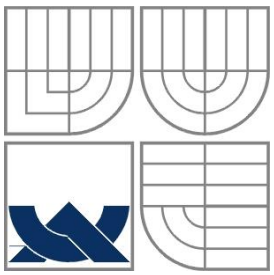
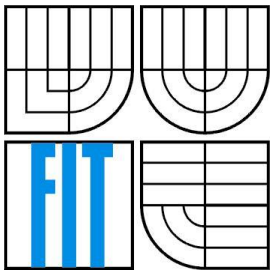


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

FYZIKÁLNÍ SIMULACE PRO HRU PINBALL

PHYSICS SIMULATION FOR PINBALL GAME

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Jakub Čermák
AUTHOR

VEDOUCÍ PRÁCE — Ing. Michal Zachariáš
SUPERVISOR

BRNO 2014

Abstrakt

Cílem této práce je vytvořit knihovnu pro fyzikální simulaci a pomocí ní vytvořit hru Pinball, na které bude vidět její funkčnost. Ve zprávě jsou popsány problémy a jejich řešení, které se v průběhu návrhu a programování objevily. Mezi těmito problémy lze vyzdvihnout tunelování, detekce a výsledek kolize. Jsou popsány i optimalizace řešení těchto problémů, které bylo možné použít pro naší specifickou knihovnu. Neméně důležité jsou principy, které snižují zátěž výsledné simulace - mezi hlavní lze zařadit správnou volbu mezí, detekce kutálení a uspání pomalu se pohybujících dynamických objektů.

Abstract

The objective of this work is to create a library for physical simulation and using the library to create a game of pinball, on which you can test functionality of the library. Problems that appeared during the creation of the library, are described in this paper along with their solutions. In particular, tunneling, detection and results of collisions. Subsequent optimization solutions to these problems for our specific library. Of no smaller importance are principles which reduce the requirements of the final simulation. Among the most important ones are the correct choice of limits, detection of rolling and sleep of slowly moving dynamic objects.

Klíčová slova

C++, detekce kolizí, výsledek kolize, tunelování, průnik, kutálení, časovač, fyzikální simulace, pinball, OpenGL

Keywords

C++, collision detection, collision response, tunneling, intersection, rolling, timer, physical simulation, pinball, OpenGL

Citace

Čermák Jakub: Fyzikální simulace pro hru Pinball, bakalářská práce, Brno, FIT VUT v Brně, 2014

Fyzikální simulace pro hru Pinball

Prohlášení

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

……
Jakub Čermák
14. května 2014

Poděkování

Děkuji svému vedoucímu Ing. Michalu Zachariáši za odborné vedení a podněty, které mi při řešení tohoto projektu poskytl.

Obsah

1 Úvod

Cílem této bakalářské práce je popsat a vytvořit knihovnu, která umožnuje fyzikální simulaci ve 2D. S pomocí této knihovny následně vytvořit hru Pinball, na které bude její funkčnost ukázána.

Slovo simulace se používá v mnoha souvislostech - testování, vzdělávání, optimalizace výkonu nebo také modelování přírodních a lidských systémů s cílem získat poznatky o jejich fungování. Pro potřeby této práce je žádoucí počítačová simulace, která simuluje situace z reálného nebo i hypotetického světa.

Simulací se využívá v mnoha odvětvích - počítačové hry a videohry, filmy, městské simulátory, simulace digitálního životního cyklu, simulace připravenosti na katastrofy, simulační výcvik, strojírenské, technologické nebo procesní simulace, simulace satelitních navigaci, letecké simulátory, finance, námořní simulátory, vojenské simulace a další. Všechna tato odvětví dnes převážně využívají počítačové simulace.

Počítačová simulace je taková, při níž modelem je počítačový program, který se pokouší simulovat model určitého systému. Ač se budeme snažit sebevíc, skutečně reálného modelu docílit nelze, proto jsou počítačové simulace vždy do jisté míry abstrakcí skutečnosti. Na nás je, jaký stupeň abstrakce zvolíme, aby výsledná simulace vyhovovala zadaným požadavkům. Úkolem simulačního programu je zjistit, jak se bude systém chovat pro zadaná vstupní data. Program neprovádí optimalizaci, tzn. nehledá, pro která vstupní data dostaneme optimální řešení. Uživatel může provádět se simulačním programem opakované experimenty, s cílem zjistit očekávané výsledky pro různá vstupní data a nalézt tak optimální řešení problému [1].

Fyzikální simulace se týkají simulací, v nichž jsou skutečné objekty nahrazeny modelovými (některé kruhy užívají termín počítačové simulace pro modelování vybraných fyzikálních zákonů). Tyto modelové objekty jsou často vybírány, protože jsou menší a levnější, než skutečný objekt nebo systém.

Zpráva je rozdělena do tří hlavních kapitol. V první si ukážeme několik existujících řešení fyzikálních simulací, druhá obsahuje popis vlastního řešení a v závěrečné si přiblížíme fungování výsledného programu.

2 Motivace

Pinball je původně druh arkádové hry (zpravidla na mince), ve které jsou body získávány ovlivňováním kuličky na herním poli ve skleněné skříince, které se říká hrací automat. Hlavním cílem hry je uhrát co nejvíce bodů. Body jsou získávány, když kulička udeří na různé cíle na herním poli. Kulička je ve hře udržována dvěma páčkami (anglicky *flippers*), které jsou umístěny ve spodní části hracího pole. Hra končí poté, co hráči propadnou všechny kuličky pod herní pole.

Historie Pinballu je už dnes v podstatě uzavřená kapitola. Na počátku osmdesátých let byla výroba hracích stolů součástí obrovského průmyslu mincových automatů, zatímco počítačové hry tvořily takřka na koleně několikačlenné skupinky nadšenců pro hrstku sobě podobných, co měli doma počítač. Dnes se situace na trhu zábavy úplně otočila. Mechanické zábavní automaty byly z trhu prakticky vytlačeny počítači a videohrami - a vyrábí-li se dnes ještě nějaké nové Pinballové stoly, je to víceméně jen okrajová, nostalgická záležitost pro pár nadšenců [2].

Nejpopulárnější verzí na počítač se stal *3D Pinball for Windows*, který se poprvé objevil na Windows XP. Původně se jednalo o součást Windows, jenže od Windows Vista ji už v nabídce Hry nenaleznete.

Obrázek 1- 3D Pinball for Windows

3 Existující řešení

Z řešení, která již existují, jsem si vybral - Box2D [3] a Chipmunk2D [4]. Zejména z toho důvodu, že jsou si do velké míry podobné a jsou zaměřeny na simulaci v rovině.

Výsledkem prostudování těchto dvou řešení, bylo sestavení základní struktury pro simulaci. Zjednodušeně řečeno se tato struktura skládá ze tří prvků: *World*, *Body* a *Shape*. Hlavní výhodou je, že co chceme, aby bylo využíváno všemi tělesy simulace (např. gravitace), vložíme do prvku *World*. Následně nestálé hodnoty těles (pozice, rychlost, …) jsou obsaženy v části *Body*. Nakonec v průběhu simulace neměnný popis tělesa (tvar, pružnost, hustota, …) uložíme v prvku *Shape*.

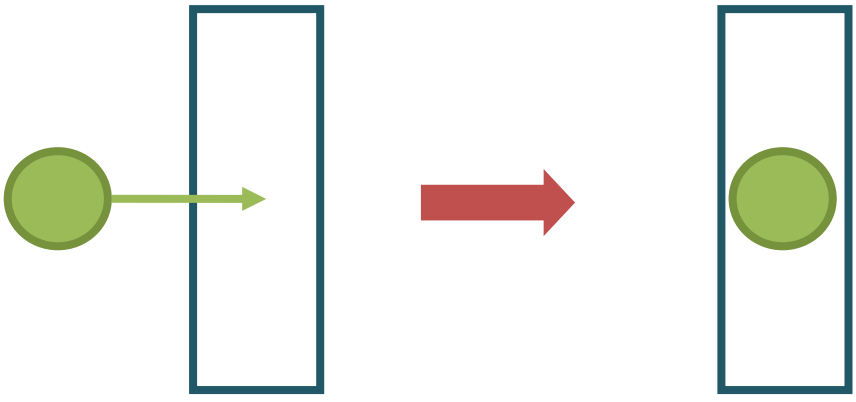
Tato struktura je základním stavebním prvkem pro Box2D i Chipmunk2D a jako takovou jsem tuto strukturu přejal. Dále se již mnoho poznatků z těchto řešení použít nedalo. Neboť jak Box2D, tak Chipmunk2D, jsou řešení pro komplexní simulaci, zatímco u našeho řešení bylo možné mnoho věcí přizpůsobit pouze pro hru Pinball.

3.1 Kolize

Nejdůležitější částí fyzikální simulace jsou kolize. Řešení kolizí se dá rozdělit na dvě podskupiny. Jsou jimi detekce kolizí a výsledek kolize. Zatímco detekce řeší kdy a kde se objekty srazí, výsledek kolize nám ovlivňuje, v jakém pohybu budou tělesa po kolizi. Od detekce kolizí lze ještě odtrhnout a řešit samostatně tzv. tunelování.

3.1.1 Detekce kolizí

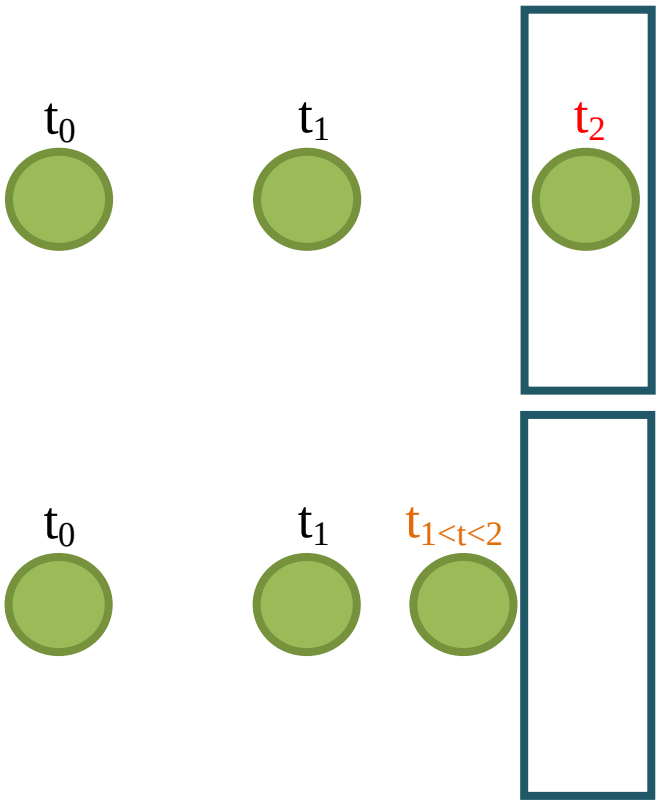
Detekce kolizí má za úkol zjistit, jestli nějaká kolice nastala a pokud ano, tak v jakých bodech objektů k ní dochází. To ale není všechno, velmi důležitou úlohou je také detekce, jestli nedošlo k průniku nebo tunelování. Průnik nám vyjadřuje, že tělesa jsou vnořena v sobě (Obrázek 2- Průnik). V takovém případě se musíme vracet v simulaci zpět a zjistit, kdy dojde ke kolizi bez toho, aby došlo ke vnoření jednoho objektu do druhého.



Obrázek 2- Průnik

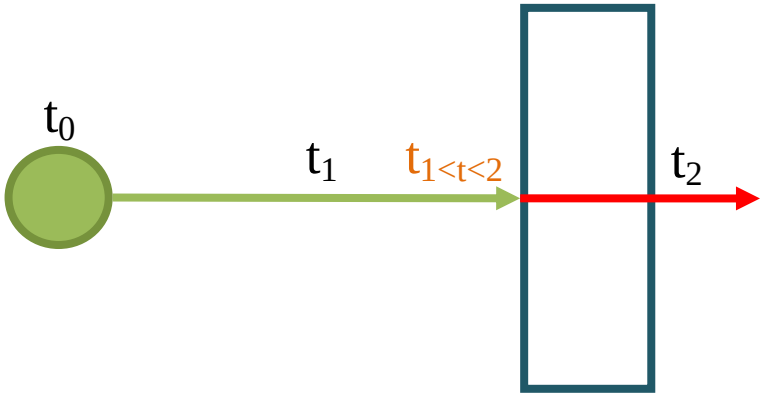
Fyzikální simulátory obvykle fungují na jednom ze dvou způsobů - detekce kolizí *posteriori* (poté co ke kolizi dojde) nebo *priori* (dříve než ke kolizi dojde) [5]. Jinak se také používá pojmenování *discrete* a *continuous* namísto *posteriori* a *priori*.

V detekci podle *posteriori (discrete*) probíhá simulace po malých krocích a po každém kroku se díváme, zda nenastala kolize nebo jestli nějaké objekty nejsou už tak blízko, že to považujeme za kolizi. Při tomto principu musíme zvlášť řešit tunelování. Pokud nastane průnik nebo tunelování, vracíme se zpět a krátíme simulační krok.



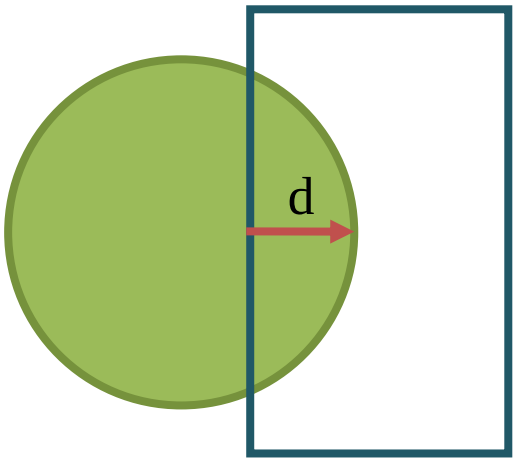
Obrázek 3 - Posteriori

Pro detekci podle *priori (continuous)* není stanoven pevný krok, ale hledáme čas, kdy se dva objekty střetnou, tzv. *time of impact*. Toto řešení se obtížněji implementuje, ale výsledná simulace je věrnější a stabilnější.



Obrázek 4 - Priori

Stabilita simulace je narušována velmi rychlými objekty, které „proletí“ přes jiné (tzv. tunelování) bez detekce kolize. Nebo kolizi detekujeme, ale velikost překryvu je tak velká, že nebude vyřešena příliš věrohodně. Tyto problémy se stabilitou se týkají pouze řešení *posteriori*, neboť *priori* sleduje dráhu pohybujících se objektů. Proto kolizi detekuje vždy (odstranění tunelování). Také je detekována v místě a čase, kdy se objekty právě srazily (nedojde k překryvu). Pro řešení *posteriori*, které může mít se stabilitou problémy, musíme zvlášť řešit tunelování, které je popsáno níže, a co nejvíce omezit překrývání objektů při kolizi, což však rapidně zvyšuje zátěž počítače. Teoreticky můžeme simulační krok krátit do takové míry, že překryv bude naprosto zanedbatelný, ale tím nepřijatelně zvýšíme náročnost simulace. Z toho důvodu musíme stanovit přijatelnou mez pro detekci kolize. Pokud tuto mez překročíme, místo kolize se nám bude jednat o průnik.



Obrázek 5 - Překryv o velikosti d

3.1.2 Výsledek kolize

Pro kvalitní výsledek kolize je důležité, aby byla detekce kolize co nejpřesnější. Výpočet se provádí pomocí impulsů [6].

Pro usnadnění orientace ve vzorcích jsou vektory označeny šipkou.

Impuls J je definován jako pomocí síly F za časovou jednotku Δt jako:

$$J = F\Delta t \quad (1)$$

Z druhého Newtonova zákona víme, že $F = ma$ a zrychlení můžeme přepsat jako změna rychlosti za čas:

$$J = F\Delta t = ma\Delta t = m\frac{\Delta v}{\Delta t}\Delta t = m\Delta t \quad (2)$$

Nakonec dostáváme, že změna rychlosti je rovna impuls děleno hmotnost tělesa:

$$\Delta v = \frac{J}{m} \quad (3)$$

Impuls dvou těles se dále vypočítá jako:

$$J = \frac{-(1+\varepsilon)\vec{v}_{ab}\vec{n}}{\vec{n}\vec{n}\left(\frac{1}{m_a}+\frac{1}{m_b}\right)} \quad (4)$$

Kde $\vec{v}_{ab}$ vyjadřuje relativní rychlost kolidujících těles, $\vec{n}$ je jednotkový vektor ve směru kolize, ε vyjadřuje elasticitu a m_a, m_b vyjadřují hmotnost těles.

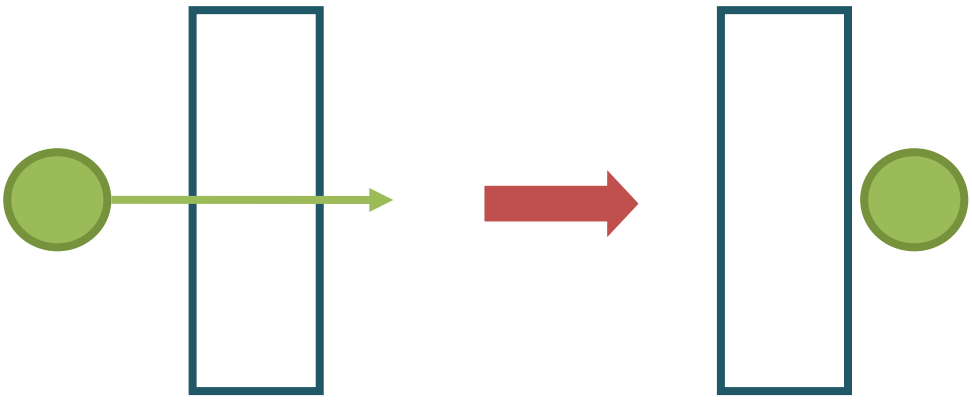
Nyní lze vypočítat rychlost těles po srážce:

$$\vec{v}_a^{+} = \vec{v}_a^{-} + \frac{J}{m_a}\vec{n}$$
$$\vec{v}_b^{+} = \vec{v}_b^{-} - \frac{J}{m_b}\vec{n} \quad (5)$$

Zde $\vec{v}^{-}$ vyjadřuje rychlost těles před kolizí a $\vec{v}^{+}$ rychlost po provedení kolize.

3.1.3 Tunelování

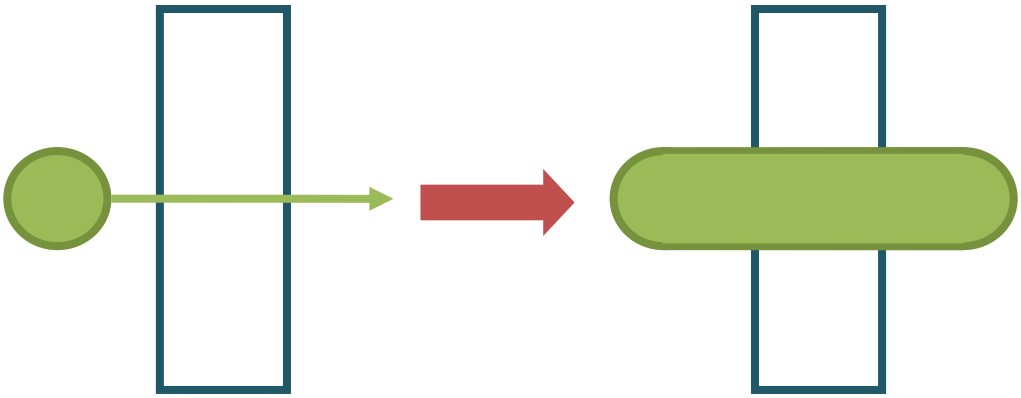
Tunelování musíme řešit, pokud se rozhodneme detekci kolizí provádět pomocí metody *posterior*. Jedná se o problém, kdy rychle pohybující se těleso projde skrz objekt bez toho, aby byla zjištěna kolize [7].



Obrázek 6 - Tunelování

Pro odstranění tohoto jevu existuje několik řešení například:

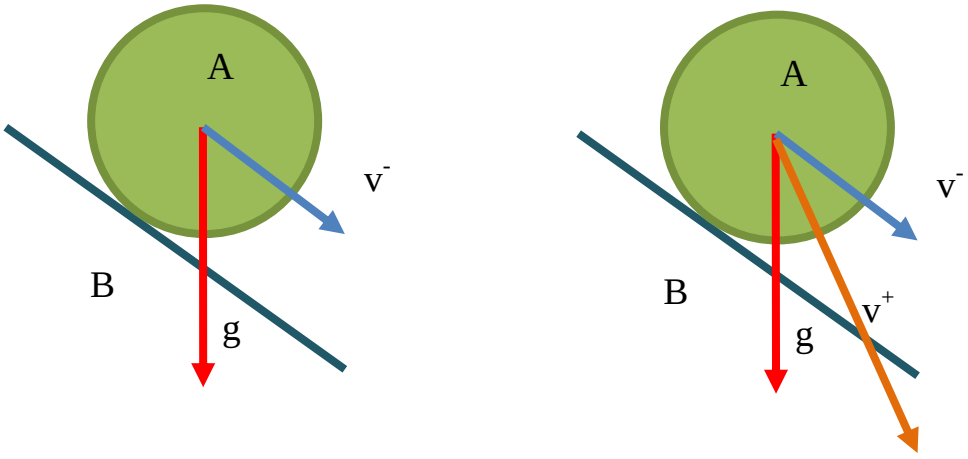
- Velmi malý simulační krok
- Detekce kolizí pomocí metody *priori*
- Protažení pohybujícího objektu na celou jeho dráhu (Obrázek 7)



Obrázek 7 - Protažení pohybujícího se objektu, pro detekci tunelování

3.2 Kutálení

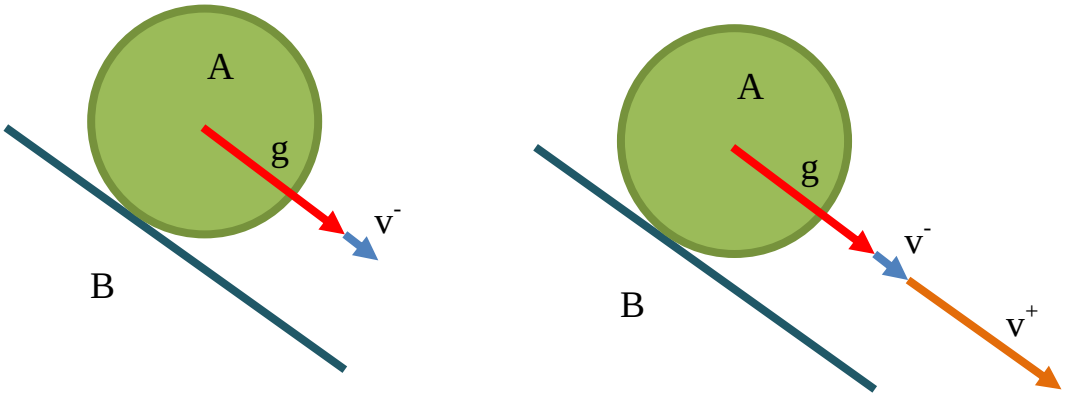
Kutálení se často označuje anglickým názvem *rolling*. Tento jev je pro fyzikální simulace velmi důležitý. Jeho použitím se jak zlepší věrohodnost simulace, tak se sníží výpočetní zátěž. Zátěž na výkon počítače je při kutálení odlehčena především kvůli redukci počtu kolizí. Kolize plynoucí z absence kutálení se vyskytují na ploše, po které se pohybuje kulatý objekt. Neboť gravitační síla působící na objekt, by nežádoucím způsobem ovlivňovala směr jeho pohybu, který by měl za následek časté střety mezi objektem a plochou, po které se pohybuje.



Obrázek 8 - Působení gravitačního zrychlení bez detekce kutálení

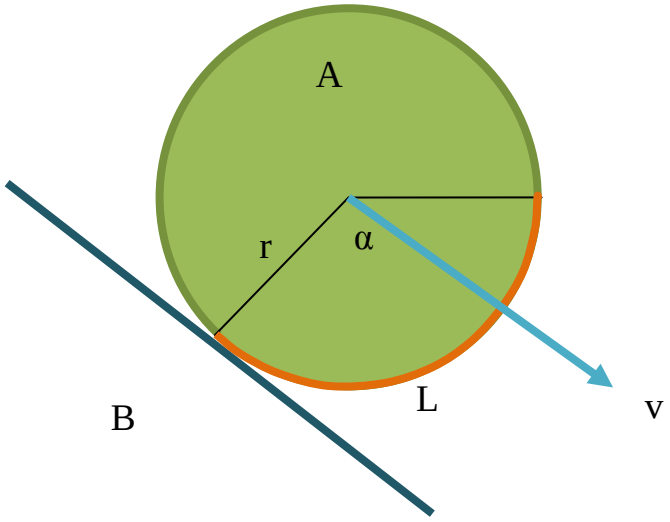
Tyto kolize by nastávaly velmi často a značně by ovlivňovaly věrohodnost celé simulace. Proto je nutné kutálení u takovýchto objektů detekovat a následně upravit gravitační zrychlení dle směru

plochy, po které se objekt kutálí. Po takovéto úpravě, nám směr výsledné rychlosti vyjde rovnoběžný s plochou, po které se objekt pohybuje a kolize, které způsobují značnou režii, již nejsou detekovány.



Obrázek 9 - Působení gravitačního zrychlení při detekci kutálení

Jak už naznačuje název jevu, dá se předpokládat, že primárním objektem, kterého se bude týkat, je kruh. Můžeme si myslet, že je nezbytně nutné použít rotaci. To však není zcela pravda, neboť v nějakých případech není zjevné, zda objekt opravdu rotuje. Proto by bylo možné rotaci vynechat a pracovat pouze s pozicemi objektů. Takové případy jsou časté. Např. Obrázek 9: není poznat, zda je objekt A otočen o nějaký úhel.



Obrázek 10 - Výpočet rotace při kutálení

$$\alpha = \frac{L}{r} \quad (6)$$

Úhel, o který se objekt otočí, je závislý na dráze, kterou kruh urazí, a poloměru. Následně tuto vzdálenost vydělíme poloměrem kruhu a získáme úhel, který nám vyjadřuje rotaci objektu.

3.3 Uspání objektů

Využívá se pro snížení výpočetní zátěže. Jedna se o to, že se těleso pohybuje velmi pomalu nebo se nehýbe vůbec. Převedeme ho do režimu spánku, ve kterém nebudeme měnit polohu, ani detekovat kolize způsobené tímto objektem. Typickým příkladem je objekt na rovné ploše. Pokud bychom takový objekt nepřevedli do spánku, pak bychom co pár milisekund museli řešit kolizi tohoto objektu s plochou, na které leží.

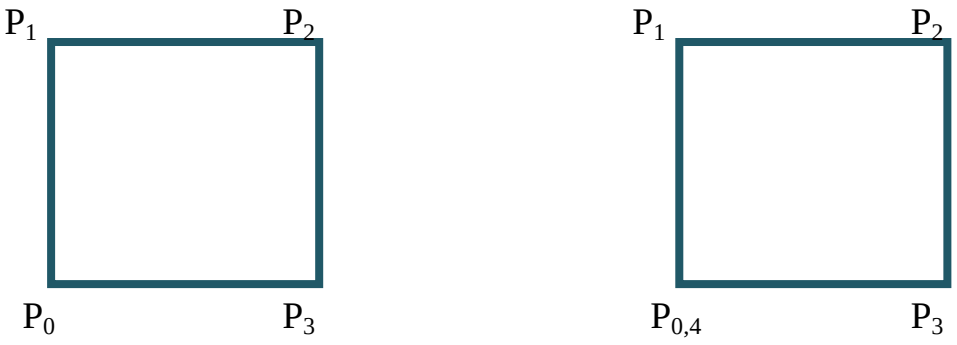
Neznamená to ovšem, že takové těleso se nemůže účastnit kolize. Pokud do uspaného tělesa narazí jiný objekt, provedeme kolizi a s největší pravděpodobností tato kolize našemu objektu dodá dostatečný impuls, aby se objekt probudil.

4 Vlastní řešení

Naše řešení fyzikální simulace je optimalizováno pro hru Pinball. Díky této optimalizaci lze efektivněji detekovat kolize a tunelování, neboť tato detekce se provádí pouze pro jediný objekt v simulaci a to kuličku.

Při simulaci jsou využívány tři druhy těles: statická, dynamická a kinetická. Každý z těchto druhů se při simulaci chová odlišně. Statické těleso se nepohybuje a je pevně připevněno na své místo. Taková tělesa jsou využita pro ohraničení herního pole a cílů, za které se dostávají body. Dynamické těleso se v Pinballu vyskytuje pouze jedno a to je kulička. Právě a pouze pro dynamická tělesa se detekují kolize a je nutné odstranit tunelování. Kinetické těleso je ukotveno pomocí kloubu (*Joint*) a dovoluje pouze rotační pohyb. Takovými tělesy jsou v našem případě páčky, kterými se hráč snaží udržet kuličku co nejdéle ve hře.

Tělesa dále nabývají jednoho ze tří tvarů a to kruh (*circle*), polygon (*polygon*) a hrany (*edges*). Kruh je určen středem a poloměrem. Polygony mohou být pouze konvexní, pokud chceme nekonvexní nebo jiný tvar, použijeme hrany. Jak jste si jistě všimli, není napsáno hrana, ale hrany, neboť tento tvar je určen dvěma a více body, které jsou spojeny. Jinak řečeno - čtverec lze vytvořit čtyřmi body jako polygon nebo pěti body pomocí hran, při čemž první a poslední bod je stejný.



Obrázek 11 - Vytváření čtverce pomocí polygonu (vlevo) a hran (vpravo)

Při optimalizaci pro hru Pinball jsem omezil kombinace druhů těles a jejich tvarů. Jedině statická tělesa mohou nabývat jakéhokoliv tvaru. Naopak dynamické, může být pouze kruh. Pro kinetická tělesa lze použít polygon a hrany, ale nesmíme zapomenout na kloub, kolem kterého se toto těleso bude otáčet.

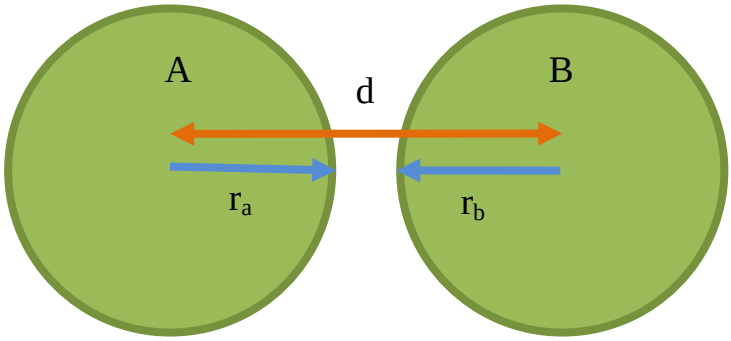
Kloub je bod, kolem kterého těleso rotuje. Není nutné, a v našem případě ani žádoucí, aby se těleso otáčelo o celých 360 stupňů. Z tohoto důvodu lze nastavit maximální a minimální úhel pro rotaci. Objekt se poté bude pohybovat pouze v námi vymezené výseči.

4.1 Kolize

Naše řešení používá detekci podle *posteriori*, tedy simulace po malých krocích. Metodu *priori* jsem si nevybral z jednoho prostého důvodu. Pokud bychom pomocí metody *priori* hledali a řešili kolize dopředu, tak pokaždé, kdy uživatel použije páčku, musíme všechny výpočty zahodit a přepočítat znovu. Proto se metoda *priori* více hodí do prostředí, kde uživatel v průběhu nezasahuje do dění simulace a všechny akce, které se provedou, jsou známy předem.

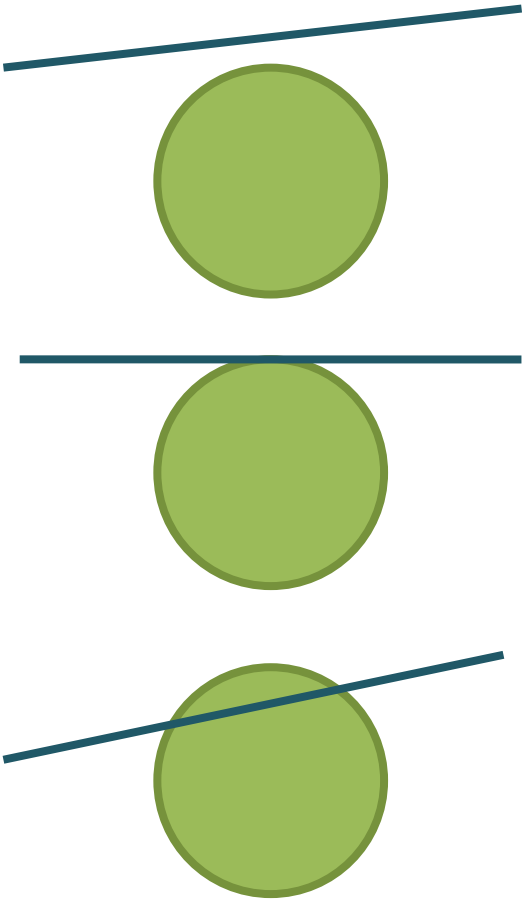
4.1.1 Detekce kolizí

Detekce je optimalizována, jak již bylo řečeno, dříve tím, že jeden z objektů účastnící se kolize bude vždy kruh. Další zjednodušení si uděláme tím, že polygon můžeme rozdělit na jednotlivé hrany. Tím se nám detekce zjednoduší pouze na měření vzdálenosti dvou kruhů a vzdálenost koule od úsečky. První z těchto měření je jednoduché. Stačí porovnat vzdálenost středů se součtem poloměrů.



Obrázek 12 – Detekce kolize koulí

Druhé měření začíná jako vzájemná poloha kružnice a přímky. V rovině mohou nastat tři různé vzájemné polohy kružnice a přímky. Rozlišujeme je podle toho, kolik mají společných bodů. Mohou nastat tyto případy - nemají žádný společný bod, mají jeden společný bod nebo mají dva společné body.



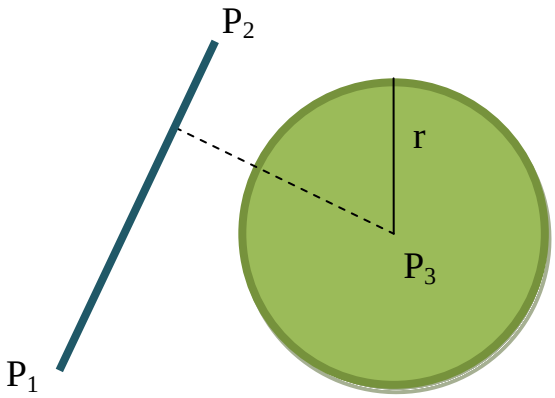
1) Žádný společný bod

2) Jeden společný bod

3) Dva společné body

Obrázek 13 - Vzájemná poloha kružnice a přímky

Náš případ, který namísto přímky požaduje úsečku, má těchto situací více. I nám ale stačí tyto tři základní - pouze následně musíme ověřit, zda nalezené průsečíky leží na úsečce. Řešení, podle kterých lze nalézt průsečíky kružnice a přímky, jsem získal upravením řešení pro 3D [11].



Obrázek 14 - Obrázek k výpočtu

Základem je kvadratická rovnice ve tvaru:

$$Ax^2 + Bx + C = 0 \quad (7)$$

Kde *A*, *B* a *C* jsou definovány jako:

$$\begin{aligned} A &= (\vec{p}_1 - \vec{p}_2) \cdot (\vec{p}_1 - \vec{p}_2) \\ B &= 2((\vec{p}_1 - \vec{p}_2) \cdot (\vec{p}_1 - \vec{p}_3)) \\ C &= ((\vec{p}_1 - \vec{p}_3) \cdot (\vec{p}_1 - \vec{p}_3)) - r^2 \end{aligned} \quad (8)$$

Následně lze průsečíky vypočítat přes řešení kvadratické rovnice:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (9)$$

Počet průsečíků P lze zjistit ještě před vypočítáním průsečíků a to podle determinantu:

$$\begin{aligned} D &= b^2 - 4ac \\ P &= \begin{cases} 0 & D < 0 \\ 1 & D = 0 \\ 2 & D > 0 \end{cases} \end{aligned} \quad (10)$$

Následně pokud nějaké průsečíky nastaly, musíme zjistit, jestli leží na dané úsečce. Pokud nějaký z průsečíků opravdu leží na úsečce, je třeba ověřit, zda se jedná o kolizi nebo průnik.

4.1.2 Výsledek kolize

Důležitým aspektem, který se týká výsledku kolizí, je fakt, že budeme pracovat s neměnnými objekty. To znamená, že objekty v simulaci nebudou měnit svůj tvar ani při nárazu. Tento aspekt do jisté míry nahradíme pružností, která nám bude ovlivňovat výslednou rychlost objektů, které se srazí. Pružnost by měla nabývat hodnot větších než 0 a menších než 1, v krajním případě 1. Pokud je hodnota pružnosti větší jak 1, objekty se srážkami zrychlují.

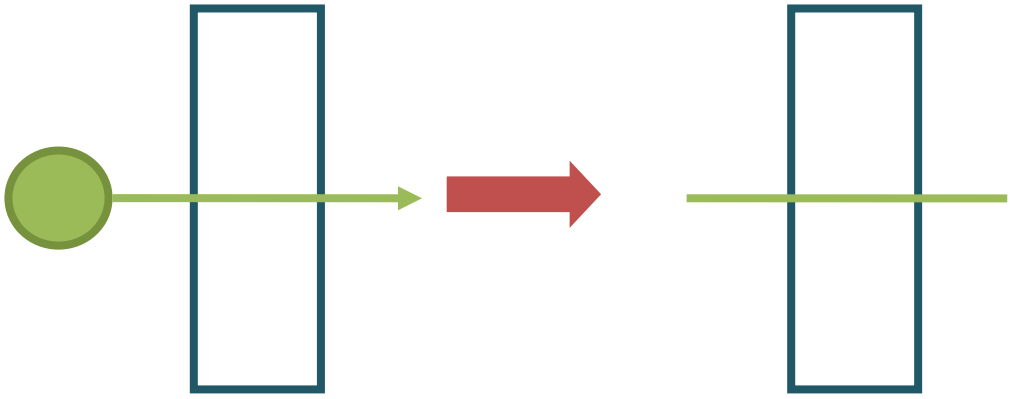
Jak již bylo zmíněno v Kapitole 3.1.2, výpočet probíhá pomocí impulsů. Pro výpočet je nutné znát jednotkový vektor ve směru kolize, rychlosti obou těles, jejich hmotnosti a pružnost. V našem řešení jsou všechny tyto hodnoty uvedeny v základní struktuře - konkrétně v prvcích *Body* a *Shape*.

Poté již při výpočtu kolize stačí dosadit tyto hodnoty do vzorce pro výpočet impulsu **(4)** a následně vypočítaný impuls použít pro výpočet nových rychlostí pro tato tělesa **(5).**

Pokud chceme tímto postupem řešit kolize, kde jeden z objektů je statický, například zem, musíme mu přiřadit velmi velkou hmotnost. V našem řešení se statickým objektům přiřadí maximální hmotnost v daném rozsahu již při vytvoření. Z tohoto důvodu při výpočtu není nutné zjišťovat, o jaký typ objektu se jedná.

4.1.3 Tunelování

Detekci tunelování do velké míry usnadnil fakt, že jediný volně pohybující se objekt, kterého se tunelování týká, je kruh. Proto nebylo potřeba vytvářet polygony, které obsáhnou celou plochu, kudy se těleso pohybuje. Ale postačilo vytvořit úsečku ze středu kruhu před simulačním krokem, do středu po provedení kroku (Obrázek 15 - Detekce tunelování pomocí úsečky) a následně porovnat nejkratší vzdálenosti mezi touto úsečkou a ostatními objekty v simulaci s poloměr pohybující se kuličky. Pokud je vzdálenost menší jak poloměr, detekovali jsme tunelování - popřípadě průnik. Jaký z těchto dvou jevů nastal, nás již nezajímá, neboť v obou případech se vracíme v simulaci a krátíme délku simulačního kroku na polovinu.

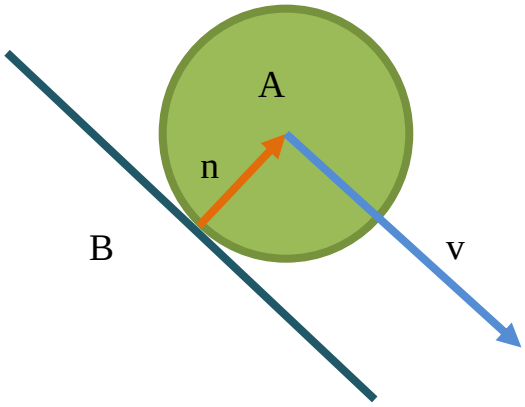


Obrázek 15 - Detekce tunelování pomocí úsečky

4.2 Kutálení

Většina důležitých poznatků o kutálení již byla popsána v Kapitole 83.2. Proto již stačí pouze doplnit několik informací o tom, jak je v našem řešení detekován tento jev a jak je nutné upravit gravitaci, která působí na kutálející se objekt.

Detekce se provádí po vyřešení kolize a to tak, že porovnáme směr pohybu dynamického kruhu, se směrem kolize mezi těmito dvěma objekty. Pokud jsou tyto směry na sebe kolmé, detekovali jsme kutálení.



Obrázek 16 - Detekce kutálení

$$\frac{n_x}{n_y} + \frac{v_y}{v_x} \leq epsilon \quad (11)$$

Dle tohoto vzorce, kde epsilon vyjadřuje toleranci, můžeme zjistit, zda jsou tyto dva vektory dostatečně kolmé, abychom mohli prohlásit, že se jedná o kutálení. Následně upravíme kolizní vektor na jednotkový tvar a uložíme si jej jako zem pro danou kouli. Pomocí tohoto jednotkového vektoru následně při výpočtu nové pozice a rychlosti koule upravíme gravitační zrychlení jako:

$$Newg = g - n \cdot (g \cdot n) \quad (12)$$

Toto *Newg* vyjadřuje gravitaci, která působí na kouli ve směru kutálení.

4.3 Uspání objektů

Z počátku jsem neměl obavy, jak vyřešit uspávání dynamické kuličky, ale již při první zkoušce bylo jasné, že jednoduchost se vytratila. Problém nastal, když při kutálení po šikmé ploše vzhůru kulička dosáhla bodu, kde se její hybnost přiblížila nule a přešla do režimu spánku. Jelikož na kuličku nepůsobila gravitační síla, kulička se již ze šikmé plochy neskutálela a zůstala zaseklá.

V tu chvíli bylo jasné, že u spícího objektu nelze gravitaci ignorovat. Proto jsem gravitační sílu nechal působit, ale jako akumulátor hybnosti. Jinak řečeno - gravitace působí na rychlost objektu, ale objekt se nepohne, dokud tato rychlost nepřekročí stanovenou mez pro „probuzení“. Tímto způsobem nenastává obrovský počet kolizí, ale pouze minimální, který pouze v malé míře zatěžuje počítač. Další výhodou je, že můžeme vynechat složitou detekci začátku a konce pro uspání, respektive probuzení. Neboť pro detekci stačí pouze kontrolovat rychlost dynamických těles po kolizi a probuzení pak probíhá při aplikaci gravitační síly na objekt, kdy gravitace způsobí dostatečný nárůst rychlosti, pro překročení stanovené meze spánku.

5 Implementace

Tato kapitola bude pojednávat o samotné implementaci hry a řešení problémů, které při ní nastaly. Implementace proběhla převážně v prostředí Visual Studio 2012 za použití programovacího jazyka C++. Pro zobrazení hry Pinball je dále použita knihovna OpenGL. Knihovna i hra je naimplementována tak, aby jí bylo možno použít v prostředí Windows i Linux.

5.1 Struktura aplikace

Jak již bylo řečeno - jedná se o hru Pinball využívající knihovnu, která byla též zadáním této práce. Samotná knihovna je vytvořena pomocí sedmi tříd jazyka C++. Při čemž jedna třída reprezentuje časovač, který není zcela nutné použít, neboť není součástí žádné jiné třídy, ale je využit až při vytváření samotné hry Pinball. Šest zbývajících tříd je možné rozdělit na dvě stejně velké skupiny, kde se jedna stará o objekty a chod simulace, a druhá se zabývá kolizemi.

5.1.1 Popis tříd

Pro lepší orientaci v aplikaci popíši velmi jednoduše hlavní třídy a jejich nejdůležitější funkce.

World – Řídící třída obsahuje gravitační zrychlení, počet simulačních kroků za sekundu, vektor těles, které do simulace vstupují, a kolize v posledním simulačním kroku. Nejdůležitější funkce této třídy jsou:

- `void AddBody(Body* b)` – Přidání nového tělesa do simulace.
- `void DeleteBody(Body* b)` – Odstranění tělesa ze simulace.
- `void Step(void)` – Provede simulační krok.
- `std::vector<Body*> MakeStep(float t)` – Výpočet nových pozic a rychlostí pro tělesa v simulaci pro čas t. Vrací vektor těles, které obsahují nově vypočítané hodnoty.

Body – Třída reprezentující tělesa, která se účastní simulace. Nese informace o typu tělesa, pozici, aktuální rotaci tělesa, rychlosti, úhlové rychlosti, hmotnosti tělesa a těžiště. Mimo tyto informace obsahuje také indikaci kutálení, uspání tělesa, existenci kloubu a ukazatel na informace o tvaru tělesa. Pro tuto třídu jsou nejčastěji používány funkce *Get*, které vrací informace o daném tělese, popřípadě *Set*, které informace nastaví, respektive upraví.

- `Body* Copy(void)` – Vrátí kopii tělesa
- `void Transfer(Body* b)` – Výměna informací mezi dvěma tělesy.

Shape – Třída nesoucí údaje o tvaru a typu tělesa. Obsahuje informace - o jaký tvar se jedná (kruh, polygon, hrany), vektor bodů, střed objektu, elasticitu, hustotu, koeficient tření a v případě kruhu i poloměr.

- `Shape(std::vector<Vec> vert, const bool poly)` – Konstruktor pro polygon a hrany. Hodnota `poly` ovlivňuje, jestli se jedná o polygon nebo hrany.
- `Shape(const float _radius, const Vec vertex)` – Konstruktor pro vytvoření kruhu.
- `float CalculateMass(void)` – Vypočítá hmotnost objektu v závislosti na hustotě objektu.
- `void CalculateCenter(void)` – Vypočítá střed objektu.
- `void MakePolygon(std::vector<Vec> vert)` – Prověří, jestli jsou body polygonu správně zadány a zda je polygon konvexní.

PreCollisionCheck – Do této třídy vstupuje dvojice vektoru těles. Jedna před provedením simulačního kroku a druhá s novými pozicemi a rychlostmi těles po simulačním kroku. Funkcí této třídy je zjistit, jestli v daném simulačním kroku nenastalo tunelování a nalezení kolizí. K tomu slouží několik funkcí:

- `bool TunelingCheck(void)` – Kontrola tunelování.
- `bool PenetrationCheck(void)` – Kontrola průniků a hledání kolizí.
- `bool CircleEdges(Body* circle, Body* edge)` – Detekce kolize mezi kruhem a hranou.
- `bool CircleCircle(Body* circle1, Body* circle2)` – Detekce kolize mezi dvěma kruhy.
- `bool CirclePolygon(Body* circle, Body* polygon)` – Detekce kolize mezi kruhem a polygonem.
- `bool CircleTwoPoints(Body* circle, Body* collisionBody, const Vec A, const Vec B)` – Detekce kolize mezi kruhem a úsečkou s body A, B.

Collision – Obdrží seznam detekovaných kolizí a vyřeší je. Také po vyřešení kolizí ověří, zda některá tělesa nesplňují podmínky pro uspání či kutálení.

- `void Solve(void)` – Vyřešení nalezených kolizí
- `void CollisionResult(Body* circle, Body* secondBody, const Vec contactPoint, const Vec p1, const Vec p2)` – Vyřešení jedné kolize

FoundCollision – Slouží k uložení informací, které potřebujeme pro vyřešení kolize - jaké objekty se srazily, v jakém bodě došlo ke srážce a v případě, že jedním objektem je hrana, uložíme i body této hrany.

5.1.2 Matematické funkce

Hlavičkový soubor, který obsahuje používané matematické funkce, je nedílnou součástí knihovny. Kromě matematických funkcí obsahuje také definici vektoru, který prostupuje celou simulací a s jeho

pomocí jsou ukládány například pozice, rychlosti, body udávající tvar objektů. Nechybí také definice operací nad vektory a to základní, jako sčítání, odčítání, tak i složitější, například skalární součin nebo rotace bodu okolo středu o daný úhel.

Tento hlavičkový soubor mimo vektorů obsahuje i funkce na zjištění vzdálenosti dvou úseček, průniků mezi kruhem a úsečkou a vzdálenosti bodu od úsečky, bez kterých by naše detekce tunelování a kolizí nefungovala.

5.2 Časovač

Časovač, v anglické literatuře znám pod názvem *timer*, je nedílnou součástí fyzikálních simulací a v našem případě ho bylo potřeba vytvořit dvakrát, jednou pro Windows a podruhé pro prostřední Linux. Oba tyto časovače pracují na stejném principu. Při inicializaci si uložíme aktuální čas a následně, pokud chceme zjistit, jaký čas uplynul od počátku simulace, odečteme aktuální čas od času při inicializaci. V prostředí Windows je použita standartní knihovna *windows.h*. Ta nám umožnuje detekci aktuálního času s přesností až na nanosekundy, i když taková přesnost není nutností. Pro Linux je použita knihovna *sys/time.h,* u které jsem zvolil přesnost na milisekundy. Taková přesnost je pro naše řešení dostatečná.

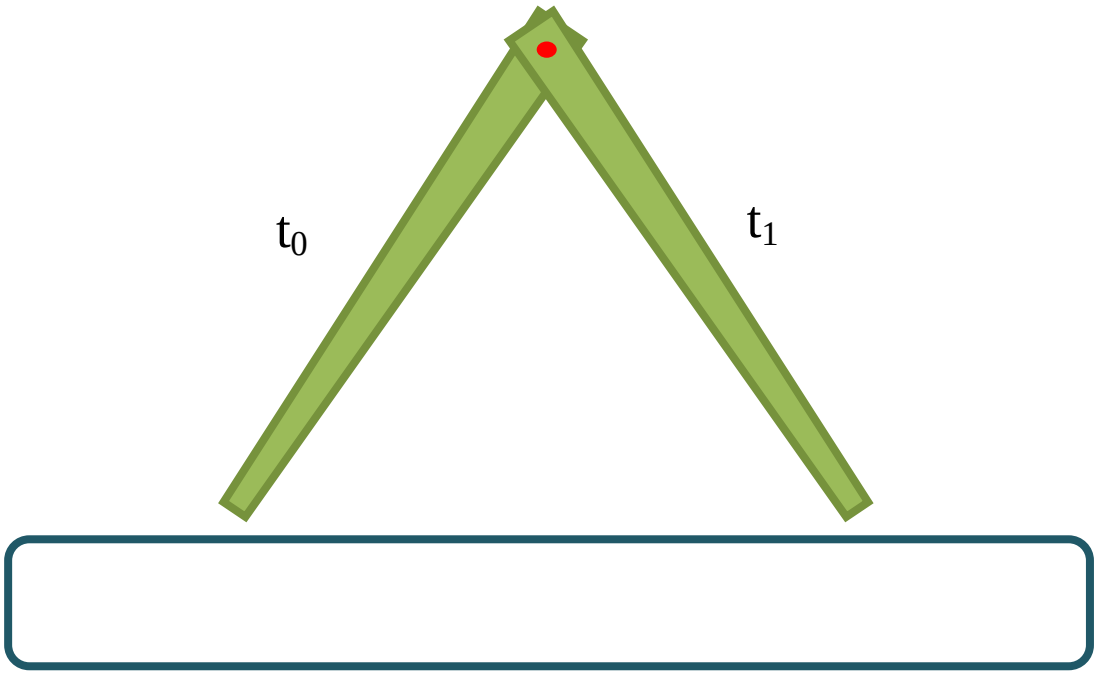
Pro vytváření hry pomocí naší knihovny není nutné tento časovač využít. Jak již bylo řečeno, je to kvůli tomu, že není součástí žádné další třídy v knihovně. Takto oddělený časovač je zvolen zejména kvůli možnosti zpomalení nebo zrychlení simulace. Ač jsem tento jev nepoužil, je možné, že někdo jiný bude chtít vytvořit zpomalenou či zrychlenou simulaci. V takovém případě by s vestavěným časovačem mohl být problém něco takového vytvořit. Ale při odděleném časovači si sami zvolíme, kdy provedeme simulační krok. Proto není problém mít dvakrát větší prodlevu mezi jednotlivými kroky, než je nastavená délka simulačního kroku, a docílit tak zpomalení simulace.

5.3 Stanovení mezí

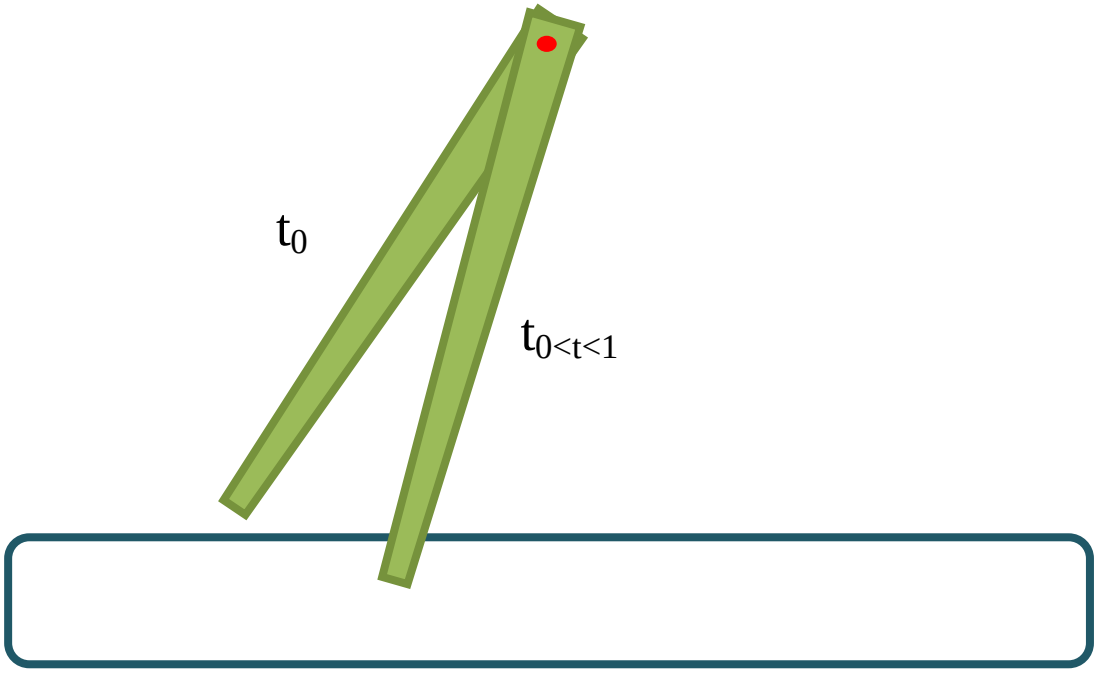
Jedná se o hodnoty, díky nimž lze výrazně snížit zátěž procesoru za cenu malých výkyvů ve stabilitě a věrohodnosti simulace. Jelikož je stabilita pro simulaci důležitá, musíme hodnoty volit s rozvahou, aby utrpěla co nejméně.

V knihovně lze nastavit pět různých mezí. Těchto pět hodnot můžeme rozdělit do dvou skupin a to podle toho, zda jsou jejich hodnoty odvozené z průběhu simulace nebo za všech okolností stejné.

Jelikož naše simulace používá proměnné typu *float*, tzn. s plavoucí desetinou čárkou, tak je dobré určit mez, která nám udává hodnotu tak malou, že jí můžeme zanedbat a nahradit nulou. Druhou mezí, která je vždy stejná, bude maximální rotace v jednom simulačním kroku. Tato mez je zavedena kvůli detekci kolizí a tunelování. V knihovně bylo nutné toto zavést, neboť objekty, které se otáčejí okolo kloubu velkou rychlostí, často unikaly detekci tunelování.



Obrázek 17 - Tunelování při rotaci



Obrázek 18 - Odstranění tunelování, omezením maximální rotace

Z obrázků výše je zřejmé, že jednoduchým omezením maximální rotace lze předejít velkým problémům, které může způsobit tunelování.

Třetí a poslední neměnná mezní hodnota již byla zmíněna. Jedná se o mez použitou při detekci kutálení (vzorec **(11)**, Kapitola 4.2). U nastavování této krajní hodnoty bylo nutné důkladně vyzkoušet, při jaké výši je neúnosně ovlivněna věrohodnost simulace a kdy výpočetní zátěž. A najít střední cestu, kde je simulace stále dostatečně věrohodná a zároveň dosáhneme znatelného snížení zátěže.

Druhá skupina mezí je složitější. Používají se pro detekci kolizí a režimu spánku. Složitější jsou, protože nelze stanovit jednotné hodnoty pro všechny případy simulace. Důvod je jednoduchý. Pokud simulace bude probíhat nad objekty, které mají velikost do desítek *p* (pixelů) a gravitační zrychlení bude mít velikost do stovky *p/s* (pixelů za sekundu), musíme hodnoty mezí nastavit v řádech jednotek *p*. Ale pokud by naše objekty dosahovaly velikosti stovek *p* a gravitační zrychlení tisíce *p/s*, budeme muset meze pro kolize a režim spánku pozměnit na desítky *p*.

Pro vyřešení takového problému se nabízí několik řešení. Můžeme volit hodnoty individuálně pro každý objekt podle jeho velikosti nebo nalézt největší, popřípadě nejmenší objekt, a zvolit meze podle něho. Dalším řešením je volit hodnoty podle gravitačního zrychlení, které je zvoleno i pro tuto knihovnu. Ať už si vybereme jakýkoliv ze způsobů, musíme ještě vyřešit, jaký poměr velikostí objektů, popřípadě gravitačního zrychlení k velikosti mezí, zvolíme. Opět zde proti sobě na vahách máme zhoršenou stabilitu a výpočetní zátěž. Osobně jsem stylem pokus omyl zvolil pro detekci kolizí 1/500 a u režimu spánku 1/40 gravitačního zrychlení. Tyto meze se při testování osvědčily, jak z pohledu výpočetní zátěže, tak věrohodnosti výsledné simulace.

5.4 Instance World

Nejprve jsem do struktury *World* chtěl zakomponovat, jak velikost okna, tak velikost simulovaného světa a následně striktně omezit poměr okna aplikace. Teprve po několika neúspěšných verzích hry jsem se rozhodl, že toto řešení není vůbec přívětivé. Neboť pracovat s tím, že poměr okna je závislý na simulovaném světě, způsobuje značné problémy, zvláště, pokud se rozhodneme v okně mít i jiné věci, než samotnou simulaci, například menu. Proto nakonec velikost simulovaného světa ani poměr či velikost okna není obsažen v naší knihovně. Všechna tato omezení proto přenechávám uživateli, který se s pomocí této knihovny rozhodne nějakou hru vytvořit.

Následně při inicializaci instance, ve které bude probíhat simulace, je potřeba pouze nastavit gravitační zrychlení a počet simulačních kroků za sekundu. Dále je také nutné před spuštěním simulace inicializovat časovač. Taková inicializace na začátku může vypadat například takto:

```
time = new Timer();
world = new World(Vec(0.0f, 980.0f));
world->SetStepsPerSec(100);
```

5.5 Vytváření objektů

Vytváření objektů je rozděleno do tří částí. V první vytvoříme tvar. Při vytváření tvaru objektu nejprve zvolíme, o jaký typ se bude jednat. Na výběr máme ze tří typů: *circle*, *polygon* a *edges*. Dále je určen jedním či více body. U kruhu jde o středový bod a jedná se tedy pouze o jeden, ale je nutné ještě zadat poloměr. V případě polygonu musíme zadat nejméně tři různé body, v opačném případě se tento polygon nevytvoří. Pro hrany platí, že je nutné zadat nejméně dva různé body, ale maximální počet není omezen a může se jednat i o body stejné, ty se následně spojí v pořadí, ve kterém jsou zadány. Při vytváření tvarů můžeme také nastavit pružnost, hustotu a hodnotu tření u daného objektu. Pokud tyto hodnoty nevyplníme, nastaví se na přednastavené.

Druhý krok zahrnuje vytvoření tělesa, u kterého musíme povinně nastavit typ a také tvar, jehož vytvoření je popsáno výše. Na výběr máme ze tří možných typů těles: statické, dynamické nebo kinetické. Při vytváření se všechny ostatní informace o tělese přednastaví. Pokud tedy chceme změnit pozici, rychlost, hmotnost atd., je třeba tak učinit až po vytvoření tělesa.

Posledním krokem je přidat vytvořené těleso do instance třídy *World*. To učiníme pomocí funkce *AddBody*, která je popsána v Kapitole 5.1.1.

Vytváření objektů využívá tři třídy, které se starají o objekty a průběh simulace (*World*, *Body* a *Shape*). Po vytvoření lze nalézt nový objekt ve vektoru těles uvnitř instance třídy *World*. Jednotlivé objekty lze z tohoto vektoru také v průběhu simulace odstranit.

Příklad vytvoření statického polygonu a následné jeho posunutí a otočení o uhel 45°:

```
std::vector<Vec> vert;
vert.push_back(Vec(0,0));
vert.push_back(Vec(100,0));
vert.push_back(Vec(0,50));
vert.push_back(Vec(100,50));

Body *b = new Body(BodyType::staticBody, new Shape(vert, true));
b->SetPosition(Vec(250,400));
b->SetAngle(45);
world->AddBody(b);
```

Následně lze tento objekt odstranit takto:

```
world->DeleteBody(b);
```

5.6 Vykreslování objektů

Vykreslování objektů probíhá po každém simulačním kroku. Pokud se detekuje tunelování nebo průnik a krátí se simulační krok, v takovém případě objekty nevykreslujeme, ale počkáme, až se dokončí celý krok a poté scénu překreslíme.

Toto překreslení celé scény (které ve většině případů začíná vymazáním plochy obrazovky barvou pozadí) však může nějakou dobu trvat a v této době by uživatel viděl problikávání způsobené postupnou změnou barev pixelů v barvovém bufferu (color bufferu).

Tomuto problikávání lze zabránit tak, že se místo jednoho barvového bufferu použijí buffery dva. Do jednoho z těchto bufferů se vykresluje (tzv. zadní buffer – back buffer) a druhý je zobrazený uživateli (tzv. přední buffer – front buffer). Po dokončení vykreslování do prvního bufferu se tento zobrazí uživateli a role bufferů se obrátí – tomuto využití dvou barvových bufferů se říká double-buffering. Důležité je, že při výměně rolí obou bufferů není zapotřebí provádět žádné přesuny dat v obrazové paměti, protože použití minimálně dvou barvových bufferů je možné na všech grafických akcelerátorech (vyrobených v tomto století).

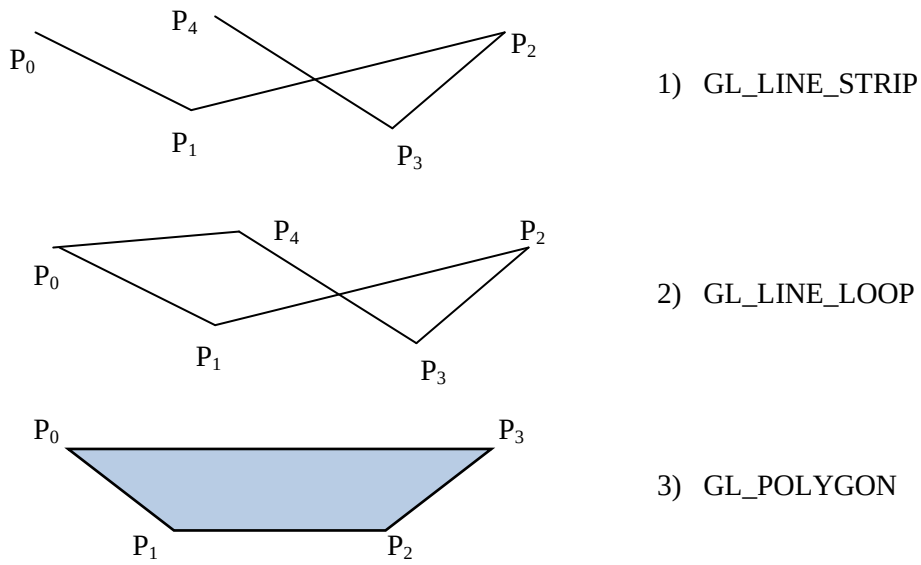
Pro úspěšné použití double-bufferingu je zapotřebí při inicializaci grafického kontextu OpenGL (OpenGL rendering context) specifikovat, že se mají vytvořit dva barvové buffery:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

Po úspěšném vytvoření dvou barvových bufferů se na konci sekvence příkazů pro vykreslení prostorové scény musí zavolat funkce pro prohození rolí obou barvových bufferů [12]:

```
glutSwapBuffers();
```

Před samotným zobrazením scény jí ovšem musíme zaplnit. Zaplnění je prováděno pomocí tří základních grafických primitiv OpenGL:



Obrázek 19 - Používaná grafická primitiva

Každý objekt používá na své vykreslení nějaké z těchto grafických primitiv. Hrany (*edges*) používají výhradně GL_LINE_STRIP. Pro polygon můžeme použít jak GL_LINE_LOOP tak GL_POLYGON, podle toho jestli chceme, aby byl objekt vyplněn.

Na kruh musíme použít jeden z algoritmů, který se na vykreslování kružnice používá. Jako nejlepší jsem shledal „Midpoint“ algoritmus, který vykresluje kružnici jako N-úhelník, jehož body vykreslíme opět pomocí GL_LINE_LOOP nebo GL_POLYGON. Následný kód pro vykreslení kruhu pomocí OpenGL může vypadat takto:

```
float c = cosf(2 * PI / numberOfSegments);
float s = sinf(2 * PI / numberOfSegments);
float t = 0;

Vec r = Vec(0.0f,circle->GetShape()->GetRadius());
Vec xy = circle->GetCenter() + circle->GetPosition();

glColor3f(0.2, 0.2, 0.2);

glBegin(GL_POLYGON);

for(int i = 0; i < numberOfSegments; i++)
{
    glVertex3f(r.x + xy.x, r.y + xy.y, 0.0);
    t = r.x;
    r.x = c * r.x - s * r.y;
    r.y = s * t + c * r.y;
}
glEnd();
```

5.7 Průběh řešení kolizí

Řešení je rozděleno do tří částí. První část je zaměřena na odstranění tunelování, neboť simulace je postavena na principu *posterior*. Druhá část již detekuje samotné kolize, popřípadě průnik. Takže první dvě části jsou zaměřeny na detekce kolizí, namísto toho třetí část vyřeší detekované kolize, které jsme našli v druhé části.

5.7.1 Průběh detekce kolizí a tunelování

Při detekci tunelování musíme nejprve ze všech dynamických objektů vytvořit adekvátní náhrady, to jsou v našem případě úsečky ze středu kuličky před a po aktuálním simulačním kroku (Obrázek 15 - Detekce tunelování pomocí úsečky). Pro všechny takto vytvořené úsečky nyní musíme změřit nejkratší vzdálenosti od ostatních objektů, jak dynamických, kde hledáme tuto vzdálenost mezi nově vytvořenými úsečkami, tak statických a kinetických. Pokud někde zjistíme, že vzdálenost je menší než poloměr pohybující se kuličky nebo že se v nějakém místě protínají, vracíme se v simulaci a zmenšujeme velikost simulačního kroku. V opačném případě konstatujeme, že se o tunelování nejedná a přistoupíme k detekci kolizí.

Detekce kolizí se provádí mezi dynamickým objektem a všemi ostatními v simulaci. Použitý princip je popsán v Kapitole: 0. Pokud při detekci nalezneme kolizi mezi dvěma objekty, uložíme si všechny potřebné údaje (objekty mezi kterými nastala, bod ve kterém nastala, počáteční a koncový

bod hrany, pro případ, že by se jednalo o kutálení), bylo možné kolizi v případě úspěšné detekce bez průniků vyřešit.

Detekce tunelování a kolizí musí být spolehlivá a přesná. V případě, že bychom kolizi detekovali špatně nebo vůbec, mohlo by to mít fatální dopad na průběh simulace, neboť při řešení výsledku kolize již nejsme schopni z uložených údajů poznat, zda výsledek je správný či nikoli.

5.7.2 Výsledek kolize

Pokud po detekci máme uložené nějaké kolize, přecházíme na jejich vyřešení. Nejprve však projdeme všechny uložené a podíváme se, jestli jsme nějakou nedetekovali dvakrát. To se může stát v případě, že kulička narazí např. na roh polygonu. Pokud nalezneme takovéto stejné kolize, vždy ponecháme pouze jednu z nich. Následně zbytek kolizí provedeme a po vyřešení všech kontrolujeme, zda nějaké dynamické objekty nesplňují podmínky pro přechod do stavu kutálení či spánku. Pokud ano, tak jim daný stav přiřadíme a vracíme upravená tělesa nazpět, kde dané objekty zobrazíme a následně u nich provedeme posun o další simulační krok, odkud opět přecházíme na detekci tunelování.

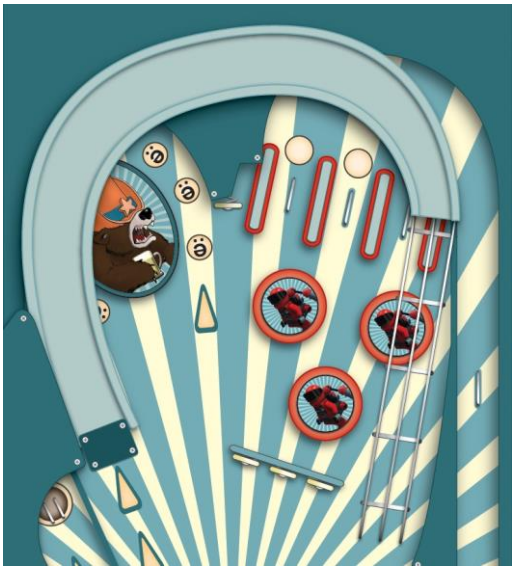
6 Možné rozšíření

Jelikož je knihovna zaměřena na hru Pinball, je zde velký prostor pro rozšíření:

- *Zohlednění rotace při kolizi* – Náročnost přidání tohoto rozšíření by nebyla příliš veliká, stačilo by pozměnit funkci `CollisionResult`. Ale jelikož v simulaci vystupuje jako dynamické těleso pouze kruh, tak nebylo nezbytně nutné tento jev implementovat, neboť výsledek na simulaci by byl neznatelný. Ovšem v případě, že budeme uvažovat o některých níže uvedených rozšíření, museli bychom přidat i toto.

- *Polygon jako dynamické těleso* – Takovéto rozšíření v aktuálním stavu knihovny odstraní všechny výhody, které jsme měli ze zaměření na hru Pinball. Proto by jeho náročnost byla opravdu velká, dalo by se říct, že většina funkcí by se musela předělat a mnoho nových přidat. Proto si nemyslím, že je dobré knihovnu se zaměřením předělávat na knihovnu obecnou, nehledě na to, že takových knihoven existuje mnoho. Z toho důvodu bych takové rozšíření nedělal a spíše se zaměřil na vylepšení stávající knihovny, kde jsou již optimalizované operace pro dynamický kruh.

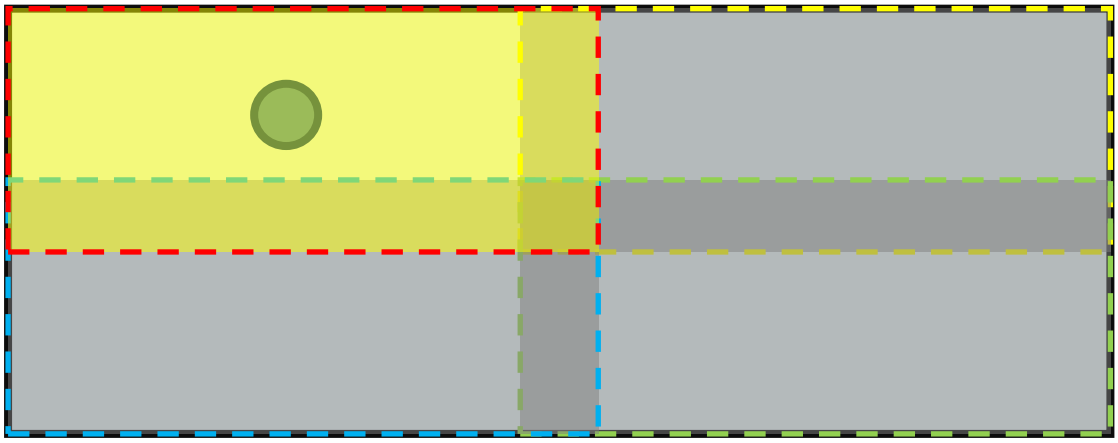
- *Více vrstvá simulace* – Do jisté míry se jedná o náhražku 3D. Jednotlivé objekty by obsahovaly informaci, o jakou vrstvu se jedná a následná detekce kolizí by probíhala pouze mezi stejnými vrstvami. Pro přechody mezi vrstvami by bylo nutné vytvořit mosty. Takové rozšíření by mělo využití i u hry Pinball (Obrázek 20 – Více vrstvá simulace).



Obrázek 20 – Více vrstvá simulace

- *Přidání třetí dimenze* – Rozšíření z 2D na 3D by nemuselo být tak obtížné, jak se na první pohled zdá. Dle mého názoru je knihovna naimplementována tak, že v případě přechodu na 3D by bylo nutné razantně pozměnit pouze detekci kolizí, detekci tunelování a výsledek kolize. Ostatní části a celkový princip fungování simulace by naznaly pouze mírné změny.

- *Rozdělení simulovaného světa* – Pokud bychom se nadále drželi zaměření na simulace, kde se bude pohybovat pouze jeden dynamický kruh, bylo by dobré zvážit rozdělení simulačního světa. Takové rozdělení bych si představoval jako uspání částí, kde se dynamické těleso právě nenachází. Mohli bychom tak vytvořit několik malých světů, kde by bylo možné detekovat kolize a tunelování pouze v jednom z nich a jen s hrstkou statických objektů. Kulička by byla vždy obsažena celá alespoň v jedné části a tak by ostatní mohly být nečinné (Obrázek 21 - Rozdělení simulovaného světa).



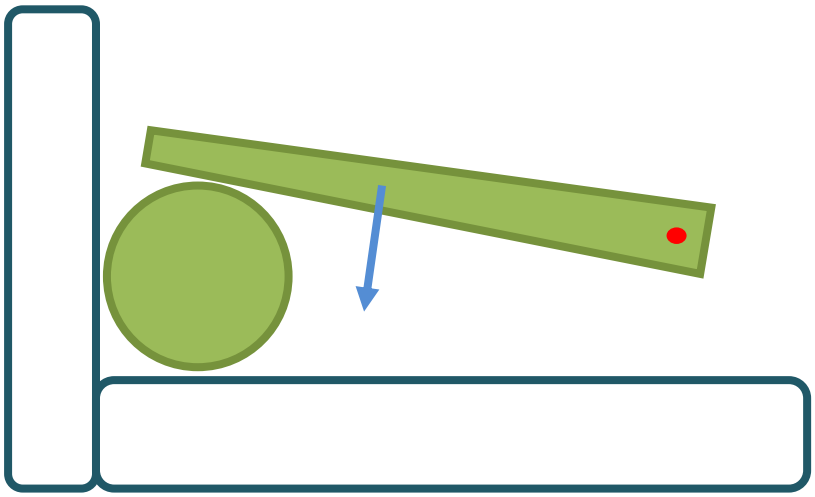
Obrázek 21 - Rozdělení simulovaného světa

- *Textury* – Myslím, že zakomponování jednoduché knihovny na načítání textur, by se této knihovně velice hodilo. Při průběžných verzích jsem zkoušel několik způsobů načítání textur, jedním z nich bylo využití knihovny SOIL (Simple OpenGL Image Library). Tu jsem nakonec zprovoznil pouze přidáním všech zdrojových souborů do projektu. Následně však nastal problém při překladu v prostředí Linux, který nepřeložil projekt s kombinovanými C a C++ zdrojovými kódy. To vedlo k tomu, že jsem knihovnu SOIL z řešení vypustil, ale myslím si, že napsání vlastního načítání textur by bylo přínosem pro celý projekt.

- *Nevyřešený problém s páčkami* – Jediný problém, který jsem řešil a nedokázal najít řešení. Naštěstí zrovna v Pinballu takový jev nenastane a tak nakonec ve výsledné aplikaci tento problém není a proto jsem tento jev neřešil tak dlouho, jak by si zasloužil.

  Jedná se o naražení dynamické kuličky na statický objekt páčkou. Jelikož se může stát, že kulička při kolizi s páčkou nebude mít kam uskočit a zároveň kinetická páčka má svojí úhlovou rychlost, která se nárazem nesníží, tak se simulace zasekne (Obrázek 22 - Zaseknutí simulace).

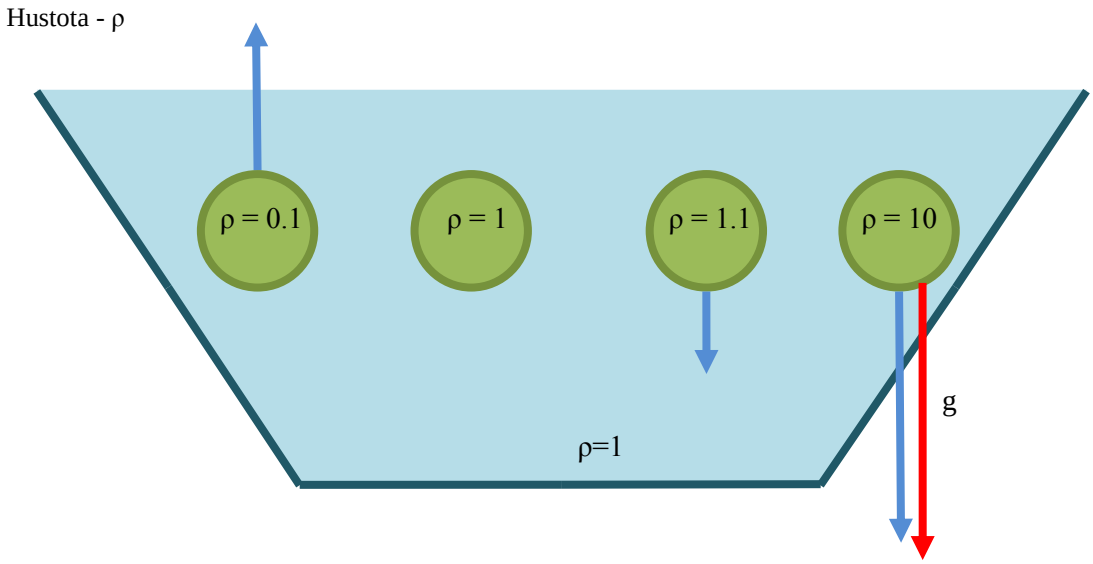
  Jediné řešení, které mě napadlo, ale neimplementoval ho, je pozastavení páčky do konce kroku, pokud s ní nastala kolize. Nejsem si jistý, jestli by takové řešení fungovalo, protože i tou jedinou srážkou na začátku každého kroku, by se kuličce předala značná rychlost a kulička by narážela do všeho kolem sebe. Vzhledem k tomu, že má téměř nulové vzdálenosti k ostatním objektům, tak by se v každém kroku detekovalo obrovské množství kolizí, které by nakonec zapříčinily, že by se simulace s velkou pravděpodobností také zasekla.



Obrázek 22 - Zaseknutí simulace

- *Speciální objekty/místa* – Jednalo by se o objekty, se kterými by se detekovaly kolize, ale už by nebyly provedené. Výsledek, co se má stát při takové kolizi, by si definoval sám uživatel. Opět by se takové objekty daly využít i v Pinballu, neboť některé verze obsahují tzv. černé díry, ve kterých se kulička zastaví a po chvíli je opět vystřelena do hry ve směru, který je předem určen. Dále by se takto dalo vyřešit přičítání bodů za to, když kulička svojí dráhou protne předem specifikované místo.

- *Různá prostředí* – Abych toto rozšíření přiblížil, mohl bych ho nazvat vodní prostředí, ale jelikož by bylo možné využít více než jen vodu, název různá prostředí se hodí více. Jednalo by se o zavedení rozdílných oblastí v rámci simulovaného světa, kde by se tělesa různě pohybovala podle své hustoty. Objekty s velkou hustotou by takové prostředí ovlivnilo jen nepatrně, naopak pokud má objekt hustotu podobnou nebo nižší, než je hustota prostředí, pak by klesal pomalu, nebo by dokonce plaval (Obrázek 23 - Různá prostředí). Do takovýchto prostředí by se dal zařadit i vzduch, ve kterém objekty s opravdu nízkou hustotou budou létat.



Obrázek 23 - Různá prostředí

7 Závěr

Průběh vytváření aplikace lze rozdělit do čtyř fází. První obsahovala nastudování existujících fyzikálních knihoven, výsledkem byla základní struktura *World, Body, Shape*. Druhá část byla implementace této struktury a seznámení s prací v OpenGL. Na konci této fáze bylo možné vytvořit objekty a následně tyto objekty zobrazit. Třetí fáze, která mi zabrala nejvíce času - jednalo se o rozpohybování dynamických objektů a zpracování detekce a výsledku kolizí. Čtvrtá fáze obsahovala detekci tunelování a optimalizaci detekce kolizí na výslednou hru.

Největší problém při vytváření knihovny jednoznačně zapříčinila detekce kolizí. Zejména pak v první verzi, když jsem ještě polygon při detekci nerozkládal na jednotlivé hrany. To ovšem nebyla jediná část, kterou jsem mnohokrát předělával a kde mi tyto průběžné verze zabraly více času, než ta konečná. Jednoznačně jsem si tak ověřil, že je lepší věnovat více času návrhu, a tím si ušetřit čas při následné zbytečně složité implementaci a přepisování kódu. Toto bych považoval za největší přínos této práce.

Výsledná knihovna i hra jsou multiplatformní a běží na operačních systémech Windows i Linux, což ještě potenciálně rozšiřuje uživatelskou základnu. Samotné použití knihovny je velmi jednoduché a následné vytvoření hry proto zabere už opravdu málo času.

Možnost dalšího vývoje se může ubírat několika směry dle rozšíření, které jsou popsány v Kapitole 6. Mezi hlavní, které bych si uměl představit, i jako zadání diplomových prací, bych zařadil rozšíření na 3D nebo zachování dvou dimenzí se zavedením vícevrstvé simulace a podpory dynamických polygonů. Dále bych si v kombinaci s těmito variantami dokázal představit i podporu různých prostředí, rozdělení simulovaného světa a zohlednění rotace při kolizi.

Literatura

[1] TÖPFER, Pavel. Programování II. *Kabinet software a výuky informatiky* [online]. 2014 [cit. 2014-05-07]. Dostupné z: http://ksvi.mff.cuni.cz/~topfer/ppt/P-7.pdf

[2] Stručná historie pinballu. *Stolni Fotbalky* [online]. [cit. 2014-05-07]. Dostupné z: http://stolnifotbalky.wz.cz/www.jz-spol.cz/htm/historie_flipper.htm

[3] CATTO, Erin. *Box2D: A 2D Physics Engine for Games* [online]. [cit. 2014-03-29]. Dostupné z: http://box2d.org/

[4] LEMBCKE, Scott. *Chipmunk: Game Dynamics* [online]. [cit. 2014-03-29]. Dostupné z: http://chipmunk-physics.net/

[5] POWER, Ken. Collision detection. *Glasnost: Institute of Technology, Carlow* [online]. 2011, 2013-10-16 [cit. 2014-03-29]. Dostupné z: http://glasnost.itcarlow.ie/~powerk/GeneralGraphicsNotes/CollisionDetection/CollisionDetection.html

[6] Game Technologies: Physics Tutorial 6: Collision Response - Impulse Methods *Newcastle University* [online]. [cit. 2014-03-29]. Dostupné z: http://research.ncl.ac.uk/game/mastersdegree/gametechnologies/physicscollisionresponse-impulsemethods/

[7] Collision Detection and Physics FAQ: How can I avoid missing collisions for fast moving objects. In: *Bullet: physics library* [online]. [cit. 2014-03-29]. Dostupné z: http://bulletphysics.org/mediawiki-1.5.8/index.php/Collision_Detection_and_Physics_FAQ

[8] Circle-Line Collision Detection Problem. In: *Game Development* [online]. 2010, 2012-09-24 [cit. 2014-04-03]. Dostupné z: http://gamedev.stackexchange.com/questions/18333/circle-line-collision-detection-problem

[9] BOURG, David,M. *Physis for game developers*. 1. vyd. Boston: O'Reilly, 2002. ISBN 0-596-00006-5.

[10] KIRMSE, Andrew. *Game programming gems 4*. Hingham: Charles River Media, 2004. ISBN 1-58450-295-9.

[11] BOURKE, Paul. Circles and spheres. *Paul Bourke* [online]. 1992 [cit. 2014-04-29]. Dostupné z: http://paulbourke.net/geometry/circlesphere/

[12] TIŠNOVSKÝ, Pavel. Grafická knihovna OpenGL: double-buffering. *ROOT.CZ* [online]. 2003 [cit. 2014-05-07]. Dostupné z: http://www.root.cz/clanky/opengl-13-double-buffering/

Příloha A - CD se zdrojovými kódy

Přiložené CD obsahuje následující soubory a adresáře:

windows	adresář obsahující projekt pro visual studio 2012
linux	adresář obsahující zdrojové kódy a makefile pro linux
app	adresář obsahující exe soubor a OpenGL dll knihovny
readme.txt	soubor s informacemi o překladu a spuštění aplikace
doc	elektronická verze textu bakalářské práce