STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor 18 - Informatika

Simulace kapalin částicovým přístupem a jejich vizualizace algoritmem Marching Cubes

Vypracoval:

Jakub Marian

Oktáva A
Gymnázium Litoměřická 726,
Praha 9 – Prosek

Praha, 2008

Abstrakt

V práci je popsán algoritmus Marching Cubes a jeho implementace (zdrojové kódy jsou psány v jazyce Object Pascal a prostředí OpenGL). Dále práce popisuje algoritmy a implementaci schématu "Prediction-relaxation" založeného na SPH (Smoothed particle hydrodynamics) za použití metody zachování dvojí hustoty (Double density relaxation) vyvinuté S. Clavetem, P. Beaudoinem, a P. Poulinem. V rámci tohoto přístupu jsou samostatně řešeny problémy týkající se viskozity, plasticity a elasticity kapaliny, interakce kapaliny s objekty a přilnavosti kapaliny k objektům. Dále se práce zabývá interakcí více kapalin – vhodným nastavením konstant je možné dosáhnout většiny běžných efektů mezi nemísitelnými kapalinami. V práci je vyřešeno optimalizované zobrazování povrchu jak jedné, tak více kapalin algoritmem Marching Cubes. Diskutuje také matematické aspekty celé teorie. Na závěr je čtenáři podán návod, jak výstup algoritmu Marching Cubes propojit s ray tracerem POV-Ray, díky čemuž je možné vytvářet fotorealistické snímky.

Obsah

Předmluva

Zapracovat do svého programu simulaci kapaliny (a případně plastických těles) interagující se svým okolím je snem mnoha herních vývojářů, tvůrců grafického software, odborníků na speciální efekty ve filmovém průmyslu a mnoha dalších. Vzhledem k náročnosti takovéto simulace – a to jak výpočetní, tak programátorské – se většinou mimo komerční sféru od řešení tohoto problému ustupuje. V současné době jsou však již i běžné počítače natolik výkonné, že výpočetní náročnost začíná pomalu ustupovat do pozadí. Co je tedy základní příčinou, že ani špičkoví „amatérští" softwaroví vývojáři v České republice se touto oblastí prakticky nezabývají? Já osobně věřím tomu, že je to důsledkem nedostatku vhodných materiálů. V době shromažďování informací jsem hledal velmi úporně, přesto se mi nepovedlo nalézt **žádný česky psaný dokument**, který by se zabýval problematikou simulace kapalin pro grafické použití. V angličtině již bylo sepsáno rozumné množství použitelných článků, problém ovšem spočívá v tom, že většina není dostupná na internetu, či je za jejich stáhnutí vyžadován nemalý finanční obnos. Krátce před dokončením tohoto textu však byla vytvořena diplomová práce Ondřeje Nováka *Simulace viskózních kapalin*, která se touto problematikou zabývá. Českému čtenáři bude jistě přínosem, nezabývá se však příliš implementací, zabývá se hlavně matematickými aspekty teorie a obecnějším popisem technik.

Důležitým prvkem v simulaci kapalin je zobrazování simulované kapaliny (samotný výpočet bez grafického znázornění nám toho mnoho neřekne). Nejpřirozenější cestou k zobrazení povrchu kapaliny je použití algoritmu **Marching Cubes**. Princip algoritmu není složitý, jeho implementace je však již poměrně náročná – a taktéž o této oblasti neexistují prakticky žádné informace v češtině. V angličtině je však již situace naštěstí o něco lepší, zde je možné získat informace jak z „vědecké sféry", tak z webových stránek amatérských vývojářů. Tento algoritmus již dlouhodobě přináší užitek nejen v oboru simulace kapalin, ale také ve **vizualizaci dat ve fyzice**, **matematice** a především **medicíně**, kde se využívá k zobrazení dat získaných počítačovou tomografií a magnetickou rezonancí, proto považuji poskytnutí obsáhlého a srozumitelného výkladu o tomto algoritmu odborné veřejnosti za více než žádoucí.

Doufám, že tento text poskytne čtenářům zajímajícím se o danou problematiku přesně to, co prozatím marně hledali (stejně jako já před začátkem práce na tomto textu). Mým hlavním cílem bylo „zaplnit" jistou mezeru v česky psané literatuře tím, že přetlumočím a rozšířím myšlenky anglicky psaných článků a vnesu do nich svůj vlastní přístup k implementaci algoritmů, jež je mnohdy ještě mnohem náročnější než zvládnutí principu algoritmů samých a kterou se anglicky psaná literatura v tomto oboru příliš nezabývá.

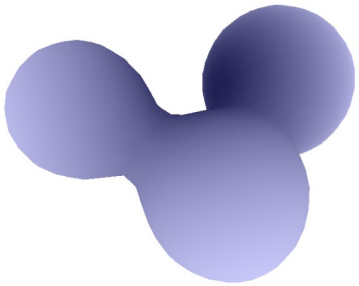
Přeji tedy čtenáři hodně štěstí (z vlastní zkušenosti vím, že ho bude potřebovat, v takto složitých programech je poměrně jednoduché udělat těžko odhalitelnou chybu; na typické chyby, kterých jsem se při implementaci dopustil já, budu v textu upozorňovat), děkuji mu za to, že byl ochoten přečíst si tuto obšírnou předmluvu až do konce, a slibuji, že následující výklad již bude stručnější...

Autor

1 Úvod

Rozeberme si nyní, čím přesně se tento text zabývá a jak je strukturován. Již zde bych rád podotkl, že tento text není učebnicí programování ani práce s OpenGL – naopak, znalost pokročilejších programovacích technik a OpenGL je od čtenáře vyžadována. K popisu algoritmů využívám „pseudojazyka“, který se z části podobá Pascalu, C++ a jiným jazykům, z části je zcela verbální. Věřím, že se srozumitelností algoritmů by neměl být problém.

Marching Cubes

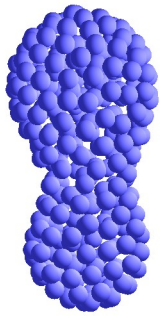


Obr. 1: Metaballs

První oddíl je věnován algoritmu Marching Cubes. Tento algoritmus slouží k vizualizaci skalárních polí v prostoru – algoritmus se snaží aproximovat určitou izoplochu (pokud by daným polem byl nějaký potenciál, pak bychom ji nazvali spíše ekvipotenciální plochou) s tím, že jsou dány hodnoty tohoto pole v nějaké diskrétní mřížce. Aproximace se zlepšuje s rostoucím množstvím bodů v mřížce – vzhledem k tomu, že je rozumné používat konstantní rozestupy bodů v mřížce ve směru všech tří os, roste výpočetní náročnost algoritmu s „rozlišením“ mřížky n jako $O(n^3)$. Chceme-li použít osvětlení a případně další efekty, je nutné použít poměrně velkou mřížku, a tak je skutečně kvalitně vypadající simulace založená na tomto a podobných vizualizačních algoritmech (vzhledem k časové složitosti) stále hudbou budoucnosti. Vzhledem k tomu, jakým tempem se vyvíjí hardware současných grafických karet, je však možné, že tato budoucnost není nikterak vzdálená. Na obr. 1 jsou zobrazeny tzv. metaballs za použití algoritmu Marching Cubes.

Simulace kapalin

V této části se budu zabývat takzvanou Smoothed Particle Hydrodynamics (dále jen SPH), rozvinu především metodu *zachování dvojí hustoty* (Double density relaxation) popsanou v článku [PVFS05]. SPH je metodou, jež vychází z fyzikálního modelu chování kapalin (z Navier-Stokesových rovnic), implementuje ho však za pomoci částic a jejich interakcí. Tyto interakce jsou popsány tzv. zjemňujícím jádrem (smoothing kernel), jehož volba je již ryze empirická, není založena na přesném fyzikálním modelu. Za cenu ne zcela fyzikálně přesného modelu je možné dosáhnout rychle pracujících, poměrně realisticky vypadajících simulací. Proto se tohoto přístupu využívá pro grafické aplikace – převážně ve filmovém a herním průmyslu. Na obrázku 2 je možné si prohlédnout dvě kapky sestávající z malého množství částic v první fázi splynutí. Čím více částic je použito, tím je model přesnější ale také pomalejší.



Obr. 2: Dvě interagující kapky

Další fází simulace kapaliny je její zobrazení. K tomuto účelu je použit algoritmus Marching Cubes popsaný v předchozí části. Je definována jistá skalární funkce tak, aby vytvořila izoplochu dobře kopírující tvar, který zaujímají částice simulované kapaliny. Poté je tato izoplocha zobrazena algoritmem Marching Cubes za pomoci určitých zrychlujících optimalizací. Tento algoritmus musí být mírně poupraven, pokud chceme simulovat interakci více různých kapalin – pro kapaliny, jež se mísí, je nutné přiřadit bodu mřížky nejen informaci o hodnotě skalární funkce, ale také je nutné rozhodnout, jakou barvu by v tomto bodě měl povrch mít (což se děje na základě informace, kolik se v okolí daného bodu nachází částic daného

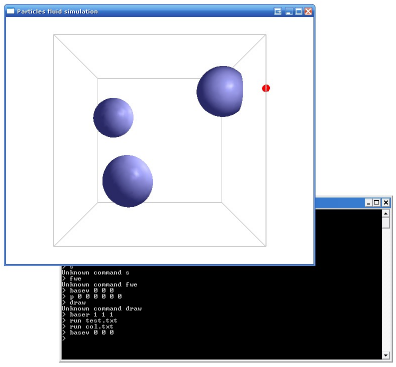
Obr. 3: Zobrazený povrch

druhu). Pro kapaliny, které se nemísí, je nutné spustit algoritmus Marching Cubes víckrát – pro každou kapalinu zvlášť, tím vznikne dojem, že mezi kapalinami existuje jasně viditelné rozhraní.

Matematický popis simulace kapalin

V této části bude SPH vysvětlena podrobněji z matematického hlediska. Důvodem faktu, že je tato kapitola umístěna až za kapitolou o implementaci simulace kapalin, je, že čtenář pravděpodobně bude tuto publikaci využívat spíše jako příručku k praktickému vývoji, většinu čtenářů matematické aspekty metody SPH zajímat příliš nebudou.

Popis ovládání a funkce ukázkového programu



Obr. 4: Screenshot programu

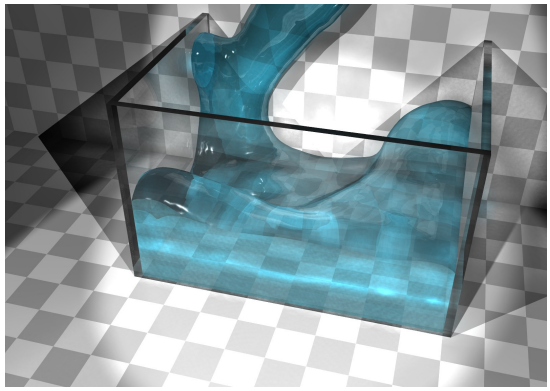
V této sekci budou vysvětleny funkce a ovládání ukázkového programu (vzhled obrazovky viz na obrázku 4). Tento program implementuje vše, co bude vysvětleno v následujícím textu – čtenáři ovšem doporučuji psát svou implementaci podle tohoto textu, vše je zde vysvětleno výrazně srozumitelněji a přehledněji. Program je psán v jazyce Object Pascal a je určen ke kompilaci ve Free Pascal Compileru.

Program sestává ze dvou oken – okna OpenGL pro vykreslování grafiky a okna příkazové řádky, pomocí něhož se do programu zadávají různé příkazy ovlivňující jeho běh. Možnosti příkazů jsou poměrně velké, program také podporuje skriptování. Díky tomu je možné navrhnout poměrně komplexní scény.

Renderování vysoce kvalitních obrázků

K renderování fotorealistických obrázků byl použit open source ray tracer POV-Ray. Ukázku toho, čeho lze po propojení s tímto ray tracerem dosáhnout, si můžete prohlédnout na obr. 5. Renderování jednoho snímku tohoto trvá obvykle na výkonném počítači 2 až 30 minut v závislosti na rozlišení a počtu použitých „fotonů“.

To, jakým způsobem je potřeba „naformátovat“ výstup programu, aby mohl spolupracovat s POV-Rayem, je v této části podrobně vysvětleno.



Obr. 5: Naplňování akvária

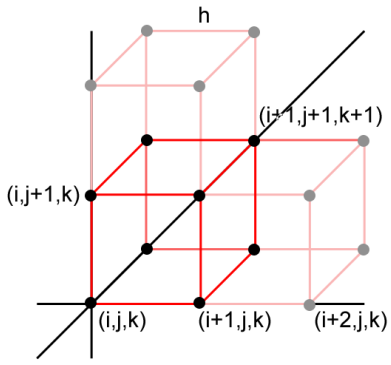
Poznámka k elektronické verzi textu

Pokud čtete tento text v jeho elektronické podobě – tj. jako .pdf soubor, možná Vám připadá, že písmo zde použité nevypadá po grafické stránce moc dobře. Pokud to není tím, že jste zarytým odpůrcem bezpatkového písma, je to pravděpodobně způsobeno tím, že máte verzi Acrobat Readeru, která z nějakého důvodu zobrazuje písmo jinak, než poté (ze stejného programu) vypadá toto písmo vytištěné na papíře. V takovém případě Vám doporučuji použít prohlížeč Foxit Reader, který vše zobrazuje správně.

2 Algoritmus Marching Cubes

2.1 Úvod

Mějme v prostoru zadané skalární pole, tj. funkci tří proměnných $f(x, y, z)$. Naším cílem bude vizualizovat plochu zadanou rovnicí $f(x, y, z) = konst$. V programu není možné znát hodnotu funkce v každém bodě prostoru, ale pouze v konečném počtu bodů zvolených podle nějakých kritérií. Nejpřirozenější cestou se jeví zjišťovat hodnoty funkce f bodech nějaké pravidelné mřížky. Nejjednodušší mřížkou je pravoúhlá mřížka se stejnými rozestupy ve všech směrech, jejíž body mají souřadnice $x = i \cdot h + x_0$, $y = j \cdot h + y_0$ a $z = k \cdot h + z_0$, kde i, j a k jsou přirozená čísla (včetně nuly), h je rozestup jednotlivých bodů mřížky a (x_0, y_0, z_0) je poloha počátku mřížky. Body mřížky jsou tedy dány třemi indexy (i, j, k), kde (0, 0, 0) je např. levý dolní roh blízký k pozorovateli (viz obr. 6) - konkrétní volba indexování je ovšem věcí konvence.



Obr. 6: Mřížka s vyznačenými krychlemi

Algoritmus Marching Cubes funguje tak, že projde všechny krychle o hraně h v této mřížce a na základě hodnot funkce v bodech mřížky (tj. v krajních bodech krychle) vygeneruje v každé krychli trojúhelníky, které co nejlépe aproximují zvolenou izoplochu. Samotný proces tvoření trojúhelníků tak závisí vždy jen na jedné dané krychli. Vrcholy vygenerovaných trojúhelníku leží vždy na hraně příslušné krychle. Problém ovšem nastává, když chceme k jednotlivým vrcholům trojúhelníků určit normály – pokud nechceme použít ploché stínování (flat shading), u kterého mají všechny body trojúhelníku stejnou normálu, musíme nějakým způsobem zprůměrovat normály bodů příslušející více krychlím. O tom viz část o výpočtu normál. Je možné také nepočítat normály vůbec a dojem prostorovosti dodat metodou, kterou jsem nazval „pseudosvětlo“ - vrcholu přiřadíme barvu v závislosti na vzdálenosti od zdroje světla. Tato metoda vytváří dokonale jemné stínování, neumožňuje však nastavit jakékoliv vlastnosti materiálu, neumožňuje pracovat s průhledností apod.

2.2 Algoritmus Marching Squares

2.2.1 Princip

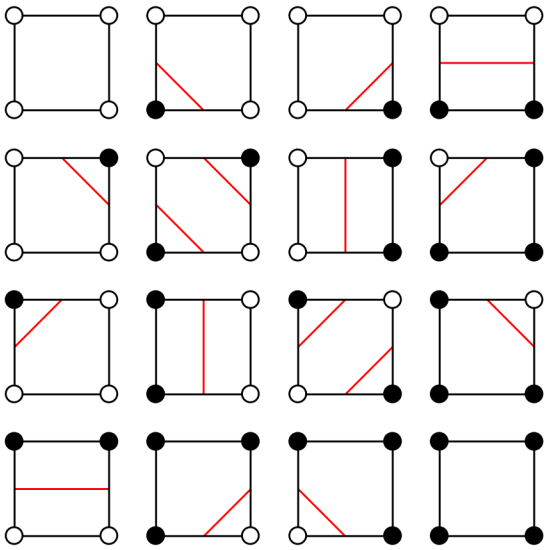
Vysvětleme si funkci algoritmu Marching Cubes nejdříve na jednodušším algoritmu – Marching Squares. Princip algoritmu je tentýž, pouze pracuje v rovině. Mějme dvojrozměrnou mřížku s indexy i a j (odpovídající krokům na ose x a y) s „rozlišením“ h (tj. posun o 1 v indexu je posunem o h v reálném rozměru) o rozměrech m x n (i nabývá hodnot 0 až m-1, j nabývá hodnot 0 až n-1). Označme $f_{ij} = f(x_0 + i \cdot h, y_0 + j \cdot h)$ hodnoty funkce f v jednotlivých bodech mřížky, f_0 hodnotu, pro níž hledáme křivku, na které je $f(x, y) = f_0$. Funkce algoritmu je následující:

Algoritmus Marching Squares

pro každý bod (i,j) mřížky

$f_{ij} := f(x_0 + i \cdot h, y_0 + j \cdot h)$
pro každý čtverec v mřížce **dělej**
    Spočítej index čtverce
    Zjisti, na kterých hranách leží koncové body úseček z tabulky úseček
    pro všechny nalezené úsečky **dělej**
        Interpoluj polohu bodu na dané hraně na základě hodnot ve mřížce
        Vykresli příslušnou úsečku

Popišme si nyní jednotlivé části algoritmu. Přiřazení hodnot funkce f snad není nutné dále rozebírat.



Obr. 7: Možné případy rozmístění vnitřních vrcholů

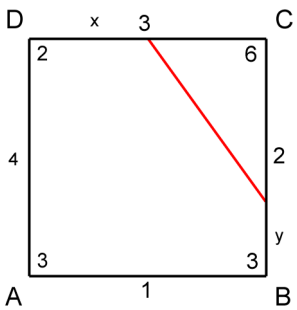
Pro každý vrchol čtverce určíme, zda je, nebo není vnitřním bodem plochy ohraničené izokřivkou (vrstevnicí) $f(x, y) = f_0$. To určíme z podmínky $f_{ij} \leq f_0$ - to, zda bod určený touto podmínkou je vnitřní, nebo vnější, určíme podle předpokládaného chování funkce f a našich požadavků na vizualizaci. Je zřejmé, že pokud je jeden bod vnějším bodem (tj. je pro něj $f_{ij} > f_0$ a druhý vnitřním bodem, tj. je $f_{mn} \leq f_0$, pak musí křivka, na které je f rovno f_0 procházet někde mezi těmito dvěma body (uvažujeme pouze spojité funkce f). Všechny možné konfigurace vnitřních a vnějších bodů ve čtverci jsou znázorněny na obr. 7. Červeně jsou vyznačeny jim příslušející úsečky. Řekněme, že tyto konfigurace jsou nějak očíslovány. Pak můžeme pro každý čtverec mřížky jednoznačně určit index příslušného čtverce (jak nejvhodněji situace očíslovat bude popsáno dále).

Když máme index čtverce, podíváme se do tabulky úseček příslušných k danému čtverci. Ta nám řekne, na kterých hranách čtverce se nachází konce jednotlivých úseček. Vzdálenost těchto bodů od vrcholů čtverce bychom mohli nechat h/2, dostali bychom však velmi hrubý výsledek. Daleko lepší možností je tuto vzdálenost nějak interpolovat. Interpolaci je vhodné zvolit na základě chování dané funkce f, dobře však postačí i jen prostá lineární interpolace.

Vysvětleme si to na příkladu – mějme čtverec jako na obrázku 8 (u vrcholů jsou zapsané hodnoty funkce). Řekněme, že hodnota $f_0 = 4$. Pak body A, B a D leží vevnitř, bod C leží vně. Index čtverce v tabulce je nejvhodnější implementovat ve dvojkové soustavě jako

$$index = \hat{A} \cdot 2^0 + \hat{B} \cdot 2^1 + \hat{C} \cdot 2^2 + \hat{D} \cdot 2^3$$

kde $\hat{A}$, $\hat{B}$, $\hat{C}$ a $\hat{D}$ jsou hodnoty 0 nebo 1, podle toho, zda jsou body A, B, C a D vnitřními body. Tento index nejsnáze spočítáme bitovou operací or.:



Obr. 8: Jeden čtverec

Index := 0
Pokud A je uvnitř **tak** Index := Index **or** 1
Pokud B je uvnitř **tak** Index := Index **or** 2
Pokud C je uvnitř **tak** Index := Index **or** 4
Pokud D je uvnitř **tak** Index := Index **or** 8

Celkem tak index nabývá hodnot od 0 do 15 a přesně nám rozlišuje jednotlivé případy. V našem konkrétním případě je index roven 1+2+8 = 11. Kdybychom se teď podívali do tabulky úseček (implementované nejspíše polem), našli bychom pro index 11 jedinou úsečku – a to z hrany 2 do hrany 3. Na základě lineární interpolace nyní spočteme vzdálenosti x a y od bodů D a B (viz obrázek). Hledáme funkci ve tvaru $I(t)=mt+n$, aby $I(a)=0$ a $I(b)=h$, kde h je délka hrany čtverce. Z těchto podmínek snadno určíme:

$$I(t)=\frac{t-a}{b-a}h$$

a zde značí hodnotu ve výchozím bodě, b hodnotu v koncovém. Pro náš konkrétní příklad vypočteme

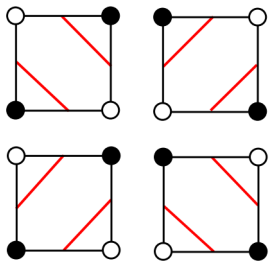
$$x=\frac{f_0-f(D)}{f(C)-f(D)}h=\frac{4-2}{6-2}h=\frac{h}{2}$$

$$y=\frac{f_0-f(B)}{f(C)-f(B)}h=\frac{4-3}{6-3}h=\frac{h}{3}$$

Jelikož polohu počítáme vždy ze dvou koncových bodů hrany čtverce bez ohledu na to, ve kterém čtverci se právě nacházíme, bude výsledná křivka na společné hraně dvou čtverců spojitá. Po projití celé mřížky tak nalezneme kompletní vrstevnici. Jelikož chyba oproti skutečné vrstevnici v principu nemůže být větší než rozlišení h mřížky, snižováním h můžeme (prakticky neomezeně) zvyšovat přesnost. Pokud ale chceme obsáhnout stále stejný region, musíme nepřímo úměrně h zvyšovat počet bodů mřížky ve směru každé z os. Náročnost algoritmu tak kvadraticky narůstá, dvakrát hustší mřížka znamená čtyřikrát delší výpočet.

2.2.2 Nejednoznačnosti algoritmu Marching Squares

Při extrakci křivky algoritmem Marching Squares vznikají nejednoznačnosti – v situacích znázorněných na obrázku 9 není možné rozhodnout, zda použít spíše úsečky znázorněné v prvním řádku, nebo ve druhém (v anglicky psané literatuře se těmto případům říká ambiguities - dvojsmysly). V rámci algoritmu Marching Squares se jedná naštěstí pouze o věc konvence – pokud první a druhý řádek nekombinujeme, výsledná křivka bude všude plynule navazovat, při použití případu z prvního řádku bude mít pouze větší tendenci utvářet nesouvislé struktury. V algoritmu Marching Squares však tyto nejednoznačnost budou způsobovat potíže – může dojít k tomu, že simulovaný povrch bude „roztržený“.



Obr. 9: Nerozhodnutelné případy

2.3 Marching Cubes

Algoritmus Marching Cubes dělá prakticky totéž, co algoritmus Marching Squares, pouze však

v prostoru. Tento algoritmus publikovali v článku [MC87] W. E. Lorensen a H. E. Cline již roku 1987. V USA nemohl být dlouhou dobu použit kvůli softwarovému patentu, nyní je však již volně použitelný i v USA. Dnešního čtenáře, pro kterého může být poněkud problém sehnat původní článek, odkazuji na poměrně dobrý popis v článcích [BO94] a [LI03]. Implementaci algoritmu v Delphi bez počítání normál nalezne čtenář na stránce [HO01], v článku [LI03] je sice výpočet normál implementován, ovšem nepříliš šťastným způsobem s neuspokojivými výsledky.

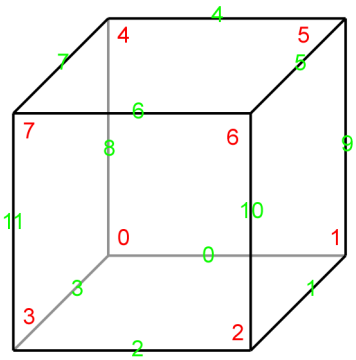
2.3.1 Algoritmus Marching Cubes bez výpočtu normál

Pusťme se do výkladu o algoritmu samém. Mějme nějakou mřížku hodnot $f_{ijk} = f(x_0 + i \cdot h, y_0 + j \cdot h, z_0 + k \cdot h)$ a hodnotu f_0, jejíž izoplochu budeme chtít zobrazit. Algoritmus pracuje následujícím způsobem:

Algoritmus Marching Cubes bez výpočtu normál

pro každý bod (i,j,k) mřížky **dělej**
    $f_{ijk} := f(x_0 + i \cdot h, y_0 + j \cdot h, z_0 + k \cdot h)$
pro každou krychli v mřížce **dělej**
    Spočítej index krychle
    Interpoluj vrcholy na aktivních hranách
    pro každý trojúhelník v tabulce trojúhelníků **dělej**
        Vykresli trojúhelník

Vzhledem k tomu, že principy jsme si již rozebrali na algoritmu Marching Squares, zaměřme se nyní spíše na konkrétní implementaci algoritmu. Podívejme se podrobněji na počítání indexu krychle. Vzhledem k tomu, že tento index budeme potřebovat k získání dat z tabulky trojúhelníků, doporučuji čtenáři držet se označení, které je použito v tomto textu, je shodné s označením v Bourkeho článku [BO94], v němž je možné najít již hotovou tabulku trojúhelníků. K počítání indexu se vrátíme ve výkladu o samotném algoritmu – zde je tento problém nastíněn, abychom si uvědomili, jak je nutné krychle a mřížku reprezentovat.



Obr. 10: Číslování vrcholů a hran

Na obrázku 10 vidíme červeně očíslované vrcholy a zeleně očíslované hrany. Index krychle určíme následujícím způsobem:

index := 0
pokud krychle.vrchol[0].hodnota < f_0 **tak** index := index or 1;
pokud krychle.vrchol[1].hodnota < f_0 **tak** index := index or 2;
pokud krychle.vrchol[2].hodnota < f_0 **tak** index := index or 4;
pokud krychle.vrchol[3].hodnota < f_0 **tak** index := index or 8;
pokud krychle.vrchol[4].hodnota < f_0 **tak** index := index or 16;
pokud krychle.vrchol[5].hodnota < f_0 **tak** index := index or 32;
pokud krychle.vrchol[6].hodnota < f_0 **tak** index := index or 64;

pokud krychle.vrchol[7].hodnota < f_0 **tak** index := index or 128;

2.3.2 Inicializace

Otázkou je, jak v praxi co nejefektivněji přiřadit hodnoty v bodech mřížky hodnotám ve vrcholech krychle. Vzhledem k tomu, že mřížka se většinou během výpočtu nemění, mění se pouze vypočtené hodnoty, je nejefektivnějším způsobem vytvořit si dvě nezávislé struktury – strukturu mřížky a strukturu krychlí. Mřížka bude trojrozměrným polem (o velikosti $M_x \times M_y \times M_z$), které musí u každého bodu uchovávat informaci o jeho poloze a hodnotě funkce v daném bodě (polohu by bylo samozřejmě možné určovat dynamicky z indexů a známé polohy mřížky, to je však zbytečně pomalé). Struktura obsahující krychle bude taktéž trojrozměrným polem, pouze o velikosti $(M_x\text{-}1) \times (M_y\text{-}1) \times (M_z\text{-}1)$, což je logické – mřížka 2 x 2 x 2 obsahuje pouze jednu krychli.

Symbolicky budu skutečnou polohu bodu mřížky na indexech [i,j,k] zapisovat jako *Mřížka[i,j,k].poloha.{zde bude v případě potřeby specifikováno x, y, z}* (k implementaci viz poznámka níže), uloženou hodnotu funkce f v bodě (i,j,k) jako *Mřížka[i,j,k].hodnota*. Krychli s „tělesovou úhlopříčkou“ [i,j,k][i+1,j+1,k+1] budu značit *Krychle[i,j,k]*, tato krychle bude mít 8 vrcholů: *Krychle[i,j,k].vrchol[0]* až *Krychle[i,j,k].vrchol[7]* (Ve skutečnosti se jedná o syntaxi Pascalu, je obdobná syntaxi v ukázkovém programu, kde jsou k implementaci použita dynamická pole). Inicializace algoritmu, který musí proběhnout před samotným průběhem algoritmu, je následující:

Deklaruj mřížku $M_x \times M_y \times M_z$ a pole krychlí $(M_x\text{-}1) \times (M_y\text{-}1) \times (M_z\text{-}1)$
Nastav polohy bodů mřížky
Přiřaď vrcholy krychle příslušným bodům mřížky

Poznámka: Výrazně čtenáři doporučuji implementovat třídu (či typ record v Pascalu) k počítání s vektory. Velmi to zjednoduší psaní programu – v první fázi vývoje by rozepsání do složek zabralo pár řádků, v pozdější fázi, kdy bude nutné počítat normály vektorovým součinem, je nanejvýše vhodné mít tuto třídu k dispozici (ale samozřejmě to není nutné, jediným účelem je pohodlí programátora).

Deklarace závisí na tom, jakým způsobem se rozhodneme mřížku a pole krychlí reprezentovat. Nastavení poloh bodů mřížky probíhá tak, že si zvolíme polohu nějakého bodu v kvádru mřížky (nejspíše bodu s indexem [0,0,0] nebo středu kvádru), zvolíme si rozlišení h, které nám udává vzdálenost bodů mřížky, nebo hrany A, B a C celého kvádru a spočítáme polohu všech bodů mřížky – jako např. v následujícím příkladu, kde udáme polohu středu krychle S a hrany A, B, C.

pro každý bod [i,j,k] mřížky **dělej**
$\quad$ Mřížka[i,j,k].***poloha***.x := $S_x + \frac{1}{2}(2\cdot i/(M_x\text{–}1) - 1)\cdot A$
$\quad$ Mřížka[i,j,k].***poloha***.y := $S_y + \frac{1}{2}(2\cdot j/(M_y\text{–}1) - 1)\cdot B$
$\quad$ Mřížka[i,j,k].***poloha***.z := $S_z + \frac{1}{2}(2\cdot k/(M_z\text{–}1) - 1)\cdot C$

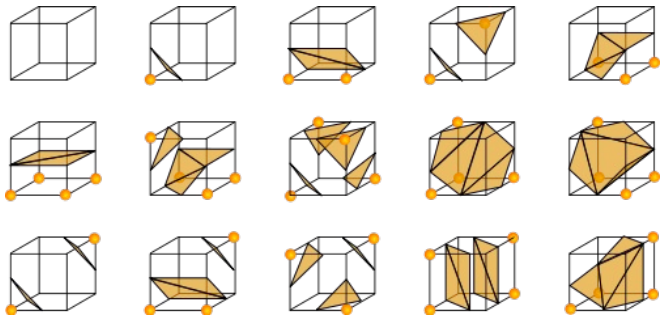
Výraz $\frac{1}{2}(2\cdot i/(M_x\text{–}1)) - 1)$ (a stejně tak zbylé dva) nabývá hodnot od $-\frac{1}{2}$ do $\frac{1}{2}$, polohy tedy probíhají

celou předpokládanou hranu A. Poznamenejme, že pokud nechceme získat deformovaný obraz, musí být A : B : C = M_x : M_y : M_z. Převodní vztahy mezi h a A, B a C snadno určíme ze zjevné podmínky $h \cdot (M_x-1) = A$ (obdobně u zbylých dvou hran).

Poznamenejme ještě, že výraz „***pro** každý bod [i,j,k]*“ *mřížky* bude pravděpodobně implementován třemi vnořenými for cykly takto:

```
for i := 0 to Mx-1 do
    for j := 0 to My-1 do
        for k := 0 to Mz-1 do Proveď něco s bodem [i,j,k]
```

Pro zjednodušení však konkrétní zápis cyklů nikde v textu nebudu vypisovat. Obdobně výraz „***pro** všechny výchozí body krychlí*“ bude symbolizovat obdobné tři cykly v mezích ještě o jedna menších.



Obr. 11: 15 možných případů rozložení vnitřních a vnějších bodů (zdroj: Wikipedie)

Mřížku tedy máme zinicializovanou, zbývá pouze zinicializovat krychle. Jelikož chceme vrcholům krychle přiřadit konkrétní body mřížky, nebudou vrcholy stejného typu, jako tyto body, budou to ukazatele na tyto body. Nechť @ je operátor, který vrací ukazatel na danou proměnnou, pak můžeme přiřazení zrealizovat takto:

```
pro každý výchozí bod (i,j,k) krychle
    Krychle[i,j,k].vrchol[0] := @Mřížka[i,j,k];
    Krychle[i,j,k].vrchol[1] := @Mřížka[i+1,j,k];
    Krychle[i,j,k].vrchol[2] := @Mřížka[i+1,j,k+1];
    Krychle[i,j,k].vrchol[3] := @Mřížka[i,j,k+1];
    Krychle[i,j,k].vrchol[4] := @Mřížka[i,j+1,k];
    Krychle[i,j,k].vrchol[5] := @Mřížka[i+1,j+1,k];
    Krychle[i,j,k].vrchol[6] := @Mřížka[i+1,j+1,k+1];
    Krychle[i,j,k].vrchol[7] := @Mřížka[i,j+1,k+1];
```

Prohlédnete-li si podrobně toto přiřazení, může se vám zdát, že neodpovídá číslovací konvenci stanovené na obr. 10. Neshoda vymizí, uvědomíte-li si, že v počítačové grafice směřuje osa y obvykle směrem vzhůru a osa z směrem „ven z obrazovky“ (pouze se na klasickou soustavu souřadnic díváme z jiného pohledu).

Poznámka: V dalším výkladu toto přiřazení budeme chápat tak, že zápis do Krychle[i,j,k].vrchol[a] je totéž jako zápis do příslušného bodu mřížky a stejně tak čtení je totéž jako čtení příslušného bodu mřížky. Konvencemi konkrétního jazyka (např. v Pascalu by bylo nutné zapisovat Krychle[i,j,k].vrchol[0]^ pro práci s hodnotou, kam směřuje ukazatel) se nebudeme zatěžovat.

2.3.3 Průběh programu

Předpokládejme, že algoritmus byl zinicializován a jednotlivým bodům mřížky byly přiřazeny hodnoty funkce f, tj. že v *Mřížka[i,j,k].hodnota* je všude správná hodnota. Vysvětleme si nejdříve jednodušší verzi bez počítání normál. Bourkeho zdrojový kód (viz [BO94] nebo soubor lookuptable.pas v ukázkovém programu) obsahuje dvě tabulky – tabulku trojúhelníků a tabulku hran příslušných k danému indexu krychle. Druhá tabulka je pouze pomocná – vzhledem k tomu, že bod na jedné hraně krychle sdílí často několik trojúhelníků, nebylo by vhodné interpolovat polohu tohoto bodu (jako je tomu v algoritmu Marching Squares, kde žádný bod nesdílí dvě úsečky) až při jeho použití. Lepší je nejdříve spočítat všechny interpolované polohy a poté již používat jen je. Zde se ovšem nabízí otázka, jak efektivně zjistit, u kterých hran vůbec máme vstoupit do funkce provádějící interpolaci; počítat interpolovanou polohu (která by samozřejmě ležela mimo krychli) u hran, na nichž neleží žádný bod vykreslovaných trojúhelníků, by bylo neefektivní. Bourke tento problém vyřešil zavedením další tabulky, tabulky hran. Vzhledem k tomu, že krychle má 12 hran, stačí nám šestnáctibitové číslo k uložení informace o tom, která hrana se podílí na vykreslování. V Bourkeho tabulce je pro každý index krychle vloženo číslo nesoucí tuto informaci – a to tím způsobem, že n-tý bit čísla je roven jedné, pokud se hrana číslo n (v souladu s konvencí na obr. 10) podílí na výpočtu.

Deklarujme tedy pole *hrany* o velikosti 12, ve kterém *hrana[n]* odpovídá hraně číslo n v Bourkeho konvenci. Obsahem prvků tohoto pole bude nějaká forma vektoru uchovávající polohu interpolovaného bodu na této hraně. Jsme ve fázi „***pro** každou krychli v mřížce*“. Předpokládejme, že index krychle jsme již určili algoritmem popsaným v úvodu. Nyní chceme na aktivních hranách (těch, na kterých leží bod nějakého v budoucnu vykreslovaného trojúhelníka) spočítat interpolovanou polohu bodu. Funkce Interpoluj(i,j,k,a,b) interpoluje polohu mezi vrcholy a a b krychle na souřadnicích [i,j,k], jak bude popsáno dále.

```
index := získaný index krychle na souřadnicích [i,j,k]      /jsme uvnitř cyklu/
bityhran := tabulka_hran[1][index]

pokud bityhran = 0 tak Přejdi k další krychli      /tato neobsahuje žádné trojúhelníky/

pokud bityhran and 1 <> 0 tak hrany[0].poloha := Interpoluj(i,j,k,0,1)   /dolní stěna/
pokud bityhran and 2 <> 0 tak hrany[1].poloha := Interpoluj(i,j,k,1,2)
pokud bityhran and 4 <> 0 tak hrany[2].poloha := Interpoluj(i,j,k,2,3)
pokud bityhran and 8 <> 0 tak hrany[3].poloha := Interpoluj(i,j,k,3,0)
pokud bityhran and 16 <> 0 tak hrany[4].poloha := Interpoluj(i,j,k,4,5)  /horní stěna/
pokud bityhran and 32 <> 0 tak hrany[5].poloha := Interpoluj(i,j,k,5,6)
pokud bityhran and 64 <> 0 tak hrany[6].poloha := Interpoluj(i,j,k,6,7)
pokud bityhran and 128 <> 0 tak hrany[7].poloha := Interpoluj(i,j,k,7,4)
pokud bityhran and 256 <> 0 tak hrany[8].poloha := Interpoluj(i,j,k,0,4) /boční hrany/
pokud bityhran and 512 <> 0 tak hrany[9].poloha := Interpoluj(i,j,k,1,5)
pokud bityhran and 1024 <> 0 tak hrany[10].poloha := Interpoluj(i,j,k,2,6)
pokud bityhran and 2048 <> 0 tak hrany[11].poloha := Interpoluj(i,j,k,3,7)
```

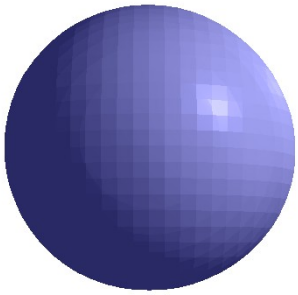
Podívejme se také na to, jak implementovat funkci Interpoluj():

1 Bourke ji nazývá EdgeTable

```
funkce Interpoluj(i,j,k,a,b) vrací vektor
        ha := Krychle[i,j,k].Vrchol[a].hodnota
        ha := Krychle[i,j,k].Vrchol[b].hodnota
        ra  := Krychle[i,j,k].Vrchol[a].poloha
        rb  := Krychle[i,j,k].Vrchol[b].poloha
        pokud ha = f0 tak
                vrať ra
        pokud hb = f0 tak
                vrať rb
        pokud ha= hb tak
                vrať ra
        jinak
                vrať (rb+((hb-f0)/(hb-ha))*(ra-rb))
```

Použití pomocných proměnných ha, hb, ***ra***, ***rb*** (vektory jsou tučně odlišeny) nebylo nutné, pouze se tím výrazně zpřehlednil zápis. Způsobů, jak volání této funkce v praxi vyřešit, je mnoho, mohou se např. předat pouze parametry a a b a ukazatel na příslušnou krychli (takovéto řešení bude rychlejší, vyžaduje méně čtení z paměti). Důvodem, proč kontrolujeme zvlášť ha = f_0 a hb = f_0 je rychlost, důvodem kontroly ha = hb je, aby nedošlo k dělení nulou.

Nyní, když známe polohy bodů na jednotlivých hranách, zbývá pouze vykreslit příslušné trojúhelníky. Bourkeho tabulka trojúhelníků je dvojrozměrné pole 256 × 16. První index určuje zjištěný index krychle, druhý index určuje vždy po třech hrany, na kterých leží vrcholy trojúhelníka. Tedy *TriangleTable[index][0]*[2], *TriangleTable[index][1]* a *TriangleTable[index][2]* tvoří informaci o prvním trojúhelníku, *TriangleTable[index][3]*, *TriangleTable[index][4]* a *TriangleTable[index][5]* tvoří informaci o druhém atd. *TriangleTable[index][15]* je vždy -1. Tuto tabulku budeme tedy číst vždy po třech údajích. Jakmile narazíme na hodnotu -1, znamená to, že jsme došli na konec seznamu trojúhelníků, a vykreslování skončí. Pole má 16 prvků, protože v algoritmu se vyskytuje nejvíce 5 trojúhelníků v jedné krychli a poslední prvek musí být -1. Všechny možné kombinace rozložení vnitřních a vnějších bodů v krychli (kromě stavu „všechny vnitřní“) můžeme vidět na obr. 11. Všechny kombinace jsou to v tom smyslu, že jakoukoliv jinou by bylo možné získat z těchto nějakou symetrií (otočením, převrácením). Přejděme k zápisu ilustrovaných myšlenek:



Obr. 12: Flat shading

```
i := 0
Pracujeme-li v OpenGL: glBegin(GL_TRIANGLES)
dokud TriangleTable[index][i] <> -1 dělej
        VykresliTrojúhelník(hrany[TriangleTable[index][i]],
                            hrany[TriangleTable[index][i+1]],
                            hrany[TriangleTable[index][i+2]])
        i := i + 3
```

2 TriangleTable je jméno, které ve svém článku pro tabulku trojúhelníků používá Bourke.

Pracujeme-li v OpenGL: glEnd()

Konkrétní implementace se liší podle jazyka a grafického rozhraní, se kterými pracujeme. Pokud pracujeme v OpenGL, je nejvýhodnější zavolat *glBegin(GL_TRIANGLES)* před cyklem vykreslujícím jednotlivé trojúhelníky a za ním zavolat g*lEnd()*, či, pokud nevykreslujeme mezitím nic jiného, je možné zavolat *glBegin(GL_TRIANGLES)* ještě před cyklem procházejícím všechny krychle a *glEnd()* až za ním. Rozepišme nyní funkci *VykresliTrojúhelník* využívající flat shading – ta nastaví všem bodům trojúhelníka stejnou normálu, díky čemuž bude mít potom v osvětlení celý trojúhelník stejnou barvu a barevné přechody nebudou plynulé (viz obrázek 12). Funkce JV() značí zde i v následujícím výkladu jednotkový vektor.

```
funkce VykresliTrojúhelník(a,b,c: prvky pole hran)
        x := c.poloha - a. poloha
        y := b.poloha - a. poloha

        normála := JV(x × y)

        glNormal3f(normála.x, normála.y, normála.z)
        glVertex3f(a.poloha.x, a.poloha.y, a.poloha.z)
        glVertex3f(b.poloha.x, b.poloha.y, b.poloha.z)
        glVertex3f(c.poloha.x, c.poloha.y, c.poloha.z)
```

Poznamenejme, že daný výpočet normály je korektní vzhledem k orientaci trojúhelníků uložených v Bourkeho tabulce. Deklarujeme-li strukturu uchovávající polohy vhodným způsobem (ve správné pořadí a se správným typem proměnných), můžeme volání funkce glVertex3f zjednodušit (a zrychlit) na volání funkce glVertex3fv(@a.**poloha**). Obdobně můžeme postupovat u normály. Postačí, deklarujeme-li strukturu, ve které uchováváme vektory, takto:

```
type vector = record
                x,y,z: GLfloat;
              end;
```

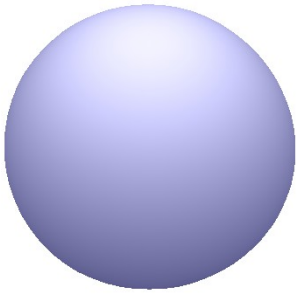
v Pascalu, resp.:

```
class vector {
        public:
                Glfloat x,y,z;
}
```

v C/C++.

Druhým možným přístupem, který můžeme použít, aniž bychom museli počítat normály, je tzv. pseudosvětlo[3] (viz obr. 13, kde je zdroj „psudosvětla“ umístěn těsně nad objektem – jedná se o tutéž kouli jako na obr. 14). Tento postup je ze všech nejméně náročný – OpenGL nemusí počítat osvětlení a náš program nemusí počítat žádné normály. Když OpenGL vypočítává osvětlení, provádí to tak, že každou barvu vypočtenou vlastnostmi materiálu a osvětlení přenásobí intenzitou danou výrazem:

$$I = \frac{1}{k_c + k_l d + k_q d^2}$$



Obr. 13: Pseudosvětlo

kde d je vzdálenost od zdroje světla. Zde použijeme přístup podobný, pevně si stanovíme polohu pseudosvětla a pak před každým voláním glVertex3f nastavíme barvu pomocí *glColor3f(I,I,I)*. Tím získáme odstíny šedi. Pokud chceme dodat určitý barevný nádech (např. modrý, jako na obrázku), přičteme k některým složkám konstantu, např. *glColor3f(I,I,I+0.1)*. Konstanty je nutné určit zkoušením vhodného nastavení pro danou scénu. Jak vidíme na obrázku, touto metodou jsme schopni získat dokonale hladký povrch i při velmi malém rozlišení mřížky; dojem z hloubky obrazu je však dosti mizerný.

2.3.4 Algoritmus Marching Cubes s výpočtem normál

Konečně jsme se dostali k nejdůležitější variantě algoritmu – té, ve které ke každému vrcholu spočteme podle určitých kritérií normálu. Inicializace algoritmu proběhne zcela stejně jako v předchozím případě. Algoritmus samotný pak k výpočtu normál potřebuje více průchodů – nejdříve spočítá normály k vykreslovaným trojúhelníkům. Ty poté pro každý bod zprůměruje, viz následující algoritmus.

Algoritmus Marching Cubes s výpočtem normál

pro každý bod (i,j,k) mřížky **dělej**
    $f_{ijk} := f(x_0 + i \cdot h, y_0 + j \cdot h, z_0 + k \cdot h)$
pro každou krychli v mřížce **dělej**
    Spočítej index krychle a ulož ho
    Interpoluj vrcholy na aktivních hranách
    pro každý trojúhelník v tabulce trojúhelníků **dělej**
        Spočítej normálu a ulož ji k příslušné hraně
pro každý vnitřní bod [i,j,k] mřížky **dělej**
    Spočti normálu na hranách [i,j,k][i+1,j,k], [i,j,k][i,j+1,k] a [i,j,k][i,j,k+1]
      zprůměrováním normál čtyř krychlí, které tyto hrany sdílí a přiřaď tuto
      normálu všem hranám podílejících se krychlí
pro všechny body [i,j,k] na povrchu mřížky **dělej**
    Spočti normály na hranách, které jsou součástí mřížky s ohledem na to, kolik
      kolik krychlí hranu sdílí, a přiřaď tuto normálu všem hranám podílejících
      se krychlí
pro každou krychli v mřížce **dělej**

3 Pojem pseudosvětlo si vymyslel autor sám, omluvte tedy nemožnost dalšího hledání termínu na internetu

Vykresli trojúhelníky

Tento princip vyžaduje mírnou úpravu struktury použité ve verzi bez výpočtu normál. U každé krychle potřebujeme po prvním průchodu uchovat její index, k tomu zavádíme proměnnou *Krychle[i,j,k].index,* a taktéž pole hran s interpolovanými polohami bodů. Toto pole již tedy nebude jen dočasnou zrychlující pomůckou při vykreslování trojúhelníků dané krychle, bude pro každou krychli uloženo v proměnné *Krychle[i,j,k].hrany*. U hrany nyní budeme uchovávat také normálu k bodu, který na ní leží, a to v proměnné *Krychle[i,j,k].hrany[a].normála*. Jak je napsáno v Bourkeho článku, je vhodné při počítání výsledné normály každého bodu provést vážený průměr normál od trojúhelníků, které bod sdílí, kde vahou má být převrácená hodnota obsahu trojúhelníka – to zajistí, že se správně zobrazí různé malé špičky, které při generování povrchu mohou vzniknout. Fakt, že vážený průměr dává skutečně lepší výsledky než průměr aritmetický, jsem sám s pozitivním výsledkem otestoval. Vzhledem k tomu, že velikost vektorového součinu vektorů určených dvěma stranami trojúhelníka je rovna dvojnásobku jeho obsahu, není nutné provádět žádný další výpočet, jednoduše vydělíme získanou normálu druhou mocninou velikosti této normály (tj. skalárním součinem normály se sebou samou), čímž zajistíme, že její délka bude nepřímo úměrná první mocnině jejího obsahu.

Předpokládejme tedy, že jsme uvnitř cyklu „***pro** každou krychli v mřížce*“, že jsou již správně nastavené hodnoty *Krychle[i,j,k].index* a všechny polohy *Krychle[i,j,k].hrany[a].poloha* a zbývá nám pouze nastavit normály. Provedeme to následovně:

```
a := 0
dokud TriangleTable[index][a] <> -1 dělej
        x := Krychle[i,j,k].hrany[TriangleTable[Krychle[i,j,k].index][a+2].poloha
                - Krychle[i,j,k].hrany[TriangleTable[Krychle[i,j,k].index][a].poloha
        y := Krychle[i,j,k].hrany[TriangleTable[Krychle[i,j,k].index][a+1].poloha
                - Krychle[i,j,k].hrany[TriangleTable[Krychle[i,j,k].index][a].poloha

        n := x × y
        n := n/(n·n)        /Pozor, že zde je myšlen skalární součin, n·n = (size(n))^2/
        Krychle[i,j,k].hrany[TriangleTable[Krychle[i,j,k].index][a]].normála := n
        Krychle[i,j,k].hrany[TriangleTable[Krychle[i,j,k].index][a+1]].normála := n
        Krychle[i,j,k].hrany[TriangleTable[Krychle[i,j,k].index][a+2]].normála := n
        a :=a + 3
```

Proměnné *Krychle[i,j,k].index* a *Krychle[i,j,k].hrany* mohou být uloženy v pomocných proměnných – ať už za účelem menšího množství požadavků na paměť či pro větší přehlednost kódu. Zde doporučuji čtenáři na chvíli se zastavit a skutečně pochopit, co tento kód dělá (přeci jen, tři vnořená pole do sebe působí poněkud odstrašujícím dojmem). Výsledkem předchozího kódu bude, že ke každé aktivní hraně bude nastavena normála, jejíž velikost bude rovna převrácené hodnotě obsahu příslušného trojúhelníka.

Nyní se dostáváme do cyklu „**pro** každý vnitřní bod [i,j,k] mřížky“. Důvodem rozdělení na vnitřní body a povrch je, že následující algoritmus by se pokusil číst hodnoty mimo rozsah pole mřížky, pokud bychom nechali indexy probíhat celou mřížkou, což by způsobilo pád programu. Pokud

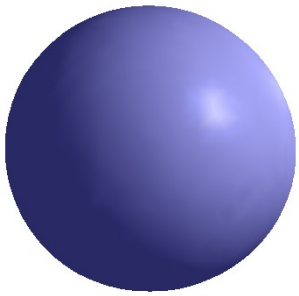
čtenáři nevadí, že se bude na okraji mřížky zobrazovat povrch špatně, nemusí implementovat vyhodnocování normál v bodech na povrchu. Je také možnost ošetřit čtení z bodů mimo mřížku podmínkami – to je ovšem poněkud pomalé. Možným řešením je také udělat mřížku ve všech směrech o 2 body větší, výpočet provést na celou mřížku, normály spočítat pouze ve vnitřních krychlích a zobrazení provést také jen na vnitřních krychlích.

Vzhledem k tomu, že „váha“ příslušné normály je uložena v její velikosti, a my potřebujeme nastavit jako normálu jednotkový vektor, vážený průměr normál spočteme tak, že je všechny vektorově sečteme a z výsledku vytvoříme jednotkový vektor. Pro každý vnitřní bod [i,j,k] tedy provedeme (***n*** je pomocná proměnná vektorového typu, JV() značí jednotkový vektor):

```
n := JV(Krychle[i,j,k].hrany[0].normála + Krychle[i,j,k-1].hrany[2].normála
        + Krychle[i,j-1,k].hrany[4].normála + Krychle[i,j-1,k-1].hrany[6].normála)
Krychle[i,j,k].hrany[0].normála := n
Krychle[i,j,k-1].hrany[2].normála := n                    /hrana [i,j,k][i+1,j,k]/
Krychle[i,j-1,k].hrany[4].normála := n
Krychle[i,j-1,k-1].hrany[6].normála := n

n := JV(Krychle[i,j,k].hrany[3].normála + Krychle[i-1,j,k].hrany[1].normála
        + Krychle[i,j-1,k].hrany[7].normála + Krychle[i-1,j-1,k].hrany[5].normála)
Krychle[i,j,k].hrany[3].normála := n
Krychle[i-1,j,k].hrany[1].normála := n                    /hrana [i,j,k][i,j,k+1]/
Krychle[i,j-1,k].hrany[7].normála := n
Krychle[i-1,j-1,k].hrany[5].normála := n

n := JV(Krychle[i,j,k].hrany[8].normála + Krychle[i,j,k-1].hrany[11].normála
        + Krychle[i-1,j,k].hrany[9].normála + Krychle[i-1,j,k-1].hrany[10].normála)
Krychle[i,j,k].hrany[8].normála := n
Krychle[i,j,k-1].hrany[11].normála := n                   /hrana [i,j,k][i,j+1,k]/
Krychle[i-1,j,k].hrany[9].normála := n
Krychle[i-1,j,k-1].hrany[10].normála := n
```

Za lomítkem je vždy napsáno, ke které hraně v řeči bodů v mřížce daný blok přísluší.

Věřím, že verzi výpočtu normál, která projde všechny hrany, u kterých nebyla spočítána normála v předchozím kódu, zvládne čtenář vypracovat sám. Jedná se o šest dvou do sebe vnořených for cyklů (jeden index je vždy konstantní), ve kterých jsou vždy vynechány hrany mimo mřížku.



Obr. 14: Smooth shading

Posledním zbývajícím krokem je zobrazení trojúhelníků. To je již triviální záležitost, stačí obdobným způsobem jako ve verzi s flat shadingem zobrazit trojúhelníky a normály nastavit na již vypočtené normály. Provedeme tedy následující kód pro všechny krychle:

```
a := 0
glBegin(GL_TRIANGLES)
dokud TriangleTable[index][a] <> -1 dělej
    glNormal3f(@Krychle[i,j,k].hrany[TriangleTable[Krychle[i,j,k].index][a]].normála)
    glVertex3f(@Krychle[i,j,k].hrany[TriangleTable[Krychle[i,j,k].index][a]].poloha)
    glNormal3f(@Krychle[i,j,k].hrany[TriangleTable[Krychle[i,j,k].index][a+1]].normála)
    glVertex3f(@Krychle[i,j,k].hrany[TriangleTable[Krychle[i,j,k].index][a+1]].poloha)
    glNormal3f(@Krychle[i,j,k].hrany[TriangleTable[Krychle[i,j,k].index][a+2]].normála)
    glVertex3f(@Krychle[i,j,k].hrany[TriangleTable[Krychle[i,j,k].index][a+2]].poloha)
    a := a + 3
glEnd()
```

Na obr. 14 Si můžete prohlédnout stejnou kouli jako na obrázcích 14 a 16 s normálami spočítanými předchozím algoritmem. Jak vidíte, oproti flat shadingu se jedná o obrovský rozdíl v kvalitě obrazu.

2.3.5 Problémy algoritmu Marching Cubes

Algoritmus Marching Cubes má své nedostatky. Hlavním problémem jsou nejednoznačnosti, které vznikají ze stejných důvodů jako u algoritmu Marching Squares. Zde však mají závažnější důsledky – nachází-li se dva nejednoznačné případy vedle sebe, může dojít k tomu, že výsledný povrch bude obsahovat díry. Čtenáře, který by chtěl tyto problémy ve své aplikaci ošetřit, odkazuji na článek [LVT03], kde je metodou rozdělení nejednoznačných případů na podpřípady dosaženo zachování požadovaných topologických vlastností zobrazovaného povrchu.

2.3.6 Metaballs

Na obrázku 1 v úvodu textu jsou zobrazeny tzv. metaballs. Zmiňme stručně, o co se jedná. Nejde o nic jiného než o izoplochu jisté speciálně definované funkce. Hodnota této funkce je závislá pouze na vzdálenosti od předem specifikovaných bodů. Jsou-li tyto body dostatečně daleko od sebe, je danou izoplochou sjednocení několika disjunktních koulí. Když se body k sobě přibližují, vzniká dojem, že se koule v jistém smyslu propojují.

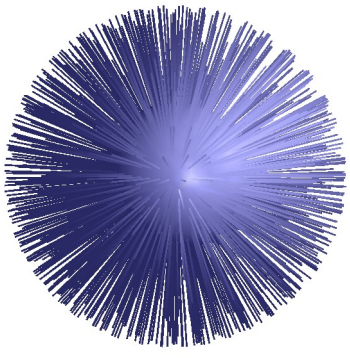
Pokud chcete naprogramovat metaballs, stačí, použijete-li vysvětlený algoritmus Marching Cubes s funkcí:

$$f(x,y,z)=\sum_P \frac{R_P^2}{(x-P_x)^2+(y-P_y)^2+(z-P_z)^2}$$

kde P probíhá přes všechny metaballs a R je poloměr metaballu. f_0 Zvolte 1.

2.3.7 Ježura

Na úplný závěr se zmiňme ještě o jednom efektu, kterého je možné s algoritmem Marching Cubes dosáhnout. Pokud si zobrazíme (jednotkové) normály k jednotlivým bodům jako úsečky se zachováním materiálu povrchu, dostaneme efekt jakési „ježury“. Ten je obzvláště zajímavý, použijeme-li ho při zobrazování kapalin – povrch



Obr. 15: „Ježura"

kapaliny se dynamicky mění a „ježura“ vypadá jako zvláštní chomáček měkkého materiálu.

3 Simulace kapalin

3.1 Úvod

V této části naleznete vysvětlení, jak v praxi simulovat kapaliny. Čtenáře, kteří jsou schopni číst v anglickém jazyce, odkazuji na článek [PVFS05], ve kterém naleznou stručnější popis problematiky a ze kterého tato práce čerpá. Nenaleznete tam však žádné detaily týkající se implementace ani vyřešenou interakci více kapalin.

Způsob, jakým bude kapalina vizualizována za pomoci algoritmu Marching Cubes bude také popsán. Konkrétní „detaily“ týkající se zobrazení – např. zobrazování povrchu kapaliny jakožto částečně průhledného materiálu, který láme a odráží okolní světlo – však nebudou implementovány; tato problematika je velmi rozsáhlá a je nad rámec tohoto textu. Zájemce, kteří chtějí tento typ vizualizace použít, odkazuji na článek [WA01], jenž popisuje principy ray tracingu, a na webový tutoriál [BI05], který dobře vysvětluje nejen teorii, ale i implementaci ray tracingu v C++. Čtenář, který se spokojí pouze s „neinteraktivním“ zobrazováním za použití dalšího softwaru, najde potřebné informace v páté kapitole.

3.2 Principy

Dá se říci, že prakticky všechny metody simulace kapalin vycházejí z Navier-Stokesových rovnic (více o nich naleznete v následující kapitole). Přístup k jejich numerickému řešení můžeme rozdělit v zásadě do dvou druhů: eulerovské[4] simulace, ve kterých máme předem definovanou mřížku v prostoru a podle určitých kritérií počítáme změny tlaku a rychlosti kapaliny v bodech této mřížky, a lagrangeovské simulace, ve kterých určujeme dané vlastnosti v konkrétních částicích kapaliny, není zde tedy žádná pevná mřížka, částice jsou unášeny pohybem kapaliny. Langrangeovský přístup je ten, který v následujícím výkladu budeme používat, vzhledem k jednoduché extrakci povrchu a (na pohled) realistickému chování je ideální k realtimeové simulaci.

3.2.1 Smoothed Particle Hydrodynamics

SPH je přístup, v němž jsou počítány interakce mezi jednotlivými částicemi metodou, jež v principu vyhovuje Navier-Stokesovým rovnicím, zavádí však tzv. smoothing kernel (dále ho budeme nazývat pouze jádro), na nějž jsou sice kladeny určité požadavky, jeho volba je však víceméně empirická. Toto jádro hraje důležitou roli při výpočtu hustoty a tlaku v kapalině (matematické principy, jimiž se tyto výpočty řídí, naleznete v kapitole 4). Důležitou vlastností jádra, jíž budeme využívat, je, že za interakčním poloměrem h je nulové. To znamená, že částice se vzájemně ovlivňují pouze na vzdálenost h.

3.2.2 Reprezentace dat

V následujícím výkladu zavedeme značení:

Částice[i] … i-tá částice
Částice[i].$\boldsymbol{R}$ … poloha i-té částice
Částice[i].$\boldsymbol{v}$ … rychlost i-té částice

4 Mnoho autorů píše „eulerovské“ s velkým E, já k tomu však z gramatického hlediska nevidím důvod.

Částice[i].***MinuléR*** … poloha i-té částice před aplikováním viskozity, pružin apod.
Částice[i].***PůvodníR*** … poloha i-té částice před začátkem kroku simulace

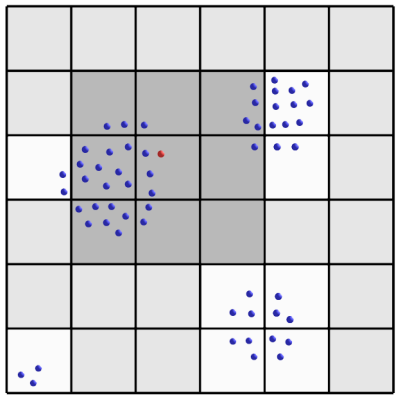
Pružina[i,j] … pružina mezi částicemi s indexy i a j
Pružina[i,j].L … délka pružiny

V ukázkovém programu je pole částic implementováno jako dynamické pole, jehož velikost se mění s tím, jak se vytváří či odstraňují částice. *Pružina* je dvourozměrné dynamické pole, jeho velikost je vždy n × n, kde n je počet částic. O jeho významu bude pojednáno v sekci o elasticitě a plasticitě.

Pokud je vaším cílem implementovat interakci více kapalin, výrazně vám doporučuji zavést si nadřazenou strukturu, která v sobě bude obsahovat pole částic příslušné kapaliny, pole pružin, hodnoty konstant příslušných ke kapalině a další potřebné vlastnosti. Jednotlivé funkce zajišťující např. viskozitu, posunutí důsledkem pružin apod. Pak volejte s parametrem určujícím, která struktura kapaliny se má použít. Nejrozumnější je dle mého názoru předávat v parametru ukazatel na strukturu, resp. předávat parametr jako „var“ v Pascalu.

3.2.3 Vyhledávání sousedů

Typickým úkonem v následujících odstavcích bude „***pro** všechny sousedy částice i proved**“***, kde sousedem částice i se míní všechny částice, jejichž vzdálenost od částice i je menší než interakční poloměr h. Nejjednodušší možností, jak projít všechny sousedy, je projít všechny částice, spočítat jejich vzdálenost od dané částice a částice, jejichž vzdálenost je větší než h, ignorovat. Vzhledem k tomu, že vyhledávání sousedních částic musíme typicky provést pro každou částici, roste náročnost takovéhoto vyhledávání kvadraticky s počtem částic, při velkém počtu částic se tak simulace stává prakticky neupočitatelnou. Řešením je pokrýt prostor, ve kterém se mohou nacházet částice, krychlemi o hraně h (upozorňuji, že tyto krychle nemají nic společného s algoritmem Marching Cubes). Každá krychle bude uchovávat informaci o tom, které částice se v ní nachází. Zapsání této informace je možné zvládnout v lineárním čase, stačí projít všechny částice. Na obr. 16 můžete vidět 2D verzi takovéto mřížky. Chceme-li projít všechny sousedy červeně zbarvené částice, stačí, projdeme-li částice v tmavě šedivě zbarvených čtvercích. Šedivě jsou zbarveny čtverce, které žádnou částici neobsahují, nemusí být tedy pro ně v principu alokována žádná paměť. Takto dynamickou strukturou se nyní nezabývejme, řekněme, že krychle jsou trojrozměrným polem o rozsahu -A .. B, kde A a B si stanovíme pro příslušný rozměr podle předpokládaných maximálních rozměrů scény (pro programátory v jazycích s Céčkovou syntaxí polí jistě nebude problém si indexy „posunout“).

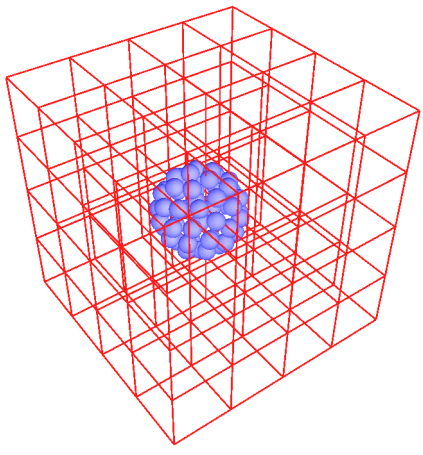


Obr. 16: Mřížka pro hledání sousedů

Provést nějakou část zdrojového kódu pro částice v okolí konkrétní částice vyžaduje projít 27 krychlí v prostoru (sousedi každé částice se nachází v okolí o velikosti 3 × 3 × 3 krychle). Definujme pole krychlí tak, že na každém indexu [x,y,z] se nachází dynamické pole *částice* obsahující indexy částic nacházejících se v dané krychli. Vykonání kódu *kód* pak bude vyžadovat projití 27 for cyklů. Poznamenejme, že *length()* je funkce vracející délku dynamického pole a že pole krychlí si označme *KrychleS[x,y,z]*, aby nám značení nekolidovalo se značením krychlí ve výkladu algoritmu Marching Cubes.

```
x := floor(Částice[i].R.x);          /Tímto kódem zjistíme indexy krychle,
y := floor(Částice[i].R.y);          ve které se nachází částice/
z := floor(Částice[i].R.z);

for c := 0 to length(KrychleS[x-1,y-1,z-1].částice)-1 do begin
        j := KrychleS[x-1,y-1,z-1].částice[c];
        kód;
end;
for c := 0 to length(KrychleS[x-1,y-1,z].částice)-1 do begin
        j := KrychleS[x-1,y-1,z].částice[c];
        kód;
end;
for c := 0 to length(KrychleS[x-1,y-1,z+1].částice)-1 do begin
        j := KrychleS[x-1,y-1,z+1].částice[c];
        kód;
end;
for c := 0 to length(KrychleS[x-1,y,z-1].částice)-1 do begin
        j := KrychleS[x-1,y,z-1].částice[c];
        kód;
end;
…
for c := 0 to length(KrychleS[x+1,y+1,z+1].částice)-1 do begin
        j := KrychleS[x+1,y+1,z+1].částice[c];
        kód;
end;
```

V ukázce nejsou vypsány všechny for cykly, princip je zřejmý – je nutné projít všechny kombinace indexů x-1, x, x+1 s y-1, y, y+1 a z-1, z, z+1. Index příslušné částice je vždy uložen v pomocné proměnné j – v kódu je totiž poměrně často využíván a vypisovat vždy KrychleS[x,y,z][c] by nebylo z programátorského hlediska efektivní.



Obr. 17: Procházené krychle

Je zřejmé, že vypisovat 27 for cyklů pokaždé, když chceme pracovat se sousedními částicemi, je nesmyslně zdlouhavé. Tento problém je možné vyřešit makry kompilátoru – definujeme makro, které obsahuje oněch 27 for cyklů a obsahuje parametr *kód* (ten může být např. uložen v jiném makru, pokud preprocesor neumí pracovat s parametry maker). Pokaždé tak zapíšeme pouze dané makro s příslušným parametrem. Na obr. 17 si můžete prohlédnout soustavu částic a všechny krychle, které byly při vyhledávání sousedů „navštíveny“. Je-li to tedy osamocená kapka, nebude snižovat výkon při výpočtu vzdálenějších skupin částic.

3.2.4 Krok simulace

Simulace funguje na principu schématu předpověď-zachování (prediction-relaxation). V tomto

schématu jsou nejdříve spočítány posuvy jednotlivých částic na základě aktuální rychlosti a pružin a na tuto polohu je aplikována funkce zachování hustoty. Z této nové polohy je pak spočtena rychlost částice pro další krok simulace. Díky tomuto přístupu zůstává hustota částice velmi dobře zachována chování systému částic je stabilní i pro poměrně velké časové kroky. Jelikož posunutí vlivem zachování hustoty je provedeno ještě před výpočtem nové rychlosti částice, má také vliv na vypočtenou rychlost. Díky tomu působí chování částic přirozeně.

Dříve, než přistoupíme k výkladu algoritmu samotného, rozeberme jednu funkci – uložení polohy částic – ta se totiž bude v algoritmu opakovat několikrát. Poznamenejme, že funkce *SetLength* je Pascalovská funkce nastavující délku (počet prvků) dynamického pole.

```
funkce Ulož polohu všech částic
        /Nastavíme délku pole částic u všech krychlí na 0, jinak bychom mohli
        počítat „duchy“/
        pro každý index [m,n,o] KrychleS dělej
                SetLength(KrychleS[m,n,o].částice,0);
        pro každou částici  i  dělej
                x := floor(Částice[i].R.x/h);          /Nejdříve zařadíme částici ke
                y := floor(Částice[i].R.y/h);          krychli/
                z := floor(Částice[i].R.z/h);

                SetLength(KrychleS[x,y,z].částice,length(KrychleS[x,y,z].částice)+1)
                KrychleS[x,y,z][length(KrychleS[x,y,z].částice)-1] := i

                Částice[i].MinuléR :=  Částice[i].R     /A uložíme polohu částice/
```

Samotný algoritmus je následující:

Výpočetní krok simulace částic jedné kapaliny

```
1.  pro každou částici  i  dělej
2.          Částice[i].v := Částice[i].v + dt·g          /Aplikujeme gravitaci/
3.  Aplikuj viskozitu                                   /Viz sekce 3.2.6/
4.  pro každou částici  i  dělej
5.          Částice[i].PůvodníR :=  Částice[i].R        /Uložíme polohu částice/
6.          Částice[i].R := Částice[i].R + dt·Částice[i].v      /Běžný posun/
7.  Ulož polohu všech částic
8.  Přepočítej pružiny                                  /Viz sekce 3.2.7/
9.  Zapůsob pružinami na částice
10. Ulož polohu všech částic
11. Aplikuj zachování dvojí hustoty                     /Viz sekce 3.2.5/
12. Ulož polohu všech částic
13. Vyřeš kolize s objekty                              /Viz sekce 3.2.8/
14.  pro každou částici  i  dělej
```

15. Částice[i].***v*** := (Částice[i].***R***-Částice[i].***PůvodníR***)/dt

„dt“ značí v celém algoritmu časový krok – čím větší dt bude, tím rychleji bude simulace probíhat. „dt“ je tedy možné použít k regulaci rychlosti simulace v závislosti na tom, jak rychle se zvládá scéna vykreslovat. Možnosti však nejsou neomezené, při velkém časovém kroku začne být simulace nestabilní. V kombinaci s ostatními konstantami se mi osvědčilo jako největší dt = 0,5, při němž je simulace absolutně stabilní.

Na řádcích 1 a 2 aplikujeme zrychlení. Ve skutečnosti nezáleží na tom, zda ho aplikujeme až po aplikování viskozity nebo před ní, jelikož viskozita je závislá jen na vzájemné rychlosti všech částic, tedy změna rychlosti všech částic o stejnou hodnotu nebude mít na viskozitu vliv.

Na řádku 3 aplikujeme viskozitu, ta ovlivňuje pouze rychlosti částic (tedy ne jejich polohy), proto ukládání polohy částic může proběhnout až poté (ale není to nutné).

Na řádku 5 uložíme současnou polohu všech částic do jiné proměnné, tuto polohu pak využijeme k výpočtu nové rychlosti.

Na řádku 6 posuneme částice na základě jejich současné rychlosti. Je rozumné posouvat částice až po aplikování viskozity, protože jinak by jinak u velmi viskózních kapalin poněkud ztratila efekt.

Na řádku 7 uložíme polohu částic. Důvodem je to, že v každé fázi je vhodné počítat posuny na základě poloh stanovených v předchozí fázi. Jelikož v principu nemůže záležet na pořadí, v jakém jsou částice uložené v paměti, musí být v dané fázi všechny posuny počítány z polohy uložené před začátkem každé fáze.

Na řádcích 8 až 9 aplikujeme posunutí na základě pružin, ty slouží k simulaci elastických a plastických jevů.

Na řádku 11 aplikujeme zachování hustoty. Právě tato funkce zajišťuje to, že se částice chovají jako kapalina. Ve skutečnosti je to jediná nutná součást simulace, částice se budou chovat jako kapalina i bez aplikace viskozity a pružin.

Na řádku 13 jsou řešeny kolize s objekty. Je zřejmé, že je správné řešit kolize s objekty až po aplikování všech ostatních posunutí, na konci simulačního kroku by se částice neměly nacházet uvnitř objektu.

Na řádcích 14 a 15 spočítáme rychlost všech částic na základě změny jejich polohy v simulačním kroku.

Jeden simulační krok samozřejmě nemusí odpovídat jednomu zobrazenému snímku. Pokud chceme dosáhnout při rychlosti 30 snímků za sekundu rychlosti, při které se simulovaná kapalina chová přibližně jako skutečná voda, je potřeba vypočítat mnoho snímků za sekundu, řádově 10. Vzhledem k tomu, že většinu výpočetního času zabírá výpočet povrchu kapaliny, není počítání mnoha kroků v každém snímku nikterak velkou ztrátou výkonu.

3.2.5 Zachování dvojí hustoty

K zachování hustoty použijeme přístup použitý v článku [PVFS05]. Standardní zachování hustoty funguje tak, že nejdříve spočteme pseudohustotu v bodě, kde se nachází daná částice na základě příspěvků od částic v jejím dosahu (ve vzdálenosti menší než h):

$$\rho_i = \sum_j (1 - \frac{r_{ij}}{h})^2 \tag{3.1}$$

kde j probíhá přes všechny sousedy částice i (pro hledání sousedů viz sekce 3.2.3), které mají menší vzdálenost od částice i než h a r_{ij} značí vzdálenost částice i a částice j. Tato hustota není skutečnou fyzikální hustotou, je to pouze vlastnost dobře charakterizující hustotu v daném bodě (ve skutečnosti se jedná o tzv. smoothing kernel, viz následující kapitola). Z této pseudohustoty spočteme pseudotlak:

$$P_i = k \cdot (\rho_i - \rho_0) \tag{3.2}$$

ρ_0 je pseudohustota, které chceme dosáhnout (kterou chceme zachovat). k je konstanta určující, jak velký tlak daný rozdíl hustot vyvolá. Pokud je $\rho_i > \rho_0$, je pseudotlak kladný a částice se snaží posunout směrem z tohoto bodu. Je-li $\rho_i < \rho_0$, je pseudotlak záporný a částice se snaží posunout směrem do tohoto bodu. Posunutí vytvoření tímto tlakem je pak pro každou dvojici částic dáno vzorcem:

$$\boldsymbol{A_{ij}} = -dt^2 P_i (1 - \frac{r_{ij}}{h}) \hat{\boldsymbol{r}}_{\boldsymbol{ij}} \tag{3.3}$$

kde $\hat{\boldsymbol{r}}_{\boldsymbol{ij}}$ je jednotkový vektor směřující od částice i do částice j. Částice i se posune o $\boldsymbol{A_{ij}}$, částice j o $-\boldsymbol{A_{ij}}$. To odpovídá výše zmíněné úvaze o pseudotlaku (při velké hustotě se částice odpuzují, při malé přitahují). Tlak je ještě škálován funkcí $-2(1 - \frac{r_{ij}}{h})$ (blíží-li se vzdálenost částic interakčnímu poloměru, částice na sebe přestávají působit). Tato „škálovací“ funkce je rovna derivaci jádra, které nám definuje hustotu (více viz následující kapitola). Síla musí být integrována dvakrát, abychom dostali změnu polohy, proto je zde časový krok zastoupen vzorcem $dt^2/2$. Dohromady tak získáme výsledek daný vzorcem 3.3.

Zachování hustoty v této podobě ovšem vede k neblahým efektům, např. ke snaze vytvářet samostatné shluky částic. Proto zavedeme tzv. blízkostní hustotu a blízkostní tlak. Blízkostní hustota bude podobná jako pseudohustota, pouze použijeme jiné jádro. Blízkostní tlak bude ovšem vždycky kladný, působící síly tedy budou pouze odpudivé, což zabrání vznikům neblahých efektů. Pseudohustotu určíme následovně:

$$\rho_{b\,i} = \sum_j (1 - \frac{r_{ij}}{h})^3 \tag{3.4}$$

Blízkostní tlak je přímo úměrný blízkostní hustotě:

$$P_{b\,i} = k_b \rho_{b\,i} \tag{3.5}$$

Posunutí je dáno vztahem (škálovací jádro je opět úměrné derivaci jádra hustoty, všechny konstanty jsou zahrnuty v konstantě k_b):

$$\boldsymbol{B_{ij}} = -dt^2 P_{b\,i} (1 - \frac{r_{ij}}{h})^2 \hat{\boldsymbol{r}}_{\boldsymbol{ij}} \tag{3.6}$$

Celkové posunutí je dáno součtem obou pousunutí ze vzorců 3.3 a 3.6:

$$\boldsymbol{D_{ij}} = \boldsymbol{A_{ij}} + \boldsymbol{B_{ij}} = -dt^2 (P_i (1 - \frac{r_{ij}}{h})^2 + P_{b\,i} (1 \frac{r_{ij}}{h})) \hat{\boldsymbol{r}}_{\boldsymbol{ij}} \tag{3.7}$$

Na částici i je aplikováno posunutí $\boldsymbol{D_{ij}}$, na částici j posunutí $-\boldsymbol{D_{ij}}$. Vzhledem k tomu, že posunutí je pro obě částice stejné a probíhá ve směru jejich spojnice, zachovává se hybnost i moment

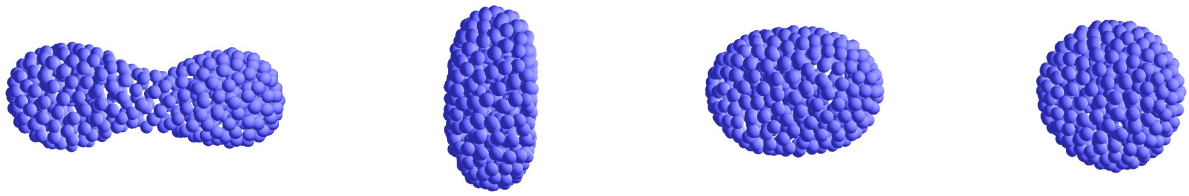
hybnosti. Vše shrnuje následující algoritmus:

Zachování dvojí hustoty

```
pro každou částici i dělej
        ρ := 0
        ρb := 0

        pro každou částici j sousedící s i dělej
                q := | Částice[j].MinuléR – Částice[i].MinuléR | / h
                pokud q < 1 tak
                        ρ := ρ + (1-q)^2
                        ρb := ρb + (1-q)^3

        P := k·(ρ- ρ0)
        Pb := kb ρb

        dr := 0
        pro každou částici j sousedící s i dělej
                q := | Částice[i]. MinuléR – Částice[j].MinuléR | / h
                pokud q < 1 tak
                        D := dt^2·(P·(1-q)+Ps·(1-q)^2)·(Částice[j]. MinuléR –
                                                        Částice[i].MinuléR )
                        Částice[j].R := Částice[j].R + D/2
                        dr := dr – D/2
        Částice[i].R := Částice[i].R + dx
```

Jako hodnoty konstant se mi osvědčilo $k = 0{,}004$, $k_b = 0{,}01$.

Zachování dvojí hustoty poskytuje také jeden přirozený efekt – povrchové napětí. Jelikož částice, které jsou dále, jsou málo ovlivněny odpudivým blízkostním tlakem (díky jeho kvadratickému jádru) a více ovlivněny přitažlivým zachováním hustoty, jsou „nasávány“ dovnitř kapky. Částice díky tomu formují koule, při menším počtu částic jsou stabilní i útvary ve tvaru disků. Přiblíží-li se k sobě dvě kapky, spojí se a vznikne jedna kapka oscilující (díky povrchovému napětí) kolem těžiště soustavy (viz obr. 18). Jak vidíme, je tedy možné vytvořit realisticky vypadající simulaci i jen na základě algoritmu zachování dvojí hustoty.



Obr. 18: Fáze oscilující kapky

3.2.6 Viskozita

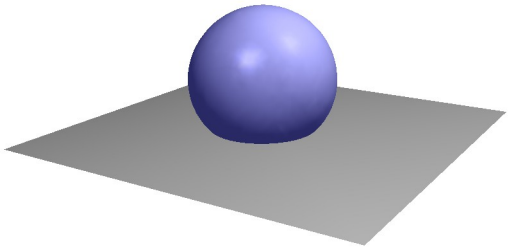
Viskozita způsobuje zpomalení vnitřního pohybu částic kapaliny. Interagují spolu vždy dvě částice v závislosti na tom, jakým směrem vzhledem ke své spojnici se pohybují, pohybují-li se kolmo ke své spojnici nebo od sebe, žádné síly nepůsobí, pohybují-li se přímo proti sobě (po své spojnici), je síla (změna rychlosti) největší. Jejich „vzájemnou rychlost“ spočteme skalárním součinem rozdílu jejich rychlostí s jednotkovým vektorem ve směru jejich spojnice. Velikost udělené hybnosti (rychlosti) je škálována lineárním jádrem a je závislá na první a druhé mocnině vzájemné rychlosti. Viz následující algoritmus:

Algoritmus viskozity

```
pro každý sousedící pár  i j  (tj. pro každé i najdi sousedící j < i) dělej
        q := | Částice[j].R – Částice[i].R | / h
        pokud q < 1 tak
                r := JV(Částice[j].R – Částice[i].R)
                u := (Částice[j].v – Částice[i].v)·r        /Pozor, jedná se o skalární
                pokud u > 0 tak                            součin dvou vektorů/
                        I := dt·(1-q)·(σ·u+β·u²)r
                        Částice[i].v := Částice[i].v – I/2
                        Částice[j].v := Částice[j].v + I/2
```

Konstanty σ a β používám různé podle toho, jak vazkou kapalinu potřebuji simulovat. Pokud mají mít znatelný efekt (např. k donucení simulované kapaliny, aby se chovala jako med), je potřeba je nastavit v řádech 0,1 až cca 3. Při větších konstantách již kapalina prakticky přestává téci.

3.2.7 Elasticita a plasticita



Obr. 19: Elastický míč

K simulaci elastického a plastického chování využijeme „pružin“ vytvořených mezi jednotlivými částicemi. U dokonale elastické deformace bychom použili pružiny, jejichž délka zůstává konstantní a při jakémkoliv vychýlení z „klidové“ polohy se snaží těleso vrátit zpět do původní stavu (takovýto dokonale elastický míč si můžete prohlédnout na obr. 19 - míč je v klidové poloze mírně zdeformován vlivem gravitace, nerozplizne se však do kapky, drží si svůj téměř perfektní kulový tvar). Pokud budeme simulovat plasticitu, bude se pružina chovat elasticky jen do určitého prodloužení, poté se začne deformovat. Posunutí je úměrné $L_{ij} - r_{ij}$, kde L_{ij} je aktuální délka pružiny mezi částicemi i a j a r_{ij} je vzdálenost částic. Když jsou částice blízko, tento faktor je kladný a pružina se snaží částice roztáhnout, když jsou moc daleko faktor je záporný a pružina se snaží stáhnout. Dále je posunutí pružinami škálováno jádrem $1 - L/h$, které zajistí, že dlouhé pružiny mají malý vliv (pružiny jsou vždy kratší než h). Zde použijeme lehce jiný přístup než v [PVFS05]. Tam jsou pružiny implementovány jako seznam objektů pružin. To má tu výhodu, že při procházení pružin za účelem aplikace posunutí na částice je potřeba pouze lineární čas. Při upravování pružin je ovšem nutné projít ke každé částici sousední částice a vyhledat příslušné pružiny, tím se dostáváme ke kubické časové složitosti. Já použiji přístup, kdy budu ukládat délku pružiny mezi částicemi v poli *Pružina[i,j]* a budu poté procházet všechny sousední částice a zjišťovat vlastnosti mezi nimi. Tak dostanu dva algoritmy s kvadra-

tickou složitostí. Samotný algoritmus posunutí na základě pružin je následující:

Posunutí pružinami

pro všechny sousední částice i j **dělej**
r := | Částice[j].***MinuléR*** – Částice[i].***MinuléR*** |
pokud r < h **a** Pružina[i,j] > 0 **tak** */pokud pružina existuje/*
D := $dt^2 \cdot k_p \cdot (1\text{-Pružina[i,j]/h}) \cdot (\text{Pružina[i,j]-r}) \cdot JV(\text{Částice[j].}\boldsymbol{MinuléR} - \text{Částice[i].}\boldsymbol{MinuléR})$
Částice[i].***R*** := Částice[i].***R*** – ***D***/2
Částice[j].***R*** := Částice[j].***R*** + ***D***/2

Toto posunutí probíhá ve směru spojnice částic, čímž je zachována celková hybnost i moment hybnosti. Před samotným posunem je ale nutné pružiny přepočítat. Při tomto procesu se deformují pružiny, jejichž délka je více než γ-krát menší či větší než jejich klidová délka. Kdyby se pružina deformovala okamžitě při jakékoliv výchylce, bylo by prodloužení pružiny dáno vztahem $\Delta L = dt\,\alpha\,(r - L)$. Ona se ovšem deformuje až po dosažení délky $(1+\gamma)L$, což zohledníme vzorcem $\Delta L = dt\,\alpha\; sgn(r-L)\cdot max(0, |r-L| - \gamma L)$. Vše shrnuje následující algoritmus:

Přepočítání pružin

pro každý pár částic i j (pro každý pár pouze jednou, tj. i < j) **dělej**
r := | Částice[j].***R*** – Částice[i].***R*** |
q := r/h
pokud Pružina[i,j] = 0 **tak** */pokud pružina neexistuje, přidáme*
Pružina[i,j] := h *pružinu s klidovou délkou h/*
d := γPružina[i,j] */určitou deformaci tolerujeme/*
pokud r > Pružina[i,j] + d **tak** */pokud jsou částice moc daleko/*
Pružina[i,j] := Pružina[i,j] + dt α (r-Pružina[i,j]-d) */natáhneme/*
pokud r < Pružina[i,j] – d **tak** */pokud jsou částice moc blízko/*
Pružina[i,j] := Pružina[i,j] – dt α (Pružina[i,j]-d-r) */stlačíme/*

pokud Pružina[i,j] > h **tak** */pokud je pružina moc dlouhá/*
Pružina[i,j] := 0 */odebereme ji/*

3.2.8 Kolize s objekty

V této sekci předpokládáme, že čtenář již má implementovaný systém vzájemných kolizí objektů, ten nebude nijak rozebírán. Implementace řešení kolizí částic s objekty také nebude řešena, budou pouze nastíněny principy algoritmu.

Nechť má každé těleso přiřazenu svou rychlost **v** a úhlovou rychlost **ω**. U každé částice potřebujeme znát její kolmou vzdálenost k povrchu tělesa a směr kolmé spojnice částice a tělesa (tj. normálu tělesa v místě, od kterého měříme vzdálenost). Vzhledem k počtu částic je vhodné ukládat tyto informace v nějaké mřížce a posléze interpolovat.

Podívejme se na algoritmus řešící kolize částic s objekty:

Řešení kolizí s objekty

pro každé těleso **dělej**
    Ulož původní pozici a orientaci
    Aplikuj posunutí a rotaci na základě rychlosti **v** a úhlové rychlosti **ω**
    Vyprázdni buffery sil a momentů sil
    pro každou částici uvnitř tělesa **dělej**
        Spočítej hybnost (rychlost) kolize **I**
        Přidej příspěvek **I** k bufferům sil a momentů sil
pro každé těleso **dělej**
    Uprav **v** a **ω** na základě sil a momentů sil
    Pohni tělesem z původní pozice a orientace na základě **v** a **ω**
Vyřeš kolize a doteky objektů
pro každou částici uvnitř tělesa **dělej**
    Spočítej kolizní hybnost (rychlost) **I**
    Aplikuj **I** na částice
    Pokud je částice stále uvnitř tělesa, přesuň ji kolmo k povrchu objektu

Relativní rychlost částice vzhledem k objektu je dána vztahem:

$$\bar{\boldsymbol{v}} = \boldsymbol{v}_i - \boldsymbol{v}_T \tag{3.8}$$

kde $\boldsymbol{v}_i$ je rychlost i-té částicé, $\boldsymbol{v}_T$ je rychlost bodu tělesa v místě, od kterého měříme vzdálenost částice. Rychlost rozdělíme do tečné a normálové složky ($\hat{\boldsymbol{n}}$ značí jednotkový normálový vektor k povrchu tělesa):

$$\bar{\boldsymbol{v}}_n = (\bar{\boldsymbol{v}} \cdot \hat{\boldsymbol{n}})\,\hat{\boldsymbol{n}} \tag{3.9}$$

$$\bar{\boldsymbol{v}}_t = \bar{\boldsymbol{v}} - \bar{\boldsymbol{v}}_n \tag{3.10}$$

Impuls počítaný v algoritmu zcela ruší normálovou složku a částečně i tangenciální na základě parametru η. Ten zajišťuje tření. Pro $\eta = 0$ tření neexistuje, pro $\eta = 1$ je absolutní. Dodaný impuls **I** je dán vzorcem:

$$\boldsymbol{I} = \boldsymbol{v}_n - \eta\ \boldsymbol{v}_t \tag{3.11}$$

Pokud by při výpočtu nastala situace, že i po provedené všech změn je částice stále uvnitř tělesa, je jednoduše přesunuta na jeho povrch bez změny hybnosti (tato situace by neměla nastávat často). Tohoto přístupu můžeme využít také u rovinných nehybných objektů – např. u podlahy, která se nachází v celé scéně. Jednoduše projdeme všechny částice, zjistíme, zda jejich souřadnice (u podlahy zpravidla y) přesahuje nějakou hranici (u podlahy zpravidla y menší než „výška“ podlahy) a pokud ano, nastavíme částici nulovou rychlost v kolmém směru (případně i aplikujeme tření) a částici přesuneme přesně na povrch hranice (podlahy). Tento způsob je velmi efektivní, vzhledem k tomu, že není nutné nic dynamicky propočítávat a projití částic je možné provést v lineárním čase.

3.2.9 Přilnavost

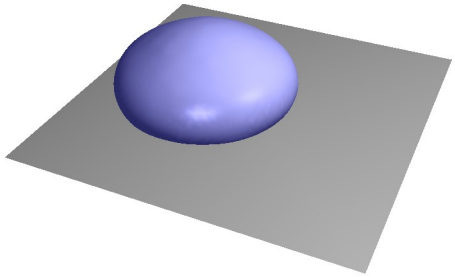
Kolize s objekty tak, jak byla popsána v předchozí sekci, předává mezi objekty částicemi

hybnost, jako kdyby se jednalo o dva pevné objekty. Částice, která narazí do objektu, tedy spadne bez jakéhokoliv odporu. Reálné kapaliny jsou ale přilnavé. Princip, jakým do algoritmu zahrneme přilnavost, je jednoduchý. Upravíme rychlost udělenou částicím v algoritmu řešení kolizí tak, aby směřovala směrem k objektu. Částice je přitahována k povrchu, pokud je její vzdálenost od povrchu menší než d_p, což je konstanta určující, na jakou vzdálenost přilnavost působí. Aby nedocházelo k chybám, mělo by být d_p poměrně malé vůči h. Nechť d značí vzdálenost mezi objektem a částicí, $\hat{\boldsymbol{n}}$ normálový vektor k povrchu tělesa. K $\boldsymbol{I}$ přičteme následující výraz:

$$\boldsymbol{I}_p = -dt\, k_p\, d\,(1-\frac{d}{d_p})\,\hat{\boldsymbol{n}} \tag{3.12}$$

kde k_p je konstanta určující „přilnavou sílu“. Tato dodaná rychlost je škálována jádrem, jež je nulové v nulové vzdálenosti (nechceme „vcucávat“ částice do objektů) a v maximální interakční vzdálenosti. Maximum má v $d_p/2$.

Pokud chcete implementovat přilnavost k „podlaze“, kde žádné rychlosti nepočítáme, přičtěte tento člen k rychlosti ihned za aplikací viskozity. Tato přilnavost k podlaze je ukázána na obr. 20. Efekt „vypuklé“ kapky je dán kombinací gravitačního působení, přilnavosti a neprostupnosti objektu.



Obr. 20: Kapka na podlaze

3.2.10 Výpočet hodnot funkce algoritmu Marching Cubes

Důležitým prvkem v simulaci kapalin je její zobrazování. Algoritmus Marching Cubes nebyl v úvodu textu vyložen nadarmo – bude tvořit kostru našeho zobrazování. K extrakci povrchu kapaliny definujme skalární funkci

$$\phi\,(\boldsymbol{r}) = \left(\sum_i (1-|\boldsymbol{r}_i-\boldsymbol{r}|/h)^2\right)^{\frac{1}{2}} \tag{3.13}$$

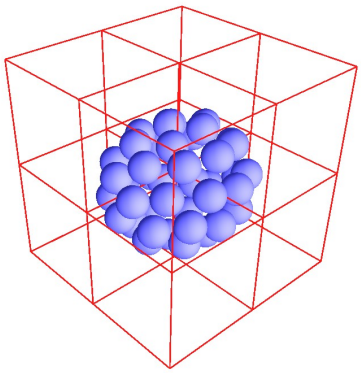
kde $\boldsymbol{r}_i$ je polohový vektor i-té částice a i probíhá přes všechny částice, jejichž vzdálenost od $\boldsymbol{r}$ je menší než h. Tato funkce dobře vystihuje vzdálenost od skupiny částic a má téměř konstantní sklon kolem extrahované izoplochy (což zaručuje přesnost interpolace mezi jednotlivými vrcholy).

V zásadě můžeme použít dva různé přístupy k optimalizaci výpočtu hodnot funkce v jednotlivých bodech mřížky. Jednodušší a mnohdy i efektivnější způsob popíšeme v této části – v apendixu je uveden druhý, náročnější přístup, u kterého je možné dále experimentovat a v některých případech je i efektivnější, čtenáři ale doporučuji považovat ho spíše za určitou zajímavost, vzhledem ke komplikacím, které jsou spojeny s jeho implementací.

Nejdříve zaveďme do programu novou proměnnou *frame*. Ta bude říkat, kolikátý frame vykreslujeme – po každém vykreslení scény (tj. po každém proběhnutí algoritmu Marching Cubes) zvedneme hodnotu *frame* o 1. Do pole Mřížka[i,j,k] přidáme parametr *frame* (bude to tedy *Mřížka[i,j,k].frame*), který bude určovat, kterému framu přísluší hodnota, která je ve daném bodě mřížky zaznamenána.

Algoritmus funguje tak, že projde všechny částice, zjistí si, na které body mřížky mohou mít vliv a, pokud je *frame* daného bodu stejný jako právě počítaný frame, připočte svůj příspěvek k hodnotě, pokud není, nejdříve hodnotu vynuluje, nastaví její *frame* na globální frame a přičte svůj příspěvek:

Výpočet hodnot funkce v bodech mřížky



Obr. 21: Optimalizované procházení krychlí

```
pro každou částici i dělej
        StartX := floor((Částice[i].R.x - h -
GridLeft)/GridH)
        StartY := floor((Částice[i].R.y - h -
GridBottom)/GridH)
        StartZ := floor((Částice[i].R.z - h -
GridFar)/GridH)
        KonecX := ceil((Částice[i].R.x + h -
GridLeft)/GridH)
        KonecY := ceil((Částice[i].R.y + h - GridBottom)/GridH)
        KonecZ := ceil((Částice[i].R.z + h – GridFar)/GridH)
        Uprav polohy StartX, StartY, StartZ, KonecX,  KonecY,  KonecZ tak, aby
        neležely mimo indexy mřížky (abychom nezapisovali do nepřidělené paměti).

        for i := StartX to KonecX do
          for j := StartY to KonecY do
            for k := StartZ to KonecZ do

                pokud Mřížka[i,j,k].frame <> frame tak
                        Mřížka[i,j,k].frame := frame
                        Mřížka[i,j,k].hodnota := 0

                q := 1 - | Částice[i].R-Mřížka[i,j,k].poloha |/ h

                pokud q > 0 tak
                        Mřížka[i,j,k].hodnota := sqrt((Mřížka[i,j,k].hodnota)^2+q^2)
```

Zde *GridH* značí rozestup bodů mřížky algoritmu Marching Cubes, *GridLeft*, *GridBottom* a *GridFar* značí po řadě x-ovou, y-ovou a z-ovou souřadnici bodu [0,0,0] algoritmu Marching Cubes. Funkce *floor* a *ceil* značí dolní a horní celou část reálného čísla. Tři vnořené for cykly pak nedělají nic jiného než projití všech krychliček algoritmu Marching Cubes, které leží v krychli o hraně 2h se středem v dané částici.

3.2.11 Zobrazování

Nyní, když máme vypočtené hodnoty funkce, mohli bychom na celou mřížku pustit standardní algoritmus Marching Cubes[5]. To by však bylo mrhání výpočetním výkonem – vždyť optimalizace, které jsou použili při výpočtu hodnot funkce, můžeme použít opět. Do pole *KrychleS* přidáme proměnnou *počítáno*, která bude obsahovat informaci o tom, zda v dané krychli mohl proběhnout nějaký výpočet – a zda tedy má smysl ji vykreslit[6]. Nejjednodušším možným přístupem by bylo

5 s hodnotou f_0 cca 2,14, tuto hodnotu je možné upravovat podle toho, jak mnoho detailů chceme v kapalině mít – pokud používáme mnoho částic, použijeme mírně menší hodnotu

6 Označení „počítáno“ je použito pro zachování jednotnosti s algoritmem uvedeným v apendixu, tam uchovává daná proměnná skutečně informaci o tom, že v dané krychli byl výpočet proveden.

projít všechny *KrychleS* a pokud by se v dané krychli nacházela částice, nastavila by se proměnná *počítáno* na pravdivou hodnotu pro danou krychli a pro všechny její sousedy. Tím je zajištěn nejdůležitější krok optimalizace – výpočet je „lokální“, povrch je vykreslován skutečně jen v okolí částic. Ale vzhledem k tomu, že se povrch nachází v poměrně malé vzdálenosti od částic, je tento výpočet zbytečně neefektivní – pokud by se nacházela v krychli jediná částice úplně vpravo, zcela jistě nemá smysl kvůli tomu procházet i levé sousedy dané krychle. Proto zavedeme parametr d, který určí minimální vzdálenost částic od stěn krychle, při které se již nastaví *počítáno* i příslušnému sousedu u dané stěny (a v kombinaci blízké polohy ke dvou stěnám i sousedu přes hranu a v kombinaci ke třem stěnám i sousedu přes vrchol). Na obr. 21 vidíme situaci, kdy se všechny částice nachází dostatečně daleko od stěn – prochází se tedy jen 8 krychlí místo 64 při procházení všech sousedů – to už je poměrně výrazná optimalizace. Vše shrnuje následující algoritmus:

Vykreslování algoritmem Marching Cubes

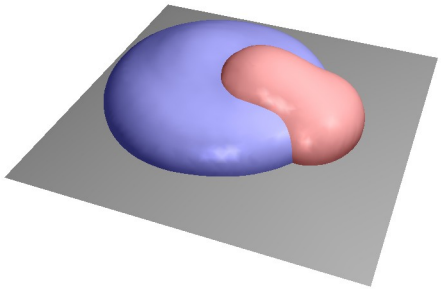
```
pro všechny indexy [m,n,o] pole KrychleS dělej
        KrychleS[m,n,o].počítáno := nepravda

pro všechny indexy [m,n,o] pole KrychleS dělej
        vlevo, vpravo, nahoře, dole, vpředu, vzadu := nepravda

        pokud length( KrychleS[m,n,o].částice) > 0 tak      // pokud je v krychli částice
                pro každou částici i v krychli [m,n,o] dělej
                        pokud Částice[i].R.x < m·h+d tak vlevo := pravda
                        pokud Částice[i].R.x > (m+1)·h-d tak vpravo := pravda
                        pokud Částice[i].R.y < n·h+d tak dole := pravda
                        pokud Částice[i].R.y > (n+1)·h-d tak nahoře := pravda
                        pokud Částice[i].R.z < o·h+d tak vzadu := pravda
                        pokud Částice[i].R.z > (o+1)·h-d tak vpředu := pravda

                KrychleS[m,n,o].počítáno := pravda
                pokud vlevo tak KrychleS[m-1,n,o].počítáno := pravda
                pokud vlevo a dole tak KrychleS[m-1,n-1,o].počítáno := pravda
                ...
                pokud dole a vpředu tak KrychleS[m,n-1,o+1].počítáno := pravda
                ...
                /Takto přirozeným způsobem nastavíme počítáno krychli
                [m,n,o] a na základě proměnných vlevo, vpravo atd. také
                některým z 26 sousedů (celkem zde tedy bude 26 podmínek)/

pro všechny indexy [m,n,o] pole KrychleS dělej
        pokud KrychleS[m,n,o].počítáno tak
                 pro všechny krychličky [i,j,k] ležící v krychli [m,n,o] dělej
                        Spočti vrcholy v krychličce [i,j,k]

pro všechny indexy [m,n,o] pole KrychleS dělej
        pokud KrychleS[m,n,o].počítáno tak
```

```
                pro všechny krychličky [i,j,k] ležící v krychli [m,n,o] dělej
                        Spočítej normály v krychličce [i,j,k]

pro všechny indexy [m,n,o] pole KrychleS dělej
        pokud KrychleS[m,n,o].počítáno tak
                pro všechny krychličky [i,j,k] ležící v krychli [m,n,o] dělej
                        Nakresli trojúhelníky v krychličce [i,j,k]
```

Poslední tři kroky odpovídají krokům algoritmu Marching Cubes s výpočtem normál, pouze je vynechán krok počítající normály na hranici mřížky – předpokládáme, že mřížka algoritmu Marching Cubes je dostatečně velká na to, aby se částice k její hranici nedostaly.

3.2.12 Interakce více kapalin

V této části vyjdeme z jistých empirických předpokladů o tom, jak spolu mohou částice různých kapalin interagovat a jak spolu interagují kapaliny na makroskopické úrovni. Předně – algoritmus by měl fungovat tak, aby bylo možné vhodnou volbou konstant docílit toho, aby různé spolu interagující kapaliny byly ve skutečnosti „stejné“, tj. aby se výsledek nelišil od simulace jedné kapaliny. To ovšem znamená, že interakci více kapalin je nutné zahrnout do zachování hustoty.



Obr. 22: Stekající kapka vody

Jak víme z každodenní zkušenosti, kapaliny se buď mísí (např. mléko s vodou, ethanol s vodou...), nebo nemísí (např. olej s vodou). Mísící se kapaliny do sebe vzájemně difundují, nemísící v sobě tvoří bublinky a vrstvy a mají mezi sebou jasně definované přechody. Proto zavedeme dvě konstanty – odpudivost ξ a difuzivnost Φ. Čím větší je ξ, tím více se částice různých kapalin navzájem odpuzují (pokud je menší než 0, přitahují se). Φ určuje tendenci částic pronikat do prostoru v druhé kapalině, kde je málo stejných částic. Je-li $\Phi > 1$, snaží se proniknout částice druhé kapaliny do první, je-li $\Phi < 1$, snaží se proniknou částice první kapaliny do druhé. Ze zkušenosti také víme, že řídké kapaliny mají malý vliv na husté, např. kapička vody, která spadne do medu, vytvoří malou kapičku na povrchu, jako by se jednalo o pevný podklad (jelikož se med a voda mísí, tato kapička se postupně do medu „rozpustí“). To znamená, že med ovlivní kapičku vody hodně, kapička vody med málo. Tohoto nepoměru docílíme tak, že částicím různých kapalin přiřadíme různé hmotnosti. To zajistí konstanta m udávající poměr hmotností částic. Tyto tři konstanty - ξ, Φ a m mohou být různé pro každou dvojici kapalin. Je-li $\xi = 0$, $\Phi = 1$ a $m = 1$ pak (pokud mají kapaliny stejně nastavené „vnitřní“ konstanty) by měly dvě kapaliny interagovat stejně jako dvě skupiny částic téže kapaliny. Mně osobně se osvědčilo Φ mírně větší než 1, $\xi = 0$, $m = 1$ pro simulaci difuze a $\Phi = 1$, $\xi = 0{,}002$ a $m > 1$ (pro viskózní kapalinu) pro simulaci nemísících se kapalin.

Někoho možná napadne, že ve skutečných kapalinách jsou částice různě daleko od sebe, co tedy zkusit nastavit různé h pro jednotlivé kapaliny. To ovšem výrazně nedoporučuji (alespoň ne za použití principu zde popsaného), má to za následek to, že kapalina s větší vzdáleností částic je „vcucnuta“ do kapaliny s menší vzdáleností a chová se z větší části, jako by vůbec kolem druhá kapalina nebyla přítomna.

Přejděme k algoritmu, z jeho zápisu bude nejlépe patrné, jak se konstanty ξ, Φ a m aplikují. Ná-

sledující algoritmus popisuje interakci dvou kapalin, pro čtenáře jistě nebude problém tento algoritmus zobecnit pro interakci více kapalin. Pole Částice1 značí částice první kapaliny, Částice2 částice druhé kapaliny, h pokládáme za stejné pro obě kapaliny.

Zachování společné dvojí hustoty při interakci více kapalin

Nejprve určíme vzájemné konstanty ρ_0, k a k_b. Můžeme použít např. aritmetický průměr těchto konstant u obou kapalin.

pro každou částici i v poli Částice1 **dělej**
ρ := 0
ρ_b := 0

/nejdříve spočteme příspěvky od vlastních částic/
pro každou částici j pole Částice1 sousedící s i **dělej**
q := | Částice1[j].***MinuléR*** – Částice1[i].***MinuléR*** | / h
pokud q < 1 **tak**
ρ := ρ + $(1-q)^2$
ρ_b := ρ_b + $(1-q)^3$

/poté spočteme příspěvky od druhých částic/
pro každou částici j pole Částice2 sousedící s i **dělej**
q := | Částice2[j].***MinuléR*** – Částice1[i].***MinuléR*** | / h
pokud q < 1 **tak**
ρ := ρ + $(1-q)^2$
ρ_b := ρ_b + $(1-q)^3$

P := $k_1(\rho- \rho_0)$ */tlak a blízkostní tlak určíme stejně jako při výpočtu*
P_b := k_{b1} ρ_b *jedné kapaliny – používáme konstanty pro Částice1/*
dr := ***0***
/nejdříve spočteme příspěvky od vlastních částic/
pro každou částici j pole Částice1 sousedící s i **dělej**
q := | Částice1[j]. ***MinuléR*** – Částice1[i].***MinuléR*** | / h
pokud q < 1 **tak**
D := $dt^2 \cdot (P \cdot (1-q) + P_s \cdot (1-q)^2) \cdot JV$(Částice1[j]. ***MinuléR*** – Částice1[i].***MinuléR***)
Částice1[j].***R*** := Částice1[j].***R*** + ***D***/2
dr := ***dr*** – ***D***/2

P := k(ρ- ρ_0)+ξ */tlak a blízkostní tlak nyní určíme pomocí společných*
P_b := k_b ρ_b *konstant obou kapalin a zvedneme ho o ξ /*

/projdeme všechny sousedící částice druhé kapaliny a spočteme posuny/
pro každou částici j pole Částice2 sousedící s i **dělej**
q := | Částice2[j]. ***MinuléR*** – Částice1[i].***MinuléR*** | / h
pokud q < 1 **tak**

$\boldsymbol{D}$:= φ·dt²·(P·(1-q)+P_s·(1-q)²)·JV(Částice2[j]. ***MinuléR*** – Částice1[i].***MinuléR***)
Částice1[j].***R*** := Částice1[j].***R*** + m·***D***/2
dr := ***dr*** – (1/m)·***D***/2

/Celkový příspěvek k pohybu přidáme k částici i, tím se zachová hybnost/
Částice[i].***R*** := Částice[i].***R*** + ***dx***

pro každou částici i v poli Částice2 **dělej**
Zde provedeme symetricky naprosto totéž, co pro každou částici v poli Částice1. Všude bude pouze prohozeny výrazy „Částice1“ a „Částice2“

Poznámka: Vzhledem k tomu, že používáme dvě pole částic, musí nám nyní také pole krychlí sousedů uchovávat dva seznamy částic. Budeme tedy mít seznam *KrychleS[m,n,o].částice1* a *KrychleS[m,n,o].částice2*.

Ještě poznamenejme, jak tento algoritmus zasadit do celkového kontextu výpočtu. Je to velmi jednoduché, provedeme to podle následujícího schématu:

Proveď všechny výpočty (posuny, viskozitu, ukládání poloh, pružiny...), které se v původním algoritmu provádí před zachováním hustoty, nezávisle na Částice1 a na Částice2

Aplikuj společné zachování hustoty

Proveď všechny výpočty (kolize, výpočet nové rychlosti), které se nachází za zachováním hustoty, nezávisle na Částice1 a Částice2

Tento algoritmus nám umožní simulovat interakci dvou kapalin, na které neaplikujeme viskozitu, nebo jedné viskózní a jedné neviskózní kapaliny (např. medu a vody). Pokud bychom chtěli simulovat interakci dvou viskózních kapalin, musíme také upravit funkci funkci výpočtu viskozity tak, aby se procházely obě dvě skupiny částic. Opět můžeme zavést konstantu říkající, jak moc se kapaliny ovlivňují vzájemně. Princip je zcela totožný, čtenář si jistě upravenou funkci řešící viskozitu zvládne napsat sám.

3.2.13 Zobrazování více kapalin

Nejjednodušším způsobem, jak dvě kapaliny zobrazit, by bylo pustit dva na sobě zcela nezávislé algoritmy Marching Cubes – s polem Částice1 a Částice2. To však nedává uspokojivé výsledky. Mezi povrchy obou kapalin vznikají mezery, což není žádoucí. K napravení tohoto nedostatku můžeme upravit způsob počítání hodnot funkce v mřížce – a to tak, že i částice druhé kapaliny budou přispívat určitým dílem k hodnotě funkce. V následujícím algoritmu určíme hodnoty v bodech mřížky pro vykreslení pole Částice1, pro pole Částice2 by byl algoritmus zcela obdobný.

Upravený algoritmus výpočtu hodnot mřížky a zobrazování pro dvě kapaliny

Zde proveď výpočet pro pole Částice1 tak, jak byl popsán v sekci o zobrazování jedné kapaliny

pro každou částici *i* pole Částice2 **dělej**
StartX := floor((Částice2[i].***R***.x - h - GridLeft)/GridH)
StartY := floor((Částice2[i].***R***.y - h - GridBottom)/GridH)
StartZ := floor((Částice2[i].***R***.z - h - GridFar)/GridH)
KonecX := ceil((Částice2[i].***R***.x + h - GridLeft)/GridH)
KonecY := ceil((Částice2[i].***R***.y + h - GridBottom)/GridH)
KonecZ := ceil((Částice2[i].***R***.z + h – GridFar)/GridH)
Uprav polohy StartX, StartY, StartZ, KonecX, KonecY, KonecZ tak, aby neležely mimo indexy mřížky (abychom nezapisovali do nepřidělené paměti).

for i := StartX **to** KonecX **do**
for j := StartY **to** KonecY **do**
for k := StartZ **to** KonecZ **do**

pokud Mřížka[i,j,k].frame <> frame **tak**
continue */Pokud políčko neovlivnily Částice1, nemá smysl ho propočítávat/*

q := 1 - | Částice2[i].***R***-Mřížka[i,j,k].***poloha*** |/ h

pokud q > 0 **tak**
Mřížka[i,j,k].hodnota := sqrt((Mřížka[i,j,k].hodnota)^2+ς·q^2)

Aplikuj standardní algoritmus Marching Cubes, tak, jak byl popsán v sekci o zobrazování jedné kapaliny.

Parametr ς určuje, jak silně se započítá příspěvek od druhých částic, rozhodně musí být $\varsigma \leq 1$. ς = 1 využijeme pravděpodobně jen v případě, že obě simulované kapaliny jsou zcela totožné, jinak bude vždy ς < 1. Pro dosažení poměrně pěkného oblého efektu dostačuje ς = 0,5 (při $f_0 = 2{,}14$). Je však nutné pro každou hodnotu f_0 najít vhodnou hodnotu ς zkoušením různých hodnot.

Pokud chceme scénu generovat ve vysoké kvalitě za použití průhlednosti, neměly by se povrchy kapalin vzájemně protínat. Za tímto účelem mi, pro $f_0 = 2{,}14$, dostačovala hodnota cca ς = 0,35.

4 Matematický popis simulace kapalin

V této kapitole budou velmi stručně popsány principy, na kterých SPH staví. Tato kapitola je do této práce zařazena spíše pro úplnost, z hlediska implementace není nutné ji číst, jedná se především o shrnutí základních myšlenek, jejichž podrobnější popis může čtenář nalézt např. v [NO07] nebo v [MCG03].

4.1.1 Navier-Stokesovy rovnice

Navier-Stokesovy rovnice jsou soustavou parciálních diferenciálních rovnic, které popisují chování tekutiny. Je možné je odvodit z podmínek zachování hmotnosti, hybnosti a momentu hybnosti. Tyto rovnice je možné řešit analyticky pouze v několika jednoduchých případech, obecně je nutné použít numerické řešení. Stojí za zmínku, že existenci a „hladkost“ tohoto řešení zatím nikdo nedokázal (ač se z fyzikální interpretace zdá, že řešení existovat musí). Vytvoření tohoto důkazu bylo zařazeno mezi tzv. Millennium Prize Problems, na něž Clay Mathematics Institute vypsal v roce 2000 odměnu 1 000 000 USD.

Vraťme se ale zpět k rovnicím. Nás zajímá pouze proudění nestlačitelných newtonovských kapalin. Charakteristickou vlastností těchto kapalin je fakt, že tečné napětí je přímo úměrné rychlosti deformace:

$$\tau = -\eta \frac{du}{dt} \tag{4.1}$$

Konstanta úměrnosti se nazývá dynamická viskozita.

Samotné Navier-Stokesovy rovnice pak díky této podmínce získají tvar

$$\rho \cdot (\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \nabla \boldsymbol{v}) = -\nabla p + \mu \nabla^2 \boldsymbol{v} + \boldsymbol{f} \tag{4.2}$$

$$\nabla \cdot \boldsymbol{v} = 0 \tag{4.3}$$

kde $\boldsymbol{v}$ je vektor rychlosti, p je tlak, ρ hustota a $\boldsymbol{f}$ pole vnějších objemových sil. Rovnice 4.3 vyjadřuje zákon zachování objemu (hmotnosti, hustoty).

4.1.2 Smoothed Particle Hydrodynamics

Metoda SPH byla původně vyvinuta pro potřeby simulace toku mezihvězdného plynu v astrofyzice. Každá částice je nositelkou určitých vlastností (hmotnost, tlak, poloha, rychlost...). Příspěvky k těmto vlastnostem jsou v každém bodě prostoru interpolovány za pomoci tzv. jádra (kernel function), které zajišťuje, že vzdálenější částice mají menší vliv než částice blízké. Toto jádro tedy „rozprostírá“ působení částice z jednoho bodu do prostoru kolem ní.

Základem SPH je tzv. integrální interpolace. Interpolovaná hodnota pole a (označme si ji A) je pro každý bod prostoru definována vztahem

$$A(\boldsymbol{r}) = \int a(\boldsymbol{r}) W(\boldsymbol{r} - \boldsymbol{r}', h)\, d\boldsymbol{r}' \tag{4.4}$$

kde $\boldsymbol{r}$ je bod, v němž vyhodnocujeme interpolovanou hodnotu. h je nezávislý parametr udávající „rozlišení prostoru“. Zvětšíme-li h na dvojnásobek a posléze počítáme se všemi polohovými vektory jako kdyby měly dvojnásobnou velikost, výsledek bude stejný.

Jádro musí být normované, tedy

$$\int_V W(\boldsymbol{r}-\boldsymbol{r}')d\,\boldsymbol{r}'=1 \tag{4.5}$$

Také musí být jádro sudé, očekáváme izotropní chování:

$$W(\boldsymbol{r},h)=W(-\boldsymbol{r},h) \tag{4.6}$$

Pokud známe $a(\boldsymbol{r})$ pouze v N bodech nějaké diskrétní mřížky, můžeme integrál 4.4 nahradit sumou:

$$A(\boldsymbol{r})=\sum_{j=1}^{N} V_j \cdot a(\boldsymbol{r}_j)\cdot W(\boldsymbol{r}-\boldsymbol{r}_j,h) \tag{4.7}$$

kde V_j je objem příslušející částici j, $\boldsymbol{r}_j$ je její polohový vektor. Objem V_j můžeme vyjádřit z hustoty částice

$$V_j=\frac{m_j}{\rho_j} \tag{4.8}$$

S použitím rovnosti 4.8 můžeme rovnici 4.8 přepsat do tvaru

$$A(\boldsymbol{r})=\sum_{j=1}^{N} \frac{m_j}{\rho_j}\cdot a(\boldsymbol{r}_j)\cdot W(\boldsymbol{r}-\boldsymbol{r}_j,h) \tag{4.9}$$

S pomocí tohoto tvaru můžeme snadno odvodit gradient a laplacián interopolované vlastnosti $A(\boldsymbol{r})$:

$$\nabla A(\boldsymbol{r})=\sum_{j=1}^{N} \frac{m_j}{\rho_j}\cdot a(\boldsymbol{r}_j)\cdot \nabla W(\boldsymbol{r}-\boldsymbol{r}_j,h) \tag{4.10}$$

$$\nabla^2 A(\boldsymbol{r})=\sum_{j=1}^{N} \frac{m_j}{\rho_j}\cdot a(\boldsymbol{r}_j)\cdot \nabla^2 W(\boldsymbol{r}-\boldsymbol{r}_j,h) \tag{4.11}$$

Vše, co tedy potřebujeme znát, je gradient a laplacián jádra.

4.1.3 Základní vztahy

Nejdříve se podívejme na hustotu, ta je pro kapalinu spojitou veličinou v prostoru, spočtěme ji tedy integrální interpolací:

$$\rho_i=\rho(\boldsymbol{r}_i)=\sum_{j=1}^{N} \frac{m_j}{\rho_j}\cdot \rho_j \cdot W(\boldsymbol{r}-\boldsymbol{r}_j,h)=\sum_{j=1}^{N} m_j \cdot W(\boldsymbol{r}-\boldsymbol{r}_j,h) \tag{4.12}$$

Dále se zaměřme na tlakovou sílu. Tlak můžeme odvodit ze stavové rovnice ideálního plynu ve tvaru

$$pV=k' \Rightarrow p\frac{1}{\rho}=k \Rightarrow p=k\rho \tag{4.13}$$

Síla odpovídající tomuto tlaku by byla vždy odpudivá (plyn se rozepíná do celého prostoru). Naším cílem je však zachovat stálou hustotu, což je možné zajistit definováním klidové hustoty:

$$p=k(\rho-\rho_0) \tag{4.14}$$

Částice má tendenci pohybovat se směrem k minimu tlakového pole (opačně ke směru gradientu). Síla vyvolaná tlakem se v rovnici 4.2 projeví jako první člen na pravé straně. Za použití rovnice 4.10 ji můžeme zapsat ve tvaru:

$$\boldsymbol{F}_i^{tlak} = -\nabla p(\boldsymbol{r}_i) = \sum_{j=1}^{N} \frac{m_j}{\rho_j} \cdot p_j \cdot \nabla W(\boldsymbol{r} - \boldsymbol{r}_j, h) \tag{4.15}$$

Takovýto tlak by ovšem neprodukoval symetrické síly, čímž by nastal problém se zachováním hybnosti a momentu hybnosti. Proto pro každé dvě částice tlakovou sílu symetrizujeme výrazem:

$$\boldsymbol{F}_i^{tlak} = \sum_{j=1}^{N} m_j \frac{p_i + p_j}{2\rho_j} \cdot p_j \cdot \nabla W(\boldsymbol{r} - \boldsymbol{r}_j, h) \tag{4.16}$$

Další silou, která působí, je síla způsobená viskozitou. Ta odpovídá druhému členu v rovnici 4.2. Tuto sílu (s ohledem na rovnici 4.11) můžeme přepsat do tvaru:

$$\boldsymbol{F}_i^{visk} = \mu \nabla^2 \boldsymbol{v} = \sum_{j=1}^{N} \frac{m_j}{\rho_j} \cdot \boldsymbol{v}_j \cdot \nabla^2 W(\boldsymbol{r} - \boldsymbol{r}_j, h) \tag{4.17}$$

Tato síla je opět nesymetrická, můžeme ji přirozeně symetrizovat takto:

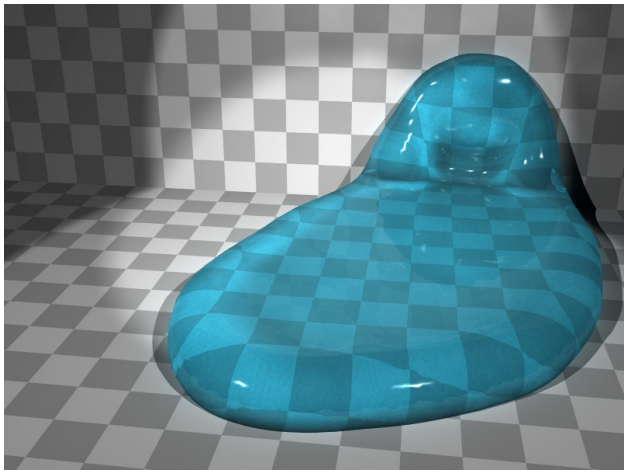
$$\boldsymbol{F}_i^{visk} = \sum_{j=1}^{N} \frac{m_j}{\rho_j} \cdot (\boldsymbol{v}_j - \boldsymbol{v}_i) \cdot \nabla^2 W(\boldsymbol{r} - \boldsymbol{r}_j, h) \tag{4.18}$$

4.1.4 Závěr

V této kapitole byly nastíněny základní matematické principy metody SPH. Výsledky zde uvedené ospravedlňují postupy implementované v kapitole 3. Čtenáře, kterého by zajímaly matematické aspekty více do hloubky, odkazuji na literaturu citovanou v úvodu.

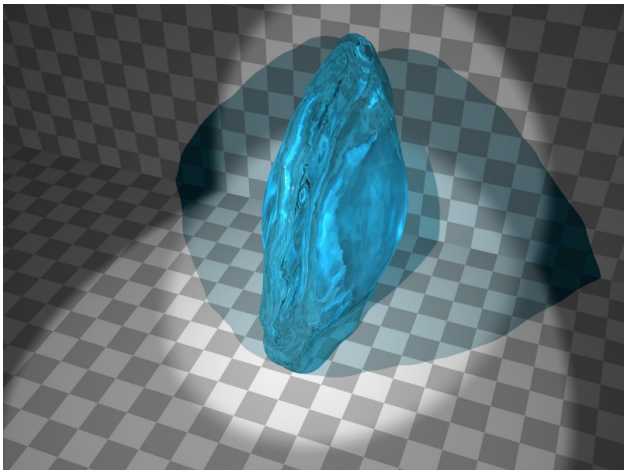
5 Renderování vysoce kvalitních obrázků

V úvodu ke třetí kapitole jsem uvedl, že se v tomto textu nebudu zabývat ray tracingem ani jinými technikami tohoto charakteru. Čtenář, jehož by tyto techniky zajímaly, byl odkázán na publikaci [WA01] a webový tutoriál [BI05]. Zmiňme však způsob, jímž je možné dosáhnout fotorealistických snímků i bez nutné implementace těchto technik – k tomu nám může posloužit již vyvinutý software. Já jsem použil open source ray tracer POV-Ray. Ten je vhodný již z několika důvodů – je zcela zdarma a používá k definici scény jednoduchý, logicky strukturovaný jazyk. Také obsahuje objekt „mesh“ (mřížka), jenž je přímo určen k definici tělesa pomocí velkého počtu trojúhelníků.



Obr. 23: Kapalina stékající ze stěny

Zde popíši pouze jednoduchý způsob, kterým jsem vytvořil obrázek 24[7]. Základem je samozřejmě simulace samotná. Zde byly v programu zadány dvě stěny – podlaha procházející bodem -2 na svislé ose (v OpenGL jde o osu y, v POV-Rayi o osu z) a stěna procházející bodem -4 na ose „směrem z obrazovky“ (V POV-Rayi je to opačně orientovaná osa y, v OpenGL osa z). Data získaná ze simulace jsou exportována jako trojúhelníky objektu *mesh* (viz dále).



Obr. 24: Kapky během srážky

Přistupme k samotnému zdrojovému kódu pro POV-Ray. Kód je okomentován a měl by být srozumitelný i programátorovi, jenž v POV-Rayi nikdy nepracoval.

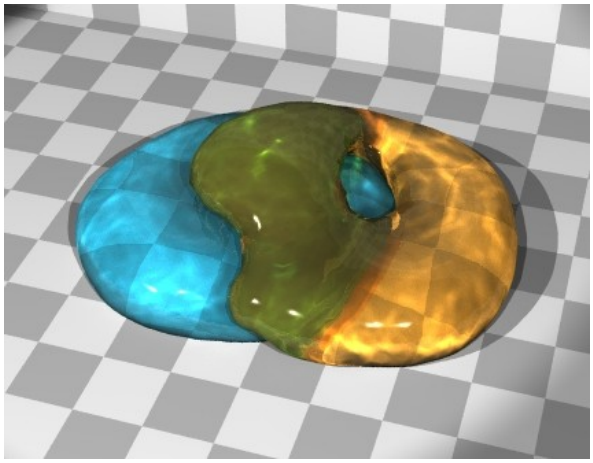
```
#include "colors.inc"          // Zde se nachází definice použitých barev
#include "glass.inc"           // Zde se nachází definice skleněných materiálů
#include "textures.inc"        // Zde se nachází použité textury

camera {                       // Nastavíme pohled
        sky <0,0,1>            // Toto zaručuje, že je kamera vzpřímená
        right <-4/3,0,0>       // Určuje poměr stran renderovaného obrázku (4:3)
        location  <30,-70,70>  // Určuje polohu kamery
        look_at   <0,10,0>     // Určuje, který bod kamera sleduje
        angle 45               // Určuje „širokoúhlost“ projekce
}
global  settings {
```

7 V tištěné verzi pravděpodobně zaniknou důležité detaily obou obrázků. Doporučuji čtenáři, aby si je prohlédl ve verzi elektronické.

```
        ambient_light Black         // Nechceme žádné „všudypřítomné“ osvětlení
        photons {                   // Říká, že chceme použít fotony – to výrazně zlepšuje
                spacing 0.5         // realističnost výsledného obrázku, ale také výrazně
        }                           // zpomaluje rendering. Čím větší je spacing, tím
}                                   // méně fotonů je použito a rendering je rychlejší.

background { color Black }          // Pozadí chceme černé

// Zde definuji osvětlení scény, používám tři různá světla

light_source {
        <100,-100,100>              // Poloha osvětlení
        color White                 // Barva světla
        spotlight                   // Typ – bodový zdroj
        radius 15                   // Konstanty rozsahu a slábnutí světla
        falloff 20
        point_at <0, 0, 0>          // Směr, kterým světlo směřuje.
}

light_source {
        <-100,-100,100>             // „Zrcadlové“ světlo k prvnímu světlu
        color White                 // Barva světla
        spotlight                   // Typ – bodový zdroj
        radius 15                   // Konstanty rozsahu a slábnutí světla
        falloff 20
        point_at <0, 0, 0>          // Světlo směřuje do stejného místa jako předchozí
}

light_source {
        <2, -50, 50>                // Světlo je blíže předmětům než ostatní
        color White                 // Barva světla
        spotlight                   // Typ – bodový zdroj
        radius 15                   // Konstanty rozsahu a slábnutí světla
        falloff 20

        // Následující tři řádky ze světla udělají sadu světel 5 x 5. Tím vzniknou jemné
        // stíny a osvětlení se stane zajímavějším.

        area_light <1, 0, 0>, <0, 1, 0>, 5, 5
        adaptive 1
        jitter
        point_at <0, 0, 0>          // Světlo směřuje do stejného místa jako ostatní
}
```

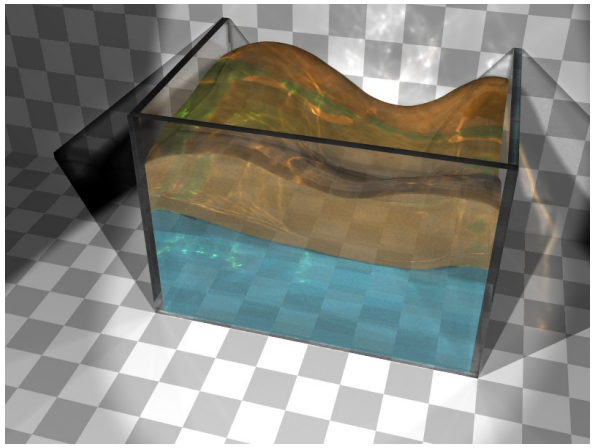
```
// Nyní vytvořím podlahu a stěnu

plane {
        <0,0,1>, -20            // Rovina zadaná rovnicí 0·x+0·y+1·z = -20
        pigment {               // Šachovnicový vzorek
                checker rgb <0.8,0.8,0.8>, rgb <0.6,0.6,0.6>
                scale 5
        }
        photons {               // Nastavení fotonů
                target          // Rovina je cílem fotonů
                collect on      // Pokud chcete modře zbarvený stín, použijte „off“
        }

}

plane {
        <0,1,0>,  40            // Rovina zadaná rovnicí 0·x+1·y+0·z = 40
        pigment {               // Šachovnicový vzorek
                checker rgb <0.8,0.8,0.8>, rgb <0.6,0.6,0.6>
                scale 5
        }
        photons {               // Nastavení fotonů
                target          // Rovina je cílem fotonů
                collect on      // Pokud chcete modře zbarvený stín, použijte „off“
        }
}

mesh {      // Zde vygenerujeme algoritmem Marching Cubes trojúhelníky povrchu
            // v1, v2 a v3 jsou vrcholy trojúhelníka, n1, n2 a n3 normály vrcholů

        smooth_triangle { <v1>, <n1>, <v2>, <n2>, <v3>, <n3> }
        // ... zde jsou takto zadány všechny trojúhelníky pomocí smooth_triangle, v této
        // scéně jsem všechny velikosti složek polohových vektorů vynásobil deseti, aby
        // souřadnice byly v rozumnějším rozsahu (proto se také roviny nacházejí na
        // souřadnicích -20 a 40, nikoliv -2 a 4).

        pigment { rgbf <0.4,0.9,1,0.8> }   // Definujeme zabarvení kapaliny
        photons {               // Fotony budou zapnuté, chceme realistický vzhled
                target          // Objekt je cílem fotonů
                refraction on   // Lom světla je zapnut
                reflection on   // Odraz je zapnut
                collect on      // Fotony se na povrchu projeví
        }
        material { M_Glass3 }   // Skleněný materiál, ideální pro naše účely
}
```

Pokud bychom chtěli zobrazovat dvě kapaliny, potřebujeme provést export do dvou objektů *mesh*, kterým nastavíme různé vlastnosti (např. různý pigment). Zde hraje velmi důležitou úlohu parametr ς při výpočtu povrchu. Na obr. 25 vidíme dvě kapaliny s ς nastaveným na 0,35. Při této hodnotě se povrchy kapalin mírně protínají – to je důvodem, proč vidíme rozhraní mezi žlutou a modrou kapalinou zeleně – díváme se na modrý povrch skrz žlutý. Pokud by byl ς menší, rozhraní by bylo žluté, protože by se žlutý povrch nacházel nad modrým a vrhal žluté odlesky. V takovéto situaci zelený povrch vypadá přirozeněji, je tedy vhodné zvolit ς o něco větší než 0,35. Pokud bychom ale zobrazovali dvě kapaliny např. v akváriu, pak by protínání povrchu vypadalo skrze sklo podivně a je nutné s hodnotou parametru experimentovat.



Obr. 25: Řídká kapalina stéká z husté

Na obr. 26 vidíme dvě kapaliny v akváriu. Hodnota ς je o něco menší než předchozím případě, kapaliny se tedy téměř neprotínají. Zde ovšem vznikne problém s tím, že v krajních oblastech jsou povrchy od sebe poměrně velmi vzdálené. Tomuto defektu je možné zabránit tak, že zobrazíme menší část povrchu, než jakou jsme skutečně spočítali. K oseknutí povrchu pouze na prostor akvária je možné v POV-Rayi použít příkaz *intersection*:



Obr. 26: Dvě kapaliny v akváriu

```
// zde je veškeré nastavení světel a scény
intersection {
        mesh {
                // zde jsou smooth_triagle, nastavení materiálo apod.
        }
        box {
                <a,b,c>, <x,y,z> // body tělesové úhlopříčky akvária
                material { M_Glass3 } // stejný materiál, jaký mají skla akvária
        }
}
```

6 Popis ukázkového programu

6.1.1 Pracovní prostředí

Vzhledem k tomu, že ukázkový program je určen především k vytváření obrázků a videí použitých v této práci (a ke studiu zdrojového kódu, což ovšem by ovšem vzhledem k úplnosti předchozího výkladu nemělo být nutné), pojměme popis jeho funkcí velmi stručně, není žádný důvod program do hloubky rozebírat (zájemci o hlubší pochopení funkcí programu si mohou prostudovat zdrojové kódy). Program sestává ze dvou oken – okna příkazové řádky a okna zobrazujícího 3D grafiku (jedná se o okno OpenGL). V po spuštění je standardně v okně OpenGL zobrazena krychle, jež ohraničuje mřížku, ve které běží algoritmus Marching Cubes. Přibližně pod kurzorem myši se nachází červený „3D kurzor“, kterým je možné pomocí pohybu myši po ploše okna pohybovat.

Příkazy v příkazové řádce je možné upravovat chování programu, vytvářet částice, spouštět skripty apod. Příkazy měnící hodnotu typu boolean mohou mít buď jeden parametr (on / off), nebo žádný parametr – pak se pouze daná pravdivostní hodnota změní na opačnou. Pokud je parametr ve tvaru „!a:b“, pak je místo něj vygenerována náhodná hodnota v rozsahu od a do b. Podívejme se na seznam příkazů:

Příkazy pro tvoření scény a řízení simulace

p x y z vx vy vz	Vytvoří částici na pozici (bx+x,by+y,bz+z) s rychlostí (bvx+vx,bvy+vy,bvz+vz)
baser x y z	Nastaví (bx, by, bz) na (x, y, z)
basev vx vy vz	Nastaví (bvx, bvy, bvz) na (vx, vy, vz)
cursorbase x y z	Nastaví (bx, by, bz) na „pozice kurzoru“ + (x, y, z)
clear	Smaže aktivní částice
freeze	Zmrazí výpočet částic

Příkazy k řízení skriptů

run name	Spustí skript „scripts/name“
runonmousedown jméno	Pokud je při výpočtu snímku stisknuto tlačítko myši, spustí se jméno
runonmouseclick jméno	Při kliknutí tlačítkem myši se spustí skript jméno
wait počet	Počká daný počet snímku před provedením dalšího příkazu
do c příkaz	Provádí příkaz c-krát

Příkazy měnící hodnotu typu boolean: applyviscosity, applystrings, drawsurface, drawparticles, lighting, smoothsading, pseudolight, gridcube, gridcubes, floor, roof, mcnewmethod.

Příkazy sloužící ke změně vlastností zobrazení: quality číslo, resolution číslo, resolutions číslo číslo číslo, gridh číslo, radius des. číslo, gridposition x y z, gridsize číslo, sphere red/green/blue

Příkazy nastavující konstanty: g x y z, h číslo, dt číslo, particles1 jméno_konstanty hodnota, particles2 jméno_konstanty hodnota

Jména konstant: h, ks, kn, kstick, ds, rho0, gamma, alpha, sigma, beta, isovalue

Ostatní příkazy

size šířka výška	Nastaví šířku a výšku okna
recreatewindow	Znovu vytvoří okno OpenGL

Dále je možné program ovládat klávesami v okně OpenGL. Ovládání je následující:

Klávesa	Význam
F1	Kurzor vytváří Částice1
F2	Kurzor vytváří Částice2
c	Zobrazí/skryje krychli mřížky
↑/↓/←/→	Posune mřížku nahoru / dolů / doleva / doprava
8/5/4/6	Začne scénou otáčet nahoru / dolů / doleva / doprava
9/7/1/3	Začne hýbat kamerou nahoru / dolů / doleva / doprava
del	Zastaví pohyb kamery
0	Zastaví otáčení scény
+ / -	Zvětšuje / zmenšuje mřížku
/ / *	Přibližuje / vzdaluje kameru
enter	Nastaví pohled do výchozí polohy
g	Zobrazí povrch „drátěně“
Page Up	Posune 3D kurzor dozadu
Page Down	Posune 3D kurzor dopředu
F12	Vytvoří screenshot jménem screen.png
F8	Začne náhrávat do records (každý snímek jako .png obrázek)
F7	Skončí nahrávání
r	Začne nahrávání .pov souborů do records
t	Ukončí nahrávání .pov souborů do records

6.1.2 Zdrojové kódy

Program je psán v jazyce Object Pascal za použití kompilátoru Free Pascal Compiler. Před čtením zdrojového kódu (které z větší části nelze doporučit, obsahuje totiž mnoho různých funkcí, které čtenář pravděpodobně nevyužije a které jsou při vlastní implementaci spíše matoucí) doporučuji čtenáři, aby si prostudoval referenční manuál Object Pascalu v dialektu Free Pascalu [FPC07], syntaxe je totiž mírně odlišná (více flexibilní) než v klasickém Turbo Pascalu či v Delphi. Taktéž čtenáři doporučuji knihu [OpenGLRB], ze které jsem čerpal při psaní funkcí starajících se o vykreslování v OpenGL.

V .zip souboru se zdrojovými kódy (který je možné získat např. na autorových stránkách www.kubaz.cz), naleznete 5 .pas souborů obsahujících samotný program. Členění je následující:

- **simulace.pas** – obsahuje samotné jádro programu, reakce na uživatelův vstup, zpracovávání příkazů a skriptů a vykreslování OpenGL scény.

- **initgl.pas** – obsahuje funkce inicializující OpenGL okno a vykreslování.
- **controls.pas** – obsahuje kód starající se o přijímání vstupu z klávesnice a myši.
- **particles.pas** – kód starající se o simulaci částic – obsahuje funkce na ukládání poloh, zachování hustoty apod. Obsahuje taktéž funkci pro výpočet hodnot mřížky.
- **particles_old.pas** – stará verze výpočtu částic, kde byly částice implementovány dvojím způsobem – pomocí dynamického pole a pomocí dynamického seznamu.
- **mcubes.pas** – obsahuje základní rutiny algoritmu Marching Cubes
- **lookuptable.pas** – obsahuje Bourkeho tabulku (viz [BO94]) upravenou Janem Hornem do Pascalovské syntaxe.
- **vectors.pas** – unita definující strukturu vektoru a přetěžující jeho operátory.
- **additional.pas** – unita definující několik pomocných funkcí.

Mimoto obsahuje .zip soubor složky records sloužící k ukládání záznamů a scripts sloužící k ukládání skriptů. K tomuto ukládání záznamů (ve formě obrázků) slouží knihovna Vampyre Imaging (viz http://imaginglib.sourceforge.net/).

Samotné zdrojové kódy zde nebudou rozebrány, nemělo by to smysl. Všechny důležité myšlenky již byly vyloženy v předcházejících kapitolách.

Apendix – alternativní metoda výpočtu hodnot funkce

Při výpočtu hodnot této funkce můžeme opět využít systému vyhledávání sousedů. K tomu si rozšíříme strukturu KrychleS, která nám nyní bude ukládat i informaci o tom, zda daná krychle již byla propočítána (jinak bychom díky procházení sousedů každou krychli[8] propočítávali několikrát).

Algoritmus výpočtu hodnot mřížky

```
pro všechny indexy [m,n,o] dělej
        KrychleS[m,n,o].počítáno := nepravda

pro všechny vnitřní indexy [m,n,o] v poli KrychleS dělej
        pokud length(KrychleS[m,n,o].částice) > 0 tak

            pro všechny indexy [mm,mn,mo] sousedů krychle [m,n,o] dělej
                pokud ne KrychleS[mm,mn,mo].počítáno tak
                    pro každý bod [i,j,k] mřížky ležící v krychli [mm,mn,mo] dělej
                        q := 0
                        pro všechny částice c v krychli [m,n,o] a jejích sousedech dělej
                                a := 1-| Částice[c].R-vektor(x,y,z) |/h
                                pokud a > 0 tak q := q +  a·a

                        Mřížka[i,j,k] := sqrt(q)

                    KrychleS[mm,mn,mo].počítáno := pravda
```

Zde popsaný algoritmus je velmi náročný na implementaci. Čtenáři jeho implementaci doporučuji pouze v případě, že chce experimentovat s efektivitou algoritmů v různých případech, jinak je rozumné implementovat algoritmus popsaný v sekci 3.2.10.

Sousedem krychle [m,n,o] rozumíme všechny krychle s indexy [m-1,n-1,o-1] až [m+1,n+1,o+1], tedy i krychli [m,n,o] samotnou (proto je nutné procházet pouze „vnitřní indexy“ [m,n,o], jinak bychom se např. pro mm = m-1 či mm = m+1 pokusili číst z pozice mimo mřížku). Rozveďme ještě, co přesně míníme pojmem „každý bod [i,j,k] mřížky ležící v krychli [mm,mn,mo]“. Známe-li polohu bodu [0,0,0] (tj. počátečního bodu) mřížky algoritmu Marching Cubes (označme si ji (X,Y,Z)), pak může být tento cyklus implementován následovně:

```
sx := floor((mm*h-X)/h_mřížka);
sy := floor((mn*h-Y)/h_mřížka);
sz := floor((mo*h-Z)/h_mřížka);

ex := floor(((mm+1)*h-X)/h_mřížka);      /při počítání krychliček by zde mělo být -1,
ey := floor(((mn+1)*h-Y)/h_mřížka);      aby se žádná krychlička nepočítala dvakrát.
ez := floor(((mo+1)*h-Z)/h_mřížka);      Ve skutečnosti by však číslo zde mohlo být i
```

8 Pojem „krychle“ zde budeme používat ve smyslu prvku pole KrychleS, pojem krychlička ve smyslu prvku pole Krychle, které slouží algoritmu Marching Cubes

```
                                        menší/
pokud je sx, sy nebo sz < 1 tak je nastav na 1    /abychom nečetli mimo mřížku/
pokud je ex, ey nebo ez přesahuje rozměry mřížky tak je nastav na maximální rozměr

for i := sx to ex do
        for j := sy to ey do
                for k := sz to ez do
                        ...
```

h zde značí interakční poloměr částic (tedy hrany KrychleS), $h_{mřížka}$ značí hranu krychličky algoritmu Marching Cubes.

Vzato do důsledku je možné algoritmus výpočtu hodnot mřížky implementovat pomocí tří vnořených for cyklů (indexy m,n,o), které v sobě obsahují 27 „projití" možných hodnot indexu [mm,mn,mo], z nichž každé obsahuje tři do sebe vnořené for cykly procházející proměnné i, j, k, které obsahují 27 for cyklů procházejících všechny částice v okolí krychle [m,n,o] (princip tohoto procházení byl již zmíněn v sekci 3.2.3).

Doporučuji čtenáři, aby si na projití 27 možných hodnot indexu [mm,mn,mo] napsal makro a v popsaném algoritmu používal dvě do sebe vnořená makra, pokud to preprocesor kompilátoru dovoluje.

Verze, kterou jsem zde popsal, však stále není zcela optimalizována. Další optimalizací může být, že jsou procházeni jen sousedi, u kterých to má smysl – nachází-li se v krychli jedna částice úplně vpravo, zcela jistě nemá smysl procházet kvůli tomu sousedy vlevo. Proto je rozumné nejdříve si jedním cyklem projít všechny částice a na základě toho rozhodnout, které krychle propočítávat. Vzdálenost od stěny krychle, při které už začneme sousedy uvažovat, označme d. Je ji potřeba určit experimentálně z daného nastavení konstant. Pro h = 0,7, f_0 = 2 a vysoké rozlišení mřížky mi postačovalo d = 0,15.

Nalezení krychlí, které je třeba propočítávat

```
vlevo, vpravo, dole, nahoře, vzadu, vpředu := nepravda
pro každou částici i v krychli [m,n,o] dělej
        pokud Částice[i].R.x < m·h+d tak vlevo := pravda
        pokud Částice[i].R.x > (m+1)·h-d tak vpravo := pravda
        pokud Částice[i].R.y < n·h+d tak dole := pravda
        pokud Částice[i].R.y > (n+1)·h-d tak nahoře := pravda
        pokud Částice[i].R.z < o·h+d tak vzadu := pravda
        pokud Částice[i].R.z > (o+1)·h-d tak vpředu := pravda

/zde vykonáme kód na základě zjištěných hodnot – např. krychle [m-1,n,o] se projde
pouze tehdy, je-li vlevo pravdivé/
```

Výsledky a závěr (a epilog autora)

V předložené práci bylo úspěšně vyřešeno zobrazování ekvipotenciálních ploch algoritmem Marching Cubes včetně výpočtu normálových vektorů. Tento algoritmus byl podrobně rozebrán a vysvětlen. Byla vyvinuta také metoda nazvaná „pseudosvětlo“, která dokáže imitovat smooth shading bez výpočtu normálových vektorů. Mimo jiné byl také popsán tzv. „ježura efekt“.

V další části byla vyřešena implementace myšlenek nastíněných v článku [PVFS05] včetně optimalizace na základě mřížky sousedů – bylo implementováno zachování dvojí hustoty, viskózní chování, plastické a elastické chování a byl vysvětlen princip interakce částic kapaliny s objekty. Byl také vyřešen princip interakce více kapalin (jež v článku [PVFS05] není řešen a je uveden jako jedno z možných pokračování výzkumu v daném oboru). Taktéž bylo vyřešeno zobrazování jedné či více kapalin (dvěma odlišnými způsoby) algoritmem Marching Cubes optimalizovaným pomocí mřížky sousedů.

V následující části byly popsány matematické detaily teorie – nejedná se však o žádné pokrokové myšlenky, jde pouze o souhrn metod popsaných v jiných článcích.

Na závěr bylo vysvětleno propojení s ray tracerem POV-Ray, díky němuž je možné vytvářet fotorealistické snímky (a z těchto snímků posléze také videa). Tím se tato práce posunuje na úroveň požadovanou profesionálními grafiky a filmaři. Je dosaženo výsledků, jež jsou publikovatelné na mezinárodních konferencích z daného oboru.

Doufám, že má práce bude české programátorské obci se zájmem o 3D grafiku výrazným přínosem. Již nyní vím, že nebyla psána nadarmo, protože první zveřejněná verze se setkala s velmi kladným ohlasem a výsledky této práce již v době dokončení současné verze využíval nejméně jeden programátor.

Na závěr bych rád poznamenal, že tvorba textu učebnicového charakteru by měla být vždy dynamickým procesem, který se vyvíjí na základě ohlasů a připomínek čtenářů. Proto mě neváhejte kontaktovat, pokud naleznete jakékoliv chyby či nedořešené problémy. Veškeré důležité kontaktní informace můžete nalézt na webu www.kubaz.cz.

Seznam použité literatury

[PVFS05] CLAVET, Simon; BEAUDOIN, Philippe; POULIN, Pierre. Particle-based Viscoelastic Fluid Simulation. *2005 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*. 2005. s. 219–228. ISBN 1-59593-198-8.

[MC87] LORENSEN, William E.; CLINE, Harvey E. Marching Cubes. A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.* 1987, vol. 21, no. 2, s. 163–169. ISSN: 0097-8930.

[BO94] BOURKE, Paul. *Polygonising a scalar field* [online]. 1994. Dostupné z URL: <http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/>.

[LI03] LINDH, Mats: *Marching Cubes*. Březen 2003. Dostupné z URL: <http://www.ia.hiof.no/~borres/cgraph/explain/marching/p-march.html>.

[HO01] HORN, Jan. *Metaballs* [počítačový program se zdojovými kódy]. 2001. Dostupné z URL: <http://www.sulaco.co.za/opengl_project_metaballs.htm>.

[LVT03] LEWINER, Thomas; LOPES, Hélio; VIEIRA, Antônio W.; TAVARES, Geovan: Efficient implementation of Marching Cubes' cases with topological guarantees. *Journal of Graphics Tools.* 2003, 8, 2, s. 1–15. ISSN: 1086-7651.

[WA01] WALD, Ingo; SLUSALLEK, Philipp: State of the Art in Interactive Ray Tracing. *EUROGRAPHICS 2001*. 2001, s. 21–42.

[BI05] BIKKER, Jacco. *Raytracing: Theory & Implementation.* 6.10.2005.Dostupné z URL: <http://www.devmaster.net/articles/raytracing_series/part1.php>.

[NO07] NOVÁK, Ondřej. *Simulace viskózních kapalin*. Praha, 2007. 52 s. Diplomová práce na Fakultě elektrotechnické ČVUT. Vedoucí diplomové práce Jaroslav Sloup.

[MCG03] MÜLLER, Matthias; CHARYPAR, David; GROSS, Markus. Particle-Based Fluid Simulation for Interactive Applications. *2003 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*. 2003, s. 154–159. ISBN 3-905673-04-5.

[FPC07] CANNEYT, Michaël Van. *Free Pascal: Reference guide* [online]. Verze 2.0 (2007). Dostupné z URL: <http://www.freepascal.org/docs/ref.pdf>.

[OpenGLRB] SHREINER, Dave; WOO, Mason; NEIDER, Jackie; DAVIS, Tom; Překlad FADRNÝ, Jiří. *OpenGL: Průvodce programátora.* 1. vyd. Brno: Computer Press, a.s., 2006. 679 s. ISBN 80-251-1275-6.