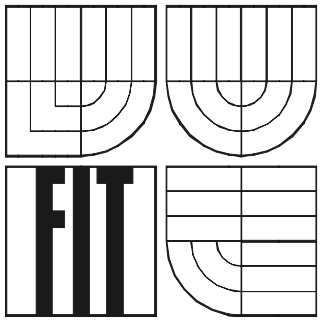
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Diplomová práce

2005 **Petr Maštera**

Zadání diplomové práce/5603/2006/xmastaG0

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2006/2007

Zadání diplomové práce

Řešitel: **Maštera Petr**
Obor: Výpočetní technika a informatika
Téma: **Simulace kolizí na základě fyzikálního modelu**
Kategorie: Počítačová grafika

Pokyny:

1. Nastudujte si teorii fyzikální srážky dvou pevných těles a kolizních algoritmů používaných v počítačové grafice.
2. Navrhněte a vytvořte programy pro výpočet těžiště a momentu setrvačnosti, které jsou nezbytné pro vyhodnocení srážky dvou pevných těles.
3. Navrhněte algoritmy řešící srážku dvou pevných těles na základě fyzikálního modelu. Algoritmy popište.
4. Algoritmy implementujte. Vytvořte jednoduchý herní engine využívající daných algoritmů.
5. Vytvořte demonstrační aplikaci "průlet tunelem".
6. Práci vystavte na internetu pod některou z open-source licencí.

Literatura:

- Open Inventor tutoriál na ROOT.CZ
- Josie Wernecke, The Inventor Mentor, Addison-Wesley Professional, 1994, ISBN: 0201624958

Při obhajobě semestrální části diplomového projektu je požadováno:

- Bez požadavků.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Pečiva Jan, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 24. ledna 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Pavel Zemčík
vedoucí ústavu

Poděkování

Chtěl bych vyjádřit poděkování vedoucímu své diplomové práce Janu Pečivovi za jeho užitečné rady při tvorbě programu a za to, že moji práci usměrňoval ke zdárnému konci.

Prohlášení

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, jsem uvedl v literatuře.

.....

Petr Maštera

Abstrakt

Diplomová práce se zabývá řešením kolizí mezi objekty scény a následným vyhodnocením těchto kolizí na základě fyzikálního modelu. Implementace všech aplikací a algoritmů je provedena v prostředí Win32 ve Visual Studiu v programovacím jazyce C++ s využitím grafické knihovny OpenGL a nadstavby Open Inventor. K práci je přiložena pomocná aplikace zabývající se výpočtem fyzikálních veličin. V demonstračních aplikacích jsou implementovány algoritmy pro detekci a vyhodnocení kolizí výbuchem, jednoduchým a fyzikálním odrazem na základě fyzikálních vzorců a vztahů. V hlavní demonstrační aplikaci „průlet tunelem“ je implementován jednoduchý herní engine. Součástí práce je diskuse objevujících se chyb s případným návrhem řešení.

Klíčová slova:

Detekce kolizí, vyhodnocování kolizí, jednoduchý odraz, jednoduché vyhodnocení kolize, fyzikální odraz, fyzikální vyhodnocení kolize, srážka pevných těles, matice momentu setrvačnosti, těžiště tělesa, Open Inventor, C++.

Abstract

This diploma thesis focuses on the collision detection between scene objects and consequent resolution of such collisions on the basis of physical model. The implementation of all the applications and algorithms is achieved in Win32 environment in Visual Studio using the programming language C++; it also employs the graphical library Open Inventor based on OpenGL. The work also includes additional application that concentrates on the calculation of physical values. The demo applications involve algorithms for detection and resolution of explosive collision by the use of a simple and physical reflection on the basis of physical formulas and relationships. The main demo application called “tunnel transit” incorporates a simple game engine. The thesis also includes a discussion over the aroused mistakes together with some suggestions how to solve them.

Key words:

Collision detection, collision resolution, simple reflection, simple collision resolution, physical reflection, physical collision resolution, collision of rigid bodies, inertia tensor, centre of mass, Open Inventor, C++.

Obsah

Úvod

Detekce kolizí objektů je základním problémem při programování grafických počítačových her. Používá se všude tam, kde potřebujete zjistit, zda jeden objekt (např. střela nebo figurka nepřítele) zasáhl druhý objekt (vaši figurku nebo kosmickou loď). S tímto problémem je úzce spjato i následné vyhodnocování kolize. Po zjištění kolize může nastat velké množství řešení, které jsou závislé na prostředí a kolidujících objektech. Například při srážce rakety a lodě může loď explodovat, při srážce míče se zdí se míč odrazí atd. Příkladů jak vyhodnocovat kolize je nespočet. Některými možnostmi se zabývá i tato práce.

Cílem diplomového projektu je naimplementovat algoritmy pro detekci a vyhodnocování kolize na základě fyzikálního modelu ve 3D prostoru a demonstrovat je na vhodných příkladech.

Cíle, které má tato práce splnit, jsou následující:

1. Seznámit se s teorií fyzikální srážky dvou pevných těles a kolizních algoritmů používaných v počítačové grafice.
2. Navrhnout a vytvořit program pro výpočet těžiště a momentu setrvačnosti, které jsou nezbytné pro vyhodnocení srážky dvou pevných těles.
3. Navrhnout algoritmy řešící srážku dvou pevných těles na základě fyzikálního modelu. Algoritmy popsat.
4. Implementovat navržené algoritmy. Vytvořit jednoduchý herní engine využívající daných algoritmů.
5. Vytvořit demonstrační aplikaci „průlet tunelem“.
6. Výsledky publikovat na internetu pod některou z open-sorce licencí.

Cílem práce je vytvořit algoritmus simulující co nejreálněji problematiku detekce a vyhodnocení kolize. K řešení problému byla použita knihovna Open Inventor a celá škála jejich objektů sloužící pro snadnou implementaci grafických aplikací. K řešení problémů spojených s pomocnou aplikací (pod názvem Veličiny) byla využita knihovna ColDet. Dále byla použita metoda billboardingu známá z programování grafických aplikací pod OpenGL. Jako vývojové prostředí pro implementaci bylo zvoleno VisualStudio.net a programovací jazyk C++.

Klíčové algoritmy projektu, které vystihují zadání projektu jsou:

- výpočet těžiště
- výpočet matice momentu setrvačnosti
- detekce kolize mezi objekty scény
- vyhodnocení kolize explozí
- vyhodnocení kolize jednoduchým odrazem
- vyhodnocení kolize na základě fyzikálního modelu

Veškeré implementované algoritmy jsou demonstrovány v aplikacích. Pro tyto aplikace bylo nutné zajistit objekty scény a jejich obsluhu, dále pak pohyb uživatele a kamery. S tím souvisí řada dalších problémů, které bylo třeba v rámci této práce vyřešit.

1. Základní pojmy

Pro tuto práci jsou důležité znalosti z oboru matematiky a fyziky [1]. Při praktickém použití fyzikálních a matematických vzorců v počítačové grafice (při algoritmizaci vzorců) nemůžeme pracovat ve spojitém prostoru, jako v rovině teoretické. Při praktickém použití musíme pracovat pouze v diskrétním prostoru, který je v počítačové grafice reprezentován kartézskou souřadnou soustavou. Proto jsou nezbytné základní znalosti geometrie, konkrétně vektorový a maticový počet a analytická geometrie [2].

Klíčovými znalosti z oboru geometrie jsou polohové vztahy základních geometrických primitiv bod, přímka, rovina, z nichž si vybrané nyní blíže popíšeme.

1.1. Vektorová rovnice přímky

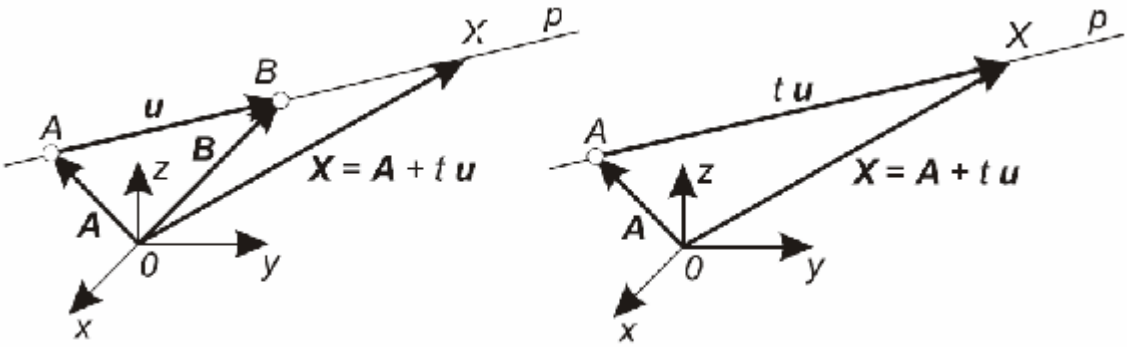
Na základě znalosti geometrie [2] uvažujme přímku p určenou dvěma různými body A, B. Směrovým vektorem přímky p rozumíme libovolný nenulový vektor, který je kolineární s vektorem $u = AB = B - A$. Pro libovolný bod X na přímce p platí:

$$X - A = t(B - A) = tu, t \in \Re \tag{1.1}$$

Tak dostáváme tzv. vektorovou rovnici přímky p:

$$X = A + tu, t \in \Re \tag{1.2}$$

V této rovnici užíváme symbolů X, A, B, které označují body, nicméně zde se jedná o polohové vektory X, A, B daných bodů (viz. obr. 1.1). Symbolem t je označen tzv. parametr. Bodu A, resp. B, přísluší parametr t = 0, resp. t = 1. Uvedená vektorová rovnice přímky je formálně stejná pro vyjádření přímek v libovolném euklidovském prostoru, tj. v E2, resp. v E3, ale i v euklidovských prostorech vyšších dimenzí.



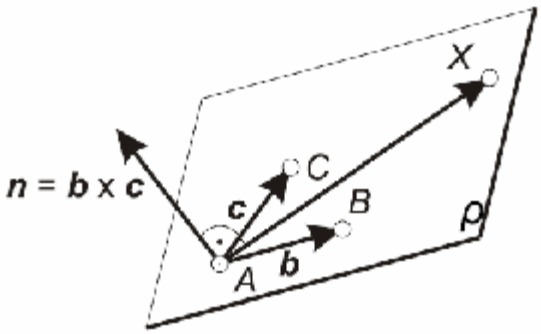
Obr. 1.1

1.2. Hessův normálový tvar rovnice roviny

Na základě poznatků z geometrie [2] můžeme rovinu popsat tak, že vektory $b = B - A$ a $c = C - A$ vynásobíme vektorově a označíme $n = b \times c$ (viz. obr. 1.2). Vektor n, tzv. normálový vektor roviny, je kolmý k nekolineárním vektorům b a c (tedy vektor n je směrovým vektorem kolmice k rovině). Hessův normálový tvar rovnice roviny ρ představuje podmínku:

$$(X - A)n = 0 \tag{1.3}$$

kde X je libovolný (obecný) bod roviny. Vycházíme z toho, že pro $X \neq A$ musí být vektory XA a n kolmé. Je-li $X = A$, pak vektor XA je nulový a daný vztah (1.3) je splněn.



Obr. 1.2

1.3. Vzájemná poloha přímky a roviny

Na základě znalostí z oblati geometrie [2] nechť je přímka p daná vektorovou rovnicí (1.2) a rovina Hessovým normálovým tvarem (1.3):

$$X = A + tu \tag{1.4}$$

$$(X - B)n = 0 \tag{1.5}$$

O vzájemné poloze přímky p a roviny lze rozhodnout na základě skalárního součinu $u \cdot n$ směrového vektoru přímky a normálového vektoru roviny, případně i vektoru "příčky" AB. Klasifikace vzájemné polohy přímky a roviny je uvedena v tabulce 1.1 ($u \neq 0, n \neq 0$).

$u \cdot n = 0$	$(B - A) \cdot n = 0$	$p \subset n$
	$(B - A) \cdot n \neq 0$	rovnoběžnost $p \not\subset n$
$u \cdot n \neq 0$		různoběžnost

Tab. 1.1

Určení průsečíku v případě různoběžné přímky a roviny provedeme pomocí výpočtu příslušného parametru t z rovnice (1.4). Je-li bod X společným bodem přímky a roviny, plyne z (1.5) a z (1.4) po jednoduché úpravě:

$$\left[\left(A-B\right)+tu\right]\cdot n=0 \tag{1.6}$$

Pro parametr t získáme lineární rovnici:

$$\left(A-B\right)\cdot n+t\cdot\left(u\cdot n\right)=0 \tag{1.7}$$

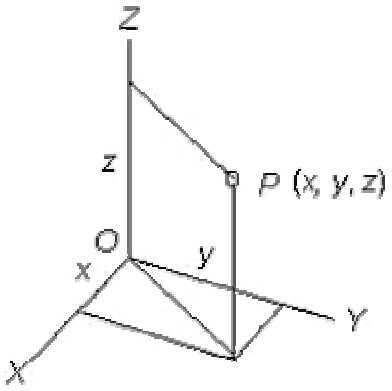
která má jediné řešení t_0, neboť skalární součin $u\cdot n\neq 0$. Hledaným průsečíkem je bod:

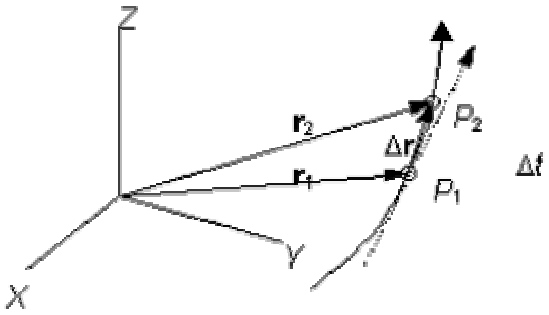
$$P=A+t_0\cdot u \tag{1.8}$$

2. Fyzika

Pro každý objekt potřebujeme znát veličiny a vztahy, které popisují chování tohoto objektu v prostoru a čase [1, 3, 4, 5]. Libovolný objekt pro zjednodušení dočasně nahradíme hmotným bodem [1, 3, 5].

2.1. Pohyb hmotného bodu





Obr. 2.1, Obr. 2.2

Pro popsání pozice hmotného bodu v kartézském souřadném systému potřebujeme znát vektor pozice r (position):

$$r = (ix, jy, kz) = (x, y, z) \tag{2.1}$$

kde i, j, k reprezentuje jednotkový vektor a x, y, z kartézské souřadnice (viz. obr 2.1). Vektorovou veličinou, která udává změnu polohy v čase, je rychlost v (velocity) (viz. obr. 2.2):

$$v = \frac{dr}{dt} = \frac{idx}{dt} + \frac{jdy}{dt} + \frac{kdz}{dt} = [\dot{x}, \dot{y}, \dot{z}] \tag{2.2}$$

Změnu rychlosti v čase naopak udává zrychlení a (acceleration):

$$a = \frac{dv}{dt} \tag{2.3}$$

Další vektorovou veličinou je síla F (force), která působí na těleso a ovlivňuje tak jeho zrychlení v závislosti na hmotnosti tělesa:

$$F = ma \tag{2.4}$$

Z této rovnice (2.4) poté odvodíme vztah pro výpočet zrychlení v závislosti na součtu všech působících sil:

$$a = \frac{\sum F}{m} \tag{2.5}$$

Pro další veličiny je třeba rozlišovat o jaký pohyb v prostoru se jedná. Obecně jsou vztahy ovlivňující těleso rozděleny na přímočarý pohyb (linear), nazývaný také translační pohyb, a na pohyb po kružnici (angular), nebo-li rotační pohyb.

2.1.1. Přímočarý pohyb

Pro přímočarý pohyb vyjadřuje hybnost míru setrvačnosti tělesa. Hybnost p (momentum) závisí na hmotnosti a rychlosti tělesa:

$$p = mv \tag{2.6}$$

Ze vzorce (2.4) a (2.6) získáme druhý Newtonův zákon, tzv. zákon zachování hybnosti:

$$F = ma = m\frac{dv}{dt} = \frac{d}{dt}(mv) = \frac{dp}{dt} \tag{2.7}$$

nebo také:

$$p = \int_{t_0}^{t} F dt + p_0 \tag{2.8}$$

$$\Delta p = p - p_0 = \int_{t_0}^{t} F dt = I \tag{2.9}$$

kde I je impuls síly působící na částici, který je stejný jako změna hybnosti částice. Hybnost částice je ovlivňována působící silou (2.8).

2.1.2. Pohyb po kružnici

Pro pohyb po kružnici potřebujeme znát moment síly, nebo-li točivý moment M (torque):

$$M = r \times F = r \times \frac{dp}{dt} = \frac{d}{dt}(r \times p) \tag{2.10}$$

kde F je působící síla a r je rameno této síly, tedy polohový vektor částice vůči středu otáčení. Vzorec (2.10) můžeme také zapsat jako:

$$M = \frac{dH}{dt} \tag{2.11}$$

z čehož plyne:

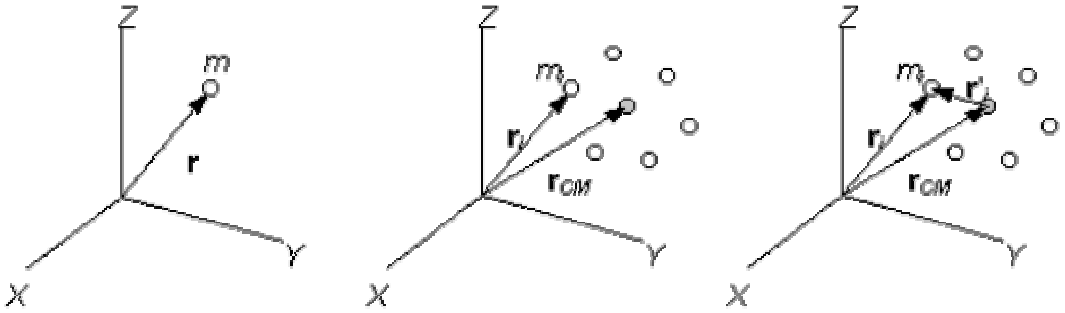
$$H = r \times p \tag{2.12}$$

kde H je úhlový moment (angular momentum).

2.2. Těžiště tělesa

Pokud se chceme přiblížit skutečnosti, nesmíme uvažovat objekt jako hmotný bod, ale jako soustavu hmotných bodů, resp. pevné těleso [1, 3, 4, 6, 7, 8]. Jelikož se při práci na počítači pohybujeme v diskrétním prostoru, nemůžeme dosáhnout reprezentace objektu jako pevného spojitého tělesa a tudíž musíme všechny rovnice řešit diskrétně. Právě kvůli diskrétnímu řešení není v praxi

triviální získat jednotlivé veličiny, jako je např. těžiště tělesa, a proto v této práci uvádím společně s fyzikálními vztahy také algoritmy pro jejich výpočet.



Obr 2.3, Obr. 2.4, Obr. 2.5

Začneme výpočtem těžiště hmotného bodu (centre of mass) (viz. obr. 2.3), také nazývaným jako první moment hmoty q (first mass moment):

$$q = rm \tag{2.13}$$

Těžiště celé soustavy je imaginární bod (viz. obr. 2.4), a to takový, že působení gravitační síly na něj má stejný účinek jako působení na celé těleso, resp. soustavu:

$$q = \sum_i q_i = \sum_i r_i m_i = \sum_i m_i \cdot r_{CM} = m_{total} \cdot r_{CM} \tag{2.14}$$

kde r_{CM} je vektor polohy těžiště a m_{total} je celková hmotnost tělesa. Důležitý je především vektor pozice těžiště (viz. obr. 2.5). Ze vzorců (2.1), (2.13) a (2.14) dostáváme:

$$r_{CM} = \frac{q}{m_{total}} = \frac{\sum_i r_i m_i}{\sum_i m_i} = \left[\frac{\sum_i m_i x_i}{\sum_i m_i}, \frac{\sum_i m_i y_i}{\sum_i m_i}, \frac{\sum_i m_i z_i}{\sum_i m_i} \right] \tag{2.15}$$

2.2.1. Algoritmy pro výpočet těžiště

Na základě vzorce (2.15) můžeme vypočítat polohu těžiště pro libovolnou soustavu bodů, nebo pro libovolný objekt s konzistentním rozložením hmoty, a to za předpokladu, že tento objekt rozdělíme na určitý konečný počet hmotných bodů. Těmto bodům se v počítačové grafice přezdívá voxely. Podobně jako v 2D rozměru je nejmenším elementem pixel, tak v 3D prostoru jím je voxel. Tedy tzv. voxel je nejmenším elementem v trojrozměrném prostoru, který je přesně definován třemi souřadnicemi (polohovým vektorem).

Pokud bychom měli těleso reprezentované pomocí voxelů, je algoritmus na výpočet těžiště poměrně jednoduchý. Za předpokladu, že hmotnost všech voxelů je stejná, postačí sečíst jednotlivé souřadnice všech voxelů a podělit je jejich počtem. Náznak algoritmu v pseudokódu jazyka C:

```
Vynuluj souřadnice těžiště
For (přes všechny voxely){
        K souřadnicím těžiště přičti souřadnice voxelu
}
každou souřadnici těžiště vyděl počtem voxelů
```

Ve většině případů je těleso reprezentováno v počítačové grafice v podobě trojúhelníkového modelu, který ovšem není vhodný pro výpočet těžiště. Takový trojúhelníkový model je nutno rozdělit na výše zmíněné voxely, resp. procházet přímo objemem tělesa. Algoritmus je proto komplikovanější. Je třeba „posunovat“ fiktivní ukazatel přes celý objem tělesa a sčítat souřadnice jednotlivých elementárních částí (voxely). Tento algoritmus je v podstatě vyčíslením určitého integrálu přes objem tělesa v diskrétním prostoru, čili suma uvedená ve vzorci (2.15). Náznak algoritmu [9] v pseudokódu jazyka C:

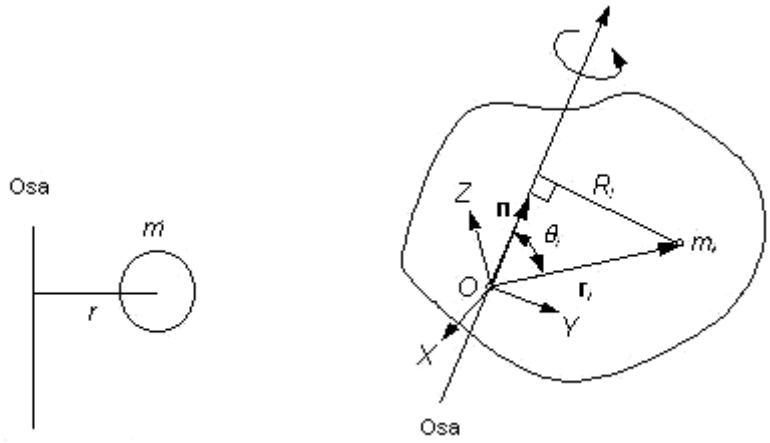
```
Vypočítej bounding box tělesa
Vynuluj souřadnice těžiště a počet voxelů
For (přes každý voxel boxu ve směru osy x )
        For (přes každý voxel boxu ve směru osy y )
                For (přes každý voxel boxu ve směru osy z )
                If (leží voxel uvnitř tělesa){
                        K souřadnicím těžiště přičti souřadnice voxelu
                        Inkrementuj počet voxelů
                        }
Každou souřadnici těžiště vyděl počtem voxelů
```

Hlavním otázkou však zůstává jak poznáme, že se nacházíme uvnitř nebo vně tělesa. Tento problém lze řešit například pomocí “pouštění” paprsku (ray) skrz těleso v nějaké z os. V místě, kde detekujeme lichou kolizi, “vstupujeme” paprskem do tělesa, naopak v místě, kde detekujeme sudou kolizi, “vystupujeme” paprskem z tělesa.

2.3. Moment setrvačnosti a matice setrvačnosti

Obdobně jako u těžiště i zde musíme brát objekt jako soustavu hmotných bodů, resp. pevné těleso [1, 3, 4, 6, 7, 8]. Takto i při stanovení momentu setrvačnosti je většina rovnic počítána v diskrétní rovině. Níže (viz. kapitola 2.3.2.) jsou uvedeny algoritmy, kterými se moment setrvačnosti počítá v praxi.

Newtonův první zákon říká, že každé těleso setrvává v klidu nebo v rovnoměrném přímočarém pohybu, dokud není nuceno působením vnějších sil svůj stav změnit [1]. Mírou setrvačnosti hmoty tělesa při translačním a rotačním pohybu je moment setrvačnosti I (Moment of Inertia) [1, 3, 4, 6, 7, 8]. Moment setrvačnosti nezávisí jen na hmotnosti objektu, ale také na jejím rozložení v tělese, proto i dvě stejně těžká tělesa mohou mít jiný moment setrvačnosti.



Obr. 2.6, Obr. 2.7

Pro zjednodušení začneme opět s jedinou částicí, u které určíme její moment setrvačnosti I, také nazývaný jako druhý moment hmoty (second mass moment):

$$I = mr^2 \tag{2.16}$$

kde m je hmotnost tělesa a r je nejkratší vzdálenost od osy rotace (viz obr. 2.6). Při srovnání se vzorci (2.13) a (2.16) vidíme rozdíl pouze v mocnině vektoru, proto rozšíření na soustavu hmotných bodů je obdobné jako u těžiště:

$$I = \sum_i m_i r_i^{\,2} \tag{2.17}$$

Z této rovnice (2.17), a na základě obrázku 2.7, odvozujeme následující:

$$I = \sum_i m_i R_i^{\,2} = \sum_i m_i \left(|r_i| \sin q_i\right)^2 = \sum_i m_i |r_i \times n|^2 = \sum_i m_i (r_i \times n) \cdot (r_i \times n) \tag{2.18}$$

kde r_i je pozice každé částice i, a n je jednotkový vektor osy rotace. Protože se pohybujeme v kartézském souřadném systému můžeme psát:

$$r_i = x_i i + y_i j + z_i k = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} \tag{2.19}$$

$$n = \cos a\, i + \cos b j + \cos c k = \begin{bmatrix} \cos a \\ \cos b \\ \cos c \end{bmatrix} \tag{2.20}$$

Aplikujeme rovnice (2.19) a (2.20), současně s převodem vektoru r na „skew-symmetric matrix“, na rovnici (2.18). Matice „skew-symmetric matrix“ je ekvivalentní vektorovému součinu, v podstatě reprezentuje 3D prostorový vektor v matici 3x3. Tím dostáváme odvození:

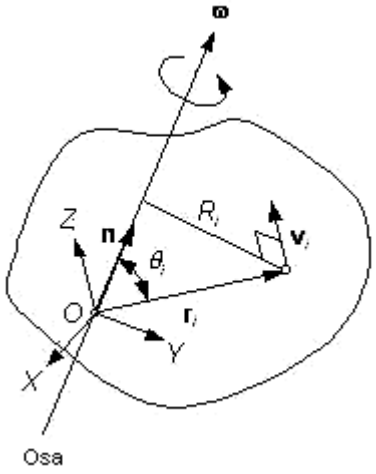
$$
\begin{aligned}
I &= \sum_i m_i (r_i \times n) \cdot (r_i \times n) \\
&= \sum_i m_i \left\{ \left(\begin{bmatrix} 0 & -z_i & y_i \\ z_i & 0 & -x_i \\ -y_i & x_i & 0 \end{bmatrix} \cdot \begin{bmatrix} \cos a \\ \cos b \\ \cos c \end{bmatrix} \right) \cdot \left(\begin{bmatrix} 0 & -z_i & y_i \\ z_i & 0 & -x_i \\ -y_i & x_i & 0 \end{bmatrix} \cdot \begin{bmatrix} \cos a \\ \cos b \\ \cos c \end{bmatrix} \right) \right\} \\
&= \sum_i m_i \left(\begin{bmatrix} y_i \cos c - z_i \cos b \\ z_i \cos a - x_i \cos c \\ x_i \cos b - y_i \cos a \end{bmatrix} \cdot \begin{bmatrix} y_i \cos c - z_i \cos b \\ z_i \cos a - x_i \cos c \\ x_i \cos b - y_i \cos a \end{bmatrix} \right) \\
&= \sum_i m_i \left[(y_i \cos c - z_i \cos b)^2 + (z_i \cos a - x_i \cos c)^2 + (x_i \cos b - y_i \cos a)^2 \right] \\
&= I_{XX} \cos^2 a + I_{YY} \cos^2 b + I_{ZZ} \cos^2 c + 2 I_{XY} \cos a \cos b + 2 I_{YZ} \cos b \cos c + 2 I_{ZX} \cos c \cos a
\end{aligned}
\tag{2.21}
$$

kde:

$$
\begin{aligned}
&I_{XX} = \sum_i m_i \left(y_i^2 + z_i^2 \right); I_{YY} = \sum_i m_i \left(z_i^2 + x_i^2 \right); I_{ZZ} = \sum_i m_i \left(x_i^2 + y_i^2 \right) \\
&I_{XY} = I_{YX} = -\sum_i m_i (x_i y_i); I_{XZ} = I_{ZX} = -\sum_i m_i (x_i z_i); I_{YZ} = I_{ZY} = -\sum_i m_i (y_i z_i)
\end{aligned}
\tag{2.22}
$$

přičemž I_{XX}, I_{YY}, I_{ZZ} se nazývají hlavní momenty setrvačnosti (moment of inertia) a I_{XY}, I_{YX}, I_{XZ}, I_{ZX}, I_{YZ}, I_{ZY} se nazývají deviační momenty setrvačnosti (produkt of inertia). Pokud je střed soustavy souřadnic O přesně v těžišti tělesa, je hmota rozprostřena rovnoměrně kolem všech os, a proto i deviační momenty setrvačnosti jsou nulové.

2.3.1. Úhlový moment



Obr. 2.8

Úhlový moment pevného tělesa, resp. soustavy hmotných bodů, můžeme vypočítat tak, že sečteme jednotlivé momenty hmotných bodů (2.12) [1, 3, 6, 7, 8]:

$$H = \sum_i r_i \times p_i \tag{2.23}$$

Dosazením vzorce (2.6) do vzorce (2.23) dostáváme:

$$H = \sum_i r_i \times (m_i v_i) \tag{2.24}$$

kde v_i je rychlost hmotného bodu opisujícího kružnici. Tuto rychlost vypočítáme jako vektorový součin úhlové rychlosti a vzdálenosti od osy otáčení:

$$v_i = w_i \times r_i \tag{2.25}$$

Dosazením vzorce (2.25) do předchozího (2.24) dostáváme:

$$H = \sum_i m_i [r_i \times (w_i \times r_i)] \tag{2.26}$$

Výše uvedené vzorce korespondují s obrázkem 2.8.

Uvažujeme-li tedy kartézský souřadný systém, můžeme obdobně jako u vztahu (2.22) za použití „skew-symmetric matrix“ odvodit následující vzorec:

$$\begin{aligned}
H &= \sum_i m_i [r_i \times (w_i \times r_i)] \\
&= \sum_i m_i \begin{bmatrix} 0 & -z_i & y_i \\ z_i & 0 & -x_i \\ -y_i & x_i & 0 \end{bmatrix} \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} \\
&= \sum_i m_i \begin{bmatrix} 0 & -z_i & y_i \\ z_i & 0 & -x_i \\ -y_i & x_i & 0 \end{bmatrix} \begin{bmatrix} w_y z_i - w_z y_i \\ w_z x_i - w_x z_i \\ w_x y_i - w_y x_i \end{bmatrix} \\
&= \sum_i m_i \begin{bmatrix} w_x (y_i^2 + z_i^2) - w_y x_i y_i - w_z x_i z_i \\ -w_x y_i x_i + w_y (z_i^2 + x_i^2) - w_z y_i z_i \\ -w_x z_i x_i - w_y z_i y_i + w_z (x_i^2 + y_i^2) \end{bmatrix} \\
&= \begin{bmatrix} \sum_i m_i (y_i^2 + z_i^2) & -\sum_i m_i (x_i y_i) & -\sum_i m_i (x_i z_i) \\ -\sum_i m_i (y_i z_i) & \sum_i m_i (z_i^2 + x_i^2) & -\sum_i m_i (y_i z_i) \\ -\sum_i m_i (z_i x_i) & -\sum_i m_i (z_i y_i) & \sum_i m_i (x_i^2 + y_i^2) \end{bmatrix} \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix}
\end{aligned} \tag{2.27}$$

Do vzorce (2.27) dosadíme výrazy (2.22), pomocí kterých výraz upravíme do jednoduší podoby:

$$H = \begin{bmatrix} I_{XX} & I_{XY} & I_{XZ} \\ I_{YX} & I_{YY} & I_{YZ} \\ I_{ZX} & I_{ZY} & I_{ZZ} \end{bmatrix} \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} = Iw \tag{2.28}$$

kde I je matice setrvačnosti (Inertia Tensor). Matice setrvačnosti reprezentuje rozložení hmoty vůči všem osám procházejícím počátkem a je konstantní pro dané těleso. Výsledný moment tělesa v prostoru je pak dán součinem matice setrvačnosti a úhlové rychlosti objektu.

2.3.2. Algoritmy pro výpočet matice setrvačnosti

Matice setrvačnosti, resp. moment setrvačnosti, je pro některá tělesa známa [1]. Pro libovolné těleso je ale nutné tento tensor vypočítat. Jak je zřejmé ze vzorce (2.22), algoritmus na výpočet matice setrvačnosti musí projít všechny diskrétní hodnoty, tzn. že tento algoritmus bude podobný algoritmu na výpočet těžiště [9, 10]. Níže uvedený náznak algoritmu v pseudokódu jazyka C je použitelný pro těleso reprezentované trojúhelníkovým modelem:

Umísti souřadný systém do těžiště tělesa
Vypočítej bounding box tělesa
Vynuluj matici setrvačnosti (IT) a počet voxelů
For (přes každý voxel boxu ve směru osy x)
    For (přes každý voxel boxu ve směru osy y)
        For (přes každý voxel boxu ve směru osy z)
        If (leží voxel uvnitř tělesa){

$$IT = IT + \begin{bmatrix} (y^2 + z^2) & -xy & -xz \\ -yx & (z^2 + x^2) & -yz \\ -zx & -zy & (x^2 + y^2) \end{bmatrix}$$

            Inkrementuj počet voxelů
        }
matici setrvačnosti vyděl počtem voxelů

Algoritmus je v základu stejný jako algoritmus pro výpočet těžiště, proto se problém s rozdělením na voxely řeší stejně jako u algoritmu pro těžiště. Jednoduší verze algoritmu pro těleso reprezentované voxeli je opět skoro stejná jak u těžiště, jen se liší v samotném výpočtu.

Pokud má matice setrvačnosti reprezentovat těleso, které bude rotovat kolem své osy (tedy kolem těžiště), musí být souřadný systém umístěn právě v těžišti. Proto platí, že pokud počítáme obě veličiny (těžiště i matici setrvačnosti), není možné je počítat v jednom průchodu. Nejprve je tedy nutné vypočítat těžiště, a poté umístit počátek souřadného systému do získaného bodu. Tyto výpočty se provádí s lokálními souřadnicemi modelu, a ne s globálními (world) souřadnicemi.

3. Detekce kolizí

Detekce kolizí hledá stavy, kdy dva objekty v prostoru pronikají do sebe [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. Tento stav se nazývá kolize a obvykle je nechtěný. Patří mezi často řešené problémy při programování počítačových her. Dále se detekce kolizí používá například v:

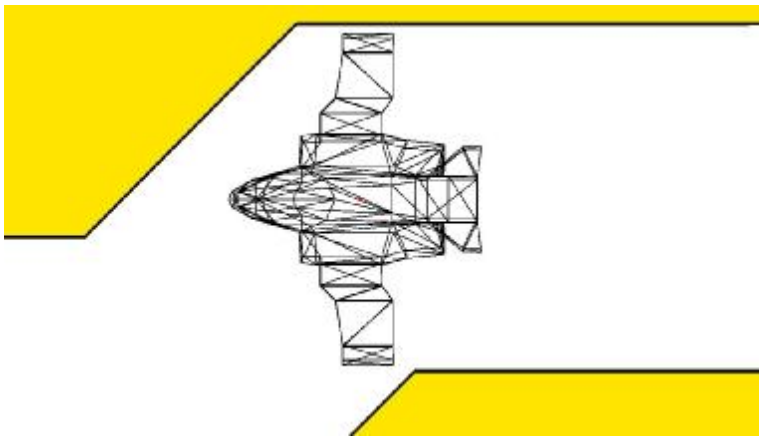
- Robotika (plánování cesty robota)
- Animační a simulační systémy (test konzistence scény, fyzikální modelování)
- CAD, strojírenský průmysl (robustní a numericky stabilní implementace)
- Molekulární modelování

3.1. Základní rozdělení detekce kolizí

Dělení kolizí probíhá především na základě „plynoucího“ času, tzn. pro statickou scénu, diskrétní a dynamickou [11].

3.1.1. Statická detekce kolizí

Detekce testu kolize se provádí pouze pro statickou scénu (viz. obr. 3.1.).



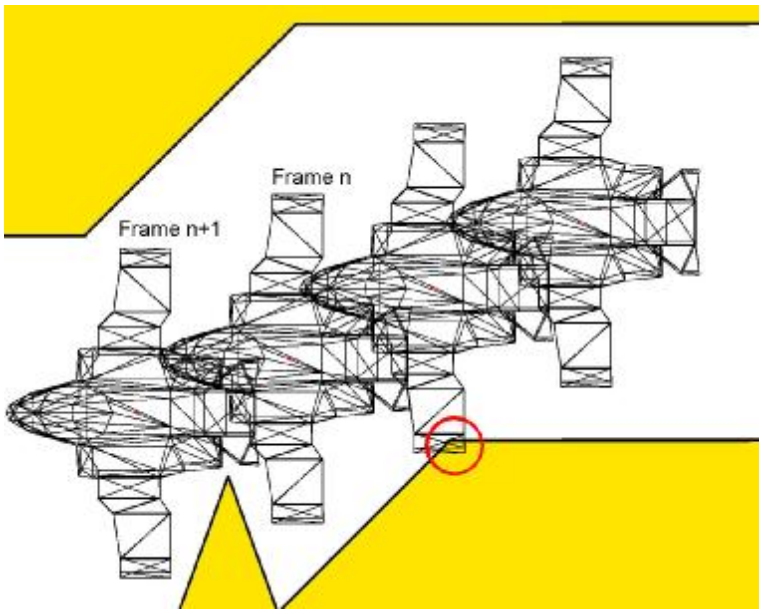
Obr. 3.1

3.1.2. Pseudo-dynamická detekce kolizí

V tomto případě se provádí testy na diskrétní množině konfigurací objektu odpovídající jeho pohybu (viz. obr. 3.2). Tento druh detekce je nejčastěji používán pro detekci kolizí v počítačových hrách. Přesnost detekce závisí na velikosti diskrétního kroku. Největším problémem pseudo-dynamické detekce je „proskočení“ objektu tenkou překážkou. Jak je znázorněno na obrázku 3.2 mezi snímkem n a snímkem n+1 dojde k tomuto „proskočení“. Na snímku n ani n+1 nedojde k detekci, ale při skutečném pohybu by ke srážce došlo.

U klasické diskrétní metody detekce se v každém kroku simulace integrují fyzikální veličiny, na základě kterých se mění poloha i natočení objektu. Při pseudo-dynamické detekci mohou nastat dva

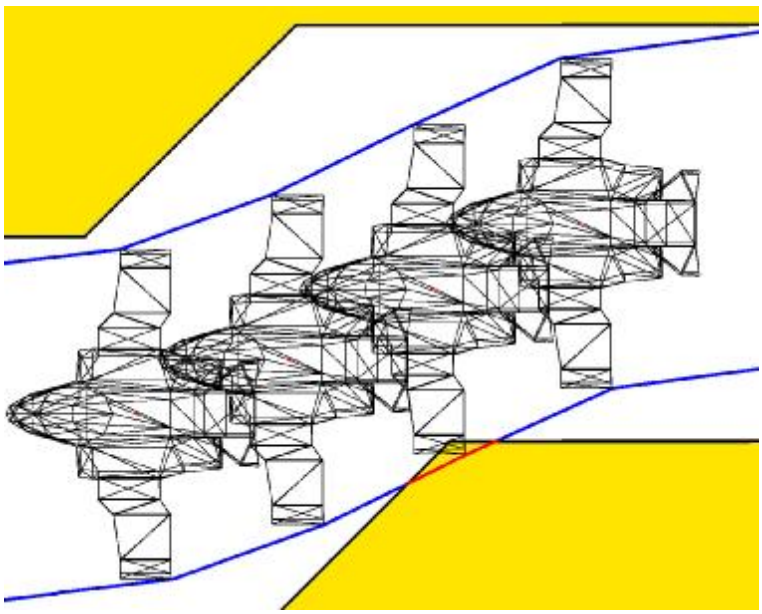
stavy, nekolizní stav a průnik objektů. Výsledkem detekce je pak průnik dvou těles, resp. všechny kolidující trojúhelníky (viz. obr 3.2).



Obr. 3.2

3.1.3. Dynamická detekce kolizí

Testuje se prostor vzniklý pohybem objektu. Tato detekce je na počítačích, které pracují v diskrétních hodnotách, nerealizovatelná. Proto se na počítačích používají vylepšené nebo specializované pseudo-dynamické metody.

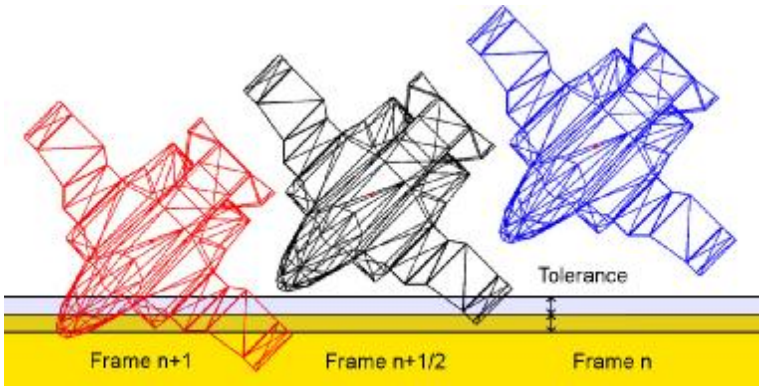


Obr 3.4

3.1.4. Optimalizovaná pseudo-dynamická detekce kolizí

Při optimalizované pseudo-dynamické detekci se v situaci, kdy dojde k průniku nebo neřešitelné kolizi, použije metoda půlení intervalu (viz obr. 3.5), ve které se „vrátíme v čase“ a zmenšíme krok na polovinu. S takto zmenšeným krokem provedeme všechny integrace znovu a vyhodnotíme kolizi. Tuto operaci opakujeme dokud nedosáhneme požadované přesnosti (viz. obr. 3.5).

Detekce kolizí představuje v reálných situacích komplexní problém, protože se často musí řešit srážky velkého množství složitě tvarovaných těles. Tento komplexní problém se rozpadá na dva hlavní podproblémy. Prvním podproblémem je zjednodušení složitého tvaru těles tak, aby se složité výpočty kolizí na úrovni trojúhelníků nemusely provádět pro všechna tělesa ve skupině. Tento problém se řeší použitím obalových těles. Druhým podproblémem je rozdělení velkého počtu objektů na menší skupiny, což nazýváme dělení prostoru. V následujících kapitolách si řešení těchto problémů detailněji popíšeme.

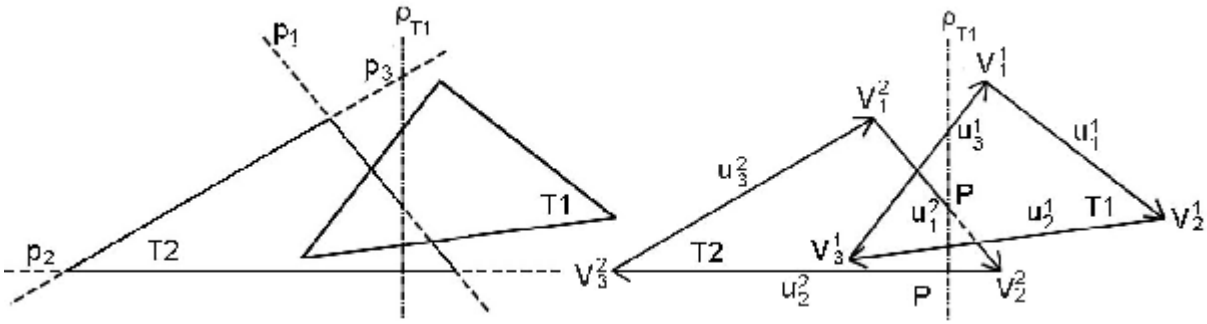


Obr. 3.5

3.2. Kolize trojúhelník vs. trojúhelník

Vesměs veškeré objekty v počítačové grafice jsou reprezentovány trojúhelníkovým modelem, proto je algoritmus detekce kolize mezi dvěmi trojúhelníky základem [13, 15, 16, 17, 19, 20]. Asi nejjednodušší algoritmus hledající kolizi mezi dvěma libovolnými trojúhelníky odvodíme ze znalostí uvedených v kapitole 1. Základní pojmy. Existují však i optimalizované, lepší či rychlejší verze tohoto algoritmu, jako například A Fast Triangle-Triangle Intersection Test [19] nebo A fast triangle to triangle intersection test for collision detection [20].

Základní úvaha je taková, že pokud libovolná hrana jednoho z trojúhelníků prochází „obsahem“ druhého trojúhelníku, dochází k průniku těchto trojúhelníků. Pokud si představíme vždy jeden z trojúhelníků jako rovinu, můžeme si druhý trojúhelník představit jako tři úsečky, resp. přímky, které tento trojúhelník tvoří (viz obr. 3.6). Pro každou přímku poté vypočítáme její průsečík s plochou. Jakmile jeden z průsečík leží v ploše ohraničené trojúhelníkem, pak jsme nalezli průsečík obou trojúhelníků. Pokud jsme však průsečík nenašli, můžeme situaci obrátit a vyměnit úlohy trojúhelníků. Trojúhelník, který tvořil plochu je nyní reprezentován přímkami a druhý trojúhelník naopak tvoří plochu.



Obr. 3.6, Obr. 3.7

3.2.1. Nalezení průsečíků

Výpočet je třeba započít určením rovnic přímek a rovin, pro které budeme počítat průsečíky. Nechť trojúhelník T_1 má vrcholy V_1^1, V_2^1, V_3^1 a vektory mezi těmito vrcholy jsou značeny u_1^1, u_2^1, u_3^1. Pro trojúhelník T_2 jsou vrcholy V_1^2, V_2^2, V_3^2 a vektory u_1^2, u_2^2, u_3^2 (viz. obr. 3.7). Vektorové rovnice přímek (1.1) a (1.2) pak budou mít následující tvar:

$$\begin{aligned} X_i^1 &= V_i^1 + tu_i^1, t \in \Re \\ X_i^2 &= V_i^2 + tu_i^2, t \in \Re \end{aligned} \tag{3.1}$$

kde i je index vrcholu, resp. vektoru přiléhajícího k vrcholu. Normálu pro jednotlivé trojúhelníky dostaneme vektorovým součtem libovolných dvou vektorů ramen trojúhelníku, např.:

$$\begin{aligned} n^1 &= u_1^1 \times u_2^1 \\ n^2 &= u_1^2 \times u_2^2 \end{aligned} \tag{3.2}$$

Získáme Hessův normálový tvar rovnice roviny (1.3) (viz kapitola 1.2 Hessův normálový tvar rovnice roviny):

$$\left(X^1 - V_1^1\right)n^1 = 0 \tag{3.3}$$

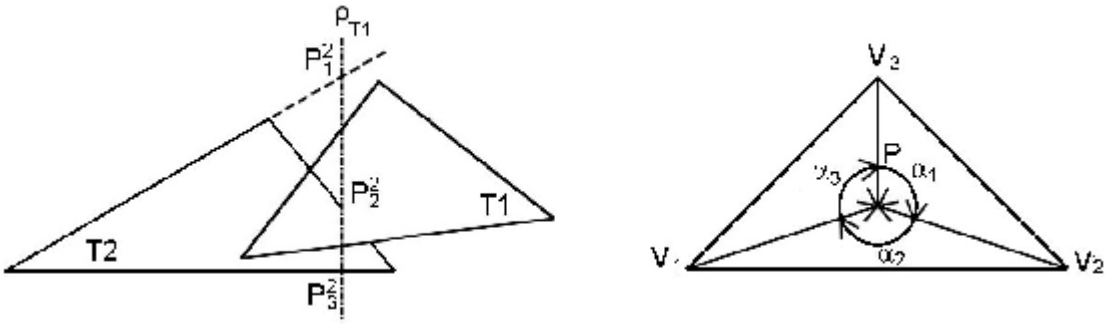
$$\left(X^2 - V_1^2\right)n^2 = 0 \tag{3.4}$$

kde jako libovolný bod na rovině použijeme jeden z vrcholů trojúhelníku, ve vzorci je konkrétně použit V_1. Nyní vypočítáme všechny průsečíky mezi přímkami X_i^1 a rovinou určenou vzorcem (3.4). Dále průsečíky mezi přímkami X_i^2 a rovinou určenou vzorcem (3.3) (viz. obr. 3.8), podle vzorců (1.6), (1.7), (1.8) uvedených v kapitole 1.3 Vzájemná poloha přímky a roviny:

$$P_i^1 = V_i^1 + t_0^k \cdot u_i^1 \tag{3.5}$$

$$P_i^2 = V_i^2 + t_0^k \cdot u_i^2 \tag{3.6}$$

kde t_0^k je parametr t, vypočítaný pro každou rovnici průsečíků (1.7).



Obr. 3.8, Obr. 3.9

3.2.2. Pozice průsečíků

Pro vypočítané průsečíky P_i^1 prvního trojúhelníku je nutné ověřit, zda se nacházejí v „obsahu“ druhého trojúhelníku T_2. To samé je třeba udělat i pro průsečíky P_i^2 a první trojúhelník T_1.

Při určování, zda se průsečík nachází uvnitř trojúhelníku, se již pohybujeme ve 2D rovině a úloha je zjednodušena na zjištění, zda se bod (průsečík) nachází uvnitř trojúhelníků (obecně polygonu). Algoritmů řešících tento problém je více, nyní si popíšeme jeden z dobře fungujících [15, 17].

Mějme trojúhelník reprezentovaný vrcholy V_1, V_2, V_3. Mezi vypočítanými průsečíky (viz. kapitola 3.2.1. Nalezení průsečíků) a těmito vrcholy vypočítáme vektor:

$$v_i = P - V_i \tag{3.7}$$

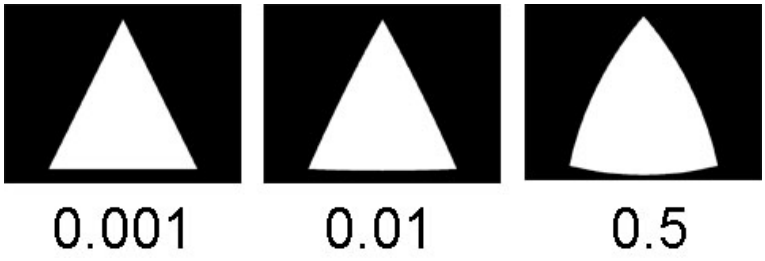
kde i je index vrcholu. Za předpokladu, že hledaný bod (průsečík) se nachází uvnitř trojúhelníku, musí být součet úhlů všech vektorů směřujících k bodu roven 180° (viz. obr. 3.9). Úhel, který svírají dva vektory, vypočítáme jako:

$$\begin{aligned} a_1 &= \arccos(v_1 \cdot v_2) \\ a_2 &= \arccos(v_2 \cdot v_3) \\ a_3 &= \arccos(v_3 \cdot v_1) \end{aligned} \tag{3.8}$$

Součet všech úhlů musí byt roven 180° ($2p$), resp. absolutní hodnota rozdílu součtu úhlů a 180° musí být menší nebo rovna zadané toleranci e :

$$\left|(a_1 + a_2 + a_3) - 2p\right| \le e \tag{3.9}$$

Jak se mění prostor vymezený trojúhelníkem pro hledání bodu v závislosti na parametru e znázorňuje obrázek 3.10. Pokud hledáme průsečíky v cyklu je možné cyklus ukončit po prvním průsečíku, který se nachází uvnitř druhého z trojúhelníků, protože došlo k nalezení hledaného průniku trojúhelníků.



Obr. 3.10

3.2.3. Aplikace na kolizi polygon vs. polygon

Většina těles je v počítačové grafice reprezentována jako trojúhelníkový model, proto je detekce kolize mezi těmito modely pouze detekcí kolize mezi všemi trojúhelníky jednoho modelu a všemi trojúhelníky druhého modelu. Algoritmy, které hledají přímo kolizi mezi reálnými objekty, jsou proto většinou jen optimalizované algoritmy pro detekci trojúhelník vs. trojúhelník. Optimalizace těchto algoritmů spočívají především v eliminaci počtu porovnávaných trojúhelníků.

3.3. Obalová tělesa

Vyhodnocovat detekci přímo mezi trojúhelníky je strojově velmi náročné, proto se objekty „zabalují“ do tzv. obalových těles [11, 14]. Základním problémem je zvolit vhodné obalové těleso, a to tak, aby obklopovalo původní model co nejtěsněji a aby testování dvou obalových těles probíhalo co nejrychleji.

3.3.1. Sphere

Koule (viz. obr. 3.11) je nejjednodušším obalovým tělesem. V testu porovnáváme vzdálenosti mezi dvěma sférami. Její nevýhodou je, že je nejméně citlivá na tvar tělesa a má největší procento obsaženého volného prostoru.

3.3.2. AABB

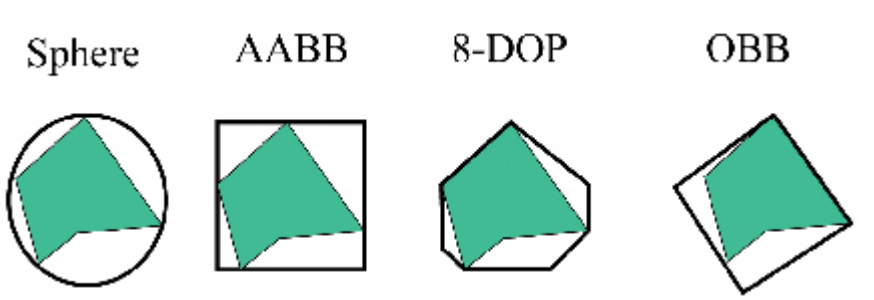
Axis Aligned Bounding Box (viz. obr. 3.11), tzv. osově orientované obalové těleso, je rovněž jednoduché, navíc v testu stačí porovnávat překrytí jednotlivých os. Teoreticky by mělo být nejrychlejší, protože se používají jen operátory porovnání. Je opět málo citlivé na tvar tělesa.

3.3.3. OBB

Oriented Bounding Box (viz. obr. 3.11), tzv. orientované obalové těleso, je obtížně vytvořitelné, ale dobře ohraničuje těleso. Test je opět založen na jednoduchých matematických operacích, je tedy relativně rychlý.

3.3.4. K-DOP

K-Discrete Orientation Polytopes (obr. 3.11), tzv. K-vrcholový orientovaný obal, je asi nejsložitější na výrobu. V testu se porovnávají rovnoběžné hrany těles. Tento obalový box nejlépe ze všech jmenovaných kopíruje tvar tělesa.



Obr. 3.11

3.4. Dělení prostoru

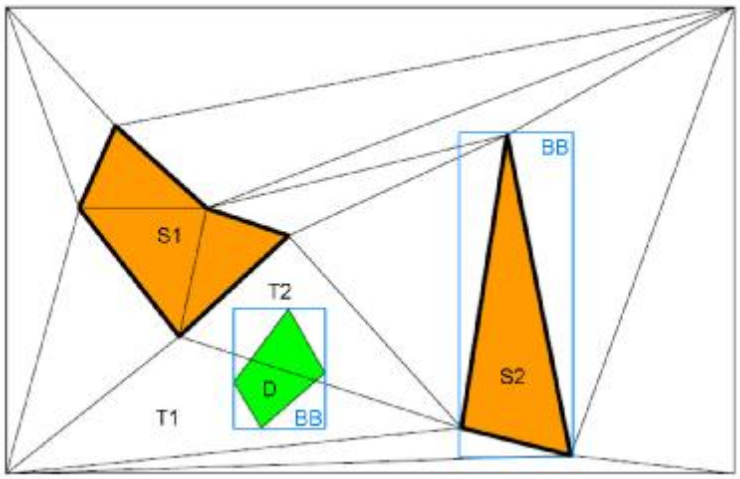
Prostor je teoreticky nekonečný a obsahuje nekonečné množství těles. V takovéto situaci je téměř nemožné počítat kolize každého objektu s každým. Snaha vyřešit tento problém vedla k zavedení technik dělení prostoru [11, 12, 14].

Základní ideou je „rozbít“ prostor na jednotlivé části s podobnou strukturou. Detekce kolizí se potom provádí jen v tomto lokálním podprostoru. Některé metody, kterými lze dělit podprostor jsou popsány v následujících odstavcích.

3.4.1. Využití sítě čtyřstěnů

Vrcholy trojúhelníků všech statických těles se propojí s nejbližšími vrcholy trojúhelníků ostatních statických těles a vytvoří tak čtyřstěny. Tyto čtyřstěny a původní tělesa mohou být reprezentovány obalovým tělesem (Bounding boxem) dále jen BB. Příklad ve 2D je znázorněn na obrázku 3.12, konkrétní zobrazený BB je osově orientovaný (AABB) a je zobrazen jen v podobě dvou těles.

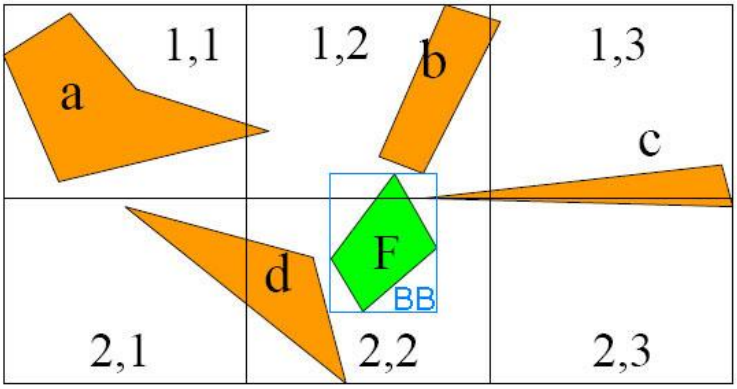
Nejprve tedy určíme všechny čtyřstěny, resp. jejich BB, které protínají hledané těleso D, resp. jeho obal (BB). Na 2D obrázku 3.12 to jsou trojúhelníky T1 a T2 (pozn. ve 3D to jsou čtyřstěny). Pro určení samotné kolize mezi objekty se provádí pouze test kolize všech takových trojúhelníků prostředí, které přiléhají k těmto čtyřstěnům, s trojúhelníky objektu D.



Obr. 3.12

3.4.2. Voxelová mřížka

Prostor se rozdělí na mřížku předem daných rozměrů (viz. obr. 3.13). V takto rozděleném prostoru určíme všechny voxely, do kterých spadá objekt F, resp. jeho BB. Na 2D obrázku 3.13 to jsou voxely (1,2) a (2,2). Samotný test kolize se provádí pouze pro trojúhelníky z daných voxelů a trojúhelníky tělesa F.



Obr. 3.13

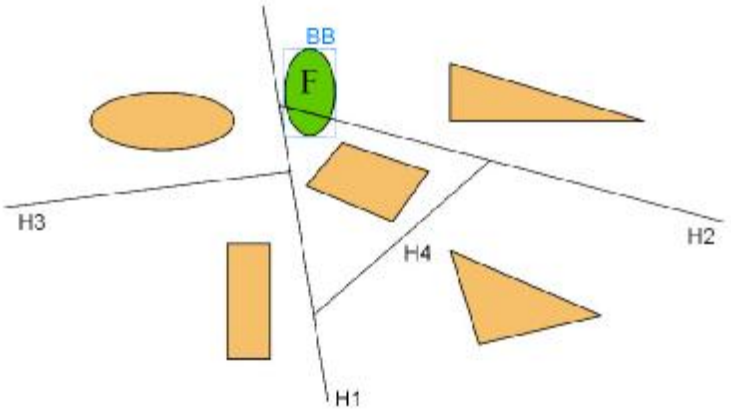
3.4.3. BSP Strom

Jednou z hojně využívaných možností jak rozdělit prostor je použití tzv. stromů [18]. Těchto stromů existuje celá řada, k těm nejpoužívanějším patří BSP Tree (viz. obr. 3.14).

Prostor je rozdělen hyper-plochou tak, aby se v každé jeho polovině nacházelo zhruba stejné množství objektů. Pokud je scéna složena převážně ze statických objektů, pak jsou rozdělovány jen statické objekty, jako je tomu na obrázku 3.14. V případě scén, u kterých je toto schéma nedostačující, tzn. převažuje počet dynamických objektů, je třeba jednou za čas, např. po přesunu většího počtu

objektů, strom znovu přerozdělit. Samotný test se provádí mezi aktuálním listovým objektem a BB testovaného objektu. Na základě polohy BB a dělící hyper-plochy se rozhodne o postupu stromem.

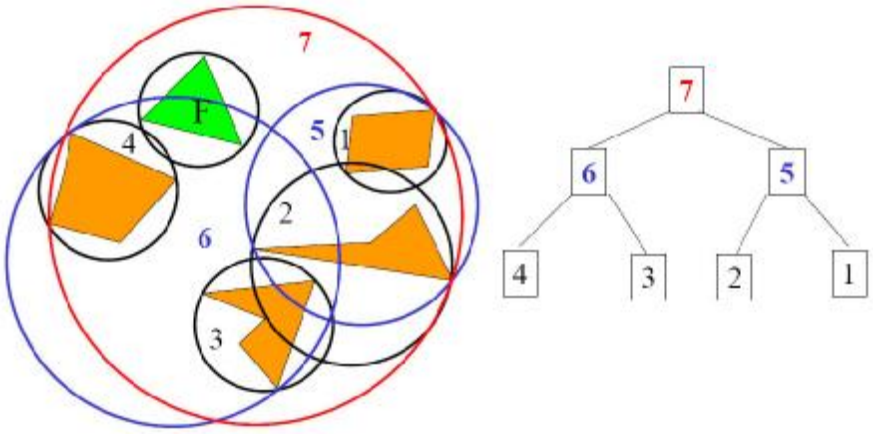
Ostatní stromové algoritmy, jako např. K-D-Tree[18], Quad-Tree [18] a jejich modifikace, se většinou liší ve formě uložení dat ve stromu a v počtu a druhu použitých hyper-ploch.



Obr. 3.14

3.4.4. Hierarchie obalových těles

Prostor lze také rozdělit jen pomocí obalových těles, které však uspořádáme do stromu. Takovýto strom může být tvořen například z obalových těles typu koule (viz. obr. 3.15). Nicméně, pro strom lze využít i ostatní obalová tělesa, případně jejich kombinace.



Obr. 3.15

4. Vyhodnocení kolize

Po zjištění kolize nastává otázka, jak takovou kolizi vyhodnotit. Vyhodnocení kolize záleží na druhu a účelu aplikace, ve které se kolize odehrává. V našem případě se soustředíme na kolize týkající se počítačových her a jejich vyhodnocování. V počítačových hrách můžeme řešit různé druhy situací, při kterých kolize nastává, například ve zjednodušené podobě:

- srážka pohyblivého objektu a pevné překážky
- srážka dvou pohybujících se těles

Tyto kolize můžeme také různě vyhodnocovat. Například při nárazu pohyblivého tělesa do pevné překážky (nebo jiného pohyblivého tělesa) může pohyblivé těleso explodovat (letící raketa narazí do domu nebo do letadla). Nebo může po detekování kolize dojít k zastavení některého z těles, případně ke vzájemnému klouzání těles (většina soudobých 3D her, kde se ovládaná postava při chůzi zastaví o překážku). Dalším častým řešením je také odraz kolidujících těles (fyzika v moderních 3D akčních hrách, např. střelba do plechovky).

V této diplomové práci se budeme zabývat převážně řešením odrazu, a to na základě reálného fyzikálního modelu srážky dvou pevných těles. Odraz můžeme řešit více způsoby. Jako jednoduchý odraz obalového tělesa sphere (koule) na základě fyzikálních vzorců pro ideálně pružnou srážku, kde se v úvahu nebere rotace ani okolní vlivy prostředí (tření, odpor vzduchu). A nebo jako fyzikální odraz dvou libovolných objektů, který se hlouběji opírá o fyzikální zákony a uvažuje jak translační tak rotační pohyb. I v tomto případě bude zanedbáno tření při srážce. Působení dalších vlivů, jako je odpor prostředí, lze zjednodušeně brát v úvahu v podobě působení vnějších sil na těleso. Nakonec si okrajově zmíníme také řešení výbuchu, které se dá provést velmi jednoduše a efektně promítnutím animace výbuchu na plochu kolmou ke kameře. Tato technika se nazývá billboarding.

4.1. Jednoduchý odraz

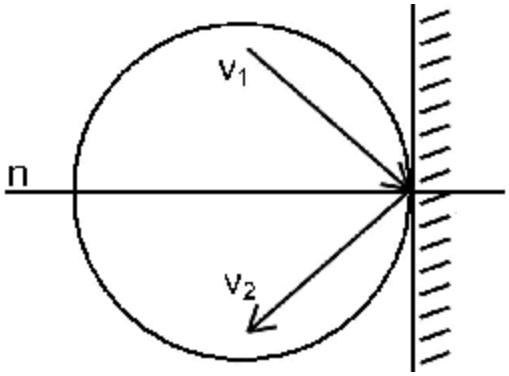
Při jednoduchém odrazu dvou koulí, resp. koule a pevného objektu, nám pomohou fyzikální zákony. Z fyzikálního hlediska řešíme šikmou dokonale pružnou srážku. Při pružné srážce se obecně mění kinetická energie jednotlivých těles, které se srážky účastní. Celková kinetická energie soustavy před srážkou i po srážce je však stejná [1].

4.1.1. Odraz koule od pevné překážky

Vzorec pro odraz koule od pevné překážky je jednoduše odvozen z poučky, kdy úhel odrazu se rovná úhlu dopadu (viz. obr. 4.1) [1]:

$$v_2 = 2(-v_1 \cdot n)n + v_1 \tag{4.1}$$

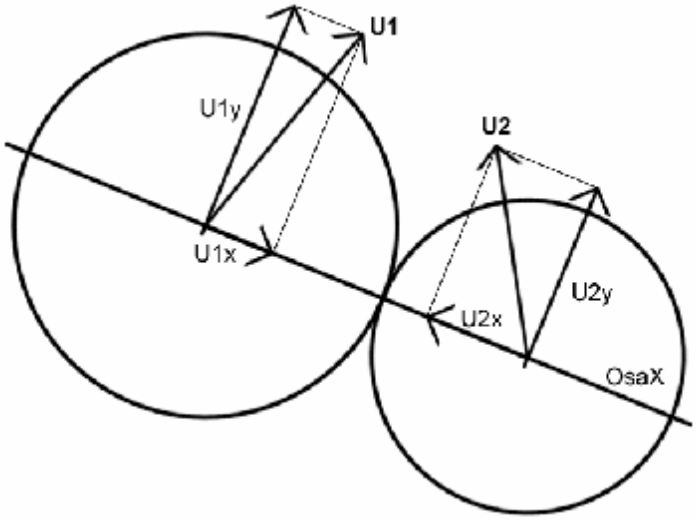
kde v_1 je původní vektor rychlosti, v_2 je nový vektor rychlosti a n je normála srážky.



Obr. 4.1

4.1.2. Srážka dvou koulí

Níže uvedené vzorce a výpočty vychází z Newtonova zákona zachování hybnosti (2.6, 2.7). Situaci při srážce dvou koulí reprezentuje obrázek 4.2.



Obr 4.2

Vektory U1 a U2 představují rychlosti koulí v čase nárazu. Středy obou těles spojuje osa X, na které leží vektory U1x a U2x, což jsou vlastně průměty rychlostí do přímky X. U1y a U2y znázorňují projekce rychlostí těles na osy, které jsou kolmé k ose X. K jejich výpočtu postačí jednoduchý skalární součin. Do následujících rovnic dosazujeme ještě čísla M1 a M2, která vyjadřují hmotnosti kolidujících koulí. Cílem je tedy vypočítat orientaci vektorů rychlosti U1 a U2 po odrazu, které budou je vyjadřovat nové vektory V1 a V2. Čísla V1x, V1y, V2x, V2y představují opět průměty. Tento postup je již odvozený například v seriálu NeHe [21], který se počítačové grafice věnuje.

Abychom mohli hledat průměty do jednotlivých os, potřebujeme nejdříve najít osu, která spojuje oba středy (těžiště) koulí:

$$OsaX = (S2 - S1) \tag{4.2}$$

kde S1 a S2 jsou polohové vektory středů koulí a OsaX je směrový vektor osy, která spojuje obě koule. Nyní je třeba najít průměty aktuálních rychlostí do jednotlivých os:

$$\begin{aligned} U1_X &= OsaX * (OsaX \cdot U1) \\ U2_X &= -OsaX * (OsaX \cdot U2) \end{aligned} \tag{4.3}$$

$$\begin{aligned} U1_Y &= U1 - U1_X \\ U2_Y &= U2 - U2_Y \end{aligned} \tag{4.4}$$

Pomocí projekcí do jednotlivých os získáme nové rychlosti:

$$\begin{aligned} V1_X &= [(U1_X * M1) + (U2_X * M2) - (U1_X - U2_X) * M2] / (M1 + M2) \\ V1_Y &= U1_Y \end{aligned} \tag{4.5}$$

$$\begin{aligned} V2_X &= [(U1_X * M1) + (U2_X * M2) - (U2_X - U1_X) * M1] / (M1 + M2) \\ V2_Y &= U2_Y \end{aligned} \tag{4.6}$$

Výsledná rychlost je poté součtem vypočítaných průmětů nových rychlostí:

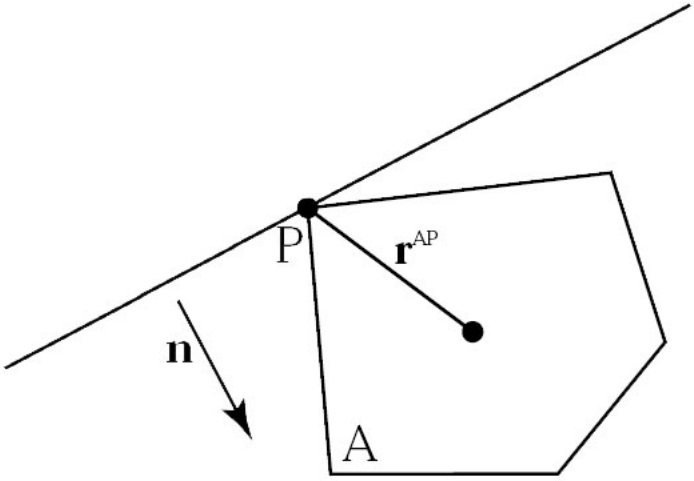
$$\begin{aligned} V1 &= V1_X + V1_Y \\ V2 &= V2_X + V2_Y \end{aligned} \tag{4.7}$$

4.2. Fyzikální odraz

Pro fyzikální odraz využijeme vzorců uvedených v kapitole 2. Fyzika. Pro řešení této problematiky jsou nutné hlubší znalosti v oblasti fyziky [3, 4, 5, 6, 7, 8]. Fyzikální odraz budeme řešit ve dvou situacích, při odrazu tělesa od pevné překážky a při srážce dvou těles navzájem.

4.2.1. Odraz od pevného objektu

Jednodušší případ srážky je situace, kdy těleso narazí do pevné překážky. V úvahu se totiž berou veličiny jen jednoho tělesa a snáze zjistíme bod a normálu kolize. Odraz objektu od pevné plochy si nyní popíšeme ve 2D rovině, kde je snadnější nastínit řešený problém (viz. obr. 4.3).



Obr. 4.3

Obrázek 4.3 znázorňuje srážku tělesa A s plošným tělesem v místě P. Vektory (ramena) směřující od středu tělesa k bodu P označíme jako r_{AP}. Dále potřebujeme znát veličiny jako rychlost, kterou značíme $v_{A,}$ úhlovou rychlost tělesa w_A a hmotnost tělesa M_A. Důležité je rovněž určit normálový vektor srážky, který je při srážce tělesa a pevné překážky normálou roviny, do které objekt narazil. Jedním ze základních výpočtů je výpočet relativní rychlosti:

$$v_{AP} = v_A + w_A \times r_{AP} \tag{4.8}$$

Bereme-li v úvahu normálový vektor, můžeme vypočítat relativní „normálovou“ rychlost:

$$v_{norm} = v_{AP} \cdot n = (v_A + w_A \times r_{AP}) \cdot n \tag{4.9}$$

kde v_{norm} je relativní „normálová“ rychlost v místě srážky, od které se budou odvozovat další výpočty. Pokud chceme brát v úvahu jinou než ideálně pružnou srážku, musíme tuto rychlost zohlednit koeficientem, který určuje míru odrazivosti. Tento koeficient se nazývá koeficient odrazu a značí se písmenem e. Úpravou vzorce (4.9) dostáváme:

$$v_{norm} = -e * v_{AP} \cdot n \tag{4.10}$$

Vztah mezi relativními rychlostmi před a po srážce udává vzorec:

$$v_2^{AP} \cdot n = -e * v_1^{AP} \cdot n \tag{4.11}$$

Při vyhodnocování výsledné rychlosti objektu musíme znát sílu, kterou se od sebe objekty odrazily. Tato síla se nazývá impuls síly a značí se J (její výpočet odvodíme později, z následujících vzorců). Díky znalosti této síly můžeme stanovit novou rychlost:

$$v_2 = v_1 + \frac{J}{M_A} \cdot n \tag{4.12}$$

a novou úhlovou rychlost:

$$w_2 = w_1 + \frac{r_{AP} * J}{I_A} \cdot n \tag{4.13}$$

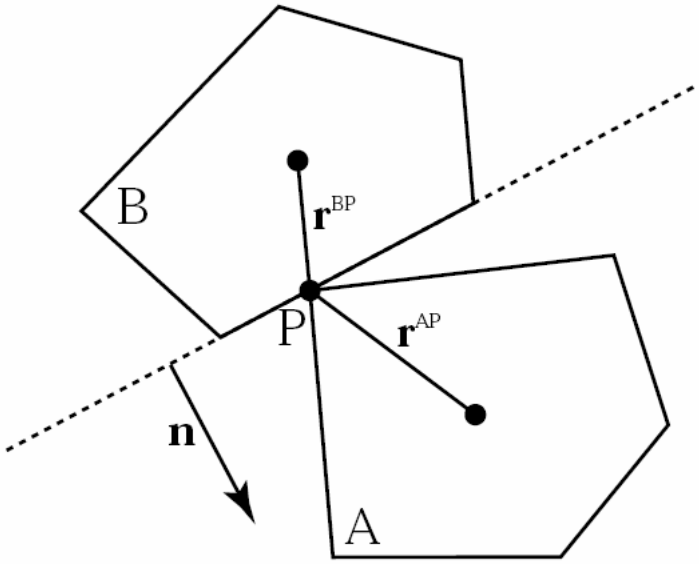
kde I_A je moment setrvačnosti tělesa (viz. kapitola 2.3. Moment setrvačnosti a matice setrvačnosti). Odvozování výsledného impulzu síly začneme rovnicí (4.11), do které dosadíme za proměnnou v_2^{AP} vztah pro výpočet relativní rychlosti (4.8), poté za v_2 a w_2 vztahy (4.12) a (4.13):

$$\begin{aligned} & v_2^{AP} \cdot n = -e * v_1^{AP} \cdot n \\ & \left(v_2^A + w_2^A \times r_{AP}\right) \cdot n = -e * \left(v_1^A + w_2^A \times r_{AP}\right) \cdot n \\ & \left(v_1^A + \frac{J}{M_A} \cdot n + \left(w_1^A + \frac{r_{AP} * J}{I_A} \cdot n\right) \times r_{AP}\right) \cdot n = -e * \left(v_1^A + w_2^A \times r_{AP}\right) \cdot n \end{aligned} \tag{4.14}$$

Nakonec z tohoto vztahu vyjádříme impuls síly J:

$$J = \frac{-(1+e) * v_1^{AB} \cdot n}{\frac{1}{M_A} + I_A^{-1} * \left[\left(r_{AP} \times n\right) \times r_{AP}\right]} \tag{4.15}$$

4.2.2. Srážka dvou těles



Obr. 4.4

Pokud se srazí dvě tělesa (viz. obr. 4.4), je situace podobná jako při srážce tělesa a plošného objektu. Pouze je třeba do výpočtu relativní rychlosti zahrnout vzájemnou relativní rychlost obou těles:

$$v_{AB} = v_{AP} - v_{BP} \tag{4.16}$$

kde v_{AP} i v_{BP} vypočítáme ze vzorce (4.8). Vzájemný vztah mezi rychlostmi je poté následující:

$$v_2^{AB} \cdot n = -e * v_1^{AB} \cdot n \tag{4.17}$$

Obdobným způsobem jako v předchozím případě odvodíme impuls síly a to tak, že do vzorce (4.17) dosadíme vztah pro relativní rychlost (4.16), kde jednotlivé rychlosti vypočteme podle vzorce (4.8):

$$\begin{aligned} & v_2^{AB} \cdot n = -e * v_1^{AB} \cdot n \\ & \left(v_2^{AP} - v_2^{BP}\right) \cdot n = -e * \left(v_1^{AP} - v_1^{AB}\right) \cdot n \\ & \left[\left(v_2^A + w_2^A \times r_{AP}\right) - \left(v_2^B + w_2^B \times r_{BP}\right)\right] \cdot n = -e * \left[\left(v_1^A + w_1^A \times r_{AP}\right) - \left(v_1^B + w_1^B \times r_{BP}\right)\right] \cdot n \end{aligned} \tag{4.18}$$

a konečně výsledné rychlosti nahradíme vzorci (4.12) a (4.13):

$$\begin{aligned} & \left[\left(v_1^A + \frac{J}{M_A} \cdot n + \left(w_1^A + \frac{r_{AP} * J}{I_A} \cdot n\right) \times r_{AP}\right) \cdot n\right] - \\ & \left[\left(v_1^B + \frac{J}{M_B} \cdot n + \left(w_1^B + \frac{r_{BP} * J}{I_B} \cdot n\right) \times r_{BP}\right) \cdot n\right] = \\ & = -e * \left(v_1^A + w_2^A \times r_{AP}\right) \cdot n \end{aligned} \tag{4.19}$$

Z tohoto vztahu vyjádříme impuls síly J:

$$J = \frac{-(1+e) * v_1^{AB} * n}{\frac{1}{M_A} + \frac{1}{M_B} + \left[\left(I_A^{-1} * (r_{AP} \times n)\right) \times r_{AP} + \left(I_B^{-1} * (r_{BP} \times n)\right) \times r_{BP}\right] * n} \tag{4.20}$$

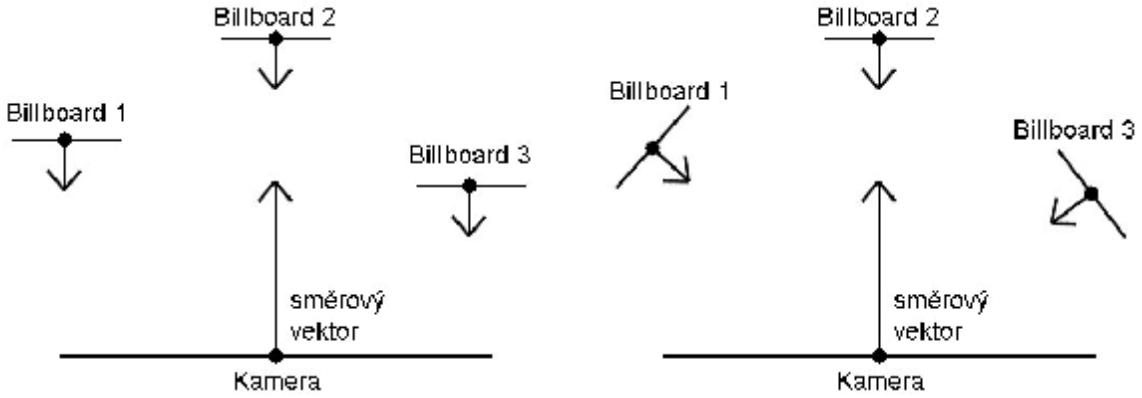
4.3. Exploze

Exploze je řešena pomocí techniky zvané billboarding [21]. Technika billboardingu mění orientaci billboardu (tj. objekt, na který je promítaná textura) tak, aby byl umístěn čelně k cíli, obvykle ke kameře. Tato technika je poměrně populární ve hrách a v dalších aplikacích, které využívají velké množství polygonů. I přes neustálý vývoj grafických karet narůstá potřeba zpracovávat stále větší počty polygonů. Billboarding umožňuje tento počet snížit použitím triku (tzv. finty) s vhodnou texturou.

Rozlišujeme dva typy billboardingu, cylindrický a sférický. Ve sférické verzi není žádný problém s orientací objektu. Zato v cylindrickém přístupu je rotace objektu svázána s vektorem rotace, a to obvykle v pozitivním směru osy Y. Sférický přístup se uplatňuje zejména u her, kde je možnost rotovat kamerou ve všech osách, např. vesmírný simulátor. Cylindrický přístup naopak ve hrách, kde dochází k pohybu po jedné rovině.

Existuje několik technik billboardingu. Podvodné (cheating), které jsou rychlé nebo jednoduché, ale neposkytují pravý billboarding, a pravé, které umožňují nastavení billboardu čelně ke kameře nebo jinému libovolnému objektu.

Podvodné metody natočí billboard tak, že normálový vektor billboardu je shodný s invertovaným vektorem kamery (viz. obr. 4.5). Pravé metody natáčí billboard kolmo na kameru (viz. obr. 4.6).


Billboard 2
Billboard 1
Billboard 3
směrový
vektor
Kamera
Billboard 2
Billboard 1
Billboard 3
směrový
vektor
Kamera

Obr. 4.5, Obr. 4.6

5. Použité nástroje

Cílem tohoto diplomového projektu je naimplementovat algoritmy pro detekci a vyhodnocování kolizí na základě fyzikálního modelu v jednoduché virtuální scéně a vytvořit pomocný program pro výpočet potřebných fyzikálních veličin.

Pomocný program pro výpočet veličin je implementován v objektově orientovaném jazyce C++ [22, 23] s využitím grafického nástroje OpenGL s použitím knihovny GLUT (The OpenGL Utility Toolkit) [24]. Další použitou knihovnou je knihovna ColDet [25].

Algoritmy pro detekci a vyhodnocování kolizí jsou demonstrované ve více aplikacích, nicméně všechny aplikace s těmito algoritmy byly implementovány v objektově orientovaném jazyce C++ ve vývojovém prostředí Visual Studio.net za použití grafické knihovny Open Inventor [26, 27]. Použité textury a modely byly vytvořeny v modelovacím studiu 3D Studio Max. Tyto modely jsou zjednodušenými verzemi modelů poskytnutých dříve pro ročníkový projekt Knihovna pro detekci kolizí mezi objekty scény.

Více o programovacím jazyce C++ je uvedeno v citované literatuře [22, 23], pro bližší seznámení s grafickou knihovnou OpenGL postačí dostupné referenční manuály.

Nyní si více přiblížíme pouze dva použité nástroje, které nejsou všeobecně používané a známé, a to knihovnu ColDet a grafickou knihovnu Open Inventor.

5.1. Knihovna ColDet

Tato knihovna slouží pro detekci kolizí na trojúhelníkové úrovni. Umožňuje provádět detekce kolizí mezi jednotlivými tělesy, mezi tělesem a paprskem, i mezi tělesem a koulí. Dále umí vyhodnotit trojúhelníky a bod srážky.

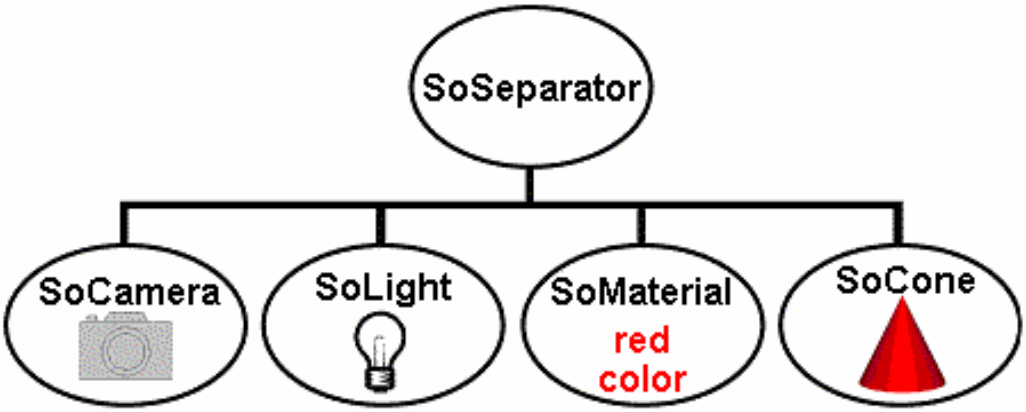
5.2. Knihovna Open Inventor

Mimo dobře zpracovaného manuálu [27] o knihovně Open Inventor existuje i článek na serveru www.root.cz:

Open Inventor je knihovna napsaná v C++ a postavená nad OpenGL, která posunuje programátora od primitivního OpenGL rozhraní na vyšší úroveň a nabízí mu rozsáhlou množinu C++ tříd. Ta podstatně zjednodušuje práci programátora a dokonce často poskytuje vyšší výkon než přímá implementace v OpenGL. Vyšší výkon je možný díky jistým optimalizacím, které Open Inventor může provádět nad daty scény. Běžný programátor také obyčejně nemá čas provádět profilování a optimalizaci renderovacích algoritmů. Proto již vyprofilované rutiny Inventoru nejsou špatnou volbou.

Design Open Inventoru vychází z konceptu grafu scény. Tedy, scéna je složena z uzlů - anglicky nodes. Nody jsou různých typů. Jedny nesou informace o geometrii těles (krychle, kužel, model tělesa), další různé atributy (barva, textury, souřadnice objektu) a také existují speciální nody, které obsahují seznam jiných nodů, anglicky zvané groups. A právě tyto groupy umožňují organizovat ostatní nody do hierarchických struktur zvaných grafy. Takovýto graf nám pak reprezentuje naši scénu. Celou problematiku si můžeme hned ukázat na prvním příkladu.

V tomto příkladu vytvoříme minimální aplikaci zobrazující červený kužel osvětlený jedním světlem. Graf scény, který budeme vytvářet, vypadá takto:



Kořen grafu tvoří objekt typu SoSeparator. Předpona So (=Scene Object) se používá pro zabránění kolizím jmen ve vašem projektu s názvy tříd v Open Inventoru. SoSeparator je třída odvozená ze SoGroup, tedy základní třídy udržující seznam jiných nodů. Třídu SoSeparator budeme používat místo SoGroup téměř vždy pro její speciální vlastnosti. Například proto, že dokáže scénu pod sebou předkompilovat do OpenGL display listu a tím urychlit proces renderování. Když se podíváme pod separátor, zjistíme, že má čtyři syny: kamera, světlo, materiál a kužel. První z nich - kamera - je speciální nod, který určuje umístění pozorovatele a některé další atributy pohledu do scény. Světlo (SoLight) osvěcuje scénu bílým světlem. Následující nod, tedy materiál, udává optické vlastnosti kužele, jednoduše řečeno - udává jeho barvu. Posledním nodem je pak vlastní kužel, což je nod specifikující geometrii tělesa. Text převzat z [26].

6. Pomocná aplikace - Veličiny

V pomocné aplikaci jsou naprogramovány algoritmy pro výpočet stálých fyzikálních veličin potřebných pro vyhodnocení kolize na základě fyzikálního modelu. Počítanými veličinami jsou těžiště tělesa a matice momentu setrvačnosti (inertia tensor). Aplikace, mimo těchto veličin, počítá pro zadaný model i jednoduchá obalová tělesa jako je koule (sphere) či osově orientované obalové těleso (AABB). Ovládání pomocné aplikace a náhled jsou přiloženy v příloze I. Ovládání pomocné aplikace a screenshot.

Při výpočtu těžiště je využit složitější algoritmus popsaný v kapitole 2.2.1. Algoritmy pro výpočet těžiště. Obdobně pro výpočet matice setrvačnosti je použit algoritmus z kapitoly 2.3.2. Algoritmy pro výpočet matice setrvačnosti. Oba algoritmy si jsou podobné, resp. oba musí „projít" objemem celého tělesa. Víme, že nelze počítat obě veličiny (těžiště, matici momentu setrvačnosti) v jednom průchodu, protože algoritmus na výpočet matice setrvačnosti musí vycházet z modelu s osou souřadnic umístěnou v těžišti. Proto je celý průchod tělesem realizován dvakrát.

6.1. Načtení a uložení modelu WRL

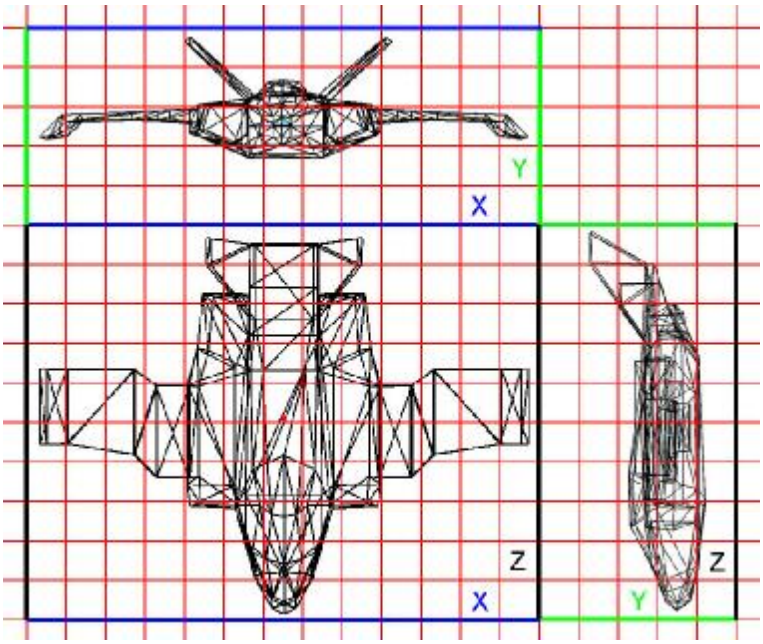
Všechny používané modely jsou načteny ze souboru *.wrl, kde je model uložen podle standartu „VRML V2.0 utf8" [28]. Tento soubor je načten standartními funkcemi jazyka C++ a zpracován tak, aby mohl byt použit v knihovně ColDet pro rozdělení modelu na voxely. WRML standart uchovává trojúhelníkový model jako pole vrcholů trojúhelníků a pole indexů do tohoto pole vrcholů. Knihovna ColDet naopak svůj trojúhelníkový model reprezentuje přímo sledem jednotlivých trojúhelníků. Proto je nutné model překonvertovat z jednoho uložení do druhého, a to načtením vrcholů s postupným indexováním a ukládáním do třídy knihovny ColDet. Při načítání všech vrcholů jsou zjištěny rovněž maximální možné souřadnice modelu (tj. primitivní obalové těleso), které se použijí při tvorbě voxelové mřížky.

Při ukládání modelu se ukládají souřadnice tak, aby počátek souřadnic ležel v těžišti modelu. Stejné souřadnice musí být použity i pro výpočet matice momentu setrvačnosti. Souřadnice se proto přepočítávají již pro algoritmus matice momentu setrvačnosti, ale do souboru se ukládají až po ukončení obou výpočtů. A opět se musí převést do standardu WRML. Společně se změněnými souřadnicemi se na konec souboru uloží i dodatečné informace ve standardu WRML Info, které obsahují matici setrvačnosti a jednoduchá obalová tělesa. Tato obalová tělesa se vypočítávají přímo při ukládání souřadnic. Pro obalové těleso koule stačí najít maximální vzdálenost vrcholu od počátku souřadnic a pro obalový kvádr minimální a maximální hodnotu v každé ze souřadnic.

6.2. Rozdělení modelu na voxely

Velkým problémem při aplikaci obou zmíněných algoritmů je rozdělení trojúhelníkového modelu na jednotlivé voxely. Vhodným řešením je například rozdělit celý prostor, vymezený obalovým tělesem modelu, na jednotlivé voxely pomocí mřížky s danou přesností, resp. s daným

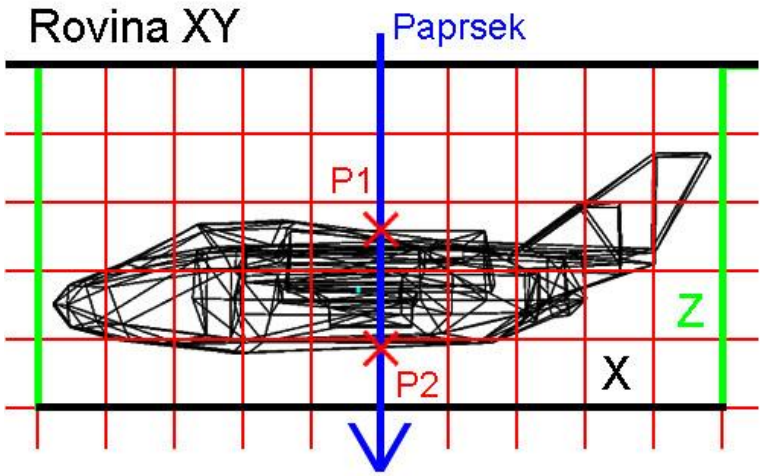
počtem dělících ploch. Rozdělení modelu pomocí mřížky je zobrazeno na obrázku 6.1, a to ve všech třech rovinách (XY, YZ, XZ).



Obr. 6.1

Algoritmus, který prochází celým objemem tělesa, využívá mřížky a cyklí se přes všechny tři souřadnice. Prvním z řešených problémů je určení, zda se má bod, daný aktuálními souřadnicemi, započítat či nikoli, tedy zda se nachází uvnitř nebo vně testovaného tělesa. Jednou z možností řešení je „spustit“ paprsek přes jednu rovinu (ve směru jedné z os) a zjišťovat kolize tohoto paprsku a modelu. Zde přichází na řadu knihovna ColDet a její funkce pro zjišťování kolize tělesa a paprsku.

Při „spuštění“ paprsku přes model dojde při první (liché) kolizi k vniknutí do tělesa a při druhé (sudé) kolizi dojde k opuštění tělesa. Spuštění paprsku z roviny XY přes osu Z znázorňuje obrázek 6.2, a to včetně vstupního průsečíku P1 a výstupního P2. S výslednými průsečíky lze požadovaného zjištění, zda se nacházím uvnitř nebo vně tělesa, dosáhnout více cestami. Záleží na způsobu použité optimalizace nebo konkrétním modelu. Například lze jednoduše porovnávat souřadnice a jako vyhovující zvolit jen ty souřadnice mřížky, které se nacházejí mezi průsečíky (na obrázku 6.2 by takto vyhovovaly dvě souřadnice mřížky). Druhým způsobem je „přitáhnout“ (zaokrouhlit) průsečík na nejbližší souřadnici mřížky. Poté se počítá se souřadnicemi mřížky mezi těmito body, a to buď včetně nebo kromě nich. Tím zde odpadá porovnávání (na obrázku 6.2 by vyhovovaly 3, resp. 1 souřadnice mřížky). V tomto diplomovém projektu je použita metoda přitažení počítající i s přitaženými souřadnicemi.

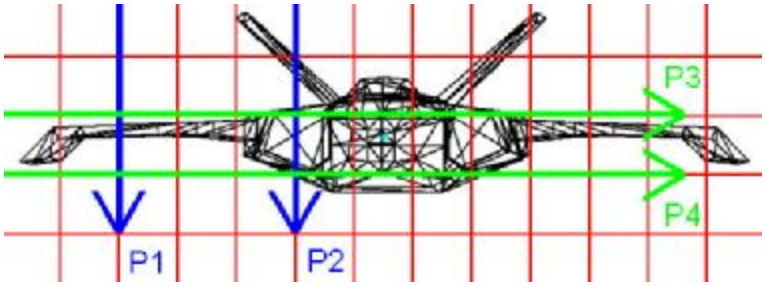


Obr. 6.2

6.2.1. Problém mřížky

Při používání „voxelové“ mřížky nastávají určité zásadní problémy, které je nutné řešit. Většina problémů se vztahuje k „tenkým“ částem modelu, jako jsou například křídla stíhačky apod.

Při průchodu paprskem dojde v takovém „tenkém“ místě k špatné detekci průsečíků nebo k vůbec žádné detekci, což záleží na zvoleném způsobu vyhodnocování průsečíků (viz. obr. 6.3). Pro modré paprsky (P1, P2, osa Y) v prvním případě vyhodnocení nedojde k nalezení žádného bodu mřížky (mezi průsečíky není žádný bod mřížky). V druhém případě dojde k chybnému vyhodnocení, protože oba průsečíky se přitáhnou ke stejné (horní) mřížce. Nastane taková situace, že bude lichý počet průsečíků (tento problém bude vysvětlen níže viz kapitola 6.2.2. Problém lichého počtu průsečíků). Konkrétně v případě paprsku P1 dojde k chybnému vyhodnocení na objemu křídla či u paprsku P2 dojde k chybnému vyhodnocení na trupu stíhačky, který se překrývá s ocasním křídlem. Pro zelené paprsky (P3, P4, osa X) nedojde k vyhodnocení objemu u křídel vůbec, protože mřížka je širší než tloušťka křídla.



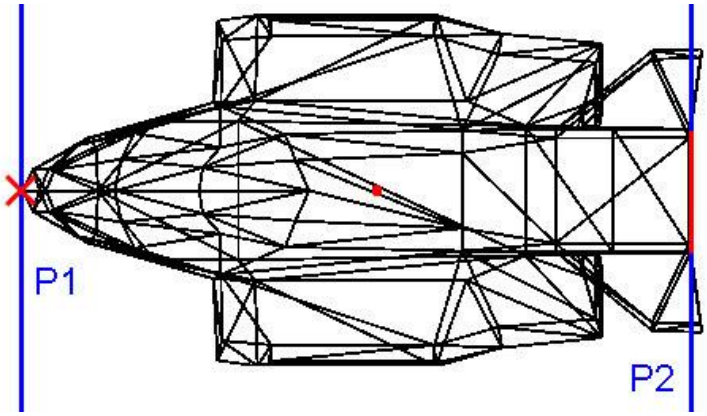
Obr. 6.3

V reálných situacích je mřížka dostatečně drobná, aby byly tyto tenké plochy započítávány, ale i přesto dochází na těchto tenkých plochách k nepřesnostem a u extrémně tenkých ploch i k chybám. Nepřesnosti jsou způsobeny malým počtem započítávaných souřadnic mřížky.

Chyby vzniklé problémem tenkých částí se dají částečně eliminovat pouštěním paprsku přes vhodnou osu nebo pouštěním paprsků postupně přes všechny osy s následným zprůměrňováním výsledků. V aplikaci je možné si zvolit přesnost právě přidáním dalšího průchodu (pouštěním paprsku) přes další jinou osu.

6.2.2. Problém lichého počtu průsečíků

Dalším problémem, který může při detekování kolizí nastat, je situace, kdy spouštěný paprsek narazí na hranu modelu nebo na vrchol modelu (viz. obr. 6.4).



Obr. 6.4

Paprsek P1 narazil na vrchol modelu a paprsek P2 na hranu, která je rovnoběžná s paprskem. Oba dva případy mají za následek lichý počet průsečíků (P1 jeden průsečík, P2 tři průsečíky), podobně jako v případě chyby s přitáhnutím na mřížku. Tento problém nelze vyloučit, lze jej však eliminovat na základě následujícího předpokladu.

Při tvorbě modelu dochází k přitažení (zaokrouhlení) také na určitou mřížku přesnosti. K tomuto zarovnávání může docházet z více důvodů. Například u některých jednoduchých modelů, kde je přesnost uvedena v celých číslech.

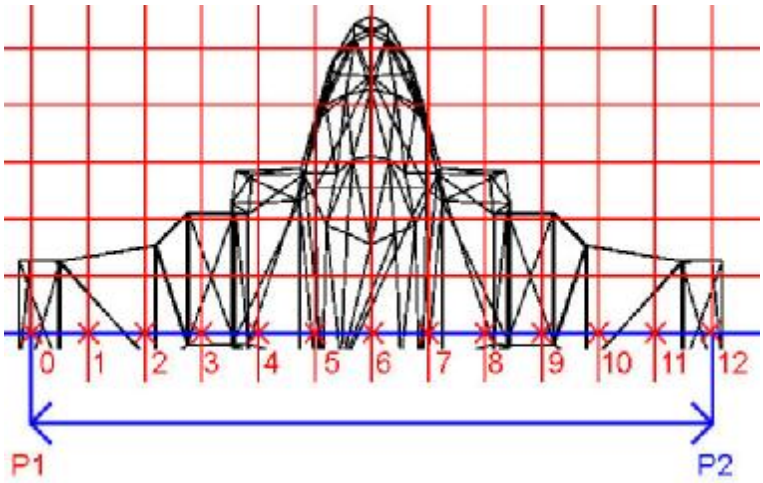
Proto můžeme naší voxelovou mřížku záměrně „posunout“ např. o 1/3 velikosti mřížky, a tím se vyhnout průchodům v celých číslech, resp. shodám s mřížkou přesnosti modelu. Pokud i přesto k chybě způsobené lichým počtem průsečíků dojde, je třeba ignorovat celý objem tohoto konkrétního paprsku, protože nejsme schopni určit, které části jsou uvnitř tělesa a které vně.

Právě problém shody souřadnic modelu a voxelové mřížky vedl v tomto projektu k velkému počtu chyb způsobených lichým počtem průsečíků. Lepších výsledných hodnot aplikace dosahovala až při jemnějším kroku (přesnější mřížce), který vedl k dlouhé době výpočtu. Po zavedení optimalizace „posunutí“ mřížky je již procentuální chyba způsobená lichým počtem průsečíků zanedbatelná (viz. analýza v kapitole 6.4. Problém dostatečné přesnosti, analýza).

6.3. Optimalizace průchodu přes objem

Zmíněný algoritmus hledání těžiště (viz. kapitola 2.2. Těžiště tělesa) lze optimalizovat díky použití metody „pouštění“ paprsku. Při korektním průchodu paprsku objemem tělesa získáme dva průsečíky (sudý počet). Standardně se sčítají všechny souřadnice mřížky od lichého (vstupujícího) průsečíku až do sudého (vystupujícího) průsečíku, resp. se sčítají všechny souřadnice mřížky mezi těmito průsečíky.

Optimalizace spočívá v současném sečtení celého objemu tělesa v části paprsku, která je uvnitř modelu. Pokud procházíme objemem například přes rovinu YZ, tzn. paprsek spouštím v ose X, budeme provádět jednoduchý součet lineární řady s X-ovou souřadnicí. Sčítáme totiž hodnoty souřadnic (x+0,y,z), (x+1,y,z), (x+2,y,z) … (x+12,y,z), což přestavuje lineární řadu. Souřadnice Y a Z se v daném průchodu nemění, stačí je tedy jen vynásobit počtem vyhovujících voxelů. Situace je znázorněna na obrázku 6.5.



Obr. 6.5

Pro každý úsek paprsku, který je uvnitř tělesa, můžeme provést následující optimalizaci (uvedeno v pseudokódu jazyka C), která nahradí poslední for cyklus v použitém algoritmu z kapitoly 2.2.1.:

```
For (pro každou část paprsku uvnitř tělesa){
        N=počet vyhovujících voxelů (uvnitř tělesa)
        Teziste.X= Teziste.X+ N/2 * (Prusecik1.X+Prusecik2.X)
        Teziste.Y= Teziste.Y+Mrizka.Y*N
        Teziste.Z= Teziste.Z +Mrizka.Z*N
}
```

Optimalizaci založenou na stejném principu (součet lineární řady) lze využít i u algoritmu pro výpočet matice momentu setrvačnosti, kde ve vzorci (2.28), resp. (2.22) můžeme součet řady uplatnit na deviační členy (I_{XY}, I_{YX}, I_{XZ}, I_{ZX}, I_{YZ}, I_{ZY}). Členy opačné (v matici symetrické) se v algoritmu nepočítají vůbec, ale přiřadí se jako opačné až na konci výpočtu. Pro hlavní členy

momentu setrvačnosti (I_{XX} , I_{YY} , I_{ZZ}) tato optimalizace uplatnit nelze, ve vzorci se totiž vyskytuje druhá mocnina a sčítaná řada v jedné ose již není ani lineární ani geometrická.

6.4. Problém dostatečné přesnosti, analýza

Oba výše zmíněné problémy (viz. kapitola 6.2.1. Problém mřížky a 6.2.2. Problém lichého počtu průsečíků) se dají řešit zpřesněním používané mřížky. I když při zjemňování mřížky roste počet lichých průsečíků, protože se častěji narazí na hranu, výsledná procentuální chyba je však menší. Se zjemňováním mřížky také roste násobná chyba při výpočtu matice momentu setrvačnosti, a to kvůli kvadrátu proměnných (2.27).

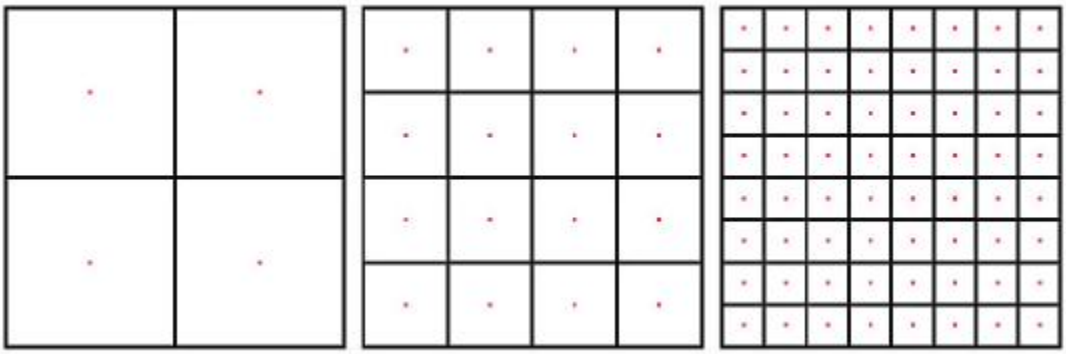
Dostatečnou přesnost stanovujeme vždy na konkrétním modelu, a to pomocí analýzy tohoto modelu. Přesnost mřížky lze určit odhadem u jednoduchých či specifických těles na základě předchozích zkušeností. V této diplomové práci byla provedena analýza nejjednoduššího modelu krychle, jednoduchého modelu jehlanu i nejsložitějšího modelu stíhačky. Pro každý model bylo provedeno dostatečné množství analýz. Na základě provedených analýz lze vyvodit závěry pro dostatečnou přesnost kroku.

6.4.1. Analýza – hrubá mřížka

Krychle je jediné těleso, u kterého lze jednoduše vypočítat těžiště i matici momentu setrvačnosti. V této práci má použitá krychle stranu o rozměru 500 jednotek. Pokud model krychle umístíme fyzicky do středu souřadné soustavy (vrcholy budou na souřadnicích 250 a -250) musí korekce těžiště vycházet (0, 0, 0), tedy těžiště se nebude posunovat. Pro výpočet matice momentu setrvačnosti použijeme vzorec (2.27). U krychle jsou však všechny strany stejně dlouhé, proto se výpočet zjednoduší:

$$I_{XX} = I_{YY} = I_{ZZ} = \sum_i m_i (2 * a_i^2) \qquad (6.1)$$

Pokud tento vzorec budeme postupně aplikovat na zjemňující se mřížku, zjistíme, že výsledky se v malých krocích velmi liší. V příloze II. Analýza krychle je uvedena analýza pro velmi hrubou mřížku s těžištěm tělesa v souřadnicích (0, 0, 0) za použití programu Veličiny, který počítá pouze matici momentu setrvačnosti. Je zde také vidět, že vždy pouze v jedné ose vznikla velká chyba, způsobená špatným upnutím na mřížku v libovolné z os. Obrázek 6.6 ilustruje použité velikosti jednotlivých mřížek krok = 250 jednotek, krok = 125 jednotek, krok = 62,5 jednotek. Červený bod přestavuje souřadnice středu mřížky.



Obr. 6.6

6.4.2. Analýza – časová náročnost

Druhá provedená analýza krychle je zaměřená především na přesnost a časovou náročnost výpočtu (viz. Příloha III. Analýza krychle). Zde je zřetelné, že přesnost výpočtu se stále mění. To platí hlavně pro matici setrvačnosti, protože se stále mění souřadnice těžiště, na kterém závisí výpočet. Od velikosti kroku 0.8333 (model je rozdělen na 600 dílků podle jedné z os) se již nemění poměr na hlavní diagonále v matici setrvačnosti, což je zásadní vlastnost počítané matice. Těžiště je určeno s přesností jedné desetiny jednotky, což je pro účely této práce také dostačující. Tato analýza však trvala zhruba 35 minut.

Obdobně byla provedena časová analýza pro model jehlanu a model stíhačky (viz. Příloha IV. Analýza jehlanu, Příloha V. Analýza stíhače). Jak je vidět na výsledcíh provedených analýz, se zmenšující se mřížkou roste přesnost výpočtu, ale také roste násobná chyba. Tato je markantní především u těles, které dobře vyplňují testovaný obalový box. Pro příklad u nejkomplikovanějšího modelu je dostačující přesnost cca 0,5 jednotky, kdy je model rozdělen zhruba na 600 dílků obdobně jako u krychle.

7. Implementace - Aplikace

7.1. Rozdělení aplikací

Aby bylo možno vyhodnocovat naimplementované algoritmy, je nutné vytvořit prostředí nebo-li virtuální scénu (demo aplikaci). Virtuálních scén bylo pro účely tohoto diplomového projektu vytvořeno několik a to tak, aby dobře demonstrovaly naimplementované algoritmy a jednoduchý herní engine.

Tři ze čtyř vytvořených aplikací mají čistě demonstrativní úkol, slouží jen k předvedení funkčnosti kolizních a vyhodnocovacích algoritmů. Všechny tři jsou umístěny do krychle, ve které se pohybují a kolidují demonstrační objekty. První aplikace, nazvaná Box – cube, obsahuje ty nejjednodušší tělesa, kvádry. Druhá aplikace, nazvaná Box – object, obsahuje různé modely jednoduchých těles. A konečně poslední aplikace, nazvaná Box – fighter, obsahuje pouze jeden složitý model stíhačky.

Hlavní aplikace, nazvaná Průlet tunelem, je demonstrací jak kolizních a vyhodnocovacích algoritmů, tak všech ostatních funkcí. Vyhodnocení kolizí je řešeno stejnými algoritmy jako v předchozích příkladech, navíc je přidáno řešení pomocí exploze, které je realizováno vystřelením rakety. Všechny aplikace se skládají se stejných programových modulů. Jelikož jsou v aplikaci použity objekty, jsou moduly rozděleny podle společných funkcí jednotlivých objektů. Hlavními moduly jsou:

- Main - hlavní modul, liší se v jednotlivých demonstračních aplikacích
- SgoManager - modul pro správu objektů, detekce a vyhodnocení kolizí
- SgObject - modul jednotlivých objektů
- SgInterface - modul obsahující funkce prostředí (ovládání, kamera)
- Billboard - modul obsluhující billboarding (explozi)

Návod k ovládání demonstračních aplikací a náhledy jsou součástí přílohy VI. Ovládání aplikace a screenshoty.

7.2. Prostředí

Samotné prostředí hlavní aplikace je tvořeno tunelem. Konfigurace tunelu je načítána z externího souboru a může být tedy editována. Tunel je tvořen jednoduchými statickými krychlemi, které obsahují statická i dynamická tělesa. Editace tunelu probíhá editací textového souboru s názvem tunel.txt, který je součástí datových souborů. Jak tento soubor editovat je podrobněji popsáno v příloze VII. Tvorba tunelu.

Prostředí je osvětlováno standardními funkcemi Open Inventoru SoPointLight a SoSpotLight, pro které jsou vytvořeny vlastní třídy (viz. příloha VIII. SgInterface). Toto světlo je v prostředí použité jako globální, tedy osvětluje celou scénu a také je „přilepeno“ na stíhačku.

7.3. Objekty v prostředí

V projektu je implementovaná třída SgObject, která umožňuje vkládat v podstatě libovolné objekty do vytvořeného prostředí. Objekt je reprezentován velkým počtem atributů a použitých metod. Převážná část atributů je fyzikálních, vyjadřující statický nebo aktuální fyzikální stav tělesa. Ve fyzikálních atributech jsou uloženy používané veličiny, které byly podrobněji charakterizovány v kapitolách 2. Fyzika a 4. Vyhodnocení kolize. Detailní popis atributů a metod je v hlavičce funkce (viz. Příloha IX. SgObject). Některé klíčové metody budou více popsány níž v tomto textu.

Samotný model objektu je reprezentován předem vytvořeným trojúhelníkovým modelem (resp. texturou), který se načítá ze souboru s modelem. V implementaci je pro uložení modelu použita třída Open Inventoru SoNode. V tomto projektu jsou použity konkrétní modely geometrických těles a stíhačky. Je možné použít další libovolné modely, ale jejich použití v ukázkovém přikladu není zapotřebí. Seznam použitých modelů, spolu s autorskými právy k těmto modelům, je uveden v souboru LICENSE, který je součástí přiložených datových souborů.

Většina reálných modelů je pro účely detekce kolizí příliš složitá, a proto se ve většině aplikací používají modely pouze dva, jeden model vykreslovací a druhý jednodušší kolizní model. V demonstrační aplikaci je zbytečné pracovat s oběma typy modelů. Proto se používají jen kolizní modely a vzniklé situace jsou pak reálné. Používání dvou kolizních modelů je naimplementované a je tudíž možné doplnit aplikaci o propracované modely. Tvorba složitých modelů, resp. textur, by zabrala příliš velké množství času a nepřinesla by žádné funkční zlepšení. Navíc tvorba modelů a textur není součástí zadání této diplomové práce.

Objekty v prostředí spravuje třída SgoManager, která zajišťuje pohyb a vzájemnou interakci všech objektů scény. V této třídě jsou také implementovány klíčové algoritmy pro detekci a vyhodnocování kolizí. Třídy SgObject a SgoManager tvoří stěžejní třídy projektu, obě budou detailněji popsány v následujícím textu.

7.4. Pohyb v prostředí

V každé scéně je potřeba zajistit patřičný pohyb kamery nebo umožnit pohled z objektů ve scéně již umístěných (např. stíhačky) po daném prostředí.

Obecně je v aplikaci implementováno prostředí s nastavitelným odporem prostředí (třecí síla). Tento odpor prostředí se projevuje jako síla působící na těleso. Ve třídě SgObject je atribut uchovávající působící síly, proto je jednoduché přidat libovolnou globálně působící sílu jako například vítr či gravitaci, která ovlivňuje všechny tělesa.

7.4.1. Kamera

V projektu je použita perspektivní kamera, realizovaná v grafické knihovně Open Inventor pomocí třídy `SoPerspectiveCamera`, která nastavuje transformační matici OpenGL. Pro tuto kameru je v projektu vytvořena speciální třída v modulu SgInterface, která umožňuje lépe pracovat s kamerou, oproti původní třídě, a nastavovat potřebné parametry (např. výška a vzdálenost od pozorovaného tělesa). Detailní popis třídy je přiložen v podobě komentářů v příloze VIII. SgInterface.

V průběhu implementace a testování bylo zjištěno, že je nutné nejprve nastavit orientaci kamery a teprve poté kamerou posunout. Při obráceném pořadí funkcí by se kamera „kousala“ a takový efekt není žádoucí.

7.4.2. Ovládání

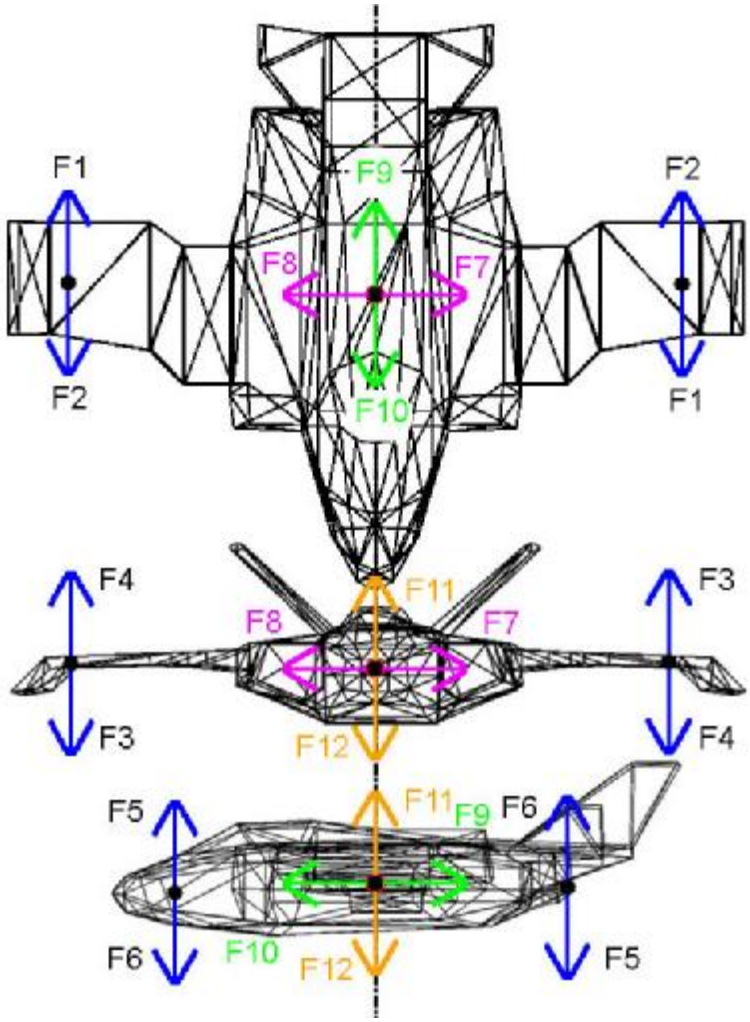
Pohyb v prostředí nemůže být jen náhodný, proto by měl být ovládaný uživatelem. Z tohoto důvodu jsou parametry třídy kamery nastavovány podle reakcí uživatele. V operačním systému Win32 jsou podněty z periferních zařízení předávány aplikaci pomocí událostí, jejichž obsluhu provádí třída SoMouseButtonEvent pro obsluhu myši a třída SoKeyboardEvent pro obsluhu klávesnice. Tyto třídy Open Inventoru mají v diplomovém projektu opět vytvořenu vlastní třídu utvořenou na míru pro účely demonstrační aplikace. Tyto třídy jsou implementovány v modulu SgInterface, a to třída SgMouse pro práci s myší a třída SgKeyboard pro práci s klávesnicí (viz. příloha VIII. SgInterface).

Pro lepší pohyb ve scéně je použito ovládání myší, z tohoto důvodu je nutné použít platformě závislé rozhraní WinAPI k ovládání kurzoru. Pro práci s kurzorem byly použity funkce GetCursorPos a SetCursorPos. Pro zlepšení ovládání se v každém snímku vrací kurzor do středové pozice okna.

7.4.3. Pohyb

Pro pohyb ve scéně jsou implementovány metody zajišťující pohyb i rotaci ve všech osách volnosti. Vzhledem k původnímu vesmírnému prostředí (použitém v ročníkovém projektu) a záměru vytvořit jednoduchý herní engine, je ovládání implementováno ve více variantách. Jednou z variant je ovládání ruční, které bere v úvahu všechny aspekty setrvačnosti tělesa. Druhé ovládání využívá funkce „auto pilota“ a snaží se stabilizovat stávající pohyb. V projektu je implementováno i třetí ovládání, a to absolutní změna polohy. Toto ovládání nepodporuje vyhodnocování kolizí, proto není v demonstrační aplikaci povoleno a slouží spíše pro testovací účely.

Otáčení a pohyb v prostoru je možné řešit mimo třídu SgObejct a nastavovat orientaci přímo pomocí funkce setOrientation. Nebo lze využít funkcí naimplementovaných ve třídě, které umožňují absolutní nebo relativní rotaci a pohyb ve všech osách. Pojmem absolutní rotace označujeme v tomto projektu rotaci, při které uživatel stiskne patřičnou klávesu a objekt rotuje, resp. pohybuje se po dobu stisknutí klávesy konstantní rychlostí, tzn. mění absolutně svoje natočení. Pojmem relativní rotace, resp. pohyb, označujeme rotaci, která je založená na fyzikálním modelu, také by se dala nazvat fyzikální. Tato rotace/translace funguje tak, že při stisku klávesy začne působit tažná síla, resp. zrychlení, která uvede objekt do pohybu. Působící síla může způsobovat libovolné rotace v závislosti na směru a umístění síly. Objekt proto obsahuje 2*6 párů „virtuálních“ motorů, které otáčí a posunují objekt ve všech osách. V ideálním případě, pro detailní simulaci, by objekt měl mít na každém rameni osy 4 páry motorů (celkem 24 motorů). Všechny stupně volnosti zachycuje obrázek 7.1, kde modré vektory reprezentují síly způsobující rotaci a barevné síly v těžišti tělesa reprezentují síly působící translaci.



Obr. 7.1

Všechny provedené změny rotace a translace je nutné uložit do uzlu Open Inventoru SoMatrixTransform, a tím je aplikovat na celý objekt. Tato transformační matice obsahuje polohu (translaci), natočení (rotaci) a měřítko (scale) daného objektu. Uložení zajišťují výše popsané funkce pro absolutní a relativní změnu rotace či translace. V popisu hlavičky třídy SgObject (viz. Příloha IX. SgObject) mají funkce pro absolutní rotaci název Rot*, pro relativní rotaci mají název Torque* a pro editaci pohybu setPos, případně updPos.

Integrace veškerých pohybových veličin (tzn. setrvačnost pohybu) i pomocných veličin každého tělesa je prováděna pomocí metody TimeStep třídy SgObject, kde se aktualizuje pozice a rotace tělesa. Dále se vyhodnocují působící síly a přepočítává se matice setrvačnosti podle aktuálního natočení tělesa.

7.5. Správa objektů

Správa objektů scény je realizována hlavní třídou SgoManager, tzv. správcem objektů. Tato třída obsahuje klíčové algoritmy detekce a vyhodnocování kolizí. Vybraným klíčovým algoritmům je

věnována samostatná kapitola a budou rozebrány později. Hlavička třídy SgoManager je uvedena v příloze X. SgoManager.

Třída SgoManager obsahuje základní funkce pro práci s objekty třídy SgObject. Objekty jsou uloženy v seznamu objektů (třída Open Inventoru SbList). Nad tímto seznamem jsou implementovány operace přidávání a odebírání jednotlivých objektů. Metody v třídě SgoManager lze rozdělit na následující skupiny:

- metody pro detekci kolizí
- metody pro vyhodnocení kolizí
- metody pro pomocné vypočty
- metoda timeStep, zajišťující chod aplikace, integraci veličin a volání ostatních metod

V této práci metody použité pro detekci kolize jsou checkCollision, checkCollisionWith, checkTriangleCollision a intersectionCallback, které zajišťují rozdělení prostoru a detekování kolize, jak již bylo teoreticky popsáno v kapitole 3. Detekce kolizí. Metodami vyhodnocujícími kolize jsou resolveCollision, resolveCollisionBALL, resolveCollisionBODY a ExplosionMissile, které zaštiují vyhodnocení kolize na základě jednoduchého odrazu, fyzikálního odrazu či exploze (viz. kapitola 4. Vyhodnocení kolize).

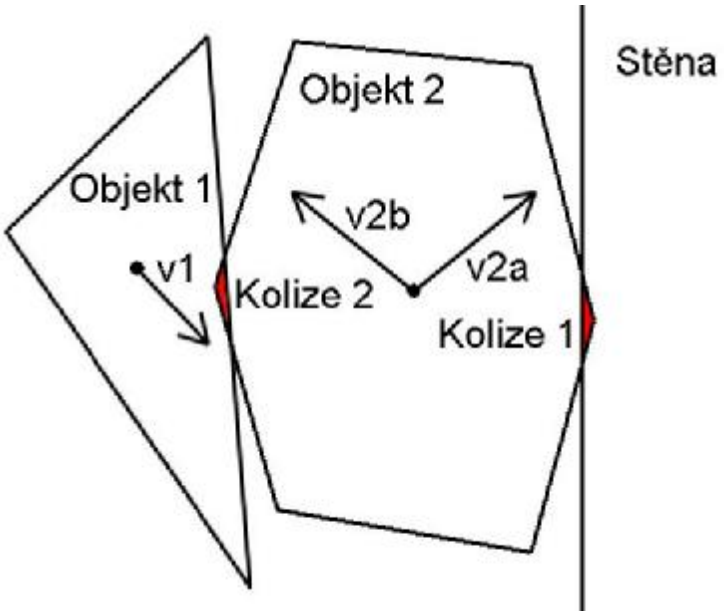
Pomocné metody zajišťují výpočty veličin nebo pomocné operace nad těmito veličinami. Jsou jimi checkID, computeTr, partOfTriangle, diffExp2, workIntersectionPoints, workIntersectionVertexs, getIntersectionPoint a computeNormal.

Metodám pro detekování a vyhodnocování kolize, včetně pomocných metod, jsou věnovány samostatné kapitoly (viz. kapitola 8. Implementace - Detekce kolize a 9. Implementace - Vyhodnocení kolize), proto si v této kapitole uvedeme pouze tu metodu, která obstarává samotnou správu objektů, tzv. metodu timeStep. Dále si také představíme problém spojený se změnou pozice a vyhodnocením kolize (viz. kapitola 7.5.1. Vyhodnocení pozice vs. detekce kolize).

Metoda timeStep obsluhuje všechny ostatní funkce v každém snímku. Pokud se ve scéně nachází běžící animace (exploze), tak ji obslouží, tzn. zavolá funkci timeStep pro běžící animaci. Dále volá funkce pro samotné detekce kolize, tedy checkCollision.

7.5.1. Vyhodnocení pozice vs. detekce kolize

Původně algoritmus řešil nejprve integraci pozice (vyhodnocení) všech objektů současně, a teprve poté zjišťovala, zda nedošlo k detekci kolize. Toto řešení je výhodné v tom, že postačí provést počet řešení rovný kombinaci bez opakování všech dvojic objektů. Nevýhodou je, že pokud dojde k posunutí více těles, u kterých byla zároveň detekována kolize, je relativně velká šance na vznik tzv. „zakousnutých“ objektů. K tomuto zakousnutí může dojít, protože nastane více kolizí současně, což znamená, že pokud nastanou u jednoho shluku objektů, dojde ke špatnému vyhodnocení kolize a možnému zaklínění těchto těles. Situaci ilustruje obrázek 7.2, kde kolidují dva objekty a stěna. Pokud dojde například nejprve k vyhodnocení kolize 1, pak kolize 2 se již počítá se špatnými hodnotami, které jsou způsobeny kolizí 1.



Obr. 7.2

Problém lze částečně eliminovat vyhodnocením pozice vždy jednoho objektu a okamžitým hledáním kolize s dalšími objekty. V tomto případě je ale potřeba provést test na kolizi u všech testovaných objektů mezi sebou, tzn. testovat každý s každým.

8. Implementace – Detekce kolize

Použité algoritmy vychází z teoretických základů uvedených v kapitole 3. Detekce kolizí. Hlavní funkcí, která je volaná z metody timeStep, je funkce checkCollision. Ta vždy aktualizuje pozici objektu (zavolá metodu timeStep, a to pro každý objekt) a provede test, zda právě posunutý objekt nekoliduje voláním metody chechkCollisionWith. Pokud dojde ke kolizi, volá i funkci pro vyhodnocení kolize resolveCollision.

8.1. Rozdělení prostoru

Rozdělení prostoru zajišťuje metoda checkCollisionWith. Aby nebyla detekce kolize tak náročná na využití zdrojů, je prostor rozdělen na jednotlivé podprostory. V našem konkrétním případě tyto podprostory tvoří stěny jednotlivých krychlí tunelu. Každá krychle tvoří jednu oblast. Dále každý objekt má svůj obal (viz. kapitola 3.3. Obalová tělesa) pro zamezení neustálé detekce na úrovni modelů (trojúhelníků). Použitými obalovými tělesy jsou koule a osově orientovaný box. Jako metoda dělení prostoru je zvolena metoda hierarchie obalových těles jedné úrovně. Použitá metoda je jednoduchá, ale v této konkrétní práci velmi účelná. Jako možnou optimalizaci lze seznam krychlí převést na jednoduchý prostorový strom a dělit tak prostor metodou stromu (viz. kapitola 3.4.3. BSP Strom).

8.2. Kolize obalových těles

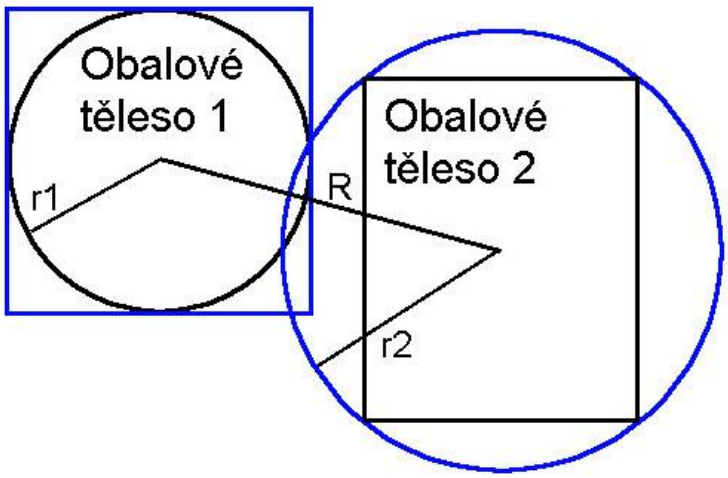
Kolize na úrovni obalových těles opět řeší funkce checkCollisionWith, která je v algoritmu použita dvakrát. Poprvé pro detekování v jakém prostoru se nacházím, a to mezi tělesy a referenčními krychlemi tvořícími tunel, a podruhé při samotné detekci kolize mezi objekty. S těmito optimalizacemi probíhá samotná detekce kolize jen u objektů ve stejném prostoru, a to na úrovni obalových těles. Při nalezení kolize obalů je na základě druhu kolidujících těles rozhodnuto o typu kolize a o případném pokračování detekce kolize na úrovni trojúhelníků. Kolize mezi obalovými tělesy mohou být trojího druhu:

- kolize koule vs. koule
- kolize box vs. box
- kolize koule vs. box

V případě kolize koule vs. koule je určení průniku (kolize) jednoduchou matematickou operací. Stačí vypočítat vzájemnou vzdálenost obou těles, a pokud je menší než součet poloměrů těchto těles, došlo ke kolizi. Toto je naznačeno na obrázku 8.1, kde r1 a r2 jsou poloměry obalových koulí a R je vzdálenost těchto koulí. Na tomto obrázku ke kolizi koulí došlo. Kolizi box vs. box lze provést postupným porovnáváním jednotlivých souřadnic BB. Celkem stačí provést 6 srovnání souřadnic:

Teleso1.Xmin < Teleso2.Xmax,	Teleso1.Xmax > Teleso2.Xmin
Teleso1.Ymin < Teleso2.Ymax,	Teleso1.Ymax > Teleso2.Ymin
Teleso1.Zmin < Teleso2.Zmax,	Teleso1.Zmax > Teleso2.Zmin

Jakmile alespoň jedno porovnání je pravdivé, došlo ke kolizi boxů. A nakonec, v posledním případě (box vs. koule), se dá vyřešit testování obalením samotných obalových těles. Tedy původní obalové těleso obalím do dalšího, a provedu jak detekci mezi koulemi tak detekci mezi boxy. Pokud nastala kolize v obou případech, původní obalová tělesa kolidují. Situaci znázorňuje obrázek 8.1, kde černé původní obalové těleso je obalené novým modrým. Na tomto obrázku ke kolizi původních obalových těles nedošlo, protože obalové boxy vzájemně nekolidují.



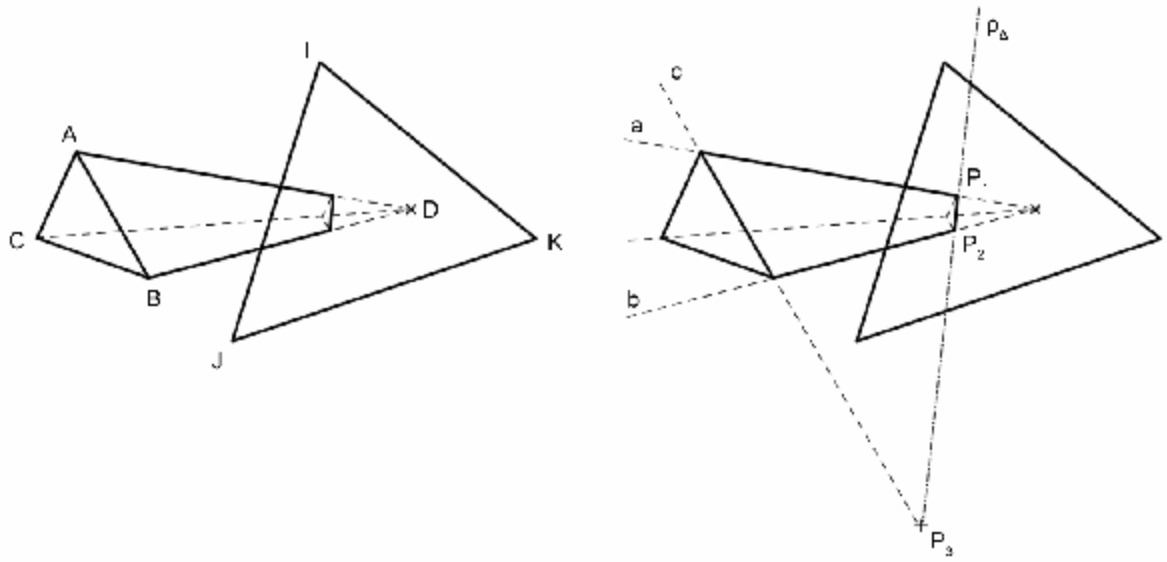
Obr 8.1

Podrobnější vyhodnocení kolize na úrovni trojúhelníků nastává jen v případě řešení kolize na základě fyzikálního modelu.

8.3. Kolize na úrovni modelů

V případě podrobnější detekce na úrovni trojúhelníků se volá metoda checkTriangleCollision , která vytvoří skrytou scénu, do níž se umístí kolidující modely. Na tuto skrytou scénu je poté aplikována funkce Open Inventoru SoIntersectionDetectionAction, která při kolizi na úrovni trojúhelníků modelů vyvolá „callback“ funkci intersectionCb, a tato následně kolizi obslouží potřebným níže popsaným způsobem. Po skončení detekce je skrytá scéna vyprázdněna a schována pro další použití.

Ve funkci intersectionCb se řeší uložení kolizních trojúhelníků a výpočet průsečíků těchto trojúhelníků. Tato funkce je postupně vyvolávána pro každou kolidující dvojici trojúhelníků. Právě pro tyto dvojice trojúhelníků hledáme jejich vzájemné průsečíky, které uložíme do seznamu kolizních bodů pro další zpracování. Průsečíky hledáme metodou podrobněji popsanou v kapitole 3.2 Kolize trojúhelník vs. trojúhelník.



Obr. 8.2, Obr. 8.3

Na obrázku 8.2 je zachycen nejjednodušší možný případ kolize, kdy několik trojúhelníků prvního tělesa pronikne do jednoho trojúhelníku druhého tělesa. Funkce intersectionCB vyhodnotí jako kolizní dvojice trojúhelníky (ABD, IJK), (ACD, IJK), (CBD, IJK). Z těchto dvojic poté vypočítáme průsečíky podle vzorce (3.5), resp. (3.6). U průsečíků musíme ověřit, zda opravdu leží v druhém trojúhelníku, a to aplikací vzorce (3.9). Pro konkrétní dvojici trojúhelníků ABD a IJK jsou pak vyhovující průsečíky P_1 a P_2 (viz. obr. 8.3).

9. Implementace – Vyhodnocení kolize

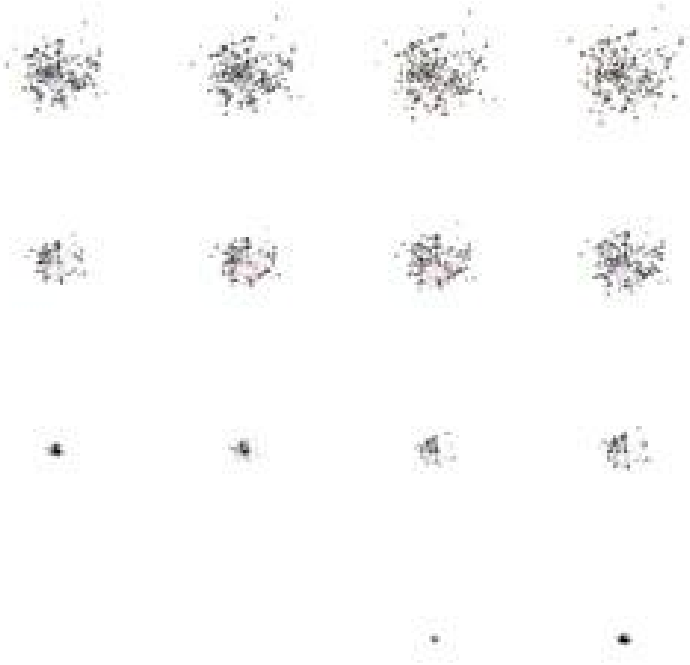
Poté, co je detekován a určen typ kolize kolidujících těles, může dojít k jejímu vyhodnocení. Vyhodnocení kolize je v této diplomové práci provedeno třemi, resp. pěti, metodami. Jsou jimi vyhodnocení jednoduchého odrazu (koule vs. pevné těleso, koule vs. koule), vyhodnocení fyzikálního odrazu (těleso vs. pevné těleso, těleso vs. těleso) a vyhodnocení explozí. Vyhodnocení srážek provádí metoda resolveCollision, která ošetří kolizi s kamerou a zavolá metody pro vyhodnocení jednotlivých druhů kolizí. Kamera je speciálním objektem, pro který se kolize v případě volné kamery vyhodnocují a v případě vázané kamery naopak nevyhodnocují.

9.1. Exploze

Kolize, která se má vyhodnotit explozí, je vyhodnocována funkcí ExplosionMissile a to tak, že nechá raketu explodovat. Prvním úkonem ošetřeným těsně před samotnou explozí je odebrání explodujícího tělesa ze seznamu objektů scény, případně přesměrování kamery na jiný objektv situacích, kdy byla kamera upnuta právě na explodovaný objekt. Následně se do scény umístí animovaný billboard s explozí na původní místo explodujícího objektu (hlavička třídy Billboard viz. Příloha XI. Billboard). Správce scény (sgoManager) má k dispozici svůj seznam již existujících billboardů (existuje instance třídy billboard), které mohou být právě používány a běží na nich animace, nebo jsou volné. V případě, že má správce volný billboard k dispozici, pouze ho inicializuje novými hodnotami a nechá spustit novou animaci. Pokud není žádný billboard volný, vytvoří správce novou instanci, kterou inicializuje a umístí na konec seznamu billboardů. Na tomto billboardu opět spustí animaci.

Za povšimnutí stojí také metody useBillboard a timeStep třídy Billboard. Začneme jednoduší metodou, tedy useBillboard, která označí billboard jako použitý a nastaví první zobrazovaný frame animace. Případně ta samá metoda volaná s parametrem nové pozice nastavuje rovnou pozici billboardu, a tudíž se již nemusí nastavovat explicitním voláním příslušné funkce.

Hlavní metodou, ve které se provádí samotná animace, je metoda timeStep, která je vyvolána v každém snímku. Tato metoda v první řadě upravuje natočení billboardu ke kameře, a to pomocí ukazatele na použitou kameru. Orientace billboardu je provedena tak, že orientace z transformační matice kamery je přiřazena do transformační matice billboardu s následným otočením o 180°, proto aby byl billboard otočený ke kameře, nikoli od kamery. Animace je optimalizována, proto není načítán každý snímek animace zvlášť, ale všechny snímky animace jsou umístěny v jedné textuře (viz. obr. 9.1).



Obr. 9.1

Takto se na textuře mění pouze texturovací souřadnice. Animace je prováděna s určitou rychlostí, kterou určuje atribut FPS. V okamžiku, kdy nastane čas k výměně snímku animace, dojde k přemapování texturovacích souřadnic pomocí funkce updateTextureCoordinate. Po provedení animace (zobrazení všech snímků) se billboard sám uvolní ze scény, tzn. již se dále nezobrazuje.

9.2. Jednoduchý odraz

Jednoduchý odraz je ze zvolených algoritmů pro vyhodnocování kolizí asi nejjednodušší. Pro dvě kolidující tělesa se vypočítá odraz jako v případě dvou koulí, a to bez ohledu na rotaci. Tento výpočet je detailně rozebrán v kapitole 4.1. Jednoduchý odraz. Po vypočtení nových rychlostí a směrů těles se objekty umístí na poslední známé nekolidující souřadnice, a tím je kolize vyřešena.

9.3. Fyzikální odraz

Fyzikální obraz byl asi nejsložitější částí projektu. Samotná teorie fyzikálního odrazu je podrobně rozepsána v kapitolách 2. Fyzika a 4.2. Fyzikální odraz. Vyhodnocení odrazu na základě fyzikálního modelu je realizováno ve funkci resolveCollisionBody, která je implementovaná podle uvedených vzorců (viz. kapitola 4.2. Fyzikální odraz).

Samotná implementace fyzikálního odrazu není tolik složitá, problematické však bylo zajišťování potřebných fyzikálních veličin. U veličin jako je rychlost, pozice nebo samotné souřadnice kolidujících těles není problém, protože tyto veličiny (atributy) jsou obsaženy buď v příslušných

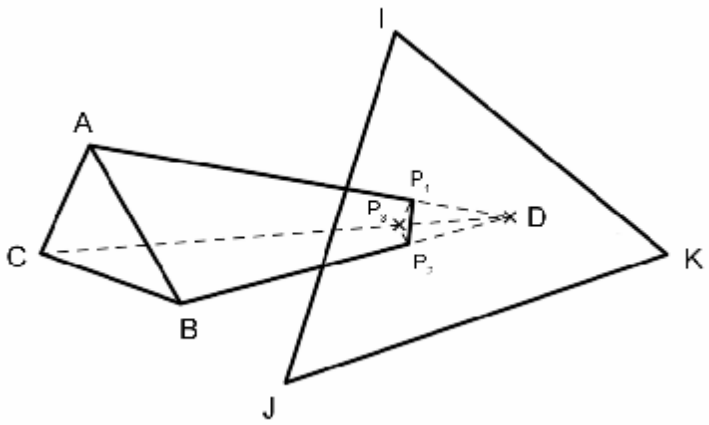
strukturách, nebo v objektu samotném. Problémovými veličinami jsou těžiště tělesa a matice momentu setrvačnosti (viz. kapitoly 2.2. Těžiště tělesa a 2.3. Moment setrvačnosti a matice setrvačnosti). Pro tyto parametry byla naimplementována speciální aplikace, počítající pouze tyto veličiny (viz. kapitola 6. Pomocná aplikace – Veličiny). Charakterizace dalších problémových veličin a situací následuje.

9.3.1. Bod kolize

Na první pohled se může zdát otázka výpočtu bodu kolize jako jednoduchá, skutečností však je, že patří k těm složitějším výpočtům v rámci tohoto projektu. Vzhledem k tomu, že se pohybujeme v rovině diskrétních funkcí a ne spojitých, jak by bylo pro fyzikální model potřeba, vzniká takto mnoho doprovodných problémů. Prvním z nich je samotná detekce kolize, při které dochází k průniku těles, právě v důsledku diskrétního posunu těchto objektů. Platí, že při průniku těles nám zanikne kolizní bod a normála, které se následně musejí vypočítat z objemu vzniklého tímto průnikem.

Již byla zmíněna funkce Open Inventoru SoIntersectionDetectionAction, resp. intersectionCb, která detekuje kolizi dvou těles a vrací všechny kolidující trojúhelníky. Zároveň vypočítává všechny průsečíky těchto trojúhelníků. Tyto průsečíky poté musíme eliminovat pouze na průsečíky tvořící přesně kolizní objem, resp. kolizní těleso. K tomu slouží další funkce workIntersectionPoints.

Funkce workIntersectionPoints eliminuje chybné kolizní body na základě předpokladu, že bod, který je ve výčtu uveden vícekrát, je právě tím hledaným kolizním bodem. Například z obrázku 9.2 průsečík P1 nalezneme jak pro dvojici trojúhelníků (ABD, IJK), tak pro (ACD, IJK), obdobně pro bod P2 to jsou dvojice trojúhelníků (ABD, IJK) a (CBD, IJK).

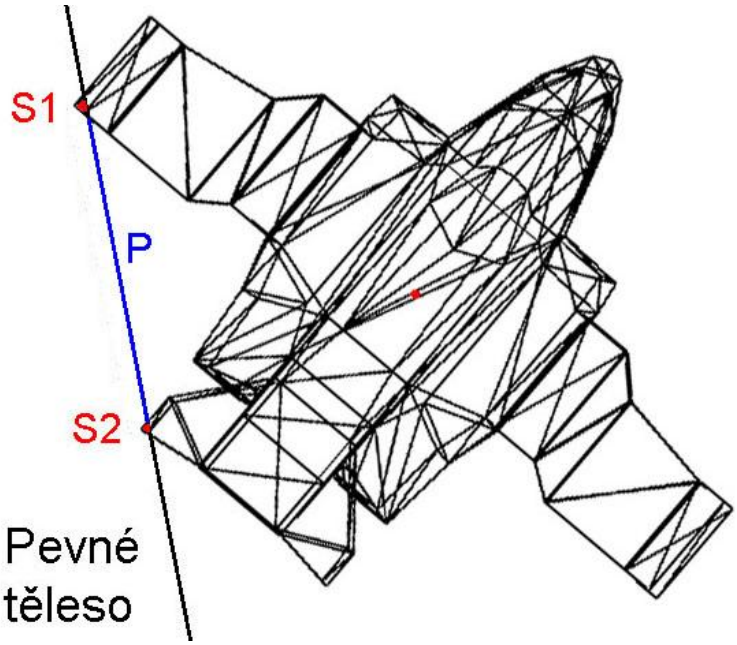


Obr. 9.2

Z takto vypočítaného kolizního tělesa, pomocí funkce getIntersectionPoint, vypočítáme kolizní bod. Ten je vypočítán tak, že jmenovaná funkce nalezne těžiště kolizního tělesa. Algoritmus pro výpočet těžiště je uveden v kapitole 2.2.1. Algoritmy pro výpočet těžiště.

Problém nastává, pokud těleso (tělesa) kolidují na více místech současně (viz. obr. 9.3). V místě S1 a S2 jsou vypočítány správné průsečíky, jenže objemové těleso je tvořené oběma shluky najednou a tedy výsledný bod srážky P (těžiště) je počítán ze špatného objemového tělesa. Tato chyba vede k tomu, že bod kolize se vypočítá nesprávně, a to někde mezi místy S1 a S2 (modrá plocha),

v závislosti na počtu průsečíků v jednotlivých místech S1 a S2. Tuto chybu pravděpodobně nelze odstranit bez použití složité heuristiky.

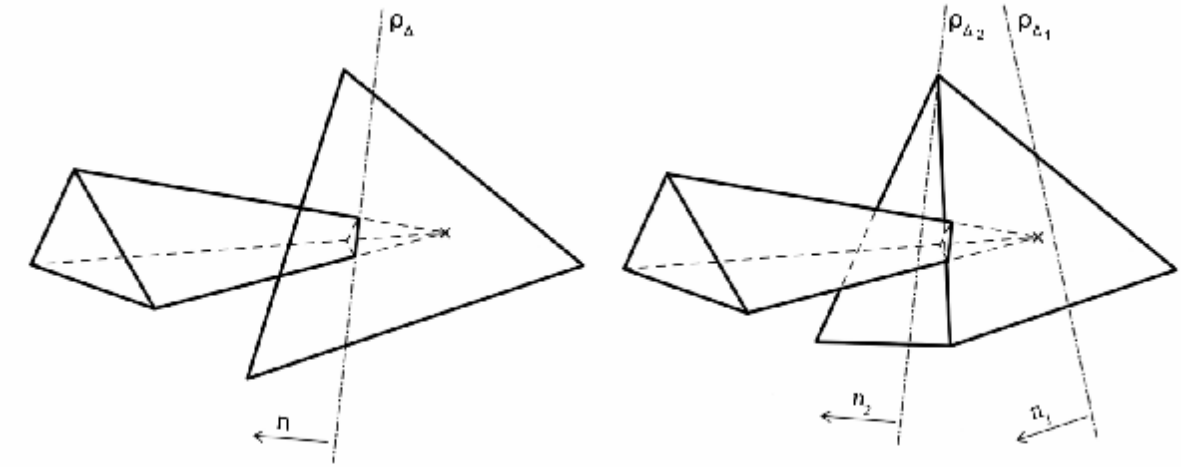


Obr. 9.3

9.3.2. Normála kolize

Výpočet normály kolize je zatížen podobnými problémy jako výpočet bodu kolize. Před samotným výpočtem normály, která se rovněž počítá na základě kolidujících trojúhelníků, je zapotřebí trojúhelníky uložené funkcí intersectionCb upravit pomocí funkce workIntersectionVertexs. Protože funkce intersectionCb hledá dvojice kolidujících trojúhelníků, mohou se v seznamu trojúhelníků každého objektu objevit stejné trojúhelníky vícekrát. Tyto výsledky jsou eliminovány právě funkcí workIntersectionVertexs**.** Po eliminaci stejných trojúhelníků můžeme přistoupit k výpočtu samotné normály.

Funkce computeNormal vypočítá normálu. Normála je ve výpočtu fyzikálního odrazu důležitá a rozhoduje, které těleso se doráží od kterého a jakým směrem. Jak již bylo zmíněno u výpočtu kolizního bodu, problém nastává při diskrétním pohybu objektů a při vyhodnocení srážky jako průniku dvou těles, proto nemůžeme jednoznačně určit, které těleso narazilo do kterého. Rozlišujeme dva druhy těchto srážek, jednoduchou a složitou (viz obrázek 9.4).



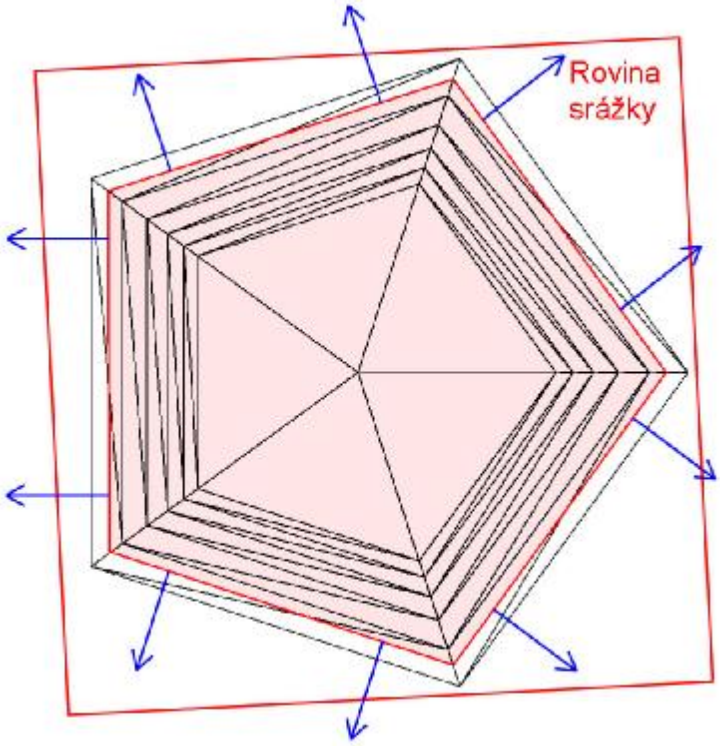
Obr. 9.4

Při jednoduché srážce proniká několik trojúhelníků jednoho tělesa do jednoho trojúhelníku druhého. Tuto srážku lze vyhodnotit tak, že normálu vypočteme z osamoceného trojúhelníku. Výsledná normála je tedy samotná normála tohoto trojúhelníku, resp. roviny, kterou trojúhelník tvoří (např. 3.2).

Při složité srážce proniká několik trojúhelníků jednoho tělesa do několika trojúhelníků druhého tělesa. Zde není jednoznačné, které těleso narazilo do kterého, proto se pro výpočet normály zvolí těleso s menším počtem kolidujících trojúhelníků. Pro zvolené těleso se vypočítají normály pro všechny tyto trojúhelníky, které se ve výsledku zprůměrňují. Normály zprůměrňujeme tak, že vektory sečteme a výsledný vektor normalizujeme. V případě srážky se statickým tělesem se automaticky počítá normála ze statického tělesa.

Problém může nastat při průniku rotačních těles (např. válec), nebo jiných plochou ukončených těles. Na obrázku 9.5 proniká 5-stěn do plochy. Při výpočtu normály dojde k situaci, kdy zprůměrňování vektorů (modré šipky) všech trojúhelníků se blíží nule. V takovém případě nedostaneme rozumný výsledek výpočtu normály a celý výpočet odrazu poté proběhne chybně. Tato chyba se pravděpodobně bez použití složité heuristiky nedá odstranit a způsobuje tak chybné odražení.

Chyba ve výpočtu normály má ve výsledném výpočtu největší váhu. Od normály se totiž odvíjí většina dalších výpočtů.



Obr. 9.5

10. Implementace - Přesnost algoritmů

Jelikož je výpočet tvořen často na sobě závislými dílčími výpočty, je výsledek zatížen velkou chybou. Výsledná chyba se dělí na chybu jednotlivých úloh, zaokrouhlovací chybu ve všech výpočtech a násobnou chybu mezi jednotlivými výpočty a závislými úlohami.

Zásadní vliv na chybu ve výpočtu mají tyto veličiny:

- normála kolize, byla popsána
- výpočet bodu kolize (kolizních bodů), též byla popsána
- přesnost těžiště
- přesnost matice setrvačnosti

Dalším faktorem ovlivňující tuto chybu je rychlost konkrétního počítače a grafické karty. Čím pomalejší je stroj, tím je menší počet snímků za sekundu, a tím je větší kolizní objem těles. Větší kolizní objem těles způsobuje větší chyby při výpočtech normály a kolizních bodů, a další problémy s nimi spojené.

10.1. Přesnost těžiště a matice setrvačnosti

Při výpočtu se určuje rameno srážky, které se vypočítává z bodu srážky, a těžiště tělesa. Pokud je v jednom z bodů chyba, jsou chybně vypočítána i ramena, která se dále používají ve výpočtu. S většinou vektorů je prováděn vektorový součet a chyba vzniklá chybným těžištěm, nebo bodem srážky, může mít poté zásadní vliv jak na velikost, tak na směr vektoru. Obdobně chyba v matici setrvačnosti má také zásadní vliv na výpočet, a to protože přes matici setrvačnosti se „pronásobuje“ většina vektorů.

S použitím matice setrvačnosti vznikl při řešení diplomového projektu další nečekaný problém. Algoritmus totiž neumí pracovat s libovolnou maticí setrvačnosti, která využívá deviační momenty. A většina vypočítaných maticí setrvačnosti je špatně podmíněná a v algoritmu nepracuje korektně. Matice obsahuje velké číselné rozdíly, velká čísla na hlavní diagonále a naopak čísla blízká nule ve zbytku matice. Zjištění, že algoritmus nepracuje korektně s „plnou“ maticí setrvačnosti, ale jen s maticí diagonální (složenou jen z momentů setrvačnosti) zabralo dlouhou dobu věnovanou ladění a testování aplikace. Tento problém se ukázal řešitelný, protože model se snažíme vycentrovat do těžiště a matice setrvačnosti pro těleso, které má střed souřadné soustavy v těžišti má deviační momenty rovny nule. Proto byla po provedené analýze a zjištění těchto skutečností upravena funkce programu veličiny tak, aby deviační momenty zaokrouhlila na nulu. Tím se matice setrvačnosti pro tělesa, která mají pevně dané těžiště, spíše upřesnila.

10.2. Zaokrouhlovací a násobná chyba

Zaokrouhlovací chyba vzniká při každém výpočtu a výpočtů je v aplikaci relativně hodně. Chyba vzniklá zaokrouhlováním je však relativně malá, či dokonce zanedbatelná, vůči ostatním chybám. Tato chyba je ale nepatrně větší než by mohla být, protože při většině výpočtů musel být použit datový typ float, který je implementován ve všech objektech knihovny Open Inventor místo datového typu double.

Násobná chyba vzniká při použití jednoho chybného výsledku v další operaci, takto se chyba neustále zvětšuje. Jelikož většina dílčích úloh je na sobě závislých, není tato chyba zanedbatelná a je nutné počítat s tím, že ovlivní výsledek odrazu. Násobná chyba se nedá odstranit, dá se však eliminovat použitím jiných metod výpočtů, případně upravením algoritmu. Největší násobná chyba byla v průběhu testování ladění a testování aplikace odstraněna. Jednalo se o chybu vzniklou při přepočítávání momentu matice setrvačnosti v každém snímku běžící aplikace. Chyba rostla velmi rychle, protože matice se počítala z matice předešlého snímku. Eliminace byla provedena úpravou algoritmu tak, že se pro výpočet používala základní matice momentu setrvačnosti, nikoliv matice z předchozího snímku.

10.3. Ovlivnění výsledku

Jelikož je výpočet ve výsledku zatížen velkým počtem chyb, dochází k chybnému vyhodnocení odrazu. Chybně je vyhodnocena jak velikost působící síly, tak její směr (vektor). Velikost působící síly by neměla způsobovat žádné viditelné problémy v aplikaci. Zato však směr odrazu již způsobuje další problémy. Většinou při mezních situacích ve výpočtu, kdy některé z vektorů svírají mezi sebou úhly blížící se 180°, 90°, 0°, může být směr vyhodnocen úplně špatně nebo s velkou chybou, a následně může dojít k „zakousnutí“ těles. K zakousnutí může dojít i při relativně malé chybě, kdy mají být výsledné vektory pod malým úhlem. Jako zakousnutí těles je označeno vzájemné a trvalé pronikání a vyhodnocování kolize. Těleso se může z tohoto stavu dostat časem samo, nebo v něm dlouhou dobu setrvávat.

Některé z těchto chyb by šly pravděpodobně vyřešit komplikovanou heuristikou. Například když dojde k nežádoucímu stavu zakousnutí, lze od sebe objekty „ručně odtrhnout“ nebo dát náhodný impuls síly oběma kolidujícím tělesům.

10.4. Chyba aproximace a statického tělesa

Každý libovolně složitý objekt je reprezentován maticí momentu setrvačnosti. Tato matice udává rozložení hmoty kolem jednotlivých os. Pro pravidelné objekty jako je kvádr či krychle je rozložení hmoty správné a použitý algoritmus pracuje relativně dobře s ohledem na další chyby. Nicméně pro nepravidelná tělesa, jako je například stíhačka, je rozložení hmoty sice ve správném poměru, nikoli v objemu (viz. obr. 10.1). Červený čtverec představuje předpokládaný objem stíhačky rozložený v jednotlivých osách, který je reprezentován maticí momentu setrvačnosti.

Obr. 10.1

Pokud nastane situace, že dojde ke srážce se statickým tělesem v místě mimo červenou oblast (srážka S1), dojde k chybnému výpočtu a tedy i k navýšení odrazové energie. Pokud dojde naopak ke srážce v místě S2, energie odrazu bude chybně snížena. Z tohoto vyplývá, že pokud dojde k sérii odrazů mimo červenou oblast, dojde ke kaskádnímu navyšování energie, které někdy (například v demonstrační aplikaci Box-Fighter) zapříčiní extrémní nárůst rychlosti a následné proskočení stěnou boxu. Při srážkách těles mezi sebou nemá tato chyba takový vliv. V řízené aplikaci (demo aplikace průlet tunelem) nebo při použití tlumení se chyba také plně neprojeví.

Chyba popsaná jako chyba aproximace není v rámci této práce matematicky podložena. Použitá literatura se problémem zabývá jen v teoretické rovině [4] a nalezená praktická řešení využívají zase jen jednoduchých objektů jako krychle a kvádr [5, 6, 7, 8]. Proto byla provedena řada testů, které nepřímo ukazují na chybu způsobenou aproximací.

Testy byly provedeny pro jednotlivé modely krychle, kvádru a stíhače v uzavřené statické krychli, obdobně jako demo příklad Box-Fighter. Pro krychli a kvádr aplikace běžela cca 1 hodinu a za tuto dobu došlo k překročení maximální energie soustavy u méně než 1% zaznamenaných kolizí. Tato chyba je způsobená vlivem ostatních chyb. Pro model stíhače však dochází k překročení energie velmi často a statistiku není možné provést, protože energie při špatných odrazech roste násobně. Počet odrazů, resp. doba správného běhu aplikace, závisí na počátečním nastavení parametrů. Demo aplikace Box-Stíhač je proto nastavena tak , aby k chybám docházelo co možná nejméně.

Jiným možným způsobem by bylo nepoužívat model jako komplexní objekt, ale rozdělit ho na pevně definované a vázané objekty, u kterých je moment setrvačnosti znám, a následně řešit problém jako soustavu takto definovaných objektů. Tento problém představuje v podstatě přesnou fyzikální simulaci a je přesahuje rámec této práce.

Obecně při srážce se statickým tělesem dochází ke změnám celkové energie, protože statické těleso „ignoruje“ působící síly. Tím dochází při srážce se statickým tělesem k porušení zákona o zachování energie, což také může způsobovat nečekané chyby při odrazu.

10.5. Chyby knihovny Open Inventoru

Ve fázi testování a konečného ladění projektu se objevila výjimka v aplikaci, která je způsobená chybou ve funkci IntersectionDetectionAction v knihovně Open Inventor. Po zajištění nových knihoven se již výjimka neobjevovala.

Další závažnou chybou Open Invetoru je problém ztráty „focusu“ při vyvolání libovolného dialogu. Tato chyba byla také vyřešena.

V originálním archivu a na stránkách knihovny Open Inventor jsou k dispozici knihovny, obsahující tyto chyby. Díky panu ing. Pečivovi jsou k dispozici v rámci této diplomové práce překompilované funkční knihovny. K projektu jsou přiloženy tyto funkční knihovny, které je nutné použít pro správnou funkci demo aplikací.

11. Etapy vývoje

Každé větší vývojové etapě projektu předcházelo studium literatury související s daným problémem. Proto si nyní představíme pouze fáze praktické implementace. Nastudování potřebných materiálů probíhalo souběžně s řešením implementovaných problémů.

11.1. Rozsah ročníkového projektu

Diplomová práce vychází z ročníkového projektu s názvem Knihovna pro detekci kolizí mezi objekty scény. První práce spočívala v seznámení s knihovnou Open Inventor a implementování triviální virtuální scény (grafického okna aplikace). Tato scéna obsahovala jednoduchou sluneční soustavu reprezentovanou koulemi.

První detekce kolizí byla provedena pouze pro obalové koule, které byly navíc pevně definovány, za použití funkce Open Inventoru SoIntersectionDetectionAction. Následné vyhodnocení probíhalo jen na úrovni vyhodnocení srážky dvou koulí.

Prvním krokem k rozumné aplikaci bylo rozdělení do té doby provedené práce na moduly a následné převedení na objekty. Byly vytvořeny základní třídy SgObject a SgoManager. Pro tyto objekty byly implementovány metody detekce kolize a vyhodnocení kolize, které byly postupně rozšiřovány. Dále přibylo vyhodnocení kolize pomocí exploze, a s tím související třída billboarding.

V druhém semestru byly v práci na ročníkovém projektu implementovány základní algoritmy pro srážku na základě fyzikálního modelu, které ovšem používaly většinou konstantních veličiny. S tímto musely být přidány algoritmy na výpočet bodu srážky a normály srážky. Nakonec byly implementovány funkce pro pohyb ve scéně a třída SgInterface.

V ročníkovém projektu fungovala jednoduchá detekce kolizí s pevně daným obalovým tělesem koule a jednoduché vyhodnocení kolize srážky koulí. Fyzikální vyhodnocení s použitím konstantních veličin fungovalo v rámci demonstrační aplikace relativně dobře.

11.2. Rozsah diplomové práce

Pro diplomovou práci bylo třeba nahradit konstantní hodnoty hodnotami vypočítanými, proto byla implementována pomocná aplikace počítající těžiště, moment setrvačnosti a obalová tělesa. Aplikace se sestávala z načtení WRLM modelu a z jeho převodu pro knihovnu ColDet. Také došlo k aplikaci algoritmů počítající těžiště a moment setrvačnosti a k řešením problému průchodů paprsků.

Dále bylo vylepšeno ovládání rozšíření objektů o metody pro ovládání stíhače. Byly naprogramovány metody ovlivňující prostředí (tření, gravitace).

Samotná detekce kolize byla rozšířena o rozdělení prostoru pomocí obalových těles. Dále byla využita vypočítaná obalová tělesa, jak koule, tak box.

S nově získanými veličinami byla upravena stávající aplikace ročníkového projektu. Vyskytla se však řada chyb a nepřesností. Bylo nutné upravit funkci pro výpočet kolizního bodu, která kolizní bod počítala nepřesně. Také byla provedena řada testů a dlouhodobé ladění programu. Rovněž bylo

nutné opětovné a hlubší studium problému fyzikálního vyhodnocení kolize. Algoritmus pro fyzikální detekci i celá aplikace byly v průběhu ladění několikrát upraveny.

Například zjištění, že algoritmus pracuje jen s diagonální maticí momentu setrvačnosti, si vyžádalo jak jednoduchou úpravu v pomocné aplikaci, tak úpravu hlavní aplikace. Také odstranění násobné chyby při aktualizaci matice setrvačnosti. Navíc byla objevena chyba v algoritmu počítajícím normálu kolize, který musel být přepracován.

Byl naimplementován textový editor umožňující načtení tunelu s objekty a upravena práce s kamerou. Závěr práce byl věnován úpravě samotných modelů, prostředí, aplikacím samotným a zpracování dokumentace. Část dokumentace byla vytvořena na základě dokumentace ročníkového projektu a byla doplněna novou literární rešerší a implementačními záležitostmi, spolu s poukázáním na zjištěné chyby a problémy s tímto spojené.

Závěr

Zadáním této diplomové práce bylo vyřešit detekci a vyhodnocování kolizí na základě fyzikálního modelu ve 3D prostoru a demonstrovat je na vhodných příkladech. K zadanému tématu byla nastudována dostupná literatura, a na jejím základě vypracována literární rešerše.

Na základě získaných znalostí a vědomostí byly navrženy algoritmy pro detekci kolizí mezi objekty scény a algoritmy na vyhodnocování těchto kolizí s ohledem na možné optimalizace. V diplomovém projektu byl implementován pomocný program pro výpočet těžiště a matice momentu setrvačnosti. Byly implementovány algoritmy řešící detekci kolizí. Dále algoritmy pro odraz koule vs. koule a pevné těleso vs. pevné těleso, které řeší odraz na základě fyzikálního modelu. Všechny algoritmy byly vytvořeny v prostředí Win32 v programovacím studiu Visual Studio za použití programovacího jazyka C++ s využitím grafické knihovny OpenGL a nadstavby Open Inventor, a jsou podrobně popsány v této dokumentaci. Pro demonstraci algoritmů byl implementován jednoduchý herní engine, ve kterém byly vytvořeny demonstrační aplikace, zvláště pak aplikace „Průlet tunelem“.

Nejsložitější algoritmy řešící fyzikální odraz dvou pevných těles jsou zatíženy velkým množstvím chyb, jak vyplývá z dokumentace. Z tohoto důvodu se mohou v aplikacích vyskytovat různé nechtěné stavy. Pro reálné použití, například v počítačových hrách, je nutné k těmto algoritmům v současné podobě přidat různé komplikované heuristiky. Konkrétně pro fyzikální problémy spojené s maticí setrvačnosti, především deviační momenty matice setrvačnosti vs. použitý algoritmus a chyba aproximace, bych však doporučil konzultaci s odborníkem v oboru fyziky a matematiky, ke které v rámci této práce z časových důvodů nedošlo.

V neposlední řadě tato práce přinesla pozitiva i mé osobě. Během vypracovávání jsem získal řadu nových poznatků a vědomostí z oboru 3D počítačové grafiky a praktické implementace fyzikálních algoritmů srážky. Zlepšil si a zdokonalil vědomosti a schopnosti již známé. Seznámil se detailně s grafickou knihovnou Open Inventor a různými programovacími technikami. Rozšířil jsem si své znalosti objektově orientovaného programování a implementace v C++.

Literatura

[1] Halliday, D., Resnick, R., & Walker, J. (2003). Fyzika (Vol. 1). VUTIUM, Brno, ISBN 80-214-1868-0, PROMETHEUS, Praha, ISBN 81-7196-213-9.

[2] Ježek, F., Míková, M., & Tomiczková, S. (2005). Geometrie pro FST 1 (skripta). Plzeň.

[3] Theoretical Foundation: Mechanical Basis of Motion Analysis (internet). 11.12.2006. <http://kwon3d.com/theory/basis.html>.

[4] Witkin, A., & Baraff, D. (1997). Physically Based Modeling: Principles and Practice (internet). 11.12.2006. <http://www.cs.cmu.edu/~baraff/sigcourse/>

[5] Hecker, Ch. (1997). Physics, Part 1: The Next Frontier (internet). 11.12.2006. <http://www.d6.com/users/checker/pdfs/gdmphys1.pdf>.

[6] Hecker, Ch. (1996). Physics, Part 2: Angular Effects (internet). 11.12.2006. <http://www.d6.com/users/checker/pdfs/gdmphys2.pdf>.

[7] Hecker, Ch. (1997). Physics, Part 3: Collision Response (internet). 11.12.2006. <http://www.d6.com/users/checker/pdfs/gdmphys3.pdf>.

[8] Hecker, Ch. (1997). Physics, Part 4: The Third Dimension (internet). 11.12.2006. <http://www.d6.com/users/checker/pdfs/gdmphys4.pdf>.

[9] Ehmann, S. (24.06.1999). Rigid Body Simulation Tutoriál (internet). 06.09.2006. <http://www.cs.unc.edu/~ehmann/RigidTutorial/>.

[10] Blow, J., & Binstock, A. J. (2004). How to find the inertia tensor (or other mass properties) of a 3D solid body represented by a triangle mesh (internet). 11.12.2006. <http://number-none.com/blow/inertia/index.html>.

[11] Sochor, J., & Tobola, P. (2005). Detekce kolize (internet). 11.12.2006. <http://www.fi.muni.cz/~sochor/PA010/Slajdy/KolizniMetody.pdf>

[12] Pelikán, J., & Tobola, P. (2003). Datové struktury pro prostorové vyhledávání (internet). 11.12.2006. <http://www.fi.muni.cz/~sochor/PA010/Slajdy/SpaceSearch.pdf>

[13] Miller, K. Collision Detection (internet). 11.12.2006. <http://www.gamespp.com/algorithms/collisionDetection.html>

[14] Advanced Collision Detection Techniques (internet). 11.12.2006 <http://www.gamespp.com/algorithms/AdvancedCollisionDetectionTechniques1.html>

[15] Pravda, J. (2001). Jak vyzrát na kolize (internet). 11.12.2006 <http://www.builder.cz/art/cpp/kolize1.html>

[16] Fauerby, K. (2000). Collision detection & Response (internet). 11.12.2006 <http://www.peroxide.dk/download/tutorials/tut10/pxdtut10.html>

[17] Collision Detection Vertex-in-Triangle Check (internet). 11.12.2006 <http://www.gamespp.com/algorithms/CollisionDetectionTutorial.html>

[18] Kolář, D. (2002). Postrelační databáze (internet, privátní stránky FIT VUT). 11.12.2006 <https://www.fit.vutbr.cz/study/courses/PRD/private/lectures/prednaskyPRD.pdf>

[19] Möller, T. A Fast Triangle-Triangle Intersection Test (internet). 11.12.2006. <http://www.cs.lth.se/home/Tomas_Akenine_Moller/pubs/tritri.pdf>.

[20] Tropp, O., Tal, A., & Shimshoni, I. (2006). A fast triangle to triangle intersection test for collision detection (internet). 11.12.2006. <http://mis.hevra.haifa.ac.il/~ishimshoni/papers/TroppTalShimshoni.pdf>.

[21] Turek, M. (2004). CZ NeHe OpenGL, vše o programování grafiky (internet). 11.12.2006. <http://nehe.ceskehry.cz/>

[22] Dostál, R. (2002). Objektově orientované programování v C++ (internet). 11.12.2006. <http://www.builder.cz/serial24.html>

[23] Prata, S. (2001). Mistrovství v C++. Computer Press, Praha, ISBN 80-7226-339-0.

[24] Kilgard, J., K. (1996). The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3 (internet). 11.12.2006. <http://www.opengl.org/documentation/specs/glut/spec3/spec3.html>

[25] Van Heesh, D. (2000). ColDet Documentation (internet). 11.12.2006. <http://sourceforge.net/projects/coldet>

[26] Pečiva, J. (2004). Seriál Open Inventor (internet). 11.12.2006. <http://www.root.cz/serialy/open-inventor/>.

[27] Systeme in Motion AS. (2005). Coin Documentation (internet). 11.12.2006. <http://doc.coin3d.org/Coin/>

[28] Carey, R., & Gavin, B. (1997). The Anotated WRML97 Reference Manual (internet). 11.12.2006. <http://www.cs.vu.nl/~eliens/documents/vrml/reference/BOOK.HTM>

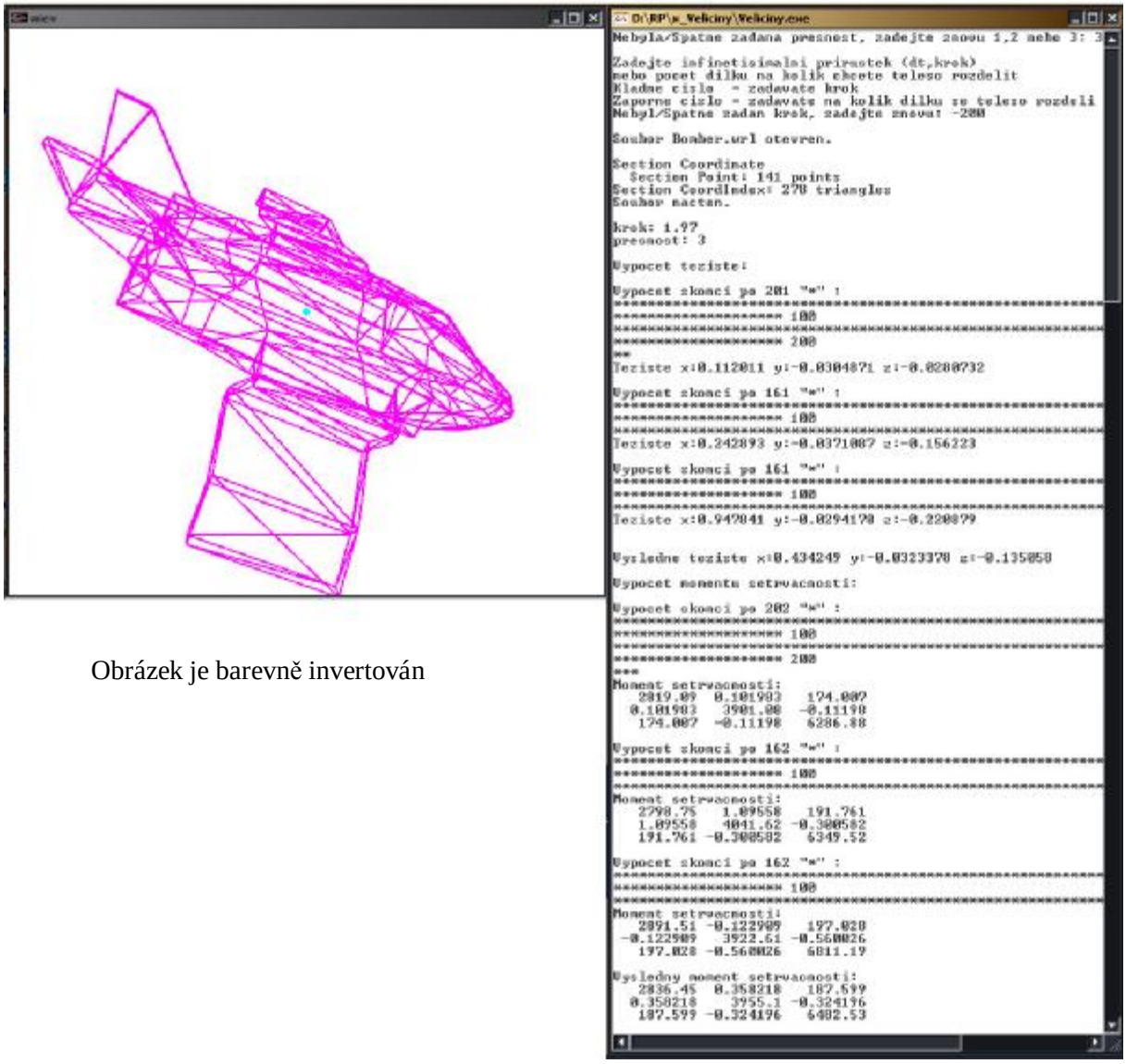
Přílohy

Příloha I. Ovládání pomocné aplikace a screenshot

Aplikace veličiny se dá spustit bez parametrů nebo s parametry. Pokud se pustí bez parametrů postupně se na vše potřebné zeptá. Spouštění s parametrem je následující:

První parametr - název souboru
Druhý parametr - krok aplikace
Třetí parametr - přesnost

Název souboru určuje jaký soubor se bude zpracovávat, povolený formát je pouze VRML V2.0 utf8. Krok aplikace určuje velikost dělící mřížky, nebo se znaménkem „-„ určuje parametr na kolik dílků se rozdělí nejdelší strana obalového boxu modelu. Rozumné hodnoty závisí na počtu jednotek modelu. Pro použité modely jsou rozumné parametry krok=0,5 jednotky nebo rozdělení modelu na cca 400-600 dílku. Třetím parametrem je přesnost určuje v kolika osách se provede výpočet (1-3). S každým stupněm přesnosti se opakuje výpočet v jedné z os, proto roste i časová náročnost.



Obrázek je barevně invertován

Příloha II. Analýza krychle

Analýza pro krychli o straně 250 jednotek. Soubor Cube.wrl (8 points, 12 triangles).

krok: 250

Moment setrvacnosti:
31262.5 0 0
0 72304.2 0
0 0 72304.2
chyb: 0, Chyba v %: 0

Moment setrvacnosti:
72304.2 0 0
0 31262.5 0
0 0 72304.2
chyb: 0, Chyba v %: 0

Moment setrvacnosti:
72304.2 0 0
0 72304.2 0
0 0 31262.5
chyb: 0, Chyba v %: 0

Vysledny moment setrvacnosti:
58623.6 0 0
0 58623.6 0
0 0 58623.6
Cas potrebny pro vypocet IT: 0s

krok: 125

Moment setrvacnosti:
39063.6 0 0
0 54594.8 0
0 0 54594.8
chyb: 0, Chyba v %: 0

Moment setrvacnosti:
54594.8 0 0
0 39063.6 0
0 0 54594.8
chyb: 0, Chyba v %: 0

Moment setrvacnosti:
54594.8 0 0
0 54594.8 0
0 0 39063.6
chyb: 0, Chyba v %: 0

Vysledny moment setrvacnosti:
49417.8 0 0
0 49417.7 0
0 0 49417.7
Cas potrebny pro vypocet IT: 0s

krok: 62.5

Moment setrvacnosti:
42024.7 0 0
0 48022.9 0
0 0 48022.8
chyb: 2
Chyba v %: 4.72183e-007

Moment setrvacnosti:
47518.3 0 0
0 41015.7 0
0 0 47518.3
chyb: 0, Chyba v %: 0

Moment setrvacnosti:
48022.9 0 0
0 48022.8 0
0 0 42024.7
chyb: 2
Chyba v %: 4.72183e-007

Vysledny moment setrvacnosti:
45855.3 0 0
0 45687.1 0
0 0 45855.3
Cas potrebny pro vypocet IT: 0s

krok: 31.25

Moment setrvacnosti:
41442.7 0 0
0 44401 0
0 0 44401
chyb: 2

Chyba v %: 8.17237e-007
Moment setrvacnosti:
44431.6 0 0
0 41503.9 0
0 0 44431.6

chyb: 0, Chyba v %: 0

Moment setrvacnosti:
44401 0 0
0 44401 0

0 0 41442.7 chyb: 2 Chyba v %: 8.17237e-007

Vysledny moment setrvacnosti:
43425.1 0 0
0 43435.3 0
0 0 43425.1
Cas potrebny pro vypocet IT: 0s

krok: 15.625

Moment setrvacnosti:
41781.8 0 0
0 43087 0
0 0 43086.2
chyb: 13
Chyba v %: 7.27511e-006

Moment setrvacnosti:
43009 0 0
0 41626 0
0 0 43008.3
chyb: 0, Chyba v %: 0

Moment setrvacnosti:
43087 0 0
0 43086.2 0
0 0 41781.8
chyb: 13
Chyba v %: 7.27511e-006

Vysledny moment setrvacnosti:
42625.9 0 0
0 42599.7 0
0 0 42625.4
Cas potrebny pro vypocet IT: 1s

Příloha III. Analýza krychle

Položka výsledné těžiště je korekce původního těžiště.

Soubor Cube.wrl
8 vrcholu
12 trojuhelniku
Velikost kroku: 50
1 pruchod: chyb 2, v %: 9.25926e-007
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 2, v %: 9.25926e-007
teziste x:-7.50291 y:-6.82263 z:-7.50292
Cas potrebny pro vypocet teziste: 1s
1 pruchod: Pocet chyb 0, v %: 0
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 0, v %: 0
Vysledny moment setrvacnosti:
45565.1 0 0
0 45612.4 0
0 0 45565
Cas potrebny pro vypocet IT: 0 s
Celkovy cas: 1s

Soubor Cube.wrl
8 vrcholu
12 trojuhelniku
Velikost kroku: 10
1 pruchod: Pocet chyb 11, v %: 7.82317e-006
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 11, v %: 7.82317e-006
teziste x:-1.70594 y:-1.77473 z:-1.70558
Cas potrebny pro vypocet teziste: 0s
1 pruchod: Pocet chyb 0, v %: 0
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 0, v %: 0
Vysledny moment setrvacnosti:
42262.4 0 0
0 42262.4 0
0 0 42262.5
Cas potrebny pro vypocet IT: 2 s
Celkovy cas: 2s

Rozdíl mezi výpočty cca 7,3 %

Soubor Cube.wrl
8 vrcholu
12 trojuhelniku
Velikost kroku: 5
1 pruchod: Pocet chyb 26, v %: 1.96003e-005
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 26, v %: 1.96003e-005
teziste x:-0.808285 y:-0.802269 z:-0.810985
Cas potrebny pro vypocet teziste: 1s
1 pruchod: Pocet chyb 0, v %: 0
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 0, v %: 0
Vysledny moment setrvacnosti:
41953.8 0 0
0 41956.3 0
0 0 41958.8
Cas potrebny pro vypocet IT: 9 s
Celkovy cas: 10s

Rozdíl mezi výpočty cca 0,73 %

Soubor Cube.wrl
8 vrcholu
12 trojuhelniku
Velikost kroku: 2.5
1 pruchod: Pocet chyb 52, v %: 4.03766e-005
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 52, v %: 4.03766e-005
teziste x:-0.415175 y:-0.420771 z:-0.414847
Cas potrebny pro vypocet teziste: 3s
1 pruchod: Pocet chyb 0, v %: 0
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 0, v %: 0
Vysledny moment setrvacnosti:
41730.1 0 0
0 41733.2 0
0 0 41736.4
Cas potrebny pro vypocet IT: 72 s
Celkovy cas: 75s

Rozdíl mezi výpočty cca 0,53 %

Soubor Cube.wrl
8 vrcholu
12 trojuhelniku
Velikost kroku: 1.25
1 pruchod: Pocet chyb 101, v %: 7.96e-005
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 101, v %: 7.96e-005
teziste x:-0.206313 y:-0.206638 z:-0.205444
Cas potrebny pro vypocet teziste: 27s
1 pruchod: Pocet chyb 0, v %: 0
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 0, v %: 0
Vysledny moment setrvacnosti:

```
  34719.1        0        0
        0   34107.2        0
        0        0   33495.3
```
Cas potrebny pro vypocet IT: 745 s
Celkovy cas: 772s

Rozdíl mezi výpočty cca 16,8 %

Soubor Cube.wrl
8 vrcholu
12 trojuhelniku
Velikost kroku: 0.833333
1 pruchod: Pocet chyb 203, v %: 0.000160787
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 203, v %: 0.000160787
teziste x:-0.134675 y:-0.135588 z:-0.136416
Cas potrebny pro vypocet teziste: 41s
1 pruchod: Pocet chyb 0, v %: 0
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 0, v %: 0
Vysledny moment setrvacnosti:
```
  20664.6        0        0
        0   20664.6        0
        0        0   20664.6
```
Cas potrebny pro vypocet IT: 2073 s
Celkovy cas: 2114s

Rozdíl mezi výpočty cca 40 %

Soubor Cube.wrl
8 vrcholu
12 trojuhelniku
Velikost kroku: 0.625
1 pruchod: Pocet chyb 391, v %: 0.000310466
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 391, v %: 0.000310466
teziste x:-0.117042 y:-0.148822 z:-0.128551
Cas potrebny pro vypocet teziste: 69s
1 pruchod: Pocet chyb 0, v %: 0
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 0, v %: 0
Vysledny moment setrvacnosti:
```
  16748.4        0        0
        0   16748.4        0
        0        0   16748.4
```
Cas potrebny pro vypocet IT: 5301 s
Celkovy cas: 5370s

Rozdíl mezi výpočty cca 18.9 %

Soubor Cube.wrl
8 vrcholu
12 trojuhelniku
Velikost kroku: 0.5
1 pruchod: Pocet chyb 252, v %: 0.000200395
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 252, v %: 0.000200395
teziste x:-0.0823201 y:-0.0823637 z:-0.082692
Cas potrebny pro vypocet teziste: 131s
1 pruchod: Pocet chyb 0, v %: 0
2 pruchod: Pocet chyb 0, v %: 0
3 pruchod: Pocet chyb 0, v %: 0
Vysledny moment setrvacnosti:
```
  15352.7        0        0
        0   15352.7        0
        0        0   15352.7
```
Cas potrebny pro vypocet IT: 11350 s
Celkovy cas: 11481s

Rozdíl mezi výpočty cca 8,3 %

Soubor Cube.wrl
8 vrcholu
12 trojuhelniku
Velikost kroku: 0.416667
1 pruchod: chyb 423, v %: 0.000336714
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 423, v %: 0.000336714
teziste x:-0.0702809 y:-0.0695268 z:-0.0686621
Cas potrebny pro vypocet teziste: 140s
1 pruchod: Pocet chyb 0, v procentech: 0
2 pruchod: Pocet chyb 0, v procentech: 0
3 pruchod: Pocet chyb 0, v procentech: 0
Vysledny moment setrvacnosti:
```
  14735.5        0        0
        0   14735.5        0
        0        0   14735.5
```
Cas potrebny pro vypocet IT: 15526 s
Celkovy cas: 15666s

Rozdíl mezi výpočty cca 4%

Příloha IV. Analýza jehlanu

Soubor cone.WRL
50 vrcholu
96 trojuhelniku
Velikost kroku: 30
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 1, v %: 4.10914e-006
3 pruchod: chyb 0, v %: 0
teziste x:-0.251553 y:74.2434 z:-0.73348
Cas potrebny pro vyteziste: 0s
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 0, v %: 0
Vysledny moment setrvacnosti:
5427.86 0 0
0 3369.5 0
0 0 5446.58
Cas potrebny pro vyIT: 0 s
Celkovy cas: 0s

Soubor cone.WRL
50 vrcholu
96 trojuhelniku
Velikost kroku: 1.5
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 42, v %: 0.000336368
3 pruchod: chyb 0, v %: 0
teziste x:-0.000981329 y:74.9884 z:-0.0138747
Cas potrebny pro vyteziste: 4s
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 37, v %: 0.000292461
3 pruchod: chyb 0, v %: 0
Vysledny moment setrvacnosti:
4919.67 0 0
0 3004.41 0
0 0 4920.54
Cas potrebny pro vyIT: 12 s
Celkovy cas: 16s

Rozdíl mezi výpočty cca 9%

Soubor cone.WRL
50 vrcholu
96 trojuhelniku
Velikost kroku: 0.75
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 153, v %: 0.00124984
3 pruchod: chyb 0, v %: 0
teziste x:0.00305156 y:74.9546 z:-0.0139443
Cas potrebny pro vyteziste: 16s
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 131, v %: 0.00105608
3 pruchod: chyb 0, v %: 0
Vysledny moment setrvacnosti:
4857.09 0 0
0 2958.3 0
0 0 4864.62
Cas potrebny pro vyIT: 79 s
Celkovy cas: 95s

Rozdíl mezi výpočty cca 1,2%

Soubor cone.WRL
50 vrcholu
96 trojuhelniku
Velikost kroku: 0.5
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 356, v %: 0.00292746
3 pruchod: chyb 0, v %: 0
teziste x:0.00402865 y:74.9739 z:-0.015635
Cas potrebny pro vyteziste: 50s
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 298, v %: 0.00241828
3 pruchod: chyb 0, v %: 0
Vysledny moment setrvacnosti:
4717.6 0 0
0 2863.78 0
0 0 4784.88
Cas potrebny pro vyIT: 345 s
Celkovy cas: 395s

Rozdíl mezi výpočty cca 2,8%

Soubor cone.WRL
50 vrcholu
96 trojuhelniku
Velikost kroku: 0.375
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 593, v %: 0.00489258
3 pruchod: chyb 0, v %: 0
teziste x:0.00399497 y:74.9786 z:-0.0141979
Cas potrebny pro vyteziste: 65s
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 499, v %: 0.00406281
3 pruchod: chyb 0, v %: 0
Vysledny moment setrvacnosti:
4085.53 0 0
0 2570.28 0
0 0 4214.26

Cas potrebny pro vyIT: 575 s
Celkovy cas: 640s

Rozdíl mezi výpočty cca 11,9%

Soubor cone.WRL
50 vrcholu
96 trojuhelniku
Velikost kroku: 0.3
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 778, v %: 0.00643174
3 pruchod: chyb 0, v %: 0
teziste x:0.00149944 y:74.9839 z:-0.00744216
Cas potrebny pro vyteziste: 103s
1 pruchod: chyb 12, v %: 9.78969e-005
2 pruchod: chyb 748, v %: 0.00610224
3 pruchod: chyb 0, v %: 0
Vysledny moment setrvacnosti:
3436.52 0 0
0 1846.7 0
0 0 3657.75
Cas potrebny pro vyIT: 1126 s
Celkovy cas: 1229s

Rozdíl mezi výpočty cca 13,2 %

Soubor cone.WRL
50 vrcholu
96 trojuhelniku
Velikost kroku: 0.25
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 1147, v %: 0.0094949
3 pruchod: chyb 0, v %: 0
teziste x:0.00361417 y:74.9864 z:-0.0152829
Cas potrebny pro vyteziste: 155s
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 1163, v %: 0.00950041
3 pruchod: chyb 0, v %: 0
Vysledny moment setrvacnosti:
3108.38 0 0
0 1551.35 0
0 0 3171.52
Cas potrebny pro vyIT: 1905 s
Celkovy cas: 2060s

Rozdíl mezi výpočty cca 13,2 %

Soubor cone.WRL
50 vrcholu
96 trojuhelniku
Velikost kroku: 0.2
1 pruchod: chyb 6, v %: 4.97343e-005
2 pruchod: chyb 1815, v %: 0.0150446
3 pruchod: chyb 5, v %: 4.14452e-005
teziste x:0.00381616 y:74.9835 z:-0.0152807
Cas potrebny pro vyteziste: 256s
1 pruchod: chyb 1, v %: 8.17971e-006
2 pruchod: chyb 1811, v %: 0.0148135
3 pruchod: chyb 6, v %: 4.90782e-005
Vysledny moment setrvacnosti:
2589.25 0 0
0 1381.33 0
0 0 2642.15
Cas potrebny pro vyIT: 3545 s
Celkovy cas: 3801s

Rozdíl mezi výpočty cca 16,7 %

Příloha V. Analýza stíhače

Soubor Bomber.wrl
141 vrcholu
278 trojuhelniku
Velikost kroku: 39.4
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 0, v %: 0
teziste x:10.5501 y:0.108323 z:-0.455971
Cas potrebny pro vyteziste: 0s
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 0, v %: 0
Vysledny moment setrvacnosti:
2953.75 0 0
0 5132.97 0
0 0 7305.88
Cas potrebny pro vyIT: 0 s
Celkovy cas: 0s

Soubor Bomber.wrl
141 vrcholu
278 trojuhelniku
Velikost kroku: 7.88
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 0, v %: 0
teziste x:2.00091 y:-0.00187711 z:-0.63596
Cas potrebny pro vyteziste: 0s
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 0, v %: 0
Vysledny moment setrvacnosti:
3414.3 0 0
0 4100.75 0
0 0 7023.78
Cas potrebny pro vyIT: 1 s
Celkovy cas: 1s

Rozdíl mezi výpočty cca 3,8%

Soubor Bomber.wrl
141 vrcholu
278 trojuhelniku
Velikost kroku: 3.94
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 0, v %: 0
teziste x:0.81494 y:-0.00104779 z:-0.221078
Cas potrebny pro vyteziste: 1s
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 0, v %: 0
Vysledny moment setrvacnosti:
3047.98 0 0
0 3907.76 0
0 0 6698.66
Cas potrebny pro vyIT: 1 s
Celkovy cas: 2s

Rozdíl mezi výpočty cca 4,6%

Soubor Bomber.wrl
141 vrcholu
278 trojuhelniku
Velikost kroku: 1.97
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 1, v %: 8.43635e-006
teziste x:0.367432 y:0.0230365 z:-0.129693
Cas potrebny pro vyteziste: 3s
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 1, v %: 8.34014e-006
Vysledny moment setrvacnosti:
2830.86 0 0
0 3950.19 0
0 0 6473.39
Cas potrebny pro vyIT: 4 s
Celkovy cas: 7s

Soubor Bomber.wrl
141 vrcholu
278 trojuhelniku
Velikost kroku: 0.985
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 24, v %: 0.000209156
teziste x:0.18518 y:-0.0187271 z:-0.0594835
Cas potrebny pro vyteziste: 10s
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 14, v %: 0.000120597
Vysledny moment setrvacnosti:
2827.53 0 0
0 3922.72 0
0 0 6420.65
Cas potrebny pro vyIT: 21 s
Celkovy cas: 31s

Rozdíl mezi výpočty cca 0,8%

Soubor Bomber.wrl
141 vrcholu
278 trojuhelniku
Velikost kroku: 0.656667
1 pruchod: chyb 1, v %: 8.81072e-006
2 pruchod: chyb 1, v %: 8.81072e-006
3 pruchod: chyb 3, v %: 2.64322e-005
teziste x:0.0875572 y:-0.00490258 z:-0.0373552
Cas potrebny pro vyteziste: 21s
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 0, v %: 0
3 pruchod: chyb 3, v %: 2.61252e-005
Vysledny moment setrvacnosti:
2817.08 0 0
0 3939.21 0
0 0 6427.45
Cas potrebny pro vyIT: 64 s
Celkovy cas: 85s

Rozdíl mezi výpočty cca 0,1%

Soubor Bomber.wrl
141 vrcholu
278 trojuhelniku
Velikost kroku: 0.4925
1 pruchod: chyb 1, v %: 8.85925e-006
2 pruchod: chyb 2, v %: 1.77185e-005
3 pruchod: chyb 8, v %: 7.0874e-005
teziste x:0.0113379 y:-0.0100701 z:-0.02211
Cas potrebny pro vyteziste: 38s
1 pruchod: chyb 1, v %: 8.75614e-006
2 pruchod: chyb 1, v %: 8.75614e-006
3 pruchod: chyb 28, v %: 0.000245172
Vysledny moment setrvacnosti:
2799.11 0 0
0 3869.25 0
0 0 6354.58
Cas potrebny pro vyIT: 133 s
Celkovy cas: 171s

Rozdíl mezi výpočty cca 1,1%

Soubor Bomber.wrl
141 vrcholu
278 trojuhelniku
Velikost kroku: 0.394
1 pruchod: chyb 1, v %: 8.88857e-006
2 pruchod: chyb 2, v %: 1.77771e-005
3 pruchod: chyb 13, v %: 0.000115551
teziste x:0.00572341 y:-0.00644004 z:-0.014955
Cas potrebny pro vyteziste: 59s
1 pruchod: chyb 1, v %: 8.78497e-006
2 pruchod: chyb 1, v %: 8.78497e-006
3 pruchod: chyb 17, v %: 0.000149344
Vysledny moment setrvacnosti:
2719.61 0 0
0 3794.16 0
0 0 6251.98
Cas potrebny pro vyIT: 260 s
Celkovy cas: 319s

Rozdíl mezi výpočty cca 1,6%

Soubor Bomber.wrl
141 vrcholu
278 trojuhelniku
Velikost kroku: 0.328333
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 3, v %: 2.67246e-005
3 pruchod: chyb 55, v %: 0.00048995
teziste x:-0.0114811 y:-0.00806898 z:-0.0103237
Cas potrebny pro vyteziste: 86s
1 pruchod: chyb 0, v %: 0
2 pruchod: chyb 5, v %: 4.40213e-005
3 pruchod: chyb 18, v %: 0.000158477
Vysledny moment setrvacnosti:
2539.67 0 0
0 3666.47 0
0 0 5764.39
Cas potrebny pro vyIT: 429 s
Celkovy cas: 515s

Rozdíl mezi výpočty cca 7,7%

Soubor Bomber.wrl
141 vrcholu
278 trojuhelniku
Velikost kroku: 0.262667
1 pruchod: chyb 3, v %: 2.67836e-005
2 pruchod: chyb 6, v %: 5.35672e-005
3 pruchod: chyb 32, v %: 0.000285692
teziste x:-0.0257787 y:-0.00530615 z:-0.00605275
Cas potrebny pro vyteziste: 136s
1 pruchod: chyb 3, v %: 2.64709e-005
2 pruchod: chyb 6, v %: 5.29418e-005
3 pruchod: chyb 52, v %: 0.000458829

Vysledny moment setrvacnosti:
2376.26 0 0
0 3370.88 0
0 0 5028.1
Cas potrebny pro vyIT: 810 s
Celkovy cas: 946s

Rozdíl mezi výpočty cca 12,7%

Soubor Bomber.wrl
141 vrcholu
278 trojuhelniku
Velikost kroku: 0.197
1 pruchod: chyb 1, v %: 8.94763e-006
2 pruchod: chyb 39, v %: 0.000348957
3 pruchod: chyb 109, v %: 0.000975291
teziste x:-0.0489834 y:-0.013875 z:-0.000208807
Cas potrebny pro vyteziste: 246s
1 pruchod: chyb 1, v %: 8.84305e-006
2 pruchod: chyb 7, v %: 6.19014e-005
3 pruchod: chyb 87, v %: 0.000769345
Vysledny moment setrvacnosti:
2225.04 0 0
0 2721.16 0
0 0 4155.43
Cas potrebny pro vyIT: 1819 s
Celkovy cas: 2065s

Rozdíl mezi výpočty cca 17.3%

Příloha VI. Ovládání aplikace a screenshoty

Hlavní aplikace průlet tunelem, se ovládá následovně:

Zobrazení nápovědy	F1/H
Volba ovládání:	
Relativní (mění se působící síla na těleso)	1
Relativní s autopilotem (default)	2
Přepnutí mezi klávesnicí a myší (default=klávesnice)	M/m

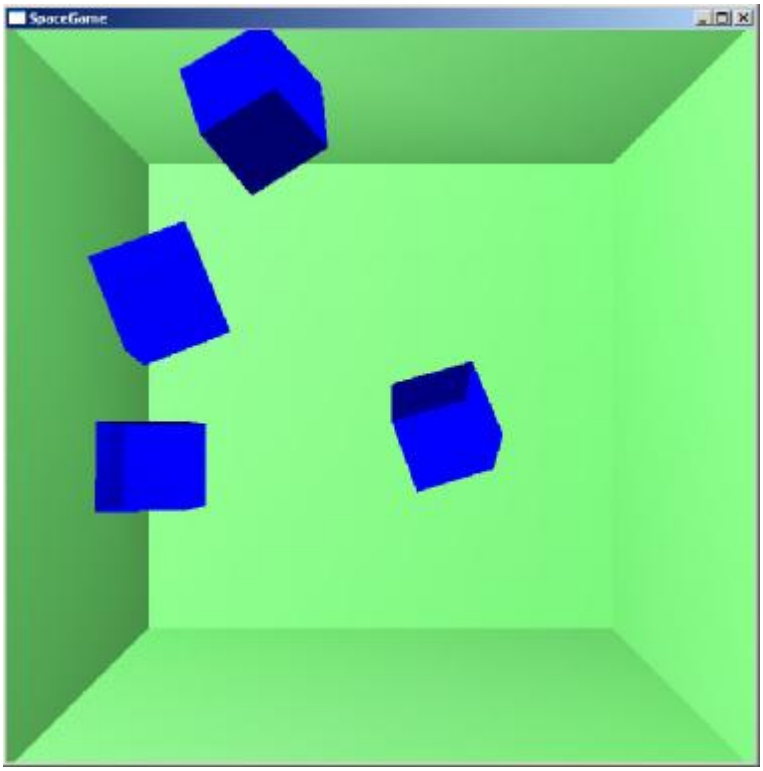
Ovládání stíhačky:	
Střelba	F, levé tlačítko myši
Ovládání translace	A,S,W,D (Q-Nahoru,E-Dolu)
Ovládání rotace šipkami	Kursorove šipky
	(CTRL+Left/Right - rotace kolem směrové osy)
Ovládání rotace myší	Myš
	(CTRL+Myš - rotace kolem směrové osy)
Zastavení veškerého pohybu (ekvivalent ruční brzdě)	SPACE

Ostatni nastavení:	
Zvýšení odporu prostředí	T
Snížení odporu prostředí	R
Přepínání kamery	V
Pozice kamery	X,C
Pozice kamery (cileni)	Tab

Ostatní aplikace jsou jen demonstrační jediné možné ovládání je:

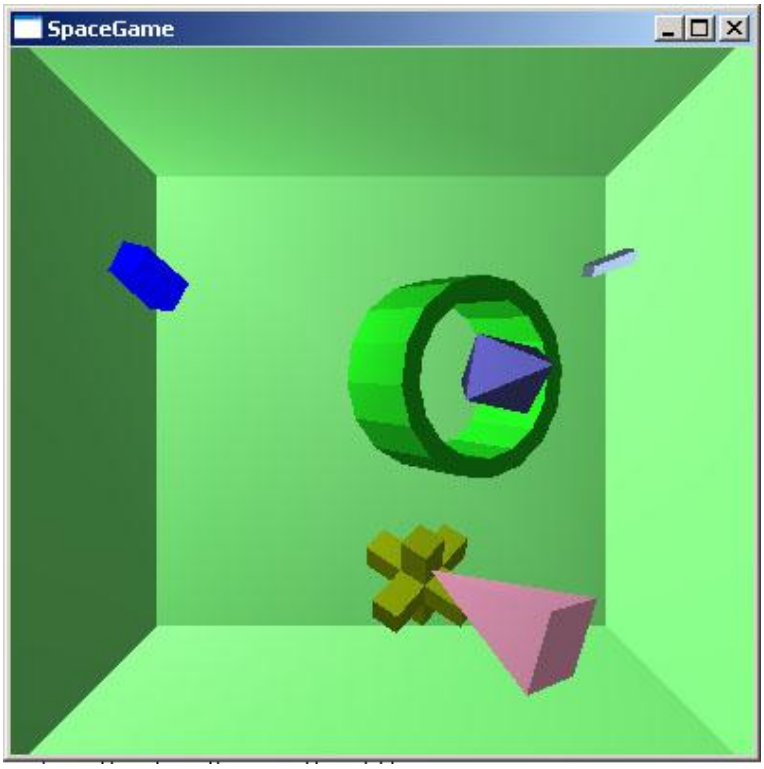
Zobrazení nápovědy	F1/H
Ostatni nastavení:	
Zvýšení odporu prostředí	T
Snížení odporu prostředí	R
Zapnutí gravitace	G+Šipka dolů
Vypnutí gravitace	G+Šipka nahoru

Aplikace Box – Cubes

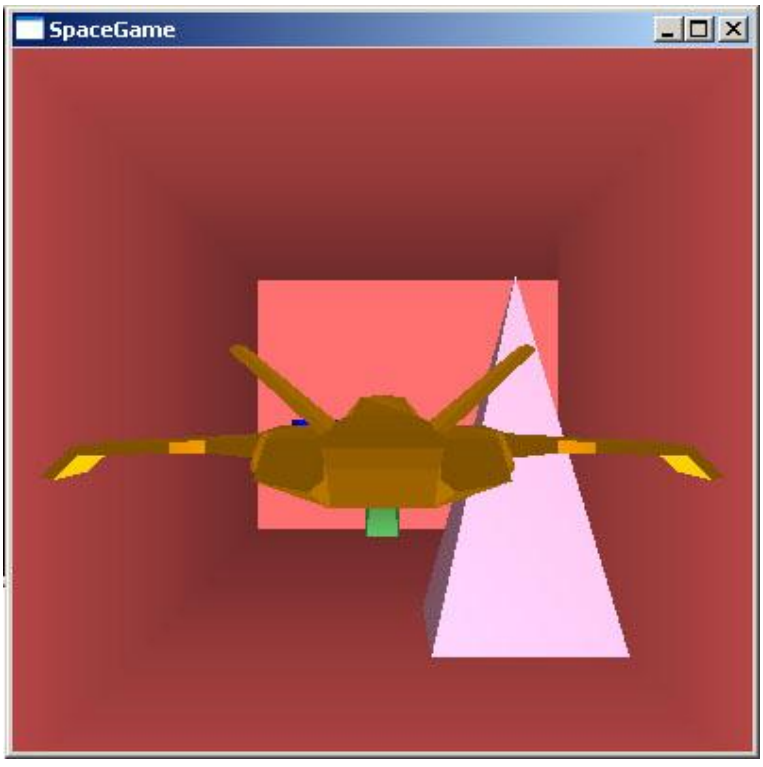


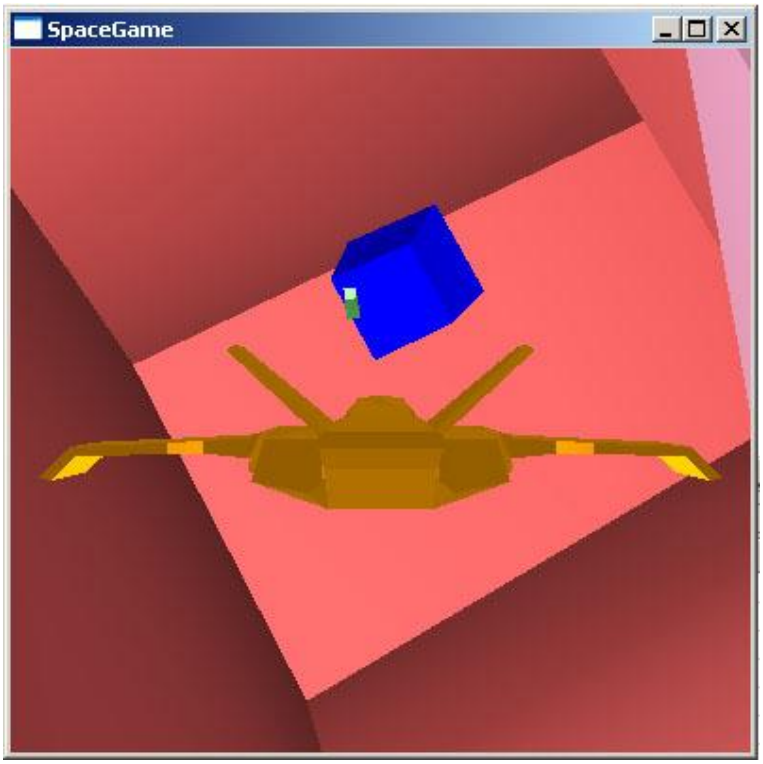
Aplikace Box - Fighter

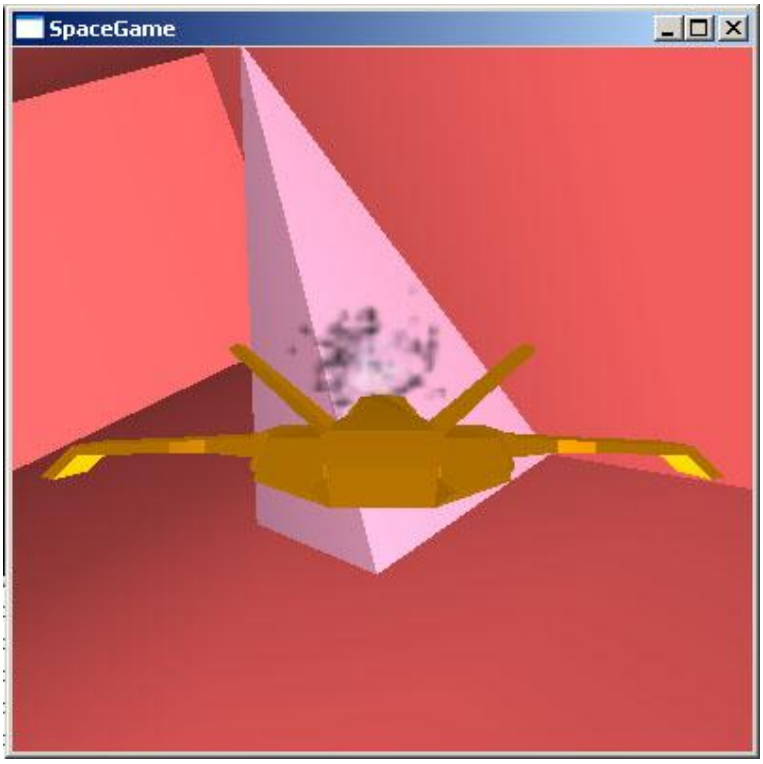
Aplikace Box – Objects



Aplikace Box – Průlet tunelem




SpaceGame


SpaceGame

Příloha VII. Tvorba tunelu

Ze souboru tunel_redme.txt:

/* komentár v textu nesmi obsahovat znaky komentáře */

přiklad vytvoření tunelu s dvěma kostkami a Kuželem v druhé kostce

druhy kostek
S - kostka s jednou volnou stěnou, pro ukončení tunelu
I - průchozí kostka
L - kostka s dvěma volnými stěnami směr otočeni je doleva (ve směru osy -z do kostky vstupuje, a -x vystupuje)
R - kostka s dvěma volnými stěnami směr otočeni je doprava (ve směru osy -z do kostky vstupuje, a x vystupuje)

následuje seznam kostek s jejich obsahem

I (0 0 0)(0 0 1) 0.0 END END
/* první kostka je prázdná 2xEND je nutné zadat */

I (0 0 -1)(0 0 1) 0.0 Frustum.wrl (0 0 0)(0 0 1) 1.0 END Kostka.wrl (0 0 0)(0 0 1) 0.0 END
/* druha kostka obsahuje staticky kužel jinak je prázdná */

obecná definice:
Tunel00 seznam_statických_těles END seznam_dynamických_těles END
Tunel01 ...

Konkrétní definice:
I (0 0 -1)(0 0 1) 0.0 Frustum.wrl (0 0 0)(0 0 1) 1.0 END Kostka.wrl (0 0 0)(0 0 1) 0.0 END

/* je důležité dodržet mezery/tabulátory editor je jen jednoduchý textový parser*/
/* za každou položkou(číslem) musí následovat mezera mimo dvou po sobě jdoucích závorek (0 0 0)(0 0 0) */

I	Kostka
(0 0 -1)	pozice kostky absolutně v jednotkách 1 kostka
(0 0 1)	osa rotace kostky (pouziva se pro kostku L a R)
0.0	uhel rotace v násobku Pi (0.5=90°, 1=180°)
Frustum.wrl	název souboru se statickým tělesem, které má kostka obsahovat
(0 0 0)	relativní pozice tělesa uvnitř kostky v jednotkách 0.5ksotky (0,0,0) = střed kostky (1,1,1) = pravý horní roh kostky !!! zaleží, kde je umístěn střed tělesa !!! - pokud je v těžišti je nutné znát vzdálenost od těžiště k "přichycené" straně
(0 0 1)	osa rotace tělesa (většinou "přichytnou" stranou ke kostce)
1.0	uhel rotace v násobku Pi (0.5=90°, 1=180°)
END	ukončení seznamu
Kostka.wrl	název souboru s dynamickým tělesem, které má kostka obsahovat
(0 0 0)	relativní pozice tělesa uvnitř kostky v jednotkách 0.5ksotky

(0,0,0) = střed kostky (1,1,1) = pravý horní roh kostky
!!! zaleží, kde je umístěn střed tělesa !!! - pokud je v těžišti je nutné znát vzdálenost od těžiště k "přichycené" straně

(0 0 1)	osa rotace tělesa (většinou "příchytnou" stranou ke kostce)
1.0	uhel rotace v násobku Pi (0.5=90°, 1=180°)
END	ukončení seznamu

!! průniky objektu jsou přípustné jen mezi statickými tělesy !!
!! za případné průniky mezi objekty nese odpovědnost tvůrce tunelu :D !!

Příloha VIII. SgInterface

```
enum t_tlacitko{Mouse1=0, LeftB=0, Mouse2=1, MiddleB=1, Mouse3=2,
RightB=2};
enum t_klavesa {keyQ=0, keyW, keyE, keyA, keyS, keyD,
                        keyLeft, keyRight, keyUp, keyDown,
                        keySpace, keyCTRL, keyTAB, keyM,
                        keyX, keyC, keyV, keyT, keyR, keyG, keyF, keyH,
                        key1, key2, key3,
                        keyF1};

class SgMouse     // trida pro mys
{
protected:
      map <t_tlacitko, SoMouseButtonEvent::Button> tlacitko;           //
mapa (kontejner) tlacitek
      map <t_tlacitko, SoMouseButtonEvent::Button>::iterator Tl; //
iterator v mape
      map <t_tlacitko, bool> stisknuto;        // seznam stisknutych
tlacitek
      map <t_tlacitko, bool>::iterator St;     // iterator do mapy
public:
      SgMouse();
      void setButton (const SoEvent * event);  // nastaveni tlacitka
      bool getButton (t_tlacitko tl);               // zyskani hodnoty
tlacitka
};

class SgKeyboard  // trida pro klavesnici
{
protected:
      map <t_klavesa, SoKeyboardEvent::Key> klavesa;       // mapa klaves
      map <t_klavesa, SoKeyboardEvent::Key>::iterator Kl;  // iterator
      map <t_klavesa, bool> stisknuta;         // mapa stisknutych klaves
      map <t_klavesa, bool>::iterator St;      // iterator

public:
      SgKeyboard();
      void setKey (const SoEvent * event);                             //
nastaveni klavesy
      bool getKey (t_klavesa key) { return stisknuta[key]; }     // zyskani
kavesy
```

```
      void falseKey (t_klavesa key) { stisknuta[key]=false; }    //
prirazeni false do klavesy
      void trueKey (t_klavesa key) { stisknuta[key]=true; }// prirazeni
tzrue do klavesy
};

class SgCamera          // trida pro kameru
{
protected:
      SoPerspectiveCamera * Camera;
      float distance;         //vzdalenost kamery od objektu
      float height;           //vyska kamery od objektu

public:
    SgCamera();
      ~SgCamera();
      SoPerspectiveCamera * getRoot() { return Camera; }

      void setRot(SbRotation rot);  //nejprve nastavit rotaci, podle ni se
pocita pozice
      void setPos(SbVec3f pos);           //opacne bude skakat kamera
      void setDistance(float d) { this->distance=d; }// nastaveni
vzdalenosti
      void setHeight(float h) { this->height=h; }          // nastaveni
vysky kamery
};
class SgLight           // trida pro svetla
{
protected:
public:
      virtual void setRoot(SoSeparator * root) {};
      virtual void setPos(SbVec3f pos) {};
      virtual void setDir(SbVec3f pos) {};
};
class SgPointLight:public SgLight   // bodove svetlo
{
protected:
      SoPointLight * Light;
public:
      SgPointLight();
      void setRoot(SoSeparator * root);
      void setPos(SbVec3f pos);
      void setDir(SbVec3f dir) {}
};
class SgConeLight:public SgLight    // kuzelove svetlo
{
protected:
      SoSpotLight * Light;
public:
      SgConeLight();
      void setRoot(SoSeparator * root);
      void setPos(SbVec3f pos);
      void setDir(SbVec3f dir);
      };
```

Příloha IX. SgObject

```
class SgObject {
public:
	enum MODEL_SET { RENDER = 0x01, COLLISION = 0x02, ALL = 0x03 };
	enum TYPE_OBJECT_BB { SPHERE = 0x01, AABB = 0x02, OBB = 0x03, KDOP =
0x04 };
	enum TYPE_COLLISION { BODY = 0x01, BALL = 0x02 };
	enum TYPE_OBJECT { OBJECT = 0x01, FIGHTER = 0x02, MISSILE = 0x03,
CAMERA = 0x04, OTHER = 0x05 };
protected:
	// Open Inventor Variables
	SgoManager * Manager;                    // sprava objektu
	SoSeparator * Root;                      // koren objektu
	SoMatrixTransform * MatrixTransform;// transformacnimatice objektu
	// Model Variables
	SoNode * Model;                          // model objektu
	SoNode * CollisionModel;                 // kolizni model objektu
	SbVec3f Scale;                           // for MatrixTanfosrm
	// User Variables
	TYPE_OBJECT_BB TypeBB;              // typ BoundingBoxu
	TYPE_COLLISION TypeCollision;       // typ jak objekt koliduje
	TYPE_OBJECT Type;                   // typ objektu
	bool Reference;                     // zda definuje vlastni podprostor
	bool Static;                        // zda je staticky objekt
	vector<int> ID;                // identifikace reference/podprostoru
	float SphereRadius;            // polomer ohranicujici koule telesa
	float ActionRadius;            // polomer akcniho radiu koule telesa
	float FieldAABB[6];     // rozmery osove orientovanehoohranicujiciho
                           kvadru telesa ([ x,-y,-z],[x,y,z])
	float FieldAAAB[6];     // rozmery osove orientovaneho akcniho kvadru
                           telesa ([-x,-y,-z],[x,y,z])
	vector<SgLight*>  Light;            // ukazatel na pouzite svetla
	bool selectLight;                   // zda je na objektu svetlo
	// Physical Variables
	float Mass;                         // hmotnost objektu
	float CoefficientOfRestitution;     // soucinitel odrayivosti
	SbVec3f Position;              // pozice objektu // for MatrixTanfosrm
	SbVec3f OldPosition;                // stara pozice
	SbVec3f Velocity;                   // rychlost
	SbVec3f AngularVelocity;            // uhlova rychlost
	SbVec3f AngularMomentum;            // uhlovy moment
	SbVec3f Force;                 // translacni sili pusobici na teleso
	SbVec3f Torque;                // rotacni sili pusobici na teleso
	SbVec3f ConstForce;            // konstantni translacni sili pusobici
na teleso
	SbVec3f ConstTorque;           // konstantni rotacni sili pusobici na
teleso
	float Damp;                    // tlumeni (odpor prostredi)
	SbRotation Orientation;        // orientace telesa
	SbRotation OldOrientation;     // stara orientace telesa
	SbRotation CameraOrientation; // orientace kamery
	SbRotation ScaleOrientation;  // for MatrixTanfosrm
	SbMatrix InertiaTensor;        // moment setrvacnosti
	SbMatrix InversIT;             // inverzni moment setrvacnosti
```

```
      SbMatrix ActualInversIT;      // moment setrvacnosti podle aktualniho
natoceni

      // Metods
      void createHiddenScene();     // vytvori pocatecni scenu a
inicializuje root
      void releaseHiddenScene();    // uvolni scenu
      void updateMatrixTransform(); // aktualizuje transformacni matici

      void loadInfo(SoNode * model);// nahraje doplnujici informace o
modelu
      void massOverIT();            // mass/IT
      SbMatrix OrthoNormalize(SbMatrix & m);// provede ortho normalizaci
matice

public:
/************** jen vybrané metody *********************/
      SgObject();
      SgObject(const char *model, const char *collisionModel = NULL,
                  const SbVec3f &pos = SbVec3f(0.f,0.f,0.f), const SbVec3f
                  &dir = SbVec3f(0.f,0.f,1.f),float radius = 0.f, float
                  weight = 1.f);
      virtual ~SgObject();

      void RotLR(const float radians);    //LeftRight //otaceni vlevo vpravo
      void RotUD(const float radians);    //UpDown    //otaceni nahoru dolu
      void RotAR(const float radians);    //all round //otaceni kolem dokola
v ose pohybu
      void TorqueLR(const float torque); //Left/Rignht     //otaceni vlevo
vpravo
      void TorqueUD(const float torque); //otaceni nahoru dolu
      void TorqueAR(const float torque); //all round       //otaceni kolem
dokola v ose pohybu

      virtual void timeStep(double time, double dt);
}
```

Příloha X. SgoManager

```
class SgoManager {
public:
      #define G 6.67e-11f
      enum CollidingType { BODY = 0x01, BALL = 0x02 };

private:
      SoSeparator * sgoRoot;       // koren sceny
      SbList<SgObject*> sgoList;   // seznam objektu sceny
      SbList<Billboard*> sgoBillboard;    // seznam billboardu


      SgObject * Active;           // aktivni objekt
      SgObject * CameraFree;       // objekt stíhačky s upnutou kamerou
      SgObject * CameraFighter;    // objekt kamery

      bool free;              // zda je kamera folna nebo upnuta
      bool fighterDestroy;    // stihac znicen
      SoPerspectiveCamera * Camera; // kamera
      SoGroup * collisionRoot;     // skupina pouzivana pro detekci kolize
      SbVec3f Force;               // translacni sili pusobici na teleso
      SbVec3f Torque;              // rotacni sili pusobici na teleso
      float Damp;                  // tlumeni

      float epsilon;
      typedef struct    // uchovava info o kolidujicich objektech
      {
            bool Collision;              // nastala kolize?
            CollidingType Type;          // typ kolize
            int Index[2];                // index kolidujicich objektu
            vector<SbVec3f> Points;            // body
            vector<SbVec3f> Triangles[2];      // trojuhelniky
            SbVec3f collidingPoint;            // bod srazky
            SbVec3f collidingNormal;           // normala srazky
      } Type_CollidingSgos;
      vector <Type_CollidingSgos > CollidingSgos;

      SoIntersectionDetectionAction * ida;  // kolizni trida
static SoIntersectionDetectionAction::Resp
      intersectionCb(void *closure,
      const SoIntersectingPrimitive *p1,
      const SoIntersectingPrimitive *p2); // callback tridy IDA

    static bool partOfTriangle(SbVec3f a, SbVec3f b, SbVec3f c);
// ukosti jaky uhle sviraji vektory
      void checkCollision(const double time, const double dt);
// detekuje kolizi
      bool checkID(SgObject * sgo1, SgObject * sgo2);
// porovnava zda jsou telesa ve stejnem prostoru
      void checkTriangleCollision(int sgo1, int sgo2);
// detekuje kolizi na trojuhelnikove urovni
      void checkCollisionWith(SgObject * sgo, int index);
// urcuje konkretni kolizi telesa se vsim ostatnim
      float computeTr(SbVec3f ot, SbMatrix m, SbVec3f o);
```

```
// pomocny vypocet pro vypocet energie teles
      void resolveCollision(const double time, const double dt);
// vyhodnocuje kolize
      void resolveCollisionBALL(int index1, int index2, bool stat);
// vyhodnocuje kolize koule vs. koule
      void resolveCollisionBODY(int index, int index1, int index2, bool
stat);
// vyhodnocuje kolize pevne teleso vs. pevne teleso
      void ExplosionMissile(int index, const double time, const double dt,
bool stat);
// vyhodnocuje explozi

      float diffExp2(SbVec3f v1, SbVec3f v2);  // rozdil druhych mocnin
      void workIntersectionPoints(int index);  // pocita body kolize
      void workIntersectionVertexs(int index); // pocita trojuhleniky
kolize
      SbVec3f getIntersectionPoint(int index); // pocita kolizni bod
      void computeNormal(int index, bool stat);// pocita normalu

public:
/************** ostatní metode neuvádím ********************/
};
```

Příloha XI. Billboard

```
class Billboard{
private:
      SoSeparator * Manager;              // spravce objektu
      SoSeparator * Root;                 // koren sceny billboardu
      SoMatrixTransform * MatrixTransform; // transformacni matice
      SbVec3f Position;                   // pozice
      SbVec3f Axis;                       // osa rotace
      float Angle;                        // uhel rotace
      SoCoordinate3 * Coordinate;         // vrcholi billboardu
      SoIndexedTriangleStripSet * TriangleStripSet;  // indexi do vrcholu
      SoTexture2 * Texture;               // textura
      SoTextureCoordinate2 * TextureCoordinate;      // koordinaty textury
      SoPerspectiveCamera * Camera;       // kamera

      float Dimension;        // rozmer billboardu
      float FPS;              // rychlost animace
      double lastTime;        // cas posledniho snimku
      int TextureDimension;   // rozmer textury pro animaci 1=bez animace
cislo 2^x
      int Frame;                    // frame animace  od 1 do TD*TD
      bool Used;                    // activni/neaktivni
(viditelny/neviditelny)

      void updateDimension(float dimension);    // aktualizuje rozmery
billboardu
      void updateTextureCoordinate(int frame); // aktualizuje texturovaci
souradnice

public:
/************** jen vybrané metody *********************/
      Billboard();
      Billboard(float dimension, char * file);

      void useBillboard();                      // použije billboard
      void useBillboard(SbVec3f pos);

      void timeStep(double time, double dt);    // aktualizuje billboard
      };
```