

Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Ondřej Mocný

Real-time fyzikální simulace pro mobilní zařízení

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: Mgr. Ondřej Sýkora

Studijní program: informatika, programování

2009

Chtěl bych poděkovat Ondrovi Sýkorovi za to, že se mi po celou dobu psaní práce věnoval a pilně hledal vecné, pravopisné i stylistické chyby, které jsem napáchal.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne: 23. června 2009

Ondřej Mocný

Obsah

1	Úvod	6
2	Analýza problému	8
2.1	Rozhlédnutí	8
2.2	Omezení určená mobilní platformou	10
2.3	Počítačová simulace	10
2.4	Mass-spring systém	11
2.5	Verletova integrace	11
2.6	Pružiny	12
2.6.1	Pořadí pružin	13
2.7	Reprezentace dynamických objektů	13
2.7.1	Elastická deformace pomocí pružin	13
2.7.2	Elastická deformace pomocí zachování tvaru	14
2.7.3	Zachování objemu aplikováním tlaku	14
2.7.4	Zachování objemu geometrickou interpretací	15
2.8	Reprezentace statických objektů	17
2.9	Reprezentace dlaždic	17
2.10	Částice	18
2.11	Kolize	18
2.11.1	Broad phase	19
2.11.2	Narrow phase	22
2.11.3	Tření	34
2.11.4	Lepení	35
2.12	Stabilita	36
3	Implementace	37
3.1	Optimalizace paměťových nároků	37
3.2	Optimalizace rychlosti	38
3.2.1	Viditelnost	38

3.2.2	Způsob psaní kódu v Javě	39
3.2.3	Nalezení úzkých hrdel	40
3.2.4	Paměť	40
3.2.5	Optimalizační programy	41
3.2.6	Instance tříd	41
3.2.7	Metody	42
3.2.8	Inlinování	42
3.2.9	Optimalizace elementárních operací	42
4	Využití v praxi	46
4.1	Příklad použití	46
4.2	Experimenty	47
5	Závěr	54
	Reference	55
A	Seznam zkratk	58
B	Seznam cizích pojmů	59
C	Obsah přiloženého CD	60

Název práce: Optimalizace fyzikální simulace pro mobilní telefony
Autor: Ondřej Mocný
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: Mgr. Ondřej Sýkora
e-mail vedoucího: ondrasej@centrum.cz

Abstrakt: V bakalářské práci se zabývám implementací knihovny pro fyzikální simulaci na mobilních telefonech. Uvádím několik možných přístupů k problému, ze kterých jeden vyberu a popíši důkladněji. Dále se zabývám obecnými postupy optimalizace programů určených pro mobilní telefony s cílem zrychlit implementovanou simulaci. Nakonec uvádím příklad použití simulátoru, na kterém demonstruji jeho výkon a použitelnost v praxi.

Klíčová slova: fyzikální simulace, mobilní telefony, optimalizace, hry

Title: Optimization of the physical simulation for mobile phones
Author: Ondřej Mocný
Department: Department of Software and Computer Science Education
Supervisor: Mgr. Ondřej Sýkora
Supervisor's e-mail address: ondrasej@centrum.cz

Abstract: In the presented work I discuss possible options of implementing a physical simulation that can be run on today's mobile phones. I describe several approaches from which I choose one and discuss it in more depth. Next, I talk about general methods of optimizing programs targeted to mobile phones and how they apply to the implemented physical simulation. Finally, I show a very simple game which demonstrates the power of the simulator and it's usability in praxis.

Keywords: physical simulation, mobile phones, optimization, games

Kapitola 1

Úvod

Mobilní telefony jsou dnes běžnou součástí každodenní komunikace a většina lidí si život bez nich ani nedokáže představit. Není proto divu, že se mnoho výrobců software během několika posledních let snaží (a velmi úspěšně) na tuto platformu zaměřit a dodávat pro ni software. Část tvoří aplikace vyvíjené na objednávku firem (různé bankovní aplikace, apod.), většinu ale zabírají mobilní hry. A právě na nich je zajímavé pozorovat, jak se v průběhu času měnily. Do značné míry to totiž kopíruje historický vývoj her na PC.

Zpočátku, když byl trh s hrami na mobilní telefony ještě poměrně prázdný, byly hry vyvíjeny často i jen jedním člověkem, který hru naprogramoval, nakreslil všechnu potřebnou grafiku a vytvořil zvuky. Uživatelé byli v té době velmi nenároční a mobilní telefony měly tak tvrdá technologická omezení, že ani nebylo možné vytvářet složitější hry. Postupem času začaly mobilní telefony dostávat rychlejší procesory, přibývala jim paměť a zákazníci začli požadovat technologicky dokonalejší hry, za které by byli ochotni zaplatit. Vznikla první vývojářská studia, které se specializovala pouze na výrobu mobilních her. Postupně se vytvořila globální distribuční síť a distributoři začli požadovat vysokou kvalitu her co se týče odladěnosti pro konkrétní mobilní telefony. Protože typů mobilních telefonů bylo už v té době na trhu velmi mnoho a každý se choval odlišně, nebylo ani možné vyvíjet mobilní hry jinak než v rámci větší firmy, která měla dostatek prostředků na to, aby mohla pro vývojáře nakoupit rozličné typy mobilních telefonů. Zároveň během této doby vymizely z trhu nejstarší modely telefonů, čímž došlo ke zmírnění technologických limitací, které ovlivňovaly kvalitu her.

Je ale zajímavé, že se tato laťka v posledních letech přestala prakticky zvyšovat, přestože výkon mobilů neustále roste. Komplexnost her se od té

doby příliš nezměnila, jen jsou již dokonale odladěné pro všechny mobilní telefony na trhu. Podle mého názoru distributoři zjistili, že zákazníci se při nákupu neřídí odbornými recenzemi, ale často jen názvem či obrázkem uvedeným u hry v katalogu. Myslím, že nevyužitého výkonu telefonů je škoda. V této práci bych proto chtěl ukázat, jakým způsobem lze tento výkon využít ke zlepšení kvality mobilních her.

Fyzikální engine je knihovna tříd a funkcí, která simuluje tuhá a deformovatelná tělesa podléhající Newtonovým zákonům. Je to dnes již standardní součást počítačových her. Zvyšuje interaktivitu prostředí, dává nové možnosti, jak zlepšit hrátelnost, a připraví pro hráče uvěřitelnější prostředí. U mobilních her tomu tak není, přestože telefony na to mají dostatečný výkon – obzvláště u her, jejichž herní svět funguje pouze ve dvou dimenzích, a nemusejí tak věnovat většinu výkonu telefonu pro kreslení 3D grafiky. Chtěl bych proto ukázat, jak je možné vytvořit komplexní fyzikální simulaci použitelnou na mobilních telefonech, a jak ji zoptimalizovat tak, aby uspokojivě fungovala i na starších a méně výkonných modelech telefonů. Rád bych zde zdůraznil, že se z výše uvedeného důvodu budu věnovat pouze simulaci ve dvou rozměrech.

Práce je rozdělena do dvou částí. V Kapitole 2 popíši, jak se dá navrhnout jednoduchý fyzikální simulátor, upozorním na problémy, které se mohou při implementaci vyskytnout, a poskytnu k nim řešení. V kapitole 3 se budu věnovat optimalizaci tohoto simulátoru pro vyšší výkon, aby byl simulátor použitelný na co nejširším spektru mobilních telefonů. V kapitole 4 pak uvedu příklad použití a poskytnu výsledky měření prováděných na skutečných mobilních telefonech, které ukazují možnosti využití simulátoru v praxi.

Kapitola 2

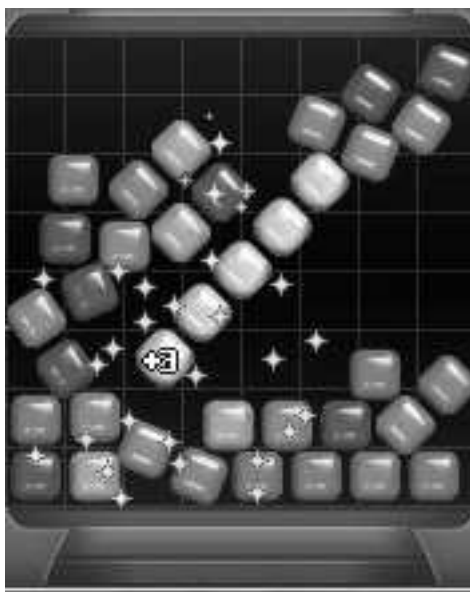
Analýza problému

Vzhledem k omezenému výkonu mobilních telefonů je potřeba vybrat takový fyzikální model, jehož simulace nebude příliš výpočetně náročná, ale zároveň bude uvěřitelná. Tedy taková, aby hráči, který bude hrát hru používající tento simulátor, připadalo, že se objekty chovají podobně jako v reálném světě. V minulosti jsem již podobnou fyzikální simulaci vytvořil [20]. Šlo o modelování objektů pomocí tuhých těles: jde o přístup, který se v oblasti počítačových her běžně používá. Je poměrně přesný – objekty, které mají být tuhé, se opravdu nedeformují, tření se počítá korektně, apod. Ale i v té nejpřímější a nejméně výpočetně náročné implementaci stále jde o příliš náročné výpočty a výsledná simulace má problémy se stabilitou a přesností.

Zvolil jsem proto jiný přístup - tzv. *mass-spring systém* (někdy označovaný jako *spring-mass systém*) [25], kdy se objekty modelují jako množina hmotných bodů spojených pružinami. Tento přístup se většinou nepoužívá z toho důvodu, že je nemožné dosáhnout naprosté tuhosti těles, a ta pak vypadají, jako by byla vytvořena z gumy. U her na mobilní telefony to ale podle mě nevádí. Výhodou je, že je takto možné simulovat i deformovatelné objekty, jak se ukáže dále v této kapitole.

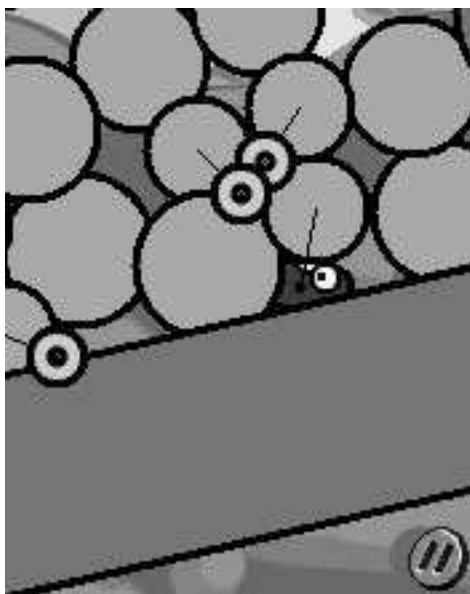
2.1 Rozhlédnutí

Existuje několik mobilních her, jejichž autoři se o použití fyzikální simulace pokusili. *Freestyle Moto-X II* [31] vydaný v roce 2006 ukázal, jak i velmi jednoduchá fyzikální simulace může významně vylepšit hratelnost. *Flexis Extreme* [29] z počátku roku 2007 už vytváří složitější simulaci s větším počtem objektů, které jsou ovšem velmi jednoduché a celá simulace probíhá pouze v prostoru jedné obrazovky mobilu. *SolaRola* [5] vydaná na konci roku 2007



Obrázek 2.1: Screenshot ze hry Flexis Extreme. Hráč má za úkol naskládat na sebe předepsaným způsobem krabičky, které se pohybují ve světě s newtonovskou fyzikou.

obsahuje komplexnější fyzikální engine podporující velmi jednoduché (pouze kulovité) tuhé a deformovatelné objekty, ale celá simulace působí velmi ne-realisticky a naopak mírně zhoršuje dojem z celé hry. Samostatný fyzikální engine *DyMiX* [20] napsaný mnou v půli roku 2007 vytváří poměrně realistickou simulaci tuhých těles, ale nepodporuje deformovatelné objekty, není příliš robustní a je náročnější na výkon procesoru, proto ho nelze použít na starších mobilních telefonech. *Afrodite* [12] je také samostatný fyzikální engine podobně jako *DyMiX*, nicméně není vytvořen s ohledem na výkon dnešních mobilních telefonů. Bohužel je velmi pomalý a v praxi téměř nepoužitelný. *Wonder Blocks* [9] je hra do jisté míry podobná hře *Flexis Extreme*, podporuje dokonce objekty složitějších tvarů, ale nejedná se o plnohodnotnou fyzikální simulaci. Objekty se totiž po umístění stanou statickou součástí herního světa a tím se rapidně sníží počet dynamických objektů a simulace se velmi zjednoduší.



Obrázek 2.2: Screenshot ze hry SolaRola. Hráč ovládá gumovou kulovitou postavičku, která při procházení světem interaguje s různými objekty tvaru koule. Na obrázku je právě zavalena několika kameny.

2.2 Omezení určená mobilní platformou

Pokud má aplikace fungovat na co největším počtu mobilních telefonů (a to je cíl většiny vývojářů), není jiná možnost, než zvolit jako prostředek k vývoji Java 2 Micro Edition (J2ME). Existují i jiné technologie, ale ty nejsou tak rozšířené. Toto je třeba vzít také v úvahu při implementaci a následné optimalizaci simulace.

2.3 Počítačová simulace

Je potřeba si uvědomit, že počítače (a sem samozřejmě patří i mobilní telefony) nejsou schopny v reálném čase efektivně analyticky řešit diferenciální rovnice pohybu. K simulaci se proto typicky přistupuje diskrétně, po krocích, a pouze se aproximuje.

Dalším problémem je, že nejsme schopni na počítači analyticky integrovat zrychlení (resp. rychlost), a získat tak rychlost (resp. novou pozici), protože se zrychlením nemůžeme pracovat v jeho explicitním vyjádření. Proto je nutné integrovat numericky. S tím je spojena celá řada problémů; například

může docházet k akumulaci chyb a numerické nestabilitě. Každý numerický integrátor má své výhody a nevýhody a je nutné vybrat takový, který nejlépe splní potřeby konkrétní simulace. Numerickou a fyzikální stabilitu počítačových simulací rozebírá např. Rhodes [27].

2.4 Mass-spring systém

Jak jsem již zmínil, objekty se při tomto přístupu modelují jako soustava hmotných bodů spojených pružinami. Je zde přitom několik možností, jak tyto hmotné body zvolit. První přístup, který využívají např. Rivers s Jamesem [28], je, že každý hmotný bod zastupuje určitou část modelovaného tělesa. Jeho hmotnost se pak nastaví na celkovou hmotnost jemu příslušné části tělesa. Používanější je ale v podobných simulacích model, ve kterém hmotné body tvoří pouze povrch tělesa a jejich hmotnosti se zvolí tak, aby jejich součet byl stejný jako celková hmotnost tělesa.

Patrně nejznámějším textem v této oblasti je *Advanced Character Physics* od Thomase Jakobsena [13], ze kterého jsem vycházel i já. Jakobsen doporučuje pro integraci pohybu hmotných bodů použít *Verletův integrátor* [3] využívaný hlavně v oblasti molekulární dynamiky, který následně velmi usnadní simulaci pružin stejně jako reakci na kolize objektů. Podobný přístup popisují také Müller, Heidelberger, Hennix a Ratcliff [23].

2.5 Verletova integrace

Tuto metodu numerické integrace vyvinul Loup Verlet pro potřeby simulace molekulární dynamiky a poprvé ji popsal ve svém článku *Computer Experiments on Classical Fluids*, kde lze najít i odvození a analýzu tohoto integrátoru [32].

Charakteristickým rysem Verletovy numerické integrace je to, že nepoužívá při výpočtu explicitně vyjádřenou rychlost, ale místo toho si každý objekt musí pamatovat svou předchozí pozici (tj. pozici před provedením integrace v předešlém simulačním kroku). Integrace pak probíhá podle následující rekurentní rovnice:

$$x_{n+1} = 2x_n - x_{n-1} + a \cdot \Delta t^2, \quad (2.1)$$

kde x_{n+1} je nová pozice, x_n je aktuální pozice, x_{n-1} je předchozí pozice, a je zrychlení a Δt je čas uplynulý od předchozího kroku simulace. Verletova

integrace dává lepší výsledky, pokud je délka kroků simulace konstantní. Zároveň pak lze eliminovat dvě násobení. Toto omezení je často problémem, ale mobilní hry běží s tak malým počtem snímků za sekundu (FPS - Frames Per Second), že pevný simulační krok je naopak výhodou - jinak by byla znatelná změna plynulosti simulace.

Nespornou výhodou Verletova integrátoru, která je také hlavním důvodem jeho rozšířenosti, je vysoká stabilita simulace. To byl i důvod, proč jsem se ho rozhodl použít. Cenou za stabilitu je nižší přesnost, to ale v omezeném prostředí mobilních telefonů nevádí.

Další výhodou je absence explicitního vyjádření rychlosti. Rychlost je vyjádřena implicitně pomocí předchozí pozice, tudíž se o ni nemusíme starat, pokud ji nechceme přímo změnit. Tato výhoda se projeví při řešení kolizí a pružin - vše, co pro to musíme udělat, je změnit pozici hmotných bodů a rychlost se upraví implicitně sama.

2.6 Pružiny

Pro dokončení mass-spring systému potřebujeme ještě určit způsob, kterým bude simulace modelovat pružiny. Pružina je definována dvěma hmotnými body p_1 a p_2 , jejichž hmotnosti jsou popořadě m_1 a m_2 a pozice \vec{x}_1 a \vec{x}_2 ; dále tuhostí k (z intervalu $(0; 1]$, $k = 1$ znamená pevný spoj, tzv. *distance joint*) a klidovou délkou r . Při aplikaci pružiny posuneme p_1 a p_2 k sobě nebo od sebe tak, aby byly po aplikaci od sebe vzdálené o r . Posuneme je tedy ve směru vektoru $\vec{v} = \vec{x}_1 - \vec{x}_2$. Délku posunu určují jejich hmotnosti (čím je bod těžší, tím méně se musí posunout), aktuální vzdálenost a klidová délka. Velikost posunutí d_1 bodu p_1 spočítáme jako

$$d_1 = -k \cdot (|\vec{v}| - r) \cdot \frac{m_2}{m_1 + m_2}. \quad (2.2)$$

Bod p_1 posunujeme směrem k p_2 , tedy proti \vec{v} , proto je d_1 záporné. Velikost posunutí d_2 bodu p_2 pak dostaneme snadno z d_1 jako $d_2 = -k \cdot (|\vec{v}| - r) - d_1$.

Tento způsob je velmi přímočarý, přesto v praxi fungující až překvapivě dobře. Výsledek je nepřesný, avšak dostačující. Rád bych upozornil na to, že při aplikaci pružin „standardním“ způsobem, kdy na p_1 a p_2 budeme působit silami, bude docházet k nestabilitě simulace, a proto je třeba se tomu vyhnout (viz Rhodes [27]).

Při tomto způsobu aplikace pružiny můžeme nějakou jinou pružinu opět „natáhnout“, i když jsme ji předtím již aplikovali, protože jeden hmotný bod

může být součástí více pružin. Tomuto jevu se obecně nedá vyhnout, ale existuje metoda nazvaná *relaxace*, kterou se můžeme alespoň blížit k ideálnímu stavu. Spočívá v tom, že všechny pružiny v systému aplikujeme iterativně několikrát za sebou, dokud nejsme s výsledkem spokojeni. V praxi postačuje nastavit pro každý objekt pevný počet iterací.

2.6.1 Pořadí pružin

Jelikož se pružiny aplikují sekvenčně (tj. postupně za sebou, nikoliv všechny najednou), nabývá na důležitosti pořadí, v jakém je procházíme, resp. v jakém je do systému přidáme. Pokud bychom je totiž procházeli jen v jednom směru, objekt by začal pomalu v opačném směru rotovat (vlivem jevu popsaného v předchozím odstavci).

Jedna možnost je postupovat podle rostoucí y-ové souřadnice hmotných bodů a postupně přidávat do systému pružiny, které budou na daný hmotný bod působit. V praxi se mi ale více osvědčilo procházet pružiny v jednom směru, ale každou druhou vynechat a v druhém průchodu projít ty, které byly předtím vynechány. Tímto způsobem objekt nedostává prakticky žádnou rotaci a nechová se neočekávaně.

2.7 Reprezentace dynamických objektů

Nyní jsme schopni vytvořit složitější objekty, tak, že jejich povrch pokryjeme hmotnými body. Čím více jich bude, tím bude jejich povrch jemnější, ale simulace bude náročnější. Po obvodu je spojíme pružinami. Dále je potřeba zajistit, aby se objekt při vnějším vlivu nedeformoval plasticky a aby se se snažil zachovat svůj objem a/nebo tvar.

2.7.1 Elastická deformace pomocí pružin

Jednou z možností je spojit pružinami i body, které spolu na povrchu objektu nesousedí. Vytvoříme tak systém pružin, které objekt zevnitř podepřou a budou bránit jeho deformaci. Máme několik možností, jak toto provést. Buď můžeme spojit jen hmotné body, které jsou z hlediska objektu „naproti“, nebo spojíme každý bod s každým, a nebo do systému přidáme jeden hmotný bod navíc, který bude tvořit střed objektu, a s ním spojíme pružinami všechny ostatní body. První dva způsoby fungují relativně dobře, ale výsledek není na pohled dostačně uspokojivý. Třetí způsob je velmi ne-

stabilní - středový bod se může dostat ven z objektu a ten se pak obrátí „naruby“.

Existují i další možnosti, jak body navzájem pospojovat, ale potřebují většinou větší počet pružin, což je pro mobilní telefony nevýhodné z hlediska výpočetní náročnosti (už tyto tři popsány způsoby jsou výpočetně velmi náročné). Zároveň je tu problém s celkovou stabilitou objektu, protože se může v některých situacích otočit „naruby“. Patrně by pomohlo rapidně zvýšit počet hmotných bodů a pružin, jako to zkusil např. West [34], ale to už by nebylo na mobilních telefonech upočitatelné. Proto je lepší zkusit na to jít jinak.

2.7.2 Elastická deformace pomocí zachování tvaru

Velmi zajímavý způsob simulace deformovatelných objektů předvedli Müller, Heidelberger, Teschner a Gross [22]. Jejich metoda je ovšem vhodná spíše pro složité objekty a na mobilní telefony je příliš sofistikovaná a náročná na výpočet. Podobný přístup implementovaný v praxi lze nalézt např. v knihovně JelloPhysics [33].

2.7.3 Zachování objemu aplikováním tlaku

Matyka [18] popsal metodu simulace deformovatelných objektů pomocí aplikace tlakové síly. Tato metoda je založena na použití modelu ideálního plynu uvnitř objektu. Velikost tlakové síly P je dána Clausius-Clapeyronovou rovnicí

$$PV = nRT \quad (2.3)$$

kde V je aktuální objem tělesa, n je počet molů plynu, R je konstanta ideálního plynu a T je teplota. Hodnota nRT je v našem případě pro jeden objekt konstantní, takže tento výraz můžeme nahradit jedinou konstantou K , kterou budeme řídit míru vyplnění objektu plynem. Jelikož se pohybujeme ve dvou rozměrech, můžeme V vypočítat jako plochu mnohoúhelníku, jehož vrcholy jsou pozice hmotných bodů na povrchu objektu. Plochu mnohoúhelníku vypočteme pomocí Gaussovy formule

$$V = \frac{1}{2} \sum_{i=0}^{n-1} x_i y_{i+1} - x_{i+1} y_i \quad (2.4)$$

kde x_i a y_i jsou souřadnice i -tého vrcholu mnohoúhelníku a n je počet vrcholů. Pokud index i je větší nebo roven n , vezmeme místo něj $(i - n)$ -tý

vrchol. Gaussovu formuli můžeme zjednodušit na výpočetně méně náročný tvar

$$V = \frac{1}{2} \sum_{i=1}^n x_i (y_{i+1} - y_{i-1}). \quad (2.5)$$

Tlakovou sílu aplikujeme na objekt tak, že pro každou hranu e určenou hmotnými body p_1, p_2 (jejich pozice označme popořadě x_1, x_2) spočteme normálový vektor \vec{n} tak, aby měl jednotkovou délku a směřoval ven z objektu, a spočítáme nové pozice x'_1, x'_2 bodů p_1, p_2 , a to následujícím způsobem:

$$x'_1 = x_1 + |e| \frac{P}{2} \vec{n} \quad (2.6)$$

$$x'_2 = x_2 + |e| \frac{P}{2} \vec{n}. \quad (2.7)$$

Tlakovou sílu je potřeba vydělit dvěma, protože působí na hranu, která je určena dvěma hmotnými body a každý z nich přispívá na určení této hrany stejnou měrou.

Při implementaci je možné tuto dvojku začlenit do konstanty K , a ušetřit tak jedno dělení (případně shift). Dále je dobré si všimnout, že při výpočtu normálového vektoru \vec{n} tento vektor normalizujeme, tj. dělíme ho jeho velikostí, a pak ho opět touto velikostí násobíme. Výpočet tedy můžeme zjednodušit takto:

$$x'_1 = x_1 + P' \vec{n}' \quad (2.8)$$

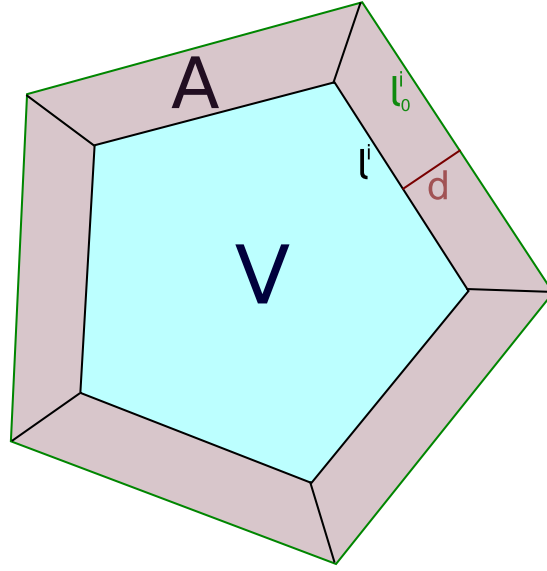
$$x'_2 = x_2 + P' \vec{n}' \quad (2.9)$$

kde \vec{n}' je vektor o velikosti $|e|$ kolmý k hraně e a směřující ven z objektu. P' je velikost tlakové síly, ale za použití konstanty K' , kde $K' = \frac{K}{2}$.

Tento přístup se v praxi osvědčil a dává na pohled hezké výsledky. Zároveň je výpočetně velmi nenáročný. Objekt se pak chová, jako by byl vyplněn plynem. Dá se tak simulovat například gumový míč nebo balonek. Nevýhoda je, že se nezachovává objem objektů. Proto se tento přístup nehodí např. pro simulaci míče naplněného vodou.

2.7.4 Zachování objemu geometrickou interpretací

Grant Kot [16] mi doporučil přistupovat k zachování objemu tak, že se na simulovaný objekt díváme jako na mnohoúhelník a snažíme se zachovat jeho plochu. Představme si situaci ukázanou na obrázku 2.3. Modrý pětiúhelník (s obsahem V a obvodem $l = \sum_{i=0}^{n-1} l^i$) je aktuální stav objektu.



Obrázek 2.3: Zachování obsahu mnohoúhelníku. V je aktuální obsah, A je cílový obsah. d je vzdálenost, o kterou musíme posunout hranu l^i podle její normály, aby ležela na hraně l_0^i .

Potřebujeme jej změnit tak, aby jeho obsah byl V_0 , což je počáteční objem objektu (ten spočítáme při inicializaci objektu v simulaci). Aby byla změna rovnoměrně rozmístěná po celém povrchu objektu, posuneme každou jeho hranu o stejnou vzdálenost d podle jejího normálového vektoru \vec{n} . Z modrého pětiúhelníku tedy dostaneme fialový pětiúhelník, a změníme tak obsah o $A = V_0 - V$. Obsah fialového pětiúhelníku je V_0 a jeho obvod je $l_0 = \sum_{i=0}^{n-1} l_0^i$. Z obrázku lze vypožorovat, že prostor mezi modrým a fialovým pětiúhelníkem je tvořen lichoběžníky o výšce d a základnách l_0^i a l^i . Také je vidět, že A se rovná součtu ploch těchto lichoběžníků. Z toho dostáváme následující vztah:

$$A = \sum_{i=0}^{n-1} d \frac{l_0^i + l^i}{2}, \quad (2.10)$$

ze kterého můžeme (po dosazení l a l_0) vyjádřit d :

$$d = \frac{2A}{l + l_0}. \quad (2.11)$$

Každá hrana je určena dvěma hmotnými body. Abychom hranu posunuli, musíme posunout oba tyto body o vzdálenost d ve směru \vec{n} . Každý bod p_i

je krajním bodem dvou sousedních hran s normálami \vec{n}_1 a \vec{n}_2 . Abychom vyhověli oběma hranám, musíme p_i posunout ve směru vektoru $\vec{v} = \vec{n}_1 + \vec{n}_2$, který normalizujeme a promítneme do něj vektor $d\vec{n}_1$ (skalárním součinem). Novou pozici x' bodu p_i tedy spočteme takto:

$$x' = x + \frac{\langle d\vec{n}_1 | \vec{v} \rangle \vec{v}}{|\vec{v}|^2}, \quad (2.12)$$

kde x je aktuální pozice bodu p_i . Jak je vidět, používáme tu podobný přístup jako při aplikaci pružiny - nepůsobíme na objekt silou, ale přímo měníme pozice jeho hmotných bodů a jejich rychlosti se jim implicitně změny samy.

Výpočet můžeme zrychlit aproximováním \vec{v} jako vektoru kolmého na vektor $x_{i-1} - x_{i+1}$, kde x_{i-1} , x_{i+1} jsou popořadě pozice bodů p_{i-1} , p_{i+1} (sousední body p_i), a dále aproximováním $d\vec{n}_1 \vec{v} \approx d\vec{v}$. Tímto způsobem normalizujeme pouze jeden vektor místo původních tří (protože už nemusíme počítat n_1 a n_2) a výsledek vypadá na pohled téměř stejně. Dalšího možného urychlení dosáhneme aproximováním rovnice 2.11 následující rovnicí:

$$d = \frac{A}{l_0}. \quad (2.13)$$

Nemusíme tedy počítat ani l .

Tato metoda se v praxi také osvědčila a simulace působí vizuálně dobře. Objekt se nyní chová, jako by byl vyplněn kapalinou, a zachová svůj objem, i když je na něj shora vržen jiný objekt.

2.8 Reprezentace statických objektů

Statické objekty jsou takové objekty, které se samy nemohou posunovat, rotovat nebo deformovat. Nemusíme je tedy simulovat pomocí hmotných bodů a pružin, jako jsme to prováděli u dynamických objektů, ale stačí pouze definovat jejich tvar a polohu. Musíme ovšem umožnit uživateli tuto polohu měnit (uživatelé se má na mysli program, který bude simulátor používat). Dá se takto vytvořit například pohyblivá platforma, výtah nebo dveře. Pro pozdější optimalizaci při detekci kolize povolíme jen konvexní tvary.

2.9 Reprezentace dlaždic

Klasickou metodou reprezentace statického světa ve 2D hrách je tzv. *tiled map*. Svět se rozdělí na čtverečky (případně jiné pravidelné geometrické

útvary, např. kosočtverce nebo šestiúhelníky) stejné velikosti, tzv. dlaždice nebo *tiles*, a ty pak určují, jak bude vypadat výřez světa jimi určený. Mapu reprezentujeme ve dvourozměrném poli tilů, kde jsou uloženy číselné hodnoty identifikující typ tilu. Typy tilů se v mapě opakují, ale pokud se vhodně zkombinují, hře to po vizuální stránce neuškodí.

Tato reprezentace světa v mobilních hrách převládá, proto je nezbytné věnovat jí zvláštní pozornost. Tily můžeme simulovat podobně jako statické objekty, ale je dobré využít toho, že se často v mapě opakují. Vytvoříme proto sadu statických objektů (pro každý tile jeden), a když budeme chtít zjistit, jak vypadá tile na místě s indexy i a j , který má v mapě číselnou hodnotu t , podíváme se do předem připravené sady statických objektů na místo určené číslem t a nalezený statický objekt přesuneme na pozici $[i \cdot s_t, j \cdot s_t]$, kde s_t je velikost jednoho tilu. Ušetříme tak paměť.

Při implementaci je vhodné jako velikost tilu zvolit mocninu dvojky. Výrazně se pak zrychlí všechny algoritmy, které potřebují velikostí tilu dělit či násobit, protože to lze provést pomocí bitového posunu, což je velmi rychlá operace.

2.10 Částice

Částicí nazývám kulovitý objekt definovaný středem a poloměrem. Můžeme jej velmi snadno simulovat pomocí jednoho hmotného bodu umístěného ve středu objektu. Detekce kolize a následná reakce na ni pak bude výpočetně nenáročná, protože se musíme starat pouze o jeden hmotný bod. Tímto způsobem lze simulovat například protivníky ve hrách, protože u nich není nutná přesná detekce kolize a aproximace koulí postačí. Nevýhodou je, že se tento typ objektů nemůže otáčet. To ale v mobilních hrách nevádí, protože rotace bitmap je zde velký problém (platforma podporuje nativně pouze rotace o násobky 90 stupňů).

2.11 Kolize

Simulátor už tedy umí simulovat pohyb objektů. To ovšem nestačí. Simulace se stane uvěřitelnou, teprve až když bude korektně reagovat na kolize objektů. Kolize navíc nastávají velmi často, obzvláště v prostředí s gravitací. Proto je třeba věnovat jim nejvyšší pozornost.

Řešení kolizí je obecně složitý problém, který nemá jednoznačné řešení, a pro každou konkrétní situaci se hodí jiný postup. Christer Ericson ve své

knize *Real-Time Collision Detection* [6] popisuje různé přístupy k detekci kolize a poskytuje konkrétní řešení včetně implementačních detailů týkajících se optimalizace a numerické stability.

Reakce na kolize je také složitý problém, ale v tomto případě je řešení poměrně jednoduché. Využijeme opět toho, že nemáme rychlost hmotných bodů explicitně vyjádřenou, a tudíž se jí nemusíme zabývat. Vše, co je třeba k vyřešení kolize, je posunout hmotné body do takové polohy, aby objekty spolu nekolidovaly. Rychlost se pak upraví implicitně sama.

V následujících sekcích si ukážeme, jak detekovat kolize. Postup rozdělíme do dvou fází. V první fázi (*broad phase*) velmi rychlým algoritmem zjistíme, které páry objektů spolu kolidovat nemohou. Zbylé dvojice pak otestujeme v druhé fázi (*narrow phase*), kde už kolize přesněji spočítáme.

2.11.1 Broad phase

Tímto pojmem se označuje fáze detekce kolize, ve které se kolize otestují jen nahruho - najdou se páry objektů, které mohou potenciálně kolidovat. Tyto páry se pak dále testují v tzv. *narrow phase*, kdy se určí, zda ke kolizi opravdu došlo.

Axis-aligned bounding box

Protože objekty testujeme jen nahruho, vystačíme si pouze s aproximací jejich tvaru. Bude-li tato aproximace dostatečně přesná a zároveň jednoduchá na sestavení a na detekci kolize, výpočet značně urychlíme, protože rychle vyřadíme páry, které spolu nekolidují (a takových je většina). Z různých často používaných aproximací jsem zvolil tzv. *axis-aligned bounding box* (AABB), neboli „kvádr“, jehož stěny (ve 2D prostoru hrany) jsou rovnoběžné s osami souřadnic. Důvodem této volby byla skutečnost, že se s ní bude dobře pracovat v algoritmu Implicit Grid popsaném dále. Existuje několik variant AABB, z nichž jsem vybral tzv. *tight AABB*, protože při použití ostatních jsem naměřil znatelně nižší výkon. Výčet všech typů lze nalézt v [6] na straně 82. Tight AABB se snaží aproximovat objekt co nejlépe (nejtěsněji). AABB spočítáme tak, že pro každou souřadnou osu najdeme minimum a maximum mezi souřadnicemi všech hmotných bodů objektu. Při změně kteréhokoliv hmotného bodu je tedy třeba AABB přepočítat; jelikož se však AABB používá jen při detekci kolize, tak tento výpočet stačí v simulačním kroku provést pouze jednou (tj. před spuštěním detektoru kolizí).

Dva AABB se protínají, právě když se protínají jejich projekce na všech

souřadných osách. Pokud je první AABB určený čtveřicí pozic hraničních bodů $(x_{1,min}, y_{1,min}, x_{1,max}, y_{1,max})$ a druhý AABB čtveřicí pozic hraničních bodů $(x_{2,min}, y_{2,min}, x_{2,max}, y_{2,max})$, pak se tyto dva AABB se překrývají, pokud zároveň platí

$$x_{1,min} \leq x_{2,max} \quad (2.14)$$

$$x_{2,min} \leq x_{1,max} \quad (2.15)$$

$$y_{1,min} \leq y_{2,max} \quad (2.16)$$

$$y_{2,min} \leq y_{1,max}. \quad (2.17)$$

Implicit Uniform Grid

Implicit Uniform Grid (IG), jak už název napovídá, rozdělí svět na stejně velké buňky, uspořádané do mřížky, a pro každý sloupec a každý řádek si pamatuje seznam objektů, které v tomto sloupci/řádku jsou. Když je pak třeba zjistit, jaké objekty jsou v buňce na pozici (x, y) , IG se podívá, jaké objekty jsou ve sloupci y a zároveň v řádku x . Tabulka je tímto způsobem reprezentována implicitně a není potřeba ji mít v paměti uloženu celou. Ericson [6] IG popisuje jako datovou strukturu použitelnou pro rychlé zjištění přítomnosti objektů v blízkém okolí testovaného objektu (toto okolí odpovídá AABB objektu). IG můžeme s výhodou použít pro broad phase, která bude pak probíhat tak, že pro každý objekt najdeme objekty v jeho okolí, a pokud se jejich AABB budou překrývat, předáme tuto dvojici dále do narrow phase.

Seznam objektů vyskytujících se v sloupci/řádku můžeme reprezentovat pomocí lineárního spojového seznamu. V praxi je ale mnohem lepší tento seznam reprezentovat pomocí bitové masky, protože známe předem maximální počet objektů přítomných v simulaci nebo ho můžeme omezit. Díky tomu lze tyto bitové masky předalokovat, a zároveň můžeme mít objekty uloženy v poli. Každý objekt tak má jedinečné identifikační číslo (*ID*), které odpovídá indexu v poli, kde je objekt umístěn. Bitovou masku pro daný sloupec/řádek definujeme tak, že bude mít na pozici i bit rovný jedné, právě když se v tomto sloupci/řádku nachází objekt s ID rovným i . Bitovou masku určující seznam objektů v buňce na pozici x, y pak dostaneme tak, že provedeme bitový součin (\wedge) na bitové masky pro y -tý sloupec a x -tý řádek. Pokud potřebujeme zjistit seznam objektů ve více buňkách najednou (a to je běžná situace, protože objekt většinou zasahuje do více buněk), stačí použít bitový součet (\vee) na bitové masky získané z jednotlivých buněk a výsledná bitová maska určuje seznam objektů ve všech prohledávaných buňkách. Navíc, díky

distributivitě bitových operací, platí

$$(x_1 \wedge y_1) \vee \dots \vee (x_1 \wedge y_n) \vee (x_2 \wedge y_1) \vee \dots \vee (x_n \wedge y_n) = \quad (2.18)$$

$$= (x_1 \vee \dots \vee x_n) \wedge (y_1 \vee \dots \vee y_n), \quad (2.19)$$

takže nemusíme procházet všechny prohledávané buňky (v čase $O(n^2)$) a pro každou z nich počítat bitovou masku. Stačí projít zvlášť všechny sloupce, použít operaci \vee na jejich bitové masky, pak totéž provést s řádky a nakonec použít na získané (dvě) výsledné bitové masky operaci \wedge (čímž se sníží potřebný čas na $O(n)$).

Nyní je třeba ještě najít způsob, jak z bitové masky získat seznam objektů, resp. seznam jejich ID. Pokud bude simulace spouštěná na platformě, která reprezentuje záporná čísla pomocí dvojkového doplňku (viz např. Bulej [2]), získáme výpočtem

$$b = b_0 \wedge -b_0 \quad (2.20)$$

z bitové masky b_0 takovou bitovou masku b , která obsahuje jedničku pouze na nejnižší pozici, na které byla jednička v b_0 . Pokud má platforma hardwarovou podporu pro dvojkový logaritmus, stačí spočítat dvojkový logaritmus z b a výsledek je pozice bitu, na kterém byla jednička v b , čili ID nalezeného objektu. V opačném případě můžeme pozici jedničkového bitu nalézt lineárním průchodem přes všechny bity, případně metodou půlení intervalu (viz např. Töpfer [30]). Poté výpočtem

$$b_1 = b_0 \text{ xor } b \quad (2.21)$$

získáme bitovou masku b_1 , která je stejná jako b_0 , jen má nulu na pozici, kde má b jedničku. Celý výpočet opakujeme, dokud není bitová maska rovná nule.

Protože je simulace dynamická a objekty se mohou pohybovat, musí IG reflektovat tyto změny. Objekty v IG budeme aproximovat pomocí jejich AABB. Vložení objektu s ID rovným i do IG proběhne tak, že projdeme všechny sloupce, do kterých zasahuje AABB objektu, v jejich bitových maskách nastavíme i -tý bit na 1 a poté totéž provedeme s řádky, do kterých AABB zasahuje. Při každé změně AABB objektu vymažeme nejprve objekt z IG a pak ho tam znovu přidáme pomocí postupu popsání výše. Odebrání probíhá podobně jako přidání, jen i -tý bit nastavíme na 0.

Implicit grid je paměťově velmi efektivní způsob, jak implementovat broad phase, a přitom není o mnoho pomalejší, než například běžný *uniform grid* (opět viz Ericson [6]). Je ale třeba vhodně zvolit velikost jedné

buňky, protože to může výkon značně ovlivnit. Implicit grid funguje nejlépe, pokud jsou všechny objekty přibližně stejně velké a velikost buňky se zvolí o trochu větší než průměrná velikost objektů. Ve své předchozí práci ([21]) jsem provedl srovnání různých přístupů k broad phase. IG mi vyšel jako nejvhodnější pro mobilní telefony, hlavně díky nízké paměťové náročnosti, proto jsem ho použil i v této práci.

2.11.2 Narrow phase

Narrow phase je druhá fáze detekce kolize, ve které je k dispozici seznam dvojic objektů, které mohou potencionálně kolidovat, a je potřeba určit přesně, zda kolidují, a zjistit o kolizi další údaje, které se použijí v reakci na kolizi.

V následujících sekcích postupně probereme problematiku kolizí různých typů objektů a ukážeme si, jak se vypořádat s jejich kolizemi.

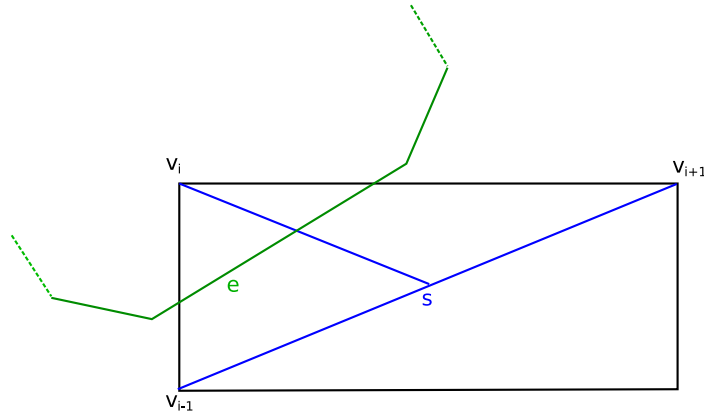
Dynamické vs. statické objekty

Náš cílový stav je takový, že žádný z hmotných bodů dynamického objektu b_1 nebude ležet uvnitř statického objektu b_2 . Pro každý hmotný bod p objektu b_1 zjistíme, zda leží uvnitř b_2 . Pokud ano, najdeme minimální vzdálenost, o kterou je třeba p posunout (tuto vzdálenost označme jako d_{mt} - minimum translation distance). Zároveň najdeme i vektor, v jehož směru p posuneme (tento vektor označme jako \vec{v}_{mt} - minimum translation vector). Abychom toho dosáhli, projdeme postupně každou hranu e objektu b_2 , zjistíme její normálu \vec{n} (jednotkový vektor směřující ven z objektu a kolmý na e) a spočítáme

$$d = \langle \vec{n} | x_p - x_a \rangle, \quad (2.22)$$

kde x_p je pozice p a x_a je pozice jednoho (libovolného) koncového bodu e . Pokud je $d < 0$, p leží uvnitř b_2 (protože statické objekty jsou vždy konvexní, viz sekce 2.8). Hodnota d_{mt} bude pak minimum ze všech $-d$ a \vec{v}_{mt} bude normálový vektor, který jsme spočítali při počítání d_{mt} . Když nyní posuneme p o $(-d_{mt} \cdot \vec{v}_{mt})$, nebude už ležet v b_2 . Je dobré si všimnout, že \vec{n} musíme normalizovat teprve tehdy, když víme, že p leží uvnitř b_2 . Znaménko d to totiž neovlivní.

Toto ale pro vyřešení kolize nestačí, protože se může stát, že některý z vrcholů b_2 bude ležet v b_1 . To vyřešíme tak, že pro každou hranu e objektu b_1 určenou hmotnými body p_1 a p_2 projdeme každý vrchol v_i objektu b_2 a zjistíme, zda v_i leží na vnitřní straně e (vnitřní strana hrany má s vnitřkem



Obrázek 2.4: Hrana e vede přes „roh“ v_i .

objektu neprázdný průnik). Pokud ano, musíme se ujistit, že e vede přes „roh“ v_i (viz obrázek 2.4). To poznáme tak, že e musí protínat úsečku mezi v_i a s , kde s je střed úsečky v_{i-1}, v_{i+1} . Jak zjistit, zda se dvě úsečky protínají, popisuje např. Ericson [6] na str. 151. Tím zabráníme řadě degenerovaných případů, kdy by se kolize vyřešila špatně nebo nedostatečně (viz obrázek 2.5). Pokud tedy v_i projde oběma testy, posuneme p_1, p_2 ve směru opačné normály $-\vec{n}$ hrany e . Velikost posunutí vypočteme jako

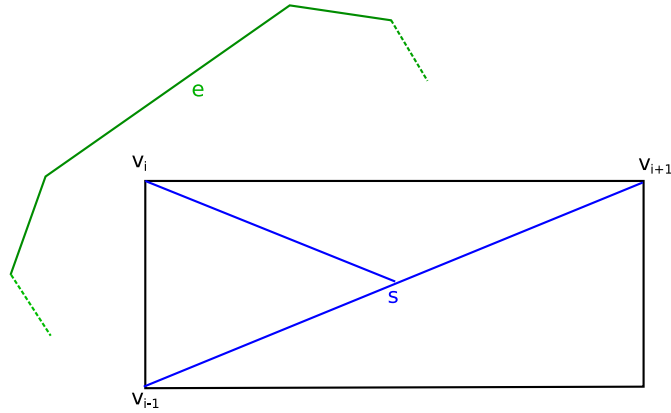
$$d_{mt} = - \langle \vec{n} | x_v - x_{p_1} \rangle, \quad (2.23)$$

kde x_v je pozice v a x_{p_1} je pozice p_1 . Opět je dobré si všimnout, že \vec{n} je nutné normalizovat až těsně před výpočtem d_{mt} , protože \vec{n} se do té doby nemění (zpracováváme stále tutéž hranu).

Správně bychom měli při posouvání p_1 a p_2 zohlednit jejich hmotnosti a pozici v promítnuté na e , ale výsledky jsou na pohled prakticky totožné, proto je lepší použít výpočetně méně náročnou verzi.

Dynamické vs. dynamické objekty

Protože se dynamické objekty mohou deformovat, nemůžeme předpokládat, že jsou vždy konvexní. Nelze tedy použít žádný z efektivnějších postupů detekce kolize pro konvexní útvary. Otestujeme proto proti sobě všechny hrany. Pro každý hmotný bod p objektu b_1 projdeme každou hranu e objektu b_2 určenou hmotnými body p_1 a p_2 . Zjistíme normálu \vec{n}_p bodu p a normálu \vec{n} hrany e . Nevyžadujeme, aby tyto dva vektory byly jednotkové, proto je nebudeme normalizovat. Bod p je uvnitř b_2 , jen pokud \vec{n}_p míří jiným směrem



Obrázek 2.5: Hrana e nevede přes „roh“ v_i , přestože v_i leží na vnitřní straně e .

než \vec{n} , čili pokud

$$\langle \vec{n}_p | \vec{n} \rangle < 0. \quad (2.24)$$

V opačném případě tuto hranu přeskočíme. Spočítáme

$$\vec{v} = x_p - x_{p_1} \quad (2.25)$$

$$s = \langle \vec{v} | x_{p_2} - x_{p_1} \rangle, \quad (2.26)$$

kde x_p je pozice p , x_{p_1} je pozice p_1 a x_{p_2} je pozice p_2 . Pokud $s < 0$, pak bod na e nejbližší k p je p_1 , $|\vec{v}|$ je kandidátem na d_{mt} a \vec{v} je kandidátem na \vec{v}_{mt} (všimněme si, že \vec{v}_{mt} zde má velikost d_{mt}). Pokud $s > |x_{p_2} - x_{p_1}|^2$, pak bod na e nejbližší k p je p_2 , ale tento případ ošetříme v další iteraci při zpracovávání sousední hrany. Poslední možnost je $0 < s < |x_{p_2} - x_{p_1}|^2$, pak bod na e nejbližší k p (nazvěme ho c) leží někde mezi p_1 a p_2 . Vzdálenost c a p spočítáme jako

$$d = \frac{\langle \vec{v} | \vec{n} \rangle}{|\vec{n}|}. \quad (2.27)$$

Hodnota d je kandidátem na d_{mt} a $\frac{\vec{n}}{|\vec{n}|}$ je kandidátem na \vec{v}_{mt} (všimněme si, že zde je \vec{v}_{mt} jednotkový vektor). Spočítáme ještě

$$h = \frac{s}{|x_{p_2} - x_{p_1}|^2} \quad (2.28)$$

pro pozdější použití v reakci na kolizi. Proměnná h určuje, jak blízko je bod dotyku objektů k bodům p_1 a p_2 , a nabývá hodnot z intervalu $[0, 1]$. Tento postup je shrnut v *Algoritmu 1*.

Nyní spočítáme vektor, podle kterého musíme posunout kolidující hmotné body, aby se kolize vyřešila. Vybereme d_{mt} jako minimum ze všech kandidátů. Za \vec{v}_{mt} zvolíme kandidáta příslušného k tomu, jehož jsme vybrali pro d_{mt} . Musíme rozlišit dvě situace. První z nich je, že jsme vybrali kandidáta v případě, kdy bylo $s < 0$. Nechť e stále označuje hranu, kterou jsme při výpočtu d_{mt} použili, a p_1, p_2 její koncové body. Posuneme p ve směru $(-\vec{v}_{mt})$ a p_1 ve směru \vec{v}_{mt} . Dohromady musíme oba body posunout o d_{mt} , ale musíme tuto vzdálenost rozdělit podle hmotností obou bodů. Navíc využijeme toho, že \vec{v}_{mt} má velikost přesně d_{mt} (při výpočtu jsme ho nenormalizovali). Nové pozice x'_p, x'_{p_1} bodů p, p_1 tedy spočítáme takto:

$$x'_p = x_p - \frac{m_{p_1}}{m_p + m_{p_1}} \vec{v}_{mt} \quad (2.29)$$

$$x'_{p_1} = x_{p_1} + \frac{m_p}{m_p + m_{p_1}} \vec{v}_{mt}. \quad (2.30)$$

Proměnná m_p určuje hmotnost p a m_{p_1} je hmotnost p_1 .

Druhá možnost je, že jsme vybrali kandidáta v případě, kdy nejbližší bod p a e byl c . Jako v předchozím případě posuneme p podle poměru hmotností bodů, avšak nyní musíme vzít v úvahu i hmotnost m_{p_2} bodu p_2 . Spočteme tedy novou pozici x'_p bodu p a posunutí \vec{u} bodů p_1 a p_2 :

$$x'_p = x_p - \frac{\frac{1}{2}(m_{p_1} + m_{p_2})}{m_p + \frac{1}{2}(m_{p_1} + m_{p_2})} d_{mt} \cdot \vec{v}_{mt} \quad (2.31)$$

$$\vec{u} = \frac{m_p}{m_p + \frac{1}{2}(m_{p_1} + m_{p_2})} d_{mt} \cdot \vec{v}_{mt}. \quad (2.32)$$

Nyní spočteme nové pozice x'_{p_1}, x'_{p_2} bodů p_1, p_2 . Rozdělíme \vec{u} podle hmotností obou bodů, ale vezmeme v potaz i koeficient h , který jsme spočetli dříve ve vztahu 2.28:

$$x'_{p_1} = x_{p_1} + \frac{m_{p_2}}{m_{p_1} + m_{p_2}} (1 - h) \cdot \vec{u} \quad (2.33)$$

$$x'_{p_2} = x_{p_2} + \frac{m_{p_1}}{m_{p_1} + m_{p_2}} h \cdot \vec{u}. \quad (2.34)$$

Na obrázku 2.6 je vidět, jak velikost koeficientu h měla vliv na rozdílnou velikost posunutí bodů p_1 a p_2 . Detekci kolize a reakci na ni je třeba zopakovat se zaměněnými objekty b_1 a b_2 . Celý postup je shrnut v *Algoritmu 2*.

V praxi se vyplatí vzít v úvahu i případ, kdy nerovnost 2.24 neplatí. Ten nastane, pokud normála testovaného bodu p má přibližně stejný směr jako

Input: Hmotný bod p prvního objektu b_1

Input: Hmotné body druhého objektu b_2

Output: Kandidáti na d_{mt} pro p

```
1 foreach Hrana  $e = (p_1, p_2)$  objektu  $b_2$  do
2    $\vec{n}_p = \text{Normála}(p)$ 
3    $\vec{n} = \text{Normála}(e)$ 
4   if  $\langle \vec{n}_p | \vec{n} \rangle \geq 0$  then
5     continue
6    $\vec{u} = \text{Pozice}(p_2) - \text{Pozice}(p_1)$ 
7    $\vec{v} = \text{Pozice}(p) - \text{Pozice}(p_1)$ 
8    $s = \langle \vec{u} | \vec{v} \rangle$ 
9   if  $s < 0$  then
10     $d = \text{Délka}(\vec{v})$ 
11     $h = 0$ 
12     $\vec{v}_d = \vec{v}$ 
13    Přidej  $d$  do kandidátů na  $d_{mt}$ 
14    Ulož k němu  $h$ ,  $\vec{v}_d$  a hranu  $e$ 
15  end
16  else if  $s < \text{Délka}(\vec{u})^2$  then
17    Normalizuj( $\vec{n}$ )
18     $d = \langle \vec{v} | \vec{n} \rangle$ 
19     $h = s / \text{Délka}(\vec{u})^2$ 
20     $\vec{v}_d = d \cdot \vec{n}$ 
21    Přidej  $d$  do kandidátů na  $d_{mt}$ 
22    Ulož k němu  $h$ ,  $\vec{v}_d$  a hranu  $e$ 
23  end
24  else
25    // Nedělej nic, vyřeší se v další iteraci
26  end
27 return Kandidáti na  $d_{mt}$ 
```

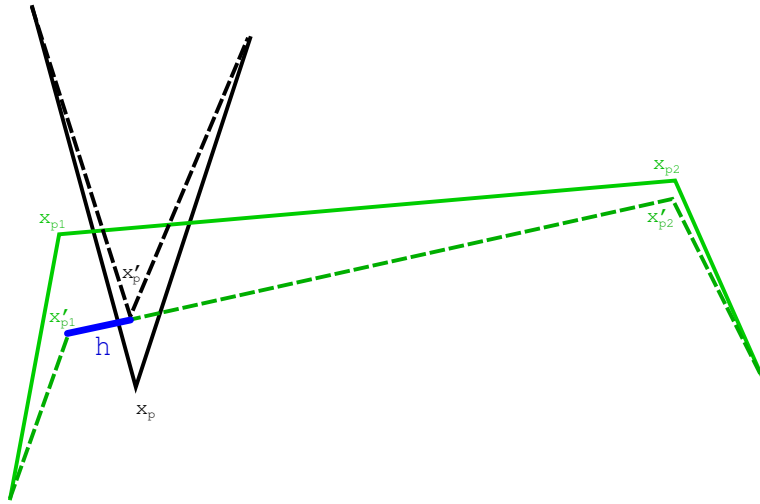
Algorithm 1: Pseudokód ilustrující nalezení kandidátů na d_{mt} pro hmotný bod p .

Input: Hmotné body objektů b_1 a b_2

Output: Upravené hmotné body objektů b_1 a b_2

```
1 foreach Hmotný bod  $p$  objektu  $b_1$  do
2   Najdi kandidáty na  $d_{mt}$  pro  $p$  vzhledem k  $b_2$ 
3    $d_{mt} =$  Vyber minimum z kandidátů na  $d_{mt}$ 
4    $h_{mt} = h$  uložené s vybraným kandidátem
5    $\vec{v}_{mt} = \vec{v}_d$  uložený s vybraným kandidátem
6    $e_{mt} = (p_1, p_2) = e$  uložená s vybraným kandidátem
7   if  $h_{mt} == 0$  then
8     //  $p_1$  je nejbližší bod k  $p$  na  $e_{mt}$ 
9      $\text{Pozice}(p) = \text{Pozice}(p) - \frac{Hmotnost(p_1)}{Hmotnost(p)+Hmotnost(p_1)}\vec{v}_{mt}$ 
10     $\text{Pozice}(p_1) = \text{Pozice}(p_1) + \frac{Hmotnost(p)}{Hmotnost(p)+Hmotnost(p_1)}\vec{v}_{mt}$ 
11  end
12  else
13    //  $c = p_1 + h(p_2 - p_1)$  je nejbližší bod k  $p$  na  $e_{mt}$ 
14     $m_e = \frac{Hmotnost(p_1)+Hmotnost(p_2)}{2}$ 
15     $\text{Pozice}(p) = \text{Pozice}(p) - \frac{m_e}{Hmotnost(p)+m_e}\vec{v}_{mt}$ 
16     $\vec{u}_{mt} = \frac{Hmotnost(p)}{Hmotnost(p)+m_e}\vec{v}_{mt}$ 
17     $\text{Pozice}(p_1) = \text{Pozice}(p_1) + \frac{Hmotnost(p_2)\cdot(1-h)}{Hmotnost(p_1)+Hmotnost(p_2)}\vec{u}_{mt}$ 
18     $\text{Pozice}(p_2) = \text{Pozice}(p_2) + \frac{Hmotnost(p_1)\cdot h}{Hmotnost(p_1)+Hmotnost(p_2)}\vec{u}_{mt}$ 
19  end
20 end
21 Zopakuj 1-43 s prohozenými  $b_1$  a  $b_2$ 
22 return Hmotné body  $b_1$  a  $b_2$ 
```

Algorithm 2: Pseudokód ilustrující detekci kolize dvou dynamických objektů a reakci na ni.



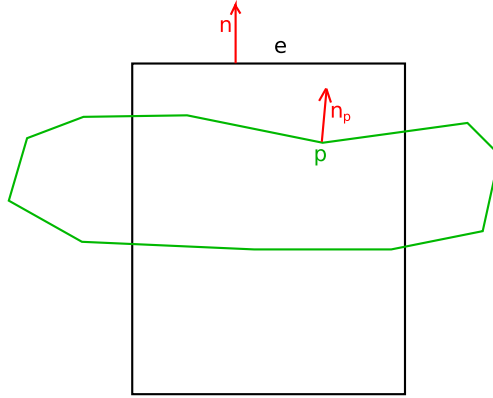
Obrázek 2.6: Vliv koeficientu h a hmotnosti bodů na velikost jejich posunutí.

normála hrany e . Odpovídá to situaci, ve které b_1 leží uvnitř b_2 nebo naopak. Na první pohled se zdá, že tento případ je nežádoucí; pokud však bude podlouhlý objekt procházet vnitřkem většího objektu (viz situace na obrázku 2.7), je patrné, že pro korektní vyřešení této kolize musíme posunout p směrem nahoru, tj. k e , přestože jejich normály míří přibližně stejným směrem. Pokud bychom to neudělali, zelený objekt by se obrátil „naruby“.

Je dobré si všimnout, že někdy děláme zbytečnou práci navíc. Bod p totiž vůbec nemusí ležet v b_2 a dokonce ani v jeho AABB. Pokud tyto dva testy provedeme hned jak je to možné, dosáhneme znatelného zrychlení. Dále můžeme otestovat, jestli p leží v AABB hrany e , což také vede ke zrychlení.

Bod v mnohoúhelníku

Abychom zjistili, zda bod leží v objektu, potřebujeme algoritmus, který detekuje přítomnost bodu v mnohoúhelníku (pohybujeme se ve dvou rozměrech a každý objekt je reprezentovaný mnohoúhelníkem). Bourke [1] popisuje algoritmus, který je založen na tom, že bod p leží uvnitř mnohoúhelníku, právě když vodorovná polopřímka \vec{r}_p s počátkem v tomto bodě protíná lichý počet hran mnohoúhelníku. Orientace polopřímky nehraje roli, proto můžeme předpokládat, že míří vpravo (podél rostoucí osy x). Projdeme tedy každou hranu e určenou body p_1, p_2 a podíváme se, zda oba leží buď pod \vec{r}_p nebo oba leží nad \vec{r}_p . Pokud ano, pak \vec{r}_p jistě hranu e neprotíná a můžeme



Obrázek 2.7: Bod p musíme posunout nahoru, přestože má normálu v přibližně stejném směru, jako hrana e .

ji přeskočit. V opačném případě spočítáme

$$s = \frac{x_{p_2} - x_{p_1}}{y_{p_2} - y_{p_1}} \quad (2.35)$$

$$h = x_{p_1} + s \cdot (y_p - y_{p_1}). \quad (2.36)$$

Hodnota s je převrácená hodnota směrnice e a h je pak x-ová souřadnice průsečíku e a přímky, která obsahuje \vec{r}_p . Pokud tento průsečík leží vpravo od p , tj. pokud

$$h \geq x_p, \quad (2.37)$$

pak leží i v \vec{r}_p , a hrana e tedy protíná \vec{r}_p . Počet protnutých hran budeme průběžně počítat a pokud nám vyjde po skončení algoritmu liché číslo, leží bod p v mnohoúhelníku. Pokud spouštíme simulaci na platformě, na které je dělení pomalé, můžeme rovnice 2.35 a 2.36 přepsat na

$$c_1 = (x_{p_2} - x_{p_1}) \cdot (y_p - y_{p_1}) \quad (2.38)$$

$$c_2 = (y_{p_2} - y_{p_1}) \cdot (x_p - x_{p_1}). \quad (2.39)$$

Pak ale musíme změnit i podmínku 2.37 na

$$(y_{p_2} \geq y_{p_1} \wedge c_1 \geq c_2) \vee (y_{p_2} \leq y_{p_1} \wedge c_1 \leq c_2) \quad (2.40)$$

(zde musíme odlišit, kdy byl jmenovatel zlomku v rovnici 2.35 kladný a kdy záporný, protože jsme jím vynásobili nerovnost 2.37). Celý postup je shrnut v *Algoritmu 3*.

Input: Bod p , body mnohoúhelníku b_2

Output: True, pokud p leží uvnitř b_2

```
1 uvnitr = FALSE
2 foreach Hranu  $e = (p_1, p_2)$  mnohoúhelníku  $b_2$  do
3   if  $p_1.y \leq p.y$  and  $p_2.y \leq p.y$  then
4     continue
5   if  $p_1.y > p.y$  and  $p_2.y > p.y$  then
6     continue
7    $c_1 = (p_2.x - p_1.x) \cdot (p.y - p_1.y)$ 
8    $c_2 = (p_2.y - p_1.y) \cdot (p.x - p_1.x)$ 
9   if  $(p_2.y \geq p_1.y$  and  $c_1 \geq c_2)$  or  $(p_2.y < p_1.y$  and  $c_1 \leq c_2)$  then
10      $uvnitr = !uvnitr$ 
11   end
12 end
13 return uvnitr
```

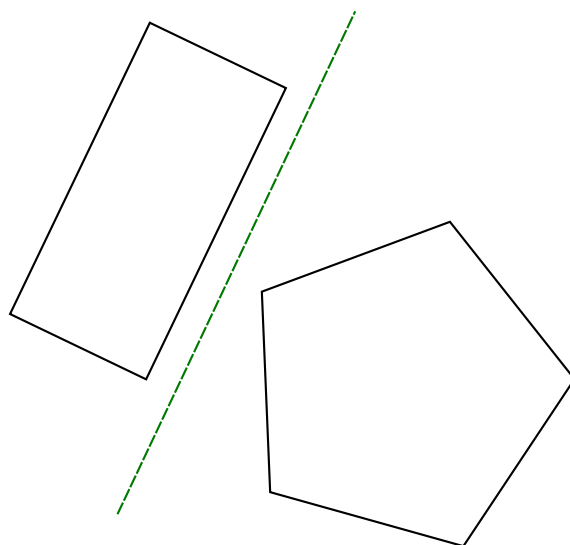
Algorithm 3: Pseudokód ilustrující detekci bodu uvnitř mnohoúhelníku.

Separating Axis Theorem

U některých objektů můžeme předpokládat, že budou po celou dobu simulace konvexní, i když jsou dynamické. Jsou to většinou objekty s menším počtem hmotných bodů a vysokým tlakem nebo velkým počtem pružin, takže nepodlehnu snadno deformaci. U těchto objektů se vyplatí použít efektivnější metodu detekce kolize. Takových metod je víc, ale já jsem vybral algoritmus využívající Separating Axis Theorem (SAT) [11], protože je ve dvou rozměrech dostačující a přitom jednoduchý na implementaci.

Separating Axis Theorem vychází z Minkovského teorií a říká, že ve dvou dimenzích se dva uzavřené konvexní útvary, z nichž alespoň jeden je konečný, nepřekrývají, právě když existuje oddělovací osa (separating axis). Oddělovací osa je taková osa, že projekce obou útvarů na tuto osu se nepřekrývají. Jinými slovy, dva konvexní útvary se nepřekrývají, právě když existuje přímka taková, že každý z útvarů leží v jiné polorovině určené touto přímkou (viz obrázek 2.8). Ke každé takové přímce navíc existuje právě jedna oddělovací osa, která je na ni kolmá. Pokud pracujeme s konvexními mnohoúhelníky, stačí zkoumat jen osy kolmé na hrany těchto mnohoúhelníků, resp. přímky rovnoběžné s hranami těchto mnohoúhelníků.

Proč tomu tak je? Ukážeme, že pokud se konvexní mnohoúhelníky b_1 ,



Obrázek 2.8: Přímka oddělující dva nepřekrývající se konvexní útvary.

b_2 nepřekrývají, tak existuje přímka rovnoběžná s některou z hran b_1 nebo b_2 , která oba mnohoúhelníky odděluje. Mějme množinu P dvojic bodů (x, y) takových, že $x \in b_1$ a $y \in b_2$, a zároveň platí, že vzdálenost $|x, y|$ je minimální možná. Z těchto dvojic si vybereme takovou (pokud existuje), že oba body jsou vrcholy b_1 a b_2 , nazvěme je a a b . Přímka l' procházející středem ab kolmá na ab odděluje b_1 a b_2 , protože díky konvexitě není žádný z bodů b_1 nebo b_2 k l blíže než a a b . Označme si jako l_i přímky procházející středem ab , které jsou rovnoběžné s některou z hran, ve kterých je a nebo b krajním bodem. Nyní z nich vyberme přímku l tak, že svírá s l' minimální úhel. Pak ale l musí b_1 a b_2 také oddělovat, jinak by došlo ke sporu s minimálností úhlu přímek l a l' .

Pokud takto specifikovaná dvojice (a, b) neexistuje, vybereme si dvojici takovou, že právě jeden z bodů je vrchol. Bez újmy na obecnosti předpokládejme, že je to a . Pak musí b ležet uvnitř nějaké hrany e . Přímka l rovnoběžná s e procházející středem ab odděluje b_1 a b_2 . Všechny body b_2 jsou totiž díky konvexitě od l dále než od e a zároveň všechny body b_1 jsou od l dále než od vrcholu a (jinak by existovala dvojice bodů z P taková, že její první složka je vrchol/hrana z p_1 , která je k e blíže než a , ale (a, b) je dvojice bodů s minimální vzdáleností, což je spor). Pokud by takto specifikovaná dvojice (a, b) neexistovala, musela by P obsahovat pouze dvojice bodů, které jsou uvnitř hran, jenže ty jsou konečné. Proto tento případ nemůže nastat.

Pokud objekty spolu kolidují, pak se projekce na všechny osy musejí pře-

krývat. Důležité pozorování je, že osa s nejmenším překryvem je \vec{v}_{mt} a velikost tohoto překryvu je d_{mt} (\vec{v}_{mt} a d_{mt} se zde vztahuje k celým objektům, ne jen k bodům/hranám, jako tomu bylo v předchozích sekcích). Tuto informaci potřebujeme dále k vyřešení kolize.

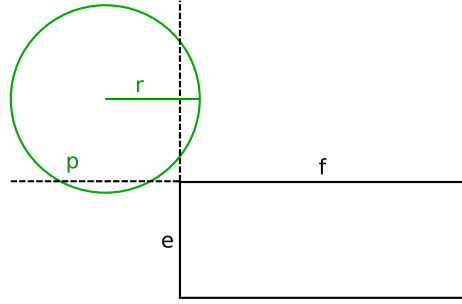
Postup algoritmu je nasnadě. Postupně projdeme všechny hrany obou objektů, zjistíme přímkou na ně kolmé, provedeme projekci obou objektů na tyto přímkou a pokud se nějaká z těchto projekcí nepřekrývá, okamžitě skončíme (našli jsme oddělovací osu). V opačném případě zjistíme, která z přímkou má nejmenší překryv projekcí a její směr prohlásíme za \vec{v}_{mt} .

Nyní je ještě potřeba získat body dotyku objektů. K tomu nám pomůže \vec{v}_{mt} . Předpokládejme, že \vec{v}_{mt} je orientován směrem od p_1 k p_2 . Projdeme všechny body p_1 a najdeme všechny body, které jsou ve směru \vec{v}_{mt} nejdál, tj. jejich skalární součin s \vec{v}_{mt} je největší. Podobně projdeme body p_2 a najdeme všechny body, které jsou ve směru \vec{v}_{mt} nejbliž, tj. jejich skalární součin s \vec{v}_{mt} je nejmenší. Pro jednoduchost předpokládejme, že tímto postupem můžeme mít z každého objektu jen dva body. Mohou tedy nastat jen 3 případy: z obou objektů máme jen jeden bod (tzn. objekty se dotýkají ve vrcholech), z jednoho objektu máme jeden bod a z druhého dva body (objekty se dotýkají vrcholem a hranou) nebo z obou objektů máme dva body (objekty se dotýkají hranami). Pouze třetí případ je třeba dále řešit, protože námi nalezené body neodpovídají přesně kontaktům, ale vrcholům určujícím hrany, které se dotýkají. Body získané z p_2 projektujeme na hranu určenou body z p_1 a podle minima a maxima této projekce hranu „uřízneme“, tj. získáme koeficient h určující pozici minima/maxima mezi p_1 a p_2 .

Kolizi pak vyřešíme stejným způsobem, jako jsme ji řešili v případě nekonvexních objektů, jen zde máme přímo hrany/vrcholy, o které je třeba se postarat. Koeficient h v případě kolize dvou hran je ten samý koeficient jako v rovnici 2.28. Více informací a zajímavé animace k SATu je možné nalézt na stránkách Metanetu [19].

Částice vs. statické objekty

Protože je částice určená jedním hmotným bodem a poloměrem, budou detekce a následná reakce na kolizi velmi podobné jako v sekci *Dynamické vs. statické objekty*. Nyní však musíme vzít v úvahu i poloměr částice. Navíc se již nemůžeme spolehnout na to, že částice koliduje s objektem, právě když přesahuje přes všechny jeho hrany směrem dovnitř něj, viz obrázek 2.9. Musíme tedy testovat částici (označme ji p , její střed c a poloměr r) i vůči vrcholům objektu (označme ho b). Nejprve zjistíme, zda je střed c



Obrázek 2.9: Částice p přesahuje přes všechny hrany černého objektu, ale přesto s ním nekoliduje.

částice p blíž k hraně e určené vrcholy v_1 a v_2 nebo k jednomu z vrcholů. Spočteme

$$s = \langle c - v_1 | v_2 - v_1 \rangle. \quad (2.41)$$

Pokud je $s < 0$, pak je c nejbliž k v_1 . V tom případě je $-(c - v_1)$ kandidátem na \vec{v}_{mt} a $(r - |c - v_1|)$ je kandidátem na d_{mt} . Pokud je $0 \leq s < |v_2 - v_1|^2$, pak je c nejbliž k e . Najdeme normálu \vec{n} hrany e tak, aby směřovala ven z b . Vektor \vec{n} je kandidátem na \vec{v}_{mt} a $(\langle \vec{n} | c - v_1 \rangle + r)$ je kandidátem na d_{mt} . Pokud je $s \geq |v_2 - v_1|^2$, pak je c nejbliž k v_2 , ale tento případ ošetříme v další iteraci, protože každý vrchol je právě ve dvou hranách. Ze všech kandidátů vybereme \vec{v}_{mt} a d_{mt} a vyřešíme kolizi stejně jako v sekci *Dynamické vs. statické objekty*, tj. posuneme p podle \vec{v}_{mt} a d_{mt} .

Částice vs. částice

Detekce kolize částic je přímočará. Stačí spočítat

$$\vec{e} = c_2 - c_1, \quad (2.42)$$

kde c_1 a c_2 jsou středy částic. Pokud je $(r_1 + r_2 - |\vec{e}|) \geq 0$, pak částice kolidují, $(r_1 + r_2 - |\vec{e}|)$ je d_{mt} a $\frac{\vec{e}}{|\vec{e}|}$ je \vec{v}_{mt} . Pro vyřešení kolize stačí posunout částice podél \vec{v}_{mt} o d_{mt} rozdělené podle jejich hmotností (čili d_{mt} pro první částici vynásobíme koeficientem $\frac{m_2}{m_1 + m_2}$, kde m_1 a m_2 jsou hmotnosti částic, obdobně pro druhou částici).

Částice vs. dynamické objekty

Kolize s dynamickými objekty probíhá podobně jako kolize se statickými objekty v sekci *Částice vs. statické objekty* s tím rozdílem, že nyní musíme

posunout i hranu e , resp. vrchol v_1 . Hodnotu d_{mt} rozdělíme mezi částici a v_1 a v_2 (pokud kolidujeme s hranou e), resp. pouze v_1 (pokud kolidujeme s vrcholem v_1) podle jejich hmotností. Hranu e posuneme změnou pozice vrcholů v_1 , v_2 , mezi které rozdělíme velikost posunutí podle koeficientu h známého z rovnice 2.28, kde za s použijeme ten z rovnice 2.41:

$$h = \frac{s}{|v_2 - v_1|^2}. \quad (2.43)$$

2.11.3 Tření

Simulace bez tření nepůsobí vizuálně dobře. Objekty na sobě nedrží, kloužou, jako by byly z ledu. Proto je potřeba tření nějakým způsobem do simulace zařadit. Tření se většinou simuluje využitím Coulombova zákona (viz např. Gomez [10]) a zvlášť se nakládá s dynamickým a zvlášť se statickým třením. Tento způsob je ale výpočetně náročný, proto jsem hledal řešení, které tak nezatíží procesor.

Máme-li hmotný bod p_1 , který kolidoval s hmotným bodem p_2 , a normálu kolize \vec{n} . Abychom nasimulovali tření, potřebujeme vynulovat relativní rychlost bodů p_1 a p_2 podél tangenty kolize \vec{t} . Tangenta kolize je jednotkový vektor kolmý na \vec{n} . Protože \vec{n} je typicky už normalizovaný vektor, nemusíme tangentu znovu normalizovat. Spočítáme

$$v_r = \langle \vec{v}_2 - \vec{v}_1 | \vec{t} \rangle, \quad (2.44)$$

kde \vec{v}_1 , \vec{v}_2 jsou rychlosti bodů p_1 , p_2 . Protože rychlosti nemáme explicitně vyjádřené (viz sekce 2.5), spočítáme je jako rozdíl aktuální pozice a předchozí pozice. Chceme, aby po změně rychlostí platilo

$$v_r = 0. \quad (2.45)$$

Toto ovšem odpovídá maximálnímu tření, které někdy není žádoucí, proto zavedeme *koeficient tření* f z intervalu $[0, 1]$ (1 odpovídá maximálnímu tření, 0 žádnému tření). Rychlost bodů tedy musíme změnit celkově o $v_r f \vec{t}$. Zároveň vezmeme v úvahu i hmotnosti m_1 , m_2 bodů p_1 , p_2 . Změny rychlostí $\Delta\vec{v}_1$, $\Delta\vec{v}_2$ spočteme takto:

$$\Delta\vec{v}_1 = v_r f \frac{m_2}{m_1 + m_2} \vec{t} \quad (2.46)$$

$$\Delta\vec{v}_2 = -v_r f \frac{m_1}{m_1 + m_2} \vec{t}. \quad (2.47)$$

Rychlost máme vyjádřenou implicitně pomocí předchozích pozic \bar{x}_1, \bar{x}_2 bodů p_1, p_2 . Pokud chceme rychlost změnit, musíme změnit \bar{x}_1 a \bar{x}_2 o $\Delta\vec{v}_1$ a $\Delta\vec{v}_2$.

Tento způsob je výpočetně velmi nenáročný, ale přesto dává na pohled hezké výsledky a pro mobilní hry je naprosto dostačující.

2.11.4 Lepení

Při implementaci příkladu zmíněného dále v kapitole 4.1 jsem potřeboval, aby mi simulace umožnila některé hmotné body „přilepit“ k jiným hmotným bodům nebo statickým objektům. Tření na lepení nestačí - například by nemohlo udržet objekt přilepený ke stropu jeskyně, aby nespadol dolů. Vhodným řešením je použít pružiny, které již simulovat umíme. Hmotný bod se může na jiný objekt přilepit, jen pokud s ním kolidoval, proto je třeba systém lepení zakomponovat do systému, který řeší kolize. Pokud tedy daný hmotný bod kolidoval s jiným hmotným bodem, přidáme do simulace pružinu spojující tyto dva body. Pokud ovšem hmotný bod kolidoval s hranou, nevíme, ke kterému hmotnému bodu pružinu připojit. Vyřešil jsem to tak, že jsem umožnil pružině spojovat jeden hmotný bod a hranu určenou jinými dvěma hmotnými body s tím, že si pružina uchovála i koeficient h známý z detekce kolize (viz rovnice 2.28). Při aplikaci pružiny se pak pomocí h spočítal skutečný bod, na který je pružina připojena.

Další problém byl, jak připojit pružinu ke statickému světu (resp. statickým objektům), protože ten není složen z hmotných bodů, nýbrž má pouze definován tvar. Umožnil jsem tedy pružině připojit se rovněž ke konkrétní pozici ve světě.

Posledním problémem bylo, co se statickými objekty, se kterými může někdo (manuálně) pohybovat. Vyřešil jsem to tak, že si každý statický objekt uchovával seznam pozic (resp. ukazatele na objekty, jež je reprezentují), na které je přilepen nějaký objekt. Při pohybu se statickým objektem pak objekt změnil i tyto pozice.

Aby se mohl objekt opět odlepit, umožnil jsem pružinám, aby se přetrhly, pokud je jejich aktuální délka větší než jistý (pro každou pružinu předem definovaný) násobek jejich původní délky. Pokud k tomu došlo, pružina se odebrala ze simulace. Zároveň jsem ale potřeboval, aby i objekt sám měl možnost přetrhnout všechny pružiny, a tak se odlepit. Proto jsem tyto „lepící“ pružiny umístil k objektu, který přilepovaly, a při odlepení jsem je jednoduše všechny odebral.

2.12 Stabilita

Díky využití Verletova integrátoru je simulace celkově velmi stabilní. Problém ale může nastat u detekce kolize, protože ta nebere v úvahu rychle se pohybující objekty. Díky tomu se může stát, že objekt během jednoho simulačního kroku „proletí“ přes jiné objekty (tento jev se označuje jako *tunelování*). Nebo s ním sice bude kolidovat, ale velikost překryvu obou objektů bude tak velká, že kolize nebude uspokojivě vyřešena.

Abychom tomuto problému předešli, musíme stanovit dostatečně malou délku simulačního kroku. Připomenu, že tato délka je v průběhu celé simulace konstantní, viz sekce 2.5. V praxi se ukazuje, že délka kroku 35 milisekund je dostatečně malá na to, aby nedocházelo k tunelování, a zároveň dost velká na to, aby mobilní telefony zvládly simulaci spočítat v reálném čase. Ani malý simulační krok ale někdy nestačí. Dobře fungujícím řešením je limitovat maximální rychlost všech hmotných bodů v simulaci, protože ty ovlivní celkovou rychlost objektů. V praxi se mi osvědčilo nastavit maximální rychlost na 8 pixelů za simulační krok. Toto omezení nevádí, protože objekty pohybující se rychleji stejně z malých displejů mobilních telefonů rychle zmizí. Protože manuální změna rychlosti hmotných bodů vyžaduje normalizaci vektoru, je dobré toto provádět jen tehdy, pokud je to opravdu třeba. V praxi je dostačující limitovat rychlost jen objektům, které v tomto simulačním kroku kolidují s jinými objekty.

Kapitola 3

Implementace

Nyní tedy máme hotovou fyzikální simulaci umožňující modelovat všechny elementy potřebné pro většinu her (statické objekty, dynamické objekty, tily, pružiny, pohybující se platformy, dveře atd.), abychom je však mohli v praxi použít, potřebujeme, aby byla simulace dostatečně rychlá i na starších modelech telefonů a zároveň nebyla paměťově náročná.

V této kapitole proto uvedu několik způsobů, jak rozpoznat místa zopmalující běh aplikace, a poskytnu rady k jejich optimalizaci.

3.1 Optimalizace paměťových nároků

Jak jsem zmínil v sekci *Implicit Uniform Grid*, je vhodné mít všechny objekty uloženy v poli. Ke každému objektu nutně potřebujeme informace vztahující se k fyzikální simulaci, takže zde prostor pro optimalizaci není a ani to není třeba, protože objektů typicky nebude příliš mnoho (řádově desítky). Použitím *Implicit gridu* je na broadphase potřeba pouze paměť lineární v šířce/výšce světa měřeno v počtu tilů, takže zde také není co optimalizovat. Problém může být jedině v reprezentaci samotných tilů, které zaberou kvadratickou paměť. V mobilních hrách nemá smysl vytvářet mapy větší než 256 x 256 tilů, což zabere 64KB paměti. Často ale nebude různých typů kolizních tilů více než 16, takže do jednoho bytu můžeme zakódovat dva tily, a zmenšit tak nároky na paměť na 32KB, což je už přibližně stejně jako paměť pro jeden větší obrázek. S ohledem na to, že dnešní telefony mají minimálně 512KB paměti, je simulace velmi nenáročná.

3.2 Optimalizace rychlosti

Daleko větší problém je implementovat simulaci takovým způsobem, aby fungovala dostatečně rychle i na starších telefonech. Jedinou masově rozšířenou technologií použitelnou pro vývoj her je J2ME, takže je nutno implementaci provést v Javě. Java Virtual Machine (JVM) je bohužel často velmi pomalá a ani Java sama není navržena primárně tak, aby poskytovala co nejvyšší výkon. Musíme se tomu tedy nějakým způsobem přizpůsobit.

Při popisu jednotlivých částí simulátoru jsem zmiňoval různé lokální možnosti optimalizace, jako např. použití Implicit gridu pro broadphase, zabránění zbytečných normalizací vektorů při detekci kolize dynamických a statických objektů a množství jiných. Jimi se tu již nebudu znovu zabývat, proto je v případě potřeby vyhledejte v předchozích kapitolách.

V následujících sekcích se budu věnovat nejprve optimalizaci fyzikální simulace na vyšší úrovni a pak postupně uvedu různé způsoby, jak zvýšit výkon J2ME aplikací na nižší úrovni.

3.2.1 Viditelnost

Objekty, které jsou mimo záběr kamery, nemá smysl simulovat, protože je hráč stejně neuvidí. Tato jednoduchá myšlenka dokáže rapidně zvýšit výkon celé simulace, ale zároveň způsobí mnoho problémů. Předně, jak efektivně zjistit, které objekty jsou vidět a které ne? Buď můžeme postupně projít všechny objekty a zjistit, jestli jejich AABB překrývá AABB kamery a pokud ano, pak objekt aktualizovat. Pokud ale budeme mít objektů velké množství, bude toto zjišťování časově náročné a navíc tento průchod budeme muset provést několikrát během jednoho simulačního kroku. Jednou při integraci pohybu, podruhé při aplikaci pružin a několikrát při detekci kolize. Naštěstí ale můžeme využít Implicit grid a dotázat se ho, jaké objekty jsou v oblasti určené AABB kamery. Tímto způsobem velmi rychle získáme seznam viditelných objektů.

Další problém je, co se bude dít na rozhraní viditelné a neviditelné části. Připomeňme, že na kolizi s objektem se testují pouze objekty s nižším ID, aby se zamezilo duplikaci detekovaných kolizí. ID je objektům přiděleno Implicit gridem. Problém je v tom, že zde může být několik objektů na sobě, z nichž jeden už viditelný bude, ale ostatní ještě ne. Viditelný objekt se posune dolů a začne kolidovat s objekty, které viditelné nejsou. Kolize přitom nebude rozpoznána - může se totiž stát, že viditelný objekt má vyšší ID než neviditelný - a viditelný objekt se postupně „propadne“ do neviditelného.

Když se kamera posune tak, že budou všechny objekty vidět, tak nebude jasné, jak kolizi vyřešit, protože velikost překryvu objektů může být příliš velká. Naštěstí lze tento problém v praxi ignorovat, protože se kolize většinou vyřeší uspokojivě a pokud je potřeba, aby se zachovala správně vždy, stačí jen přeházet ID objektů nacházejících se blízko u sebe.

Mnohem větší problém je ovšem s objekty spojenými pružinami. Představme si provazový most. K mostu se přiblížíme zleva, první objekt bude viditelný, začne se tedy simulovat a pohybovat směrem dolů. V souvislosti s tím přes pružinu potáhne za sebou i objekt napravo od něho, který v tu chvíli nebude vidět, až se tyto objekty dostanou natolik dolů, že se most vizuálně „rozbije“. Navíc, jakmile se ostatní dosud neviditelné objekty dostanou do záběru kamery, zaktivují se pružiny na ně připojené, které mohou být v tomto okamžiku nataženy například na desetinásobek původní délky, a doslova vystřelí okolní objekty daleko mimo hranice simulačního světa. Tento problém jsem vyřešil tak, že jsem každému objektu definoval „sousedy“, čili objekty, které jsou s ním spojeny pružinami. Pokud simulace označí některý objekt jako viditelný, označí zároveň jako viditelné i jeho sousedy a rekurentně opět jejich sousedy. Tímto způsobem jsem zajistil, že se v simulačním kroku aktualizoval celý most.

Posledním problémem je, jak zjistit viditelné pružiny, protože ty v Implicit gridu uloženy nejsou. Stačí je ovšem umístit do datové struktury jednoho z objektů, na který je daná pružina připojena.

3.2.2 Způsob psaní kódu v Javě

Programy se v Javě tvoří způsobem, který významně souvisí s její objektivostí. Aplikace se logicky rozvrhne do objektů, které se pak implementují pomocí javovských tříd. Plně se dodržují tři základní principy objektově orientovaného programování: zapouzdřenost, abstrakce a polymorfismus (více informací o Javě a objektově orientovaném programování lze najít např. v [4]). S tím pak souvisí i řada dalších praktik, které jsou každému programátorovi v Javě známé.

Bohužel, při tvorbě her (nebo jiných aplikací náročných na výkon) musíme tento styl programování opustit, protože může rapidně snížit výkon a zvýšit paměťovou náročnost i celkovou velikost aplikace. Právě ta byla v minulosti velkým problémem, protože některé telefony jsou schopny spustit pouze aplikace, jejichž celková velikost není větší než 64KB (včetně obrázků, zvuků a všech dalších potřebných zdrojů). Doba těchto telefonů je sice pryč, ale stále zde existují určité limity, i když volnější. Hlavním problémem je,

že i prázdná třída v Javě zabere po zkompilování do bytekódu přibližně půl KB. Proto se typicky snažíme maximálně snížit celkový počet tříd. V minulosti obsahovaly i hry od největších a nejúspěšnějších společností často jen jednu třídu, ve které byla implementovaná celá hra. Dnes už není takovéto extrémní řešení nezbytné, ale stále se vyplatí nevytvářet novou třídu, pokud to není opravdu nutné.

3.2.3 Nalezení úzkých hrdel

Základní pravidlo optimalizace zní: nepředpokládej. Znamená to, že nemá smysl optimalizovat nějakou část programu, pokud jen *předpokládáme*, že právě v ní je program pomalý. Zároveň se nevyplatí *předpokládat*, že námi provedená optimalizace program opravdu zrychlila. Je proto třeba mít prostředky umožňující změřit výkon jednotlivých částí programu.

Pojmem *úzké hrdlo* se označuje místo v programu, které je zdatelně pomalejší oproti jiným částem kódu, a tudíž celý program brzdí. Problém je ovšem, jak taková místa najít. Potřebovali bychom změřit, jak dlouho trvá provedení jednotlivých metod, protože podle toho úzké hrdlo poznáme. V J2ME je nejpřesnějším měřítkem času funkce *currentTimeMillis()* ve třídě *System*, která však měří čas jen v milisekundách, což není dostatečně přesné. Naštěstí je součástí WirelessToolkitu [26] (WTK) program Profiler, který toto měření provádí na úrovni JVM a dobu provedení měří v počtech cyklů procesoru; tyto výsledky jsou již použitelné. Zároveň po skončení běhu vytvoří přehlednou statistiku, z níž lze poznat, která metoda spotřebovala nejvíce času, a je tedy třeba se jí věnovat.

Výsledky Profileru závisí výrazně na implementaci JVM emulátoru, která se může velmi lišit od těch, jež jsou použity v různých mobilních telefonech; je to však jediný prostředek, který máme k dispozici. Praxe ukazuje, že skoro ve všech případech, kdy došlo k odstranění úzkého hrdla nalezeného Profilerem, došlo i ke zrychlení na skutečných telefonech.

3.2.4 Paměť

Obecně bychom se měli pokud možno vyhnout průběžnému zaplňování paměti, protože to může mít neblahý vliv i na výkon. Správu paměti má v Javě na starosti *Garbage Collector*, což je část JVM, která zajišťuje mazání nepoužívaných objektů. Když průběžně dochází k rychlému zaplňování paměti, je *Garbage Collector* aktivován velmi často a v ten moment se může aplikace i na několik milisekund zastavit.

K měření zátěže paměti poslouží opět WTK a jeho program Memory Monitor. Tento program průběžně sleduje naplnění paměti a výsledky zobrazuje v grafu. Je zde také vidět, kdy je aktivován Garbage Collector, případně je možné ho spustit ručně. Po skončení programu vytvoří přehlednou statistiku, z níž je patrné, do jaké míry se jednotlivé objekty instancovaly a z jakých metod k tomu docházelo. Pomocí tohoto programu tedy můžeme opět identifikovat místa, kterým bychom se měli věnovat.

Co ovšem Memory Monitor neměří, je velikost obrázků v paměti. Není to ostatně ani dobře možné, protože různé mobilní telefony ukládají obrázky v paměti v různých formátech. To nás sice nemusí při implementaci fyzikální simulace zajímat, ale každá mobilní hra obrázky používá. Tady pomůže program Pstros [24]. Pstros je emulátor, který ovšem umí také zobrazit všechny obrázky aktuálně načtené v paměti a zároveň spočítá celkový počet jejich pixelů, což už může sloužit jako měřítko toho, kolik místa v paměti obrázky zabírají.

3.2.5 Optimalizační programy

Existují programy, které jsou schopné program vytvořený v Javě automaticky zoptimalizovat. Jedním z nich je Proguard [17], který přeložené třídy Javy *zaobfuskuje*, tzn. nahradí všechny identifikátory tříd a proměnných krátkým (často jednopísmenným) názvem. Tím se celková velikost programu zmenší a zároveň se zabrání dekompilování a případnému zneužití. Kromě toho umí Proguard kód zoptimalizovat tak, že je ve výsledku mírně rychlejší. Proto se bezpochyby vyplatí alespoň ho zkusit použít.

Dalším zajímavým programem je JavaGO [15], který umí inlinovat (viz sekce 3.2.8) některé metody, takže tuto operaci není třeba dělat ručně.

3.2.6 Instance tříd

Jedním z největších problémů Javy je instanciování velkého počtu malých objektů. JVM na stolních počítačích si s tím už dokáže poradit, ale na mobilních telefonech je to časově nejnáročnější operace; dokáže rapidně snížit výkon aplikace a je třeba se jí vyhnout. Při implementaci simulace nastane tento problém hlavně při použití třídy pro vektor uchovávající informace o pozici, rychlosti, zrychlení, síle, atd. Při počítání s vektory totiž každá operace musí vytvořit novou instanci této třídy. Podle mých zkušeností vznikla většina úzkých hrdel hlavně z tohoto důvodu. Řešení je jednoduché; v kritických sekcích nepoužívat vektor, ale místo něj zavést dvě proměnné

standardního typu (*int* nebo *float*) a všechny operace ručně inlinovat (viz sekce 3.2.8). Velikost kódu se tím příliš nezvýší, protože většina operací s vektory se vejde do jednoho příkazu. Zároveň se sníží nároky na paměť, protože jedna instance třídy nesoucí dva *inty* zabírá cca 20 bytů, kdežto dvě proměnné typu *int* zabírají jen 8 bytů. Na druhou stranu ovšem není dobré simulaci implementovat přímo bez použití třídy pro vektory, protože to dokáže značně znepráhlednit kód, a způsobit tak chyby. Inlinovat se tedy vyplatí až v době, kdy celá simulace funguje korektně a jde nám hlavně o zvýšení rychlosti.

3.2.7 Metody

Každé volání metody s sebou nese nějakou režii, která je sice velmi malá, ale na mobilních telefonech přesto znatelná. Proto je dobré vyhnout se tzv. getterům a setterům. Jsou to metody, které přistupují k datovým položkám třídy, a zajišťují tak zapouzdřenost. Mnohem rychlejší je však přistupovat k datům přímo, což zároveň zmenší i výslednou velikost zkompilevané třídy, jelikož se tak zbavíme nemalého množství kódu.

Často se také doporučuje rozdělit metodu do několika menších podmetod, pokud je tělo metody příliš velké. To sice pomůže přehlednosti, ale opět to v důsledku nezbytné režie vede ke zpomalení běhu aplikace; je proto vhodnější se tomu v tomto případě vyhnout.

3.2.8 Inlinování

Jazyky C/C++ dávají možnost tzv. inlinování funkcí provádějících málo operací. Znamená to, že na místo volání této funkce se zkopíruje její tělo. Dosáhne se tím určitého zrychlení, protože se vyhneme režii spojené s voláním metody, ale na druhou stranu to zvýší velikost zkompilevaného kódu. Java bohužel inlinování nenabízí, a tak je někdy nutné to udělat ručně - tj. zkopírovat tělo metody na místo jejího volání a upravit názvy jejích parametrů na názvy skutečných proměnných.

3.2.9 Optimalizace elementárních operací

Pokud jsme již optimalizace na vyšších úrovních provedli, musíme v zájmu dalšího zrychlení sestoupit na nižší úroveň a optimalizovat jednotlivé operace prováděné s daty. Opět platí pravidlo, že se nevyplatí *předpokládat*; musíme proto najít (nebo vyrobit) nástroj, který dokáže změřit, kolik času jednotlivé

operace zaberou. Takovýmto nástrojem je např. SPMark [8]. Z výsledků měření uskutečněných na různých telefonech je zřejmé, které operace je třeba zoptimalizovat či kterým bude lepší se vyhnout.

Dělení

Dělení je bezpochyby jednou z výpočetně nejnáročnějších operací. Na dnešních PC to možná už neplatí, ale mobilní telefony nemají tak pokročilý hardware. Proto je nutné se dělení - tam, kde je to možné - raději vyhnout. Pokud pracujeme s typy *int* či *long*, můžeme dělení mocninou dvou (tj. nějakým 2^n) nahradit bitovým posunem doprava (o n). Stejně můžeme zrychlit i násobení mocninou dvou, to však není tak výpočetně náročné.

Když zavádíme nějakou konstantu, o níž víme, že s ní budeme často dělit či násobit, je vhodné ji zvolit tak, aby byla mocninou dvou, a pak místo ní použít její dvojkový logaritmus. Příkladem je třeba šířka/výška jednoho tilu.

Stejně tak je pomalá i operace *mod*, tj. zjištění zbytku po celočíselném dělení. Pokud chceme zjistit

$$y = x \text{ mod } 2^n, \quad (3.1)$$

můžeme *mod* nahradit operací bitového součinu s číslem o jedna menším, tzn. můžeme rovnici přepsat takto:

$$y = x \wedge (2^n - 1). \quad (3.2)$$

Přístup do pole

Přístup k určitému prvku pole je překvapivě až čtyřikrát pomalejší než násobení. Zde není pro optimalizaci příliš prostoru, takže je třeba se přístupu do pole vyhnout nebo ho alespoň minimalizovat. To se dá zajistit zavedením lokální proměnné, do které uložíme hodnotu prvku v poli, a s ní pak budeme dále pracovat namísto opakovaného přístupu do pole. Protože v Javě jsou všechny proměnné typu objekt referencí, nemusíme se bát toho, že by někde docházelo ke zbytečnému kopírování dat - přinejhorším se zkopíruje jen adresa umístění objektu v paměti.

Float vs. fixed-point

Operace nad daty typu *float* jsou nepoměrně pomalejší než operace nad daty typu *int*. *Float* není na drtivé většině dnešních telefonů hardwareově implementován, takže se výpočty s ním softwarově emulují. Navíc starší telefony *float* vůbec nepodporují. Proto je potřeba se obejít bez něj.

V začátcích tvorby počítačových her se používala tzv. *fixed-point* čísla; využil jsem je i já při implementaci simulace. Jde o způsob, při kterém se čísla reprezentují pomocí typu *int*, ale pracuje se s nimi, jako by předem pevně určený počet bitů (označme n) ležel za desetinnou čárkou. Pokud chceme takto reprezentovat číslo x , musíme ho posunout o n bitů doleva. Pro sečtení dvou čísel stačí sečíst jejich *fixed-point* reprezentace. Abychom mohli dvě čísla vynásobit, musíme jejich reprezentace nejdříve převést do datového typu, který je schopen pojmut dvakrát větší počet bitů, protože až tak se může při násobení číslo zvětšit. Přetypujeme tedy obě reprezentace do typu *long*, vynásobíme je a následně provedeme bitový posun doprava o n bitů, aby se nám desetinná čárka neposunula. Výsledek pak převedeme na *int*. Dělení provedeme podobně, ale opačně: reprezentace čísel převedeme na *long*, na dělení provedeme bitový posun doleva o n bitů, takto získaná čísla vydělíme a výsledek převedeme na *int*. Protože jsou bitové posuny velmi rychlé, výpočet se příliš nezpomalí (při implementaci se ukázalo, že dojde ke zpomalení o cca 20 %).

Porovnávání čísel

Porovnávání dvou čísel typu *long* je až dvakrát pomalejší než porovnávání dvou čísel typu *int*. Proto pokud máme výsledek *fixed point* operace uložen v proměnné typu *long* (toto je často potřeba pro zabránění přetečení), je výhodnější před porovnáním výsledek převést do proměnné typu *int*, protože přetypování je velmi rychlá operace.

Vector

Vector je jedna z knihovnických tříd nacházejících se v JVM. Vector implementuje tzv. *gumové pole*, neboli pole, které nemá předem přesně danou velikost a může se zvětšovat/zmenšovat. Přístup k prvkům je v konstantním čase, ale tato konstanta je tak vysoká, že se Vector v praxi nevyplatí používat.

Pro hry je typické, že v nich předem víme, jaký bude maximální možný počet objektů najednou na hracím poli. Kdyby totiž tento počet totiž nebyl omezený, nemohli bychom zaručit stabilní rychlost hry. Proto je výhodné použít klasické pole, které je mnohem rychlejší než Vector. Pokud někde opravdu potřebujeme gumové pole, je lepší ho implementovat samostatně. Výjimkou jsou situace, ve kterých výkon nepotřebujeme, jako například při načítání hry. Tam naopak můžeme Vector bez problémů použít.

Lineární spojový seznam

Jsou situace, ve kterých by se hodilo použít lineární spojový seznam, a to kvůli jeho konstantní složitosti přidání a odebrání prvku. Proto jsem ho implementoval a provedl měření jeho výkonu. Ukázalo se, že režie spojená s operacemi nad lineárním spojovým seznamem je ve velkém počtu případů větší, než když použijeme pro stejnou úlohu klasické pole. Proto je třeba používat lineární spojový seznam opatrně, *nepředpokládat*, že bude pracovat rychleji než pole, ale místo toho jeho výkon opravdu změřit.

Těmito triky lze program znatelně zrychlit, bohužel ale často za cenu značného zneprůhlednění kódu. Nicméně pokud potřebujeme, aby simulace běžela uspokojivě i na starších modelech telefonů, nezbyvá nám nic jiného.

Kapitola 4

Využití v praxi

V této kapitole uvedu příklad praktického použití simulace a následně měřením výkonu simulace testované na skutečných mobilních telefonech doložím použitelnost simulátoru v praxi. Aplikace je i se zdrojovým kódem přiložena na CD.

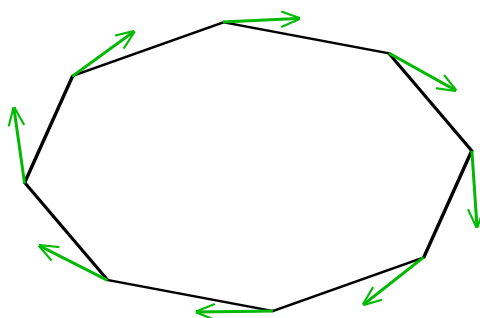
4.1 Příklad použití

Pro demonstraci praktického využití simulátoru jsem vytvořil základ hry, ve které hráč ovládá deformovatelný kulovitý objekt, kterému budu říkat *blob*. Blob dokáže skákat, lepit se na stěny i jiné objekty a interagovat s nimi. Hráč může také měnit fyzikální vlastnosti blobu.

Implementace je poměrně přímočará, simulátor nabízí prostředky, se kterými si vystačíme, včetně lepení na stěny a jiné objekty. Blob jsme schopni simulovat pomocí zachování objemu metodami popsány v sekcích 2.7.3 a 2.7.4. Přesto se vyskytly problémy, které bylo nutné nějak řešit.

Jelikož simulace aktualizuje pouze objekty, které jsou vidět, nemá smysl aktualizovat herní objekty, které nejsou vidět (herní objekt obsahuje simulační objekt a některé další informace, jako je počet životů, vizuální reprezentaci atd.). K tomu bychom mohli využít Implicit grid v simulátoru, ale pak bychom museli nějak navázat herní objekty na simulační objekty i obráceně. Já jsem to vyřešil tak, že jsem prošel všechny objekty a otestoval jejich AABB na průnik s AABB kamery. Na výkon to nemělo prakticky žádný vliv, protože objektů ve hře není mnoho.

Další problém je, jak s blobem pohybovat. Jednou možností je aplikovat na všechny hmotné body sílu ve směru požadovaného pohybu. Tento způsob nedává ale na pohled moc hezké výsledky. Druhý způsob spočívá v roztočení



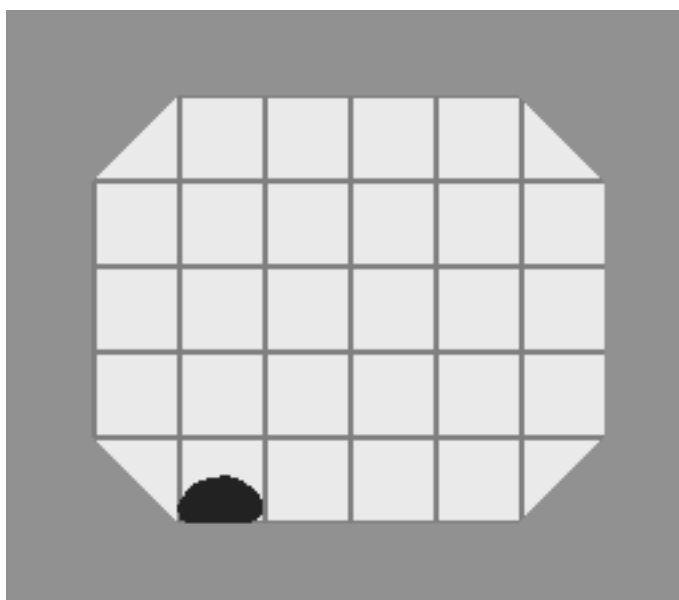
Obrázek 4.1: Roztočení blobu působením silou na body na jeho povrchu.

blobu aplikováním síly na všechny hmotné body ve směru tečen na povrch blobu procházejících těmito body, viz obrázek 4.1. Pak se ovšem blob nepohybuje, pokud se nedotýká žádného povrchu a na hráče to nepůsobí dobře. Mnohem lepší je proto oba tyto způsoby zkombinovat.

Posledním problémem je, jak nasimulovat skákání. Aplikujeme-li na všechny hmotné body stejnou sílu, výsledek působí nepřirozeně; blob se vznese do vzduchu, jako kdyby na něj přestala působit gravitace. Lepší výsledky dostaneme, pokud velikost síly určíme podle vzdálenosti hmotného bodu od země, tj. čím je hmotný bod blíž k zemi, tím menší silou na něj budeme působit; body bližší pevnému povrchu k němu pak přilnou.

4.2 Experimenty

Abych ukázal, že je simulátor v praxi opravdu použitelný, provedl jsem několik testů na skutečných telefonech. Každý test odpovídá jedné mapě aplikace *Blob* popsané v předchozí sekci. Jednotlivé mapy mají různý počet porůznu rozmístěných objektů a měřil jsem minimální, maximální a průměrnou délku simulačního kroku (FD). Minimální (resp. maximální) délka simulačního kroku je minimum (resp. maximum) všech délek přes celý průběh měření. Průměrná délka simulačního kroku se průměruje přes všechna naměřená data. Při měření jsem počkal, dokud se tato hodnota nestabilizuje. Testoval jsem na telefonech *Nokia 3100* (nejnižší třída), *Nokia 6070*, *Motrola V547*, *Siemens SX1*, *Samsung D500* (nižší třída), *Nokia N70* (střední třída), *Nokia N95* (vyšší třída).



Obrázek 4.2: Mapa použitá pro test „Klidový stav“. Tmavé části jsou dynamické objekty, šedé části jsou zdi. Pozadí je bílé se šedivou mřížkou.

Klidový stav

Abych mohl hodnotit jednotlivé testy, změřil jsem, kolik času zabere jeden simulační krok, pokud simulace obsahuje pouze jeden objekt kolidující s jednoduchým statickým světem. V následující tabulce je pro každý typ telefonu uvedena minimální, maximální a průměrná délka simulačního kroku (FD) s přesností na půl milisekundy.

Telefon	min FD [ms]	max FD [ms]	avg FD [ms]
<i>Nokia 3100</i>	32	38	34
<i>Nokia 6070</i>	32	40	33
<i>Motorola V547</i>	24	26	25
<i>Siemens SX1</i>	8	14	9
<i>Samsung D500</i>	0	0	0
<i>Nokia N70</i>	0	0	0
<i>Nokia N95</i>	0	0	0

Statické objekty

V prvním testu jsem zkoumal, jaký vliv má členitost a složitost statického světa na náročnost simulace. Pro testování jsem použil mapu *test1*, ve které jsem nechal padat blob úzkým prostorem s velkým počtem ostrých rohů, takže zde docházelo k mnoha kolizím se statickým světem.

Telefon	min FD [ms]	max FD [ms]	avg FD [ms]
<i>Nokia 3100</i>	36	39	36
<i>Nokia 6070</i>	26	43	29
<i>Motorola V547</i>	24	36	29
<i>Siemens SX1</i>	7	15	8
<i>Samsung D500</i>	0	0	0
<i>Nokia N70</i>	0	0	0
<i>Nokia N95</i>	0	0	0

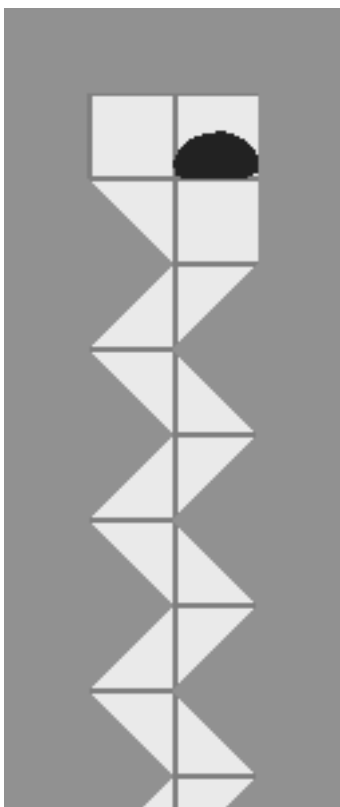
Pokud tuto tabulku porovnáme s tabulkou měření v klidovém stavu, vidíme, že členitost statického světa nemá prakticky vliv na rychlost simulace.

Řídce rozmístěné dynamické objekty

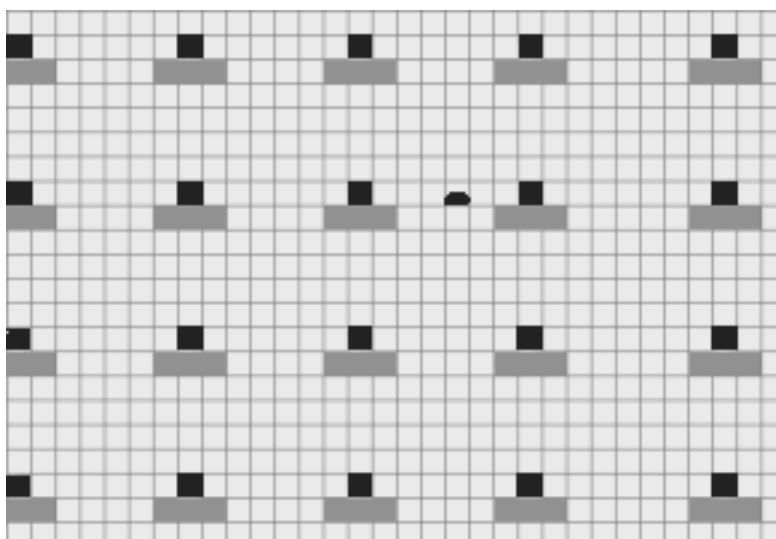
V tomto testu jsem zkoumal, jak si simulace poradí s velkou mapou s velkým počtem dynamických objektů (64 beden). Objekty byly rozmístěny tak, aby byly najednou vidět maximálně dva z nich.

Telefon	min FD [ms]	max FD [ms]	avg FD [ms]
<i>Nokia 3100</i>	30	40	37
<i>Nokia 6070</i>	28	42	34
<i>Motorola V547</i>	20	43	25
<i>Siemens SX1</i>	7	14	7
<i>Samsung D500</i>	0	5	3
<i>Nokia N70</i>	0	0	0
<i>Nokia N95</i>	0	0	0

Je zřejmé, že ani velký počet objektů nemá vliv na rychlost simulace, pokud jsou objekty od sebe dostatečně vzdáleny, takže jich není viditelných mnoho najednou.



Obrázek 4.3: Výsek mapy použité pro test „Statické objekty”. Na mapě je vidět blob padající do úzkého klikatého tunelu.



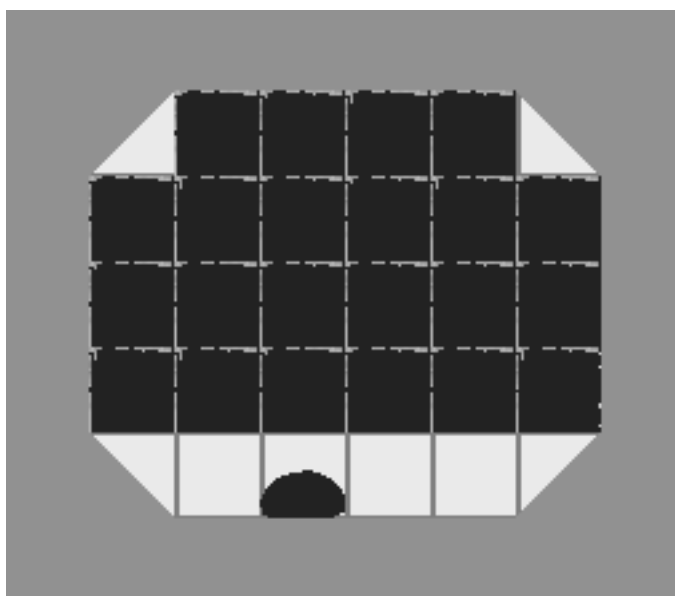
Obrázek 4.4: Výsek mapy použité pro test „Řídce rozmístěné dynamické objekty”. Na mapě je velké množství dynamických objektů (tmavé části), které jsou ale umístěné daleko od sebe, takže jen malý počet z nich je viditelný najednou.

Hustě rozmístěné dynamické objekty

V kontrastu s předchozím testem jsem zkoumal, jaký dopad na rychlost simulace bude mít velký počet objektů (22) všech viditelných najednou.

Telefon	min FD [ms]	max FD [ms]	avg FD [ms]
<i>Nokia 3100</i>	477	590	502
<i>Nokia 6070</i>	358	510	446
<i>Motorola V547</i>	484	700	522
<i>Siemens SX1</i>	161	220	183
<i>Samsung D500</i>	31	52	35
<i>Nokia N70</i>	7	15	9
<i>Nokia N95</i>	3	10	6

Z tabulky vidíme, že v tomto případě se simulátor stává takřka nepoužitelným, protože doba simulačního kroku rapidně převyšuje požadavky na real-timovou simulaci. Takovýchto případů se proto musíme při použití simulátoru vyvarovat.



Obrázek 4.5: Mapa použitá pro test „Hustě rozmístěné dynamické objekty“. Dynamické objekty (tmavé části) jsou umístěné těsně vedle sebe, takže dochází k mnoha kolizím.

Zátěžový test

Pokud chceme simulátor v praxi použít, je dobré znát jeho limitace na konkrétních telefonech. Zjistil jsem proto maximální počet dynamických objektů, při jejichž simulaci bude simulační krok trvat maximálně 200 ms. Takovou prodlevu mezi snímky ještě člověk vnímá jako „plynulý“ pohyb.

Telefon	Počet objektů
<i>Nokia 3100</i>	13
<i>Nokia 6070</i>	14
<i>Motorola V547</i>	15
<i>Siemens SX1</i>	44
<i>Samsung D500</i>	129
<i>Nokia N70</i>	386
<i>Nokia N95</i>	369

Vzhledem k tomu, že v praxi nemá smysl simulovat více než 10 objektů viditelných najednou, tak i na pomalejších telefonech je simulátor zřejmě použitelný.

Ze všech tabulek je patrné, že i ty nejpomalejší mobilní telefony si velmi dobře poradí se simulací, ve které není větší množství objektů blízko u sebe. Pokud je tato situace vyžadována, je stále možné simulátor použít s tím, že na některých telefonech nižší třídy aplikace poběží příliš pomalu. Na ně lze ovšem vytvořit jinou (odlehčenou) verzi aplikace. Tento postup je dnes v oblasti vývoje mobilních her běžný.

Kapitola 5

Závěr

Ukázal jsem, jak vytvořit fyzikální simulaci způsobem nenáročným na výkon procesoru, přitom však velmi robustní a vizuálně uspokojivou. Dále jsem uvedl několik postupů vedoucích k optimalizaci programů napsaných v J2ME. Tím jsem poskytl všechno, co je nezbytné k úspěšné implementaci fyzikální simulace na mobilní telefony. Nakonec jsem předvedl, jak se tento simulátor dá použít; vytvořil jsem jednoduchou hru a pomocí ní jsem změřil výkon simulátoru na několika skutečných mobilních telefonech. Simulátor je implementován ve formě J2ME knihovny na přiloženém CD spolu se zmiňovanou hrou.

Základem simulátoru je *mass-spring* systém – soustava hmotných bodů spojených pružinami. Z těch jsou tvořeny všechny dynamické objekty. Během simulačního kroku se nejprve zintegrují pohybové rovnice hmotných bodů pomocí numerické Verletovy integrace, která zaručí vysokou stabilitu simulace a odstraňuje nutnost explicitně pracovat s rychlostí. Následně dojde k detekci kolize mezi všemi dvojicemi objektů urychlené pomocí speciálních datových struktur. Nakonec se nalezené kolize objektů vyřeší přesunutím hmotných bodů do takových pozic, ve které spolu objekty nekolidují. Zvláštním způsobem se řeší zachování objemu těles zabraňující jejich zhroucení do sebe.

Z výsledků provedených experimentů je vidět, že simulátor se dá v praxi dobře použít. Proto si myslím, že má práce byla úspěšná. Dokládá to i využití popsaného simulátoru v komerčně vydané mobilní hře *Gish Mobile* [7].

Reference

- [1] Bourke P.: *Determining if a point lies on the interior of a polygon*, <http://local.wasp.uwa.edu.au/~pbourke/geometry/insidepoly/>, 1987.
- [2] Bulej L.: Přednáška *Principy počítačů a operačních systémů*, <http://dsrg.mff.cuni.cz/teaching/nswi120/>, 2009.
- [3] Côté A. S.: *Democritus: A Molecular Dynamics tutorial*, <http://www.compsoc.man.ac.uk/~lucky/Democritus/>, 2001.
- [4] Eckel B.: *Thinking in Java*, Prentice-Hall, Inc., 2002.
- [5] Eidos Mobile: *SolaRola*, <http://www.eidosmobile.com/solarola>, 2007.
- [6] Ericson Ch.: *Real-Time Collision Detection*, Morgan Kaufmann, 2005.
- [7] Erphenic, Hardwire: *Gish Mobile*, <http://www.gishmobile.com/>, 2009.
- [8] Futuremark: *SPMark Java*, <http://www.futuremark.com/products/spmark/spmarkjavajs184/>, 2008.
- [9] Gameloft: *Wonder Blocks*, <http://www.gameloft.com/mobile-games/wonder-blocks/?animid=54730>, 2008.
- [10] Gomez M.: *Coping with Friction in Dynamic Simulations*, Game Programming Gems 3, Charles River Media Inc., 2002.
- [11] Gottschalk S., Lin M. C., Manocha D.: *OBBTree: A Hierarchical Structure for Rapid Interference Detection*, Proceedings of SIGGRAPH'96, 1996.

- [12] Hecl R.: *Afrodite*, http://www.freemobil.cz/view_article.php?article_id=7, 2007.
- [13] Jakobsen T.: *Advanced Character Physics*, http://www.gamasutra.com/resource_guide/20030121/jacobson_01.shtml, 2003.
- [14] Kishonti Informatics LP: *JBenchmark*, <http://www.jbenchmark.com>, 2007.
- [15] Knizhnik K.: *JavaGO*, <http://www.garret.ru/~knizhnik/javago/ReadMe.htm>, 2006.
- [16] Kot G.: osobní konzultace, 29.9.2007.
- [17] Lafortune E.: *Proguard*, <http://proguard.sourceforge.net/>, 2008.
- [18] Matyka M., Ollila M.: *A pressure model for soft body simulation*, <http://www.ep.liu.se/ecp/010/007/ecp01007.pdf>, Proc. of Sigrad, Umea, November 2003.
- [19] Metanet software: *Basic Collision Detection and Response*, <http://www.harveycartel.org/metanet/tutorials/tutorialA.html>, 2004.
- [20] Mocný O.: *DyMiX*, <http://dymix.hardwire.cz>, 2007.
- [21] Mocný O.: *Zápočtový program na předmět Algoritmy a datové struktury II*, <http://physics.hardwire.cz/mirror/CollTest.zip>, 2007.
- [22] Müller M., Heidelberger B., Teschner M., Gross M.: *Meshless Deformations Based on Shape Matching*, http://www.beosil.com/download/MeshlessDeformations_SIG05.pdf, Proceedings of SIGGRAPH'05, 2005.
- [23] Müller M., Heidelberger B., Hennix M., Ratcliff J.: *Position Based Dynamics*, <http://www.matthiasmueeller.info/publications/posBasedDyn.pdf>, 2006.
- [24] Olejník M.: *Pstros*, <http://www.volny.cz/molej/pstros/>, 2007.
- [25] Oliviera G.: *Exploring Spring Models*, http://www.gamasutra.com/features/20011005/oliveira_01.htm, 2001.
- [26] Sun Microsystems: *Wireless Toolkit*, <http://java.sun.com/javame/downloads/index.jsp>, 2008.

- [27] Rhodes G. S.: *Stable Rigid-body Physics*, http://www.gamasutra.com/features/gdcarchive/2001/rhodes_paper.pdf, 2001.
- [28] Rivers A., James D.: *FastLSM: Fast Lattice Shape Matching for Robust Real-Time Deformation*, <http://www.fastlsm.com/>, 2007.
- [29] Telcogames: *Flexis Extreme*, <http://www.flexisextreme.com>, 2007.
- [30] Töpfer P.: *Algoritmy a programovací techniky*, Prometheus, 1995.
- [31] Xendex: *Freestyle Moto-X II*, <http://www.xendex.com/index.php?&page=0&categoryid=4&gameid=146>, 2006.
- [32] Verlet L.: *Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules*, *Phys. Rev.* 159, (1967) 98–103.
- [33] Walaber: *JelloPhysics*, <http://www.walaber.com/index.php?action=showitem&id=16>, 2007.
- [34] West M.: *Blob Physics*, <http://cowboyprogramming.com/2007/01/05/blob-physics/>, Game Developer Magazine, 2006.

Příloha A

Seznam zkratek

- AABB — Axis Aligned Bounding Box; kvádr, jehož stěny jsou rovnoběžné s osami souřadnic.
- FD — Frame Delay; doba, která uplyne mezi dvěma následujícími simulačními kroky.
- FPS — Frames Per Second; počet simulačních kroků provedených za vteřinu.
- IG — Implicit Grid; datová struktura pro efektivní detekci kolize v *broad phase*.
- J2ME — Java 2 Micro Edition; více na <http://java.sun.com/javame>.
- JVM — Java Virtual Machine; program, který je schopný spouštět programy napsané v Javě.
- MTD — Minimum Translation Distance; nejmenší vzdálenost, o kterou je potřeba posunout objekty, aby spolu nekolidovaly.
- MTV — Minimum Translation Distance; vektor, v jehož směru je potřeba posunout objekty, aby spolu nekolidovaly.
- SAT — Separation Axis Theorem; věta o konvexních mnohoúhelnících a jejich průniku.
- WTK — Wireless TollKit; sada nástrojů pro vývoj J2ME softwaru.

Příloha B

Seznam cizích pojmů

- Broad phase — první fáze detekce kolize, ve které se objekty otestují jen nahrubo.
- Fixed-point numbers — reprezentace reálných čísel taková, že pro celou a desetinnou část je vyhrazen konstantní počet bitů.
- Mass-spring system — množina hmotných bodů spojených pružinami.
- Narrow phase — druhá fáze detekce kolize, ve které se kolize mezi objekty spočítá přesněji.

Příloha C

Obsah přiloženého CD

CD přiložené k této bakalářské práci obsahuje text práce ve formátu PDF a soubory k ročníkovému projektu „Fyzikální simulace pro mobilní telefony”, na který tato práce navazuje. Ty zahrnují instalační balíčky aplikace Blob, zdrojové soubory projektu a konfigurační soubory pro Eclipse IDE. Přiložena je i specifikace projektu, uživatelská dokumentace a programátorská dokumentace. Uživatelská dokumentace popisuje nainstalování a ovládání aplikace. Programátorská dokumentace popisuje postup zkompilování aplikace v Eclipse IDE a zapojení knihovny do jiných projektů.