přístupu k tomuto typu paměti. NVIDIA poskytuje určité postupy, jak se těmto konfliktům vyhnout, ale jejich zvládnutí nebylo z časových důvodů možné zahrnout do tohoto projektu. Další možné vylepšní se týká metody marching cubes. Ta se totiž zatím v mé aplikaci není schopná celá provádět na GPU. Je to dáno chrakteristikami tohoto algoritmu. Ovšem například s použitím atomických operací či již zmiňované knihovny CUDPP by bylo možné i tuto metodu komplentně naprogramovat pod technologií CUDA. Dokonce jsem již jednu takovouto implementaci viděl.

Tento projekt měl pro mě ještě jeden přínos - naučil jsem se programovat jazykem C# a k programovaní využívat programu Visual Studio.

Na úplný závěr bych rád podotknul, že jsem se v tomto projektu soustředil hlavně na praktickou část. I proto není tato technická zpráva příliš rozsáhla. Chcete-li více proniknout do mého díla, doporučuji Vám vyzkoušet si aplikaci FluidSim, která je výsledkem praktické části. K této aplikaci je napsána malá nápověda, která je spolu s aplikací na přiloženém CD. Určité informace o projektu Vám dozajista poskytne i programová dokumentace, která je také na přiloženém CD.

Literatura

[1] Clavet S., Beaudoin P., Poulin P., Particle-based Viscoelastic Fluid Simulation, In SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, New York, 2005, page 219

[2] Dongwoon Lee, Physics-Based Simulations for Fluid Mixtures, [Research paper], University of Toronto, 2007

[3] Novák J., Visualization of particles in force fields, [Master's thesis], Czech Technical University

[4] Premoze S., Tasdizen T., Bigler J., Lefohn A., Whitaker R.T., Particle-based simulation of fluids, In Eurographics, 22, 2003, page 401

[5] Sharman J., The Marching Cubes Algorithm, http://www.exaflop.org/docs/marchcubes/

[6] NVIDIA, CUDA - Compute Unified Device Architecture - Programming Guide, 2007

[7] http://www.gpgpu.org/ - Stránka věnovaná obecným výpočtum na grafických adaptérech

Seznam obrázků

Seznam příloh

Příloha 1. CD