Vysoké učení technické v Brně

Fakulta informačních technologií

Ročníkový projekt

Petr Maštera

2006

Zadání ročníkového projektu/4623/2005/xmaste00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2005/2006

Zadání ročníkového projektu

Řešitel: **Maštera Petr**
Obor: Výpočetní technika a informatika
Téma: **Knihovna pro detekci kolizí mezi objekty scény**

Pokyny:

1. Seznamte se s teorií virtuálních scén a algoritmy pro detekci kolizí.
2. Vytvořte jednoduchou virtuální scénu a implementujte základní algoritmy pro detekci kolizí mezi objekty scény.
3. Vytvořte algoritmy pro pohyb uživatele ve scéně tak, aby uživatel nemohl procházet pevnými objekty.
4. Proveďte měření časové náročnosti a diskuzi nad možnými dalšími směry vývoje.
5. Výsledky publikujte na internetu.

Literatura:

- Sánchez-Crespo, Core Techniques and Algorithms in Game Programming, 2003
- Open Inventor tutoriál na ROOT.CZ
- Josie Wernecke, The Inventor Mentor, Addison-Wesley Professional, 1994, ISBN: 0201624958

Rozsah:

Vytištěná technická zpráva o rozsahu nejméně 20 normovaných stran, úplná technická a programová dokumentace včetně zdrojových textů programů na přiložené disketě.

Vedoucí: **Pečiva Jan, Ing.**, UPGM FIT VUT
Datum zadání: 15. října 2005
Datum odevzdání: 2. května 2006

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 svého semestrálního projektu 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

Práce se zabývá řešením kolizí mezi objekty scény a následným vyhodnocením těchto kolizí. V ročníkovém projektu byly zadané algoritmy řešeny pomocí některých funkcí grafické knihovny Open Inventor a aplikováním fyzikálních vzorců a vztahů. Algoritmy pro detekci kolizí a algoritmy pro řešení kolize výbuchem, jednoduchým a fyzikálním odrazem jsou implementovány v programu, který je součástí práce.

Klíčová slova

Kolize, detekce kolizí, jednoduchý odraz, jednoduché vyhodnocení kolize, fyzikální odraz, fyzikální vyhodnocení kolize, billboarding, 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 ročníkového projektu je naimplementovat algoritmy pro detekci a vyhodnocování kolize v 3D prostoru a demonstrovat je na vhodných příkladech.

Cíle, které tato práce má splnit, jsou:

1. Seznámit se s teorií virtuálních scén a algoritmů pro detekci a vyhodnocování kolizí.
2. Vytvořit jednoduchou virtuální scénu a implementovat základní algoritmy pro detekci kolizí mezi objekty scény.
3. Vytvořit algoritmy pro pohyb uživatele ve scéně tak, aby uživatel nemohl procházet pevnými objekty.
4. Provést měření časové náročnosti a diskuzi nad možným dalším směrem vývoje.
5. Výsledky publikovat na internetu.

V ročníkovém projektu bylo snahou vytvořit algoritmus simulující co nejreálněji problematiku detekce a vyhodnocení kolize. K řešení problému byla použita knihovna Open Inventor a celá škála jejích objektů sloužící pro snadnou implementaci grafických aplikací. 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 splňují zadání projektu, jsou:

- detekce kolize mezi objekty scény
- vyhodnocení kolize explozí
- vyhodnocení kolize jednoduchým odrazem
- vyhodnocení kolize na základě fyzikálního modelu

Veškeré implementované algoritmy jsou demonstrovány v jednoduché virtuální scéně. Pro tuto virtuální scénu je nutné zajistit objekty scény a jejich obsluhu, dále pak pohyb uživatele a kamery. S tím souvisí další problémy, které je třeba v rámci této práce vyřešit:

- vytvoření virtuální scény
- vytvoření objektů scény
- zajištění pohybu objektů ve scéně

- zajištění pohybu uživatele, kamery ve scéně
- použití billboardingu pro vyhodnocování kolize explozí

Písemná práce je rozdělena na dvě hlavní kapitoly:

- Teoretický úvod (literární rešerše)
- Implementaci (praktické řešení práce)

Teoretický úvod k ročníkovému projektu

1. Detekce kolizí

Detekce kolizí hledá stavy, kdy dva objekty v prostoru pronikají do sebe. Tento stav se nazývá kolize a obvykle je vždy nechtěný. Patří mezi často řešené problémy při programování počítačových her. Dále se detekce kolizí používá například v:

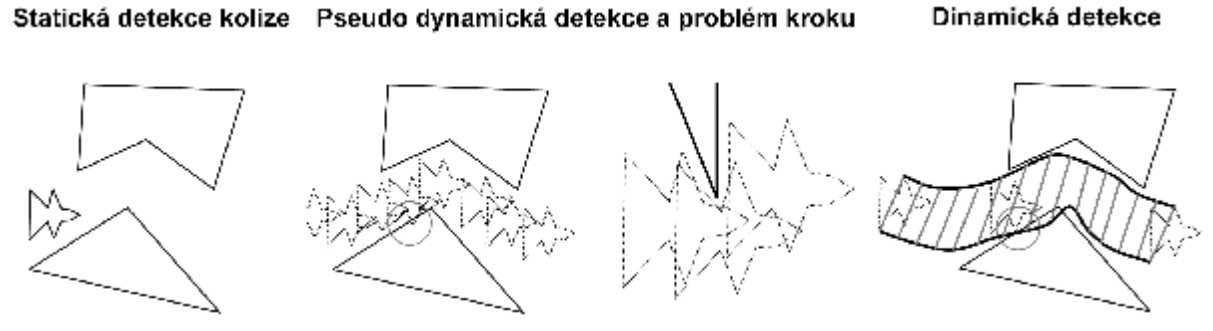
- Robotice (plánování cesty robota)
- Animačních a simulačních systémech (test konzistence scény, fyzikální modelování)
- CAD, strojírenském průmyslu (robustní a numericky stabilní implementace)
- Molekulárním modelování

Základní rozdělení detekce kolizí:

Statická detekce kolizí. Provádí se jeden test v konkrétní konfiguraci těles. Používá se pro statické scény.

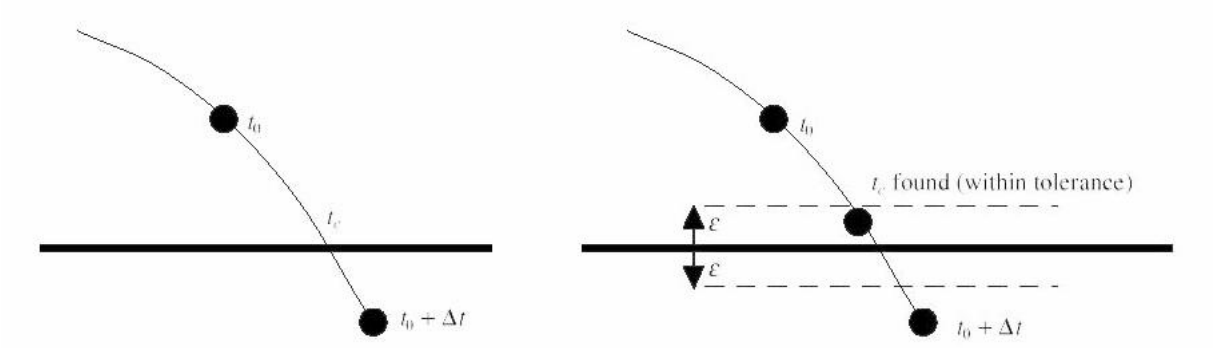
Pseudo-dynamická detekce kolizí. Provádí se testy na diskrétní množině konfigurací objektu odpovídající jeho pohybu. Tento druh detekce je nejčastěji použit pro detekci kolizí v počítačových hrách. Přesnost detekce závisí na velikosti diskrétního kroku.

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



U klasické diskrétní metody detekce se v každém kroku simulace integrují fyzikální veličiny, na základě kterých se mění poloha a rotace objektu. Při pseudo-dynamické detekci mohou nastat dva stavy, nekolizní stav a průnik objektů.

Při optimalizované pseudo-dynamické detekci pokud dojde k průniku, tak se „vrátíme v čase“ a zmenšíme krok na polovinu, a znovu integrujeme hodnoty. Tuto operaci opakujeme dokud nedosáhneme požadované přesnosti. Rozdíl je patrný z obrázku.



V projektu nemohla být použita dynamická detekce, protože byla použita funkce Open Inventoru, která je pseudo-dynamická a neumožňuje se „vracet v čase“.

Detekce kolizí je v reálných situacích komplexní problematika, protože se často musí řešit kolize velkého množství složitě tvarovaných těles. Tento problém se rozpadá na dva podproblémy. Prvním je rozdělení velkého počtu objektů na menší skupiny, uvedený problém nazýváme dělením prostoru. Druhým problémem je zjednodušení složitého tvaru těles, aby se složité výpočty kolizí na úrovni trojúhelníků nemusely provádět pro všechna tělesa ve skupině. Tato problematika se nazývá hierarchie obalových těles. V následujících bodech si řešení těchto problémů blíže popíšeme.

1.1. 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.

Základní ideou je „rozbít“ prostor na jednotlivé části s podobnou strukturou. Detekce kolizí se poté provádí jen v tomto lokálním podprostoru. Metody, kterými lze dělit podprostor, jsou:

- Využití sítě čtyřstěnů
- Voxelová mřížka
- BSP strom

V mé práci je použito netypické, ale pro tento případ scény vhodné a jednoduché, sférické řešení dělení prostoru. Sférické oblasti v projektu jsou reprezentovány jednotlivými slunečními soustavami nebo planetami.V tomto případě se měří vzdálenost středů dvou objektů (příp. kamery a středu objektu). Pokud je obdržená hodnota menší než pevně daná hranice (konkrétně v tomto projektu akční rádius sluneční soustavy), nachází se těleso v daném sférickém prostoru.

1.2. Hierarchie obalových těles

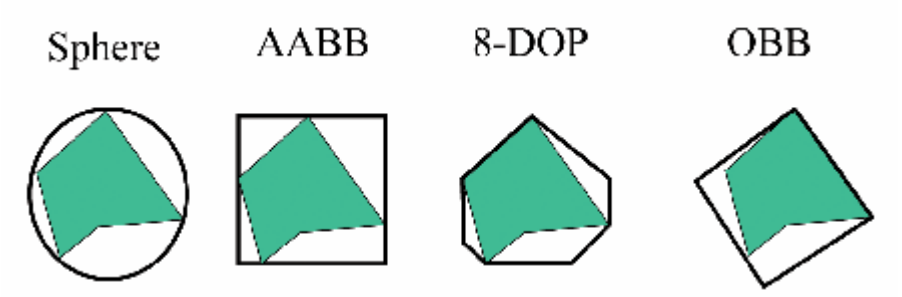
Veškeré objekty v počítačové grafice jsou reprezentovány trojúhelníky. Vyhodnocovat detekci přímo mezi trojúhelníky je strojově velmi náročné, proto se objekty „zabalují“ do obalových těles. Základním problémem je zvolit vhodné obalové těleso tak, aby obklopovalo původní model co nejtěsněji a aby testování dvou obalových těles probíhalo co nejrychleji.

Sphere je nejjednodušším obalovým tělesem. V testu porovnáváme vzdálenosti mezi dvěma sférami. Její nevýhodou je, že je nejméně citlivá na tvar tělesa, má největší procento obsaženého volného prostoru.

AABB Axis Aligned Bounding Box (osově orientované obalové těleso) je jednoduché a v testu stačí porovnávat překrytí jednotlivých os. Teoreticky by mělo být nejrychlejší, protože se požívají jen operátory porovnání. Je opět málo citlivé na tvar tělesa.

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

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



2. Vyhodnocování kolizí

Při zjištění kolize nastává otázka jak takovou kolizi vyhodnotit. Vyhodnocení kolize záleží na druhu aplikace 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:

- 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 srážce dojít k zastavení některého z těles, případně

vzájemnému klouzání těles. Dalším řešením je odraz těles. Na otázku jak vyhodnotit srážku by se našlo spousty dalších řešení, v projektu se budeme zabývat dvěmi zvolenými řešeními, a tím je výbuch a odraz.

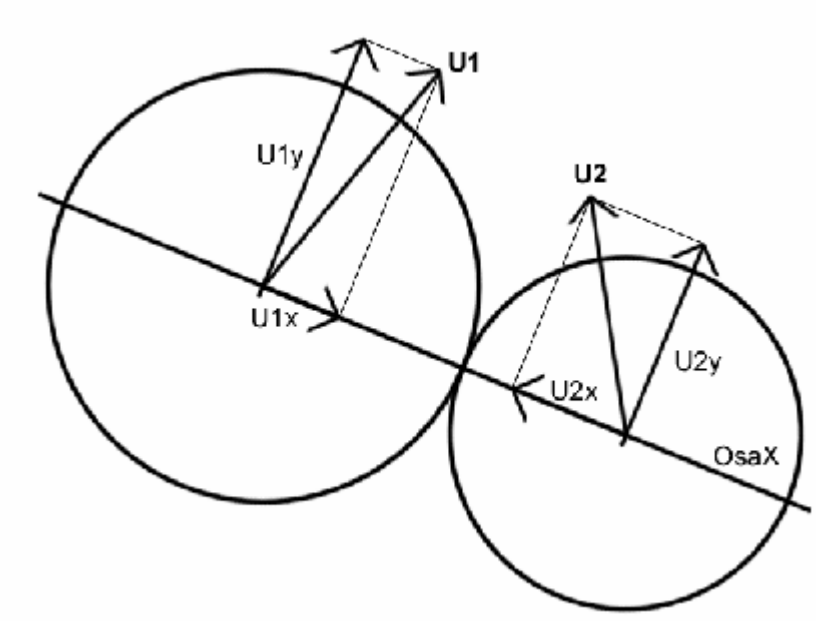
Problém výbuchu je řešen jednoduše a elegantně promítnutím animace výbuchu na plochu kolmou ke kameře. Tato technika se nazývá billboarding a budeme se jí zabývat v samostatné kapitole. Proto se nyní soustředíme na problém odrazu. V projektu se zabýváme jednoduchým a fyzikálním odrazem. Jednoduchý odraz je řešen také na podle fyzikálních zákonů, ale jen pomocí jednoduchých vzorců pro srážku a odraz dvou koulí, kde se nebere v úvahu rotace tělesa a další působící vlivy. Proto je nazván jednoduchý. Fyzikální odraz se hlouběji opírá o fyzikální zákony a bere v úvahu rotaci tělesa. V následujících dvou kapitolách si objasníme vzorce, které jsou v projektu použity.

2.1. Jednoduchý odraz (koule)

Při jednoduchém odrazu dvou koulí nám pomohou fyzikální zákony. Z fyzikálního hlediska řešíme šikmou dokonale pružnou srážku a odraz dvou koulí. Pružná srážka je definována poučkou:

Při pružné srážce se obecně mění kinetická energie jednotlivých těles, která se srážky účastní. Celková kinetická energie soustavy před srážkou i po srážce je však stejná.[citace]

Dále uvedené vzorce a výpočty vycházejí z fyzikální podstaty problému, jejich odvození není podstatné, protože v projektů využíváme jen výsledné odvozené vztahy. Tyto vztahy nyní vysvětlím. Situaci při srážce dvou koulí reprezentuje následující obrázek:



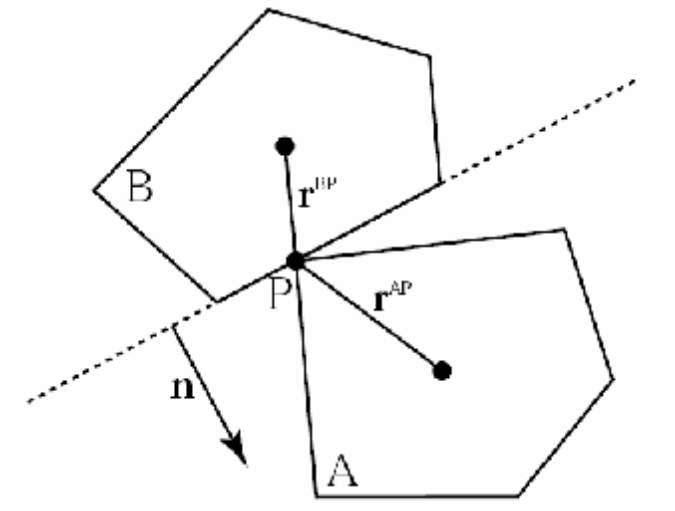
Vektory U1 a U2 představují rychlost koulí v čase nárazu. Středy dohromady spojuje osa X, na které leží vektory U1x a U2x, což jsou vlastně průměty rychlosti do roviny X. U1y a U2y jsou

projekce rychlosti na osu, která je kolmá k ose X. K jejich výpočtu postačí jednoduchý skalární součin. Do následujících rovnic dosazujeme ještě čísla M1 a M2, která vyjadřují hmotnost koulí. Snažíme se vypočítat orientaci vektorů rychlosti U1 a U2 po odrazu. Budou je vyjadřovat nové vektory V1 a V2. Čísla V1x, V1y, V2x, V2y jsou opět průměty. Postup výpočtu objasním v jednotlivých bodech:

- Najdeme osu (vektor) spojující oba středy OsaX= (střed2 - střed1)
- Osu normalizujeme na jednotkový vektor.
- Nalezneme projekce do X-ové osy **U1x = OsaX * (OsaX dot U1),**
  U2x = -OsaX * (-OsaX dot U2)
- Nalezneme projekci do Y-ové osy **U1y = U1 - U1x, U2y = U2 - U2x**
- Nalezneme nové rychlosti
  V1x = ((U1x * M1) + (U2x * M2) - (U1x - U2x) * M2) / (M1 + M2),
  V2x = ((U1x * M1) + (U2x * M2) - (U2x - U1x) * M1) / (M1 + M2),
  V1y = U1y, V2y = U2y
- Vypočítáme konečné rychlosti **V1 = V1x + V1y, V2 = V2x + V2y**

2.2. Fyzikální odraz

Fyzikální odraz nejprve nastíníme v 2D rovině, a poté uvedené vzorce převedeme do třetího rozměru. Nejlépe začít obrázkem, který názorně ukazuje srážku dvou objektů:



Obrázek ukazuje srážku tělesa označeného A s tělesem označeným B v místě srážky P. Vektory (ramena) směřující od středu tělesa k bodu P označíme jako r_{BP} a r_{AP}. Dále potřebujeme znát rychlost v místě srážky, které označíme v_{AP} pro těleso A a v_{BP} pro těleso B. Při výpočtu však počítáme s relativní rychlostí v bodě P, kterou označíme v_{AB} a vypočítáme:

1. $$v_{AB} = v_{AP} - v_{BP}$$

Důležité je určit, které těleso narazilo do kterého, abychom mohli určit směr normálového vektoru srážky. V našem případě narazilo těleso A do tělesa B, proto je normálový vektor n kolmý na plochu tělesa B v místě nárazu tělesa A. Pokud známe normálový vektor, můžeme vypočítat relativní „normálovou“ rychlost.

$$v_{AB} * n = (v_{AP} - v_{BP}) * n \tag{2.}$$

Každá srážka není dokonale pružná, proto se zavádí koeficient odrazu značený e. Koeficient odrazu ovlivňuje výslednou sílu (rychlost).

$$v_2^{AB} * n = -e * v_1^{AB} * n \tag{3.}$$

Při vyhodnocování výsledné rychlosti objektů musíme znát sílu, se kterou se od sebe objekty odrazily. Tato síla se nazývá impuls síly a značí se J. Na základě této síly můžeme vyhodnotit rychlosti po odraze.

$$v_2^A = v_1^A + \frac{J}{M_A} * n \tag{4a.}$$

$$v_2^B = v_1^B + \frac{J}{M_B} * n \tag{4b.}$$

Rovnici 3, která udává relativní rychlost před a po srážce, a rovnici 1, která definuje relativní rychlost, dosadíme do rovnic 4a a 4b.

$$(v_2^A - v_2^B) * n = -e * (v_1^A - v_1^B) * n$$

$$v_1^A * n + \frac{J}{M_A} * n * n - v_1^B * n + \frac{J}{M_B} * n * n = -e * v_1^{AB} * n \tag{5.}$$

Z této rovnice vyjádříme impuls síly J.

$$J = \frac{-(1+e) * v_1^{AB} * n}{n * n * \left(\frac{1}{M_A} + \frac{1}{M_B} \right)} \tag{6.}$$

Zatím jsme řešili srážku, bez ohledu na rotaci, nyní přidáme k odvozeným vzorcům i rotaci. Pro výslednou rychlost s ohledem na rotaci dostáváme rovnici pro objekt A, které platí i pro objekt B.

$$v_2^{AP} = v_2^A + w_2^A * r_{AP} \tag{7.}$$

Nyní přepíšeme rovnice 4a a 4b do podoby pro rychlost i úhlovou rychlost pro objekt A, pro objekt B jsou rovnice opět podobné. Ve vzorci se objevila nová veličina I, která reprezentuje moment setrvačnosti tělesa.

$$v_2^A = v_1^A + \frac{J}{M_A} * n \tag{8a.}$$

$$w_2^A = w_1^A + \frac{r_{AP} * J * n}{I_A} \tag{8b.}$$

Výsledný impuls síly odvodíme obdobně jako v předcházejícím případě. Začneme rovnicí 3, do které dosadíme rovnici 1. a nahradíme v rovnici 7.. Poté dosadíme do rovnic 8a a 8b pro objekt A i B a vyjádříme J.

$$J = \frac{-(1+e) * v_1^{AB} * n}{\frac{1}{M_A} + \frac{1}{M_B} + \frac{(r_{AP} * n)^2}{I_A} + \frac{(r_{BP} * n)^2}{I_B}} \qquad 9.$$

Tímto jsme dosáhli odvození impulsu síly pro 2D rovinu a pomocí tohoto impulsu můžeme vypočítat novou rychlost a novou úhlovou rychlost objektů. Pro 3D prostor se některé vektorové veličiny „rozrostou“ na matice a výpočet se tím značně zkomplikuje. Setrvačnost tělesa I je nyní udána maticí místo vektorem. Již nebudu rozebírat složité odvození a napíši jen vzorce použité pro samotný výpočet fyzikálního odrazu ve 3D .

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

$$v_2^A = v_1^A + \frac{J}{M_A} * n$$

$$w_2^A = w_1^A + \frac{r_{AP} * J * n}{I_A}$$

$$v_2^B = v_1^B + \frac{J}{M_B} * n$$

$$w_2^B = w_1^B + \frac{r_{BP} * J * n}{I_B}$$

3. Billboarding

Billboarding je technika, která mění orientaci objektu tak, aby byl čelně k cíli obvykle ke kameře. Tato technika je poměrně populární ve hrách a v aplikacích, které potřebují velké množství polygonů. I přes neustálý vývoj grafických karet narůstá potřeba zpracovávat stále větší počty polygonů. Billboarding umožňuje tento počet snížit použitím triku (finty) s vhodnou texturou.

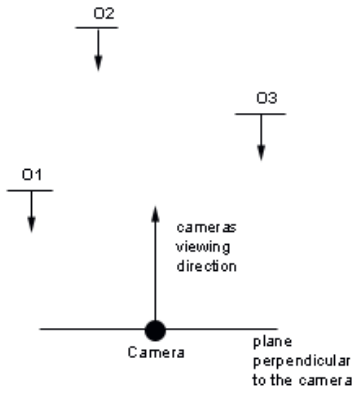
Klasickým příkladem je strom. I na skromnou reprezentaci stromu je potřeba velké množství polygonů. Billboarding nám umožňuje nahradit geometrii jednoduchou texturou aplikovanou na čtverec či dva trojúhelníky. Zároveň garantuje, že textura je vždy natočená čelně ke kameře, proto si uživatel nikdy neuvědomí, že „strom“ je ve skutečnosti plochý čtverec textury, dokud jím neprojde skrz, v tomto případě je trik (finta) odhalen. Nevýhodou je, že stínování stejné textury (stromu) nebude správné pro všechny orientace. 3D strom pozorovaný z různých míst bude vykazovat rozdílný vzhled

přijmutím fixního nebo přinejmenším nekorelovaného pohybu světla s kamerou. Pro 2D bilboard, pokud použijeme stejnou texturu ve všech orientacích, to bude vypadat tak, že se světlo pohybuje s kamerou. Možným řešením je mít více textur pro různé pohledy nebo blanding (viz 4.)

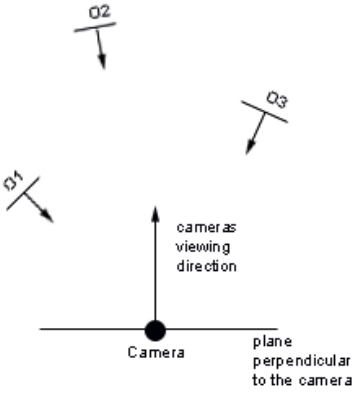
Rozlišujeme dva typy billboardingu cylindrický a sferický. Ve sférické verzi není žádný problém s orientací objektu. Zatímco v cylindrickém přístupu je rotace objektu svázána s vektorem rotace, a to obvykle v pozitivním směru osy Y.

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

Podvodné metody natočí billboard tak, že normálový vektor billboardu je shodný s invertovaným vektorem kamery viz. obrázek.



Pravé metody natáčí billboard kolmo na kameru viz. obrázek.



Implementace

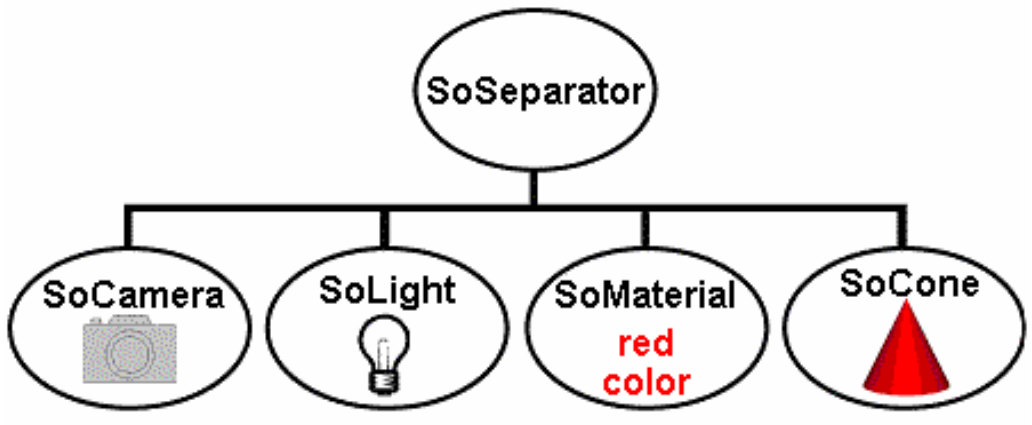
Cílem projektu je naimplementovat algoritmy pro detekci a vyhodnocování kolizí v jednoduché virtuální scéně.

Projekt je implementován v objektově orientovaném jazyce C++ ve vývojovém prostředí Visual Studion.net za použití grafické knihovny Open Inventro, použité textury a modely byly vytvořeny v modelovacím studiu 3D Studio Max. Tyto modely jsou zjednodušené verze modelů poskytnutých pro ročníkový projekt. Přiblížení programovacího jazyka C++ není nutné, stejně tak jako znalost vývojového prostředí Visual Studio.net a 3D Max Studia. Zato knihovna Open Inventor není zatím běžně používaná a rozšířená, proto ji v dokumentaci přiblížím. O knihovně Open Inventor existuje množství kvalitních článků, ze kterých jsem čerpal, a pro základní přiblížení uvádím citaci z http://www.root.cz/clanky/open-inventor/:

Open Inventor to 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.

Samotný projekt je rozdělen do 5modulů, z nichž každý má specifický účel. Stručně popíši rozdělení a účel modulů.

Main.cpp, Main.h - hlavní modul programu.

- Zajišťuje inicializaci a rozjetí rendrovací smyčky
- Zajišťuje vytvoření a inicializaci všech tříd (umístění objektů)
- Provádí obsluhu systémových událostí

SgObject.cpp, SgObject.h – modul pro objekty scény

- Obsahuje atributy potřebné pro rendrování
- Obsahuje atributy potřebné pro fyzikální a jiné výpočty
- Obsahuje metody pro nastavování a ovládání objektů

SgoManager.cpp, SgoManager.h – modul pro obsluhu a vzájemnou interakci objektů scény

- Uchovává seznam všech objektů scény
- Řeší algoritmus detekce kolizí mezi objekty
- Řeší algoritmus pro vyhodnocování kolize

SgInterface.cpp, SgInterface.h – modul pro interakci uživatele s prostředím

- Obsahuje třídu pro práci s kamerou
- Obsahuje třídu pro práci s klávesnicí
- Obsahuje třídu pro práci s myší.

Billboard.cpp, Billboard.h – modul pro techniku billboardingu

- Vytváří samotný billboard
- Obsahuje metody pro nastavení parametrů billboardu

4. Prostředí a pohyb v prostředí

4.1. Prostředí

Aby bylo možno vyhodnocovat naimplementované algoritmy, je nutné vytvořit prostředí (virtuální scénu). Jako vhodné prostředí byl vybrán vesmír z důvodu návaznosti na diplomovou práci a také s ohledem na původní záměr utvořit hru SpaceGame (3D akční hru ve vesmírném prostředí). Pro tvorbu virtuálního prostředí se dají použít různé techniky, např. tzv. skybox, což je krychle obklopující scénu pokrytá vhodnou texturou, nebo tzv. skydome, což je polokoule obklopující povrch (zem) pokrytá vhodnou texturou. Protože ve vesmírném prostředí skydome pozbývá smyslu, v projektu je použit skybox, u kterého je nutné vyřešit perfektní navazování použitých textur.

4.2. Objekty v prostředí

V projektu je implementovaná třída SgObject, která umožňuje vkládat v podstatě libovolné objekty do prostředí. Objekt je reprezentován předem vytvořeným modelem a případně jeho texturou. Konkrétně v projektu jsou použity modely planet a stihačů. Je možné použít další libovolné modely, ale jejich použití v ukázkovém příkladu není zapotřebí, navíc modely jsou pro účely detekce kolizí příliš složité a je nutná jejich úprava na jednoduší verzi, navíc úprava modelů není hlavní součástí projektu. Proto je použit jen jeden model stíhače.

Dále je implementovaná 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, proto budou detailněji popsány níže.

4.3. 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.

V projektu je použita perspektivní kamera, realizovaná v grafické knihovně OpenInventor 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 a nastavovat potřebné parametry.

Hlavicka třídy SgCamera:

```
class SgCamera{
protected:
      SoPerspectiveCamera * Camera;
      //uzel (node) Open Inventoru pro realizaci samotné kamery
      float distance;   //vzdálenost kamery od objektu
      float height;     //výška kamery od objektu
public:
```

```
        SgCamera(); //vytvoření instance SoPerpectiveCamera
        ~SgCamera();//zrušení instance SoPerpectiveCamera
        SoPerspectiveCamera * getRoot(){return Camera;};
        //vrací ukazatel na sebe
        void setRot(SbRotation rot); //nastavuje orientaci (otočení) kamery
        void setPos(SbVec3f pos);    //nastavuje pozici kamery
};
```

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

Pohyb v prostředí nemůže být náhodný, musí být ovládaný uživatelem, proto 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 projektu opět vytvořenu vlastní třídu utvořenou na míru pro účely projektu. Tyto třídy jsou implementovány v modulu SgInterface. Jsou to třída SgMouse pro práci s myší a třída SgKeyboard pro práci s klávesnicí.

Hlavička třídy SgMouse:

```
class SgMouse{
protected:
        map <t_tlacitko, SoMouseButtonEvent::Button> tlacitko;
        // pole používaných tlačítek
        map <t_tlacitko, SoMouseButtonEvent::Button>::iterator Tl;
        // iterátor do tohoto pole
        map <t_tlacitko, bool> stisknuto;
        // indikace zda je tlačítko použito
        map <t_tlacitko, bool>::iterator St;
        // iterátor do tohoto pole
public:
        SgMouse();  //konstruktor nastavuje defaultně klavesi pro ovládání
        void setButton (const SoEvent * event);//nastavuje stisknuté tlačítko
        funkce je volána při stisku tlačítka
        bool getButton (t_tlacitko tl);
        // vrací tru pokud je tlačítko stisknuto
};
```

Třída SgKeyboard má stejné funkce, které místo tlačítek myši obsluhují klávesy z klávesnice.

Pro pohyb ve scéně jsou implementovány metody pro pohyb a otáčení ve všech osách, a to jak pomocí relativních, tak absolutních hodnot. Tyto metody jsou implementovány ve třídě SgObject. Vzhledem k vesmírnému prostředí je implementován pohyb jak s ohledem na setrvačnost, tak na absolutní pohyb. Ve výsledku naimplementované metody umožňují stíhačce, případně kameře, reálný pohyb ve vesmíru se zvolením použitých motorů.

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

5. Realizace objektů a třída SgObject

Realizaci objektů a jednotlivé algoritmy implementované ve třídě SgObject sloužící pro funkčnost projektu popíši přímo u konkrétních metod nebo atributů v obsáhlé hlavičce třídy. Ve výpisu jsou uvedeny jen použité atributy a metody, samotná třída SgObject je připravována pro další použití v diplomové práci, a proto obsahuje vetší množství atributů a metod. Použití a význam některých atributů je uveden níže v popisu metod pracujících s těmito atributy.

```
class SgObject {
public:
  enum MODEL_SET { RENDER = 0x01, COLLISION = 0x02, ALL = 0x03 };
  enum TYPE_OBJECT { SUN = 0x01, PLANET = 0x02, OTHER = 0x03 };
  // Výčtové typy pro typ modelu a typ vesmírného objektu

protected:
  SgoManager * Manager;
  SoSeparator * Root;
  SoMatrixTransform * MatrixTransform;
  SoNode * Model;
  SoNode * CollisionModel;
  // instance objektů knihovny Open Inventor
```

Důležitou třídou je transformační matice SoMatrixTransform, ve které je uložena poloha (translation) a natočení (rotation) objektu, případně zvětšení (scale). Dále pak třída SoNode, která reprezentuje samotný model a kolizní model uložený ve formátu WRL. Objekt má dva modely z důvodu optimalizace. Algoritmus pro řešení kolizí pracuje s jednodušším kolizním modelem, ale vykreslován je složitější model.

```
  SbVec3f Scale;              // případné měřítko modelu
  float CollisionRadius;      // kolizní rádius pro sférický obal
  float ActionRadius;         // akční rádius pro sférický obal vyšší vrstvy
  // následuje výpis atributů sloužících nejen pro fyzikální účely
  float Mass;                 float CoefficientOfRestitution;
  SbVec3f Position;           SbVec3f OldPosition;
  SbVec3f CentrOfMass;        SbVec3f Velocity;
  SbVec3f AngularVelocity;    SbVec3f AngularMomentum;
  SbVec3f Force;              SbRotation Orientation;
  SbRotation OldOrientation;  SbRotation ScaleOrientation;
  SbMatrix InertiaTensor;
  TYPE_OBJECT Type;     // atribut pro identifikaci objektu
  int ID;               // detailní identifikace objektu
  bool Select;          // určuje zda je objekt právě vybrán
```

```
  void createHiddenScene();
  void releaseHiddenScene();
  //metody pro vytvoření a zrušení instancí OpenInventoru
  void updateMatrixTransform();//aktualizuje transformační matici

public:
  SgObject();
  SgObject(const char *model …);
  // konstruktory, inicializuji objekt na defaultní hodnoty
  virtual ~SgObject();//destruktor

  SbVec3f getDirVel() const;
  // metoda poskytující směr pohybu vpřed
  SbVec3f getDirVec() const;
  SbVec3f getUpVec() const;
  SbVec3f getLeftVec() const;
  // metody poskytující levý, horní a přední směr natočení objektu
```

Následuje několik vybraných metod nastavujících atributy, v jiných vývojových prostředích známé jako „properties“. Tyto metody jsou při objektově orientovaném programování nezbytné k zapouzdření objektu. A dále je uvedeno několik metod upravujících atributy.

```
  SgoManager * getManager();
  void setManager(SgoManager * manager);
  void setCollisionRadius(float radius);
  float getCollisionRadius();
  void setMass(float mass);
  float getMass();
  // klasické metody na nastavování atributů
  void updPos(const SbVec3f &deltaP);
  void updPos(const float x, const float y, const float z);
  void updOrientation(const SbVec3f &axis, const float radians);
  // metody na úpravu atributů
```

Otáčení v prostoru je možné řešit mimo třídu SgObejct a nastavovat orientaci přímo pomoci funkce setOrientation, nebo využít funkcí naimplementovaných, které umožňují absolutní nebo relativní rotaci ve všech osách. Pojmem absolutní rotaci označuji rotaci, při které uživatel stiskne patřičnou klávesu a objekt rotuje po dobu stisknutí klávesy konstantní rychlostí. Pojmem relativní rotaci označuji rotaci, která je založená na fyzikálním modelu. Také by se dala nazvat fyzikální rotací. Tato rotace 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 6 párů „virtuálních“ motorů, které otáčí objektem ve všech osách. V ideálním případě, pro detailní simulaci, by objekt měl mít na každém rameni osy 4 páry motorů (24 motorů). Můžeme použít libovolný počet motorů, a tím upravit manévrovatelnost daného objektu.

Obdobné řešení jako pro rotaci je pro pohyb. Opět můžeme nastavovat pohyb buď mimo třídu, nebo v třídě. Absolutní a relativní pohyb je ve stejném významu jako u absolutní a relativní rotace. Pro ideální simulaci pohybu by objekt měl mít 2 motory (vpřed,vzad) v každé ose (6 motorů) Virtuální motory které slouží pro rotaci mohou sloužit i pro pohyb a naopak. V praxi by ale realizace pohybu a

rotace byla řešena jen několika motory, a z nich by se odvozovala výsledná manévrovací schopnost objektu.

Ve výpisu třídy uvádím jen metody pro rotaci. Metody pojmenované Rot, zajišťují absolutní rotaci objektu. Metody pojmenované Torque zajišťují relativní otáčení objektu.

```
  void RotLR(const float radians)
      { this->Orientation*=SbRotation(this->getUpVec(),radians); }
  void RotUD(const float radians)
      { this->Orientation*=SbRotation(this->getLeftVec(),radians); }
  void RotAR(const float radians)
      { this->Orientation*=SbRotation(this->getDirVec(),radians); }
  void TorqueLR(const float torque)
      { this->AngularMomentum+=getLeftVec().cross(getDirVec()*torque); }
  void TorqueUD(const float torque)
      { this->AngularMomentum+=getDirVec().cross(getUpVec()*torque); }
  void TorqueAR(const float torque)
      { AngularMomentum+=>getLeftVec().cross(getUpVec()*torque); }

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

Poslední uvedenou metodou je metoda timeStep, která zajišťuje v každém snímku započítání působících sil a integraci fyzikálních veličin, jako například rychlost, poloha, matice momentu setrvačnosti atd.

6. Správa objektů a třída SgoManager

Správa objektů scény je realizována hlavní třídou SgoManager tzv. správcem objektů. Samotnou správu objektů a klíčové algoritmy popíši přímo u konkrétních metod jako u třídy SgObject. I třída SgoManager je připravována pro další použití, proto uvádím jen důležité metody a atributy.

Popis třídy SgoManager je nejrozsáhlejší, protože popisuje algoritmy pro detekci a vyhodnocování kolizí. Také se odkazuje na jiné třídy a jiné materiály uvedené výše.

```
class SgoManager {
public:
  #define G 6.67e-11f   // definice gravitační konstanty
  enum CollidingType { Null=0, Sphere, Body, Explosion };
  // výčtový typ pro druh kolize

private:
  SoSeparator * sgoRoot;
  //hlavní kořen manageru (první podkořen celé scény)
  SbList<SgObject*> sgoList;  // seznam všech objektů
  SbList<Billboard*> sgoBillboard;  // seznam billboardů, pro exploze
  SgObject * Active;    // ukazatel na aktivní objekt
  int CameraFree;       // index do pole objektu, objekt kamery
  int CameraFighter;    // index do pole objektu, objekt stihacky
```

```
  SoPerspectiveCamera * Camera; // instance třídy pro perspektivní kameru
  SoGroup * collisionRoot;      // uzel, pro skrytou scénu detekce kolizí
  float epsilon;        // určuje minimální možnou vzdálenost dvou bodů

  // struktura pro uchování kolizních objektů a údajů potřebných pro
vyřešení úspěšné vyřešení kolize
  struct{
        bool Collision;       // určuje zda došlo ke kolizi
        int Index[2];         // index do pole objektů pro kolidující tělesa
        vector<SbVec3f> Points;           // průnikové body kolize
        vector<SbVec3f> Triangles[2];
        // trojúhelníky, které se zúčastnily kolize, pro obě tělesa
        SbVec3f collidingPoint;           // kolizní bod (bod srážky)
        SbVec3f collidingNormal;          // kolizní normála
  } CollidingSgos;

public:
  SgoManager();  // konstruktor, inicializuje objekt na defaultní hodnoty
  ~SgoManager(); // destruktor, uvolní všechny objekty scény

  setFreeCamera(int i) { this->CameraFree=i; }
  // nastavuje index do pole objektů, pro objekt který reprezentuje kameru
  setFighterCamera(int i) { this->CameraFighter=i; }
  // nastavuje index do pole objektů, pro stíhačku
  setActiveFighter() { this->Active=this->sgoList[CameraFighter]; }
  // nastaví kameru na stíhačku
  setActiveFree() { this->Active=this->sgoList[CameraFree]; }
  // nastaví kameru na objekt, který reprezentuje kameru
  SoSeparator * getSgoRoot() const { return sgoRoot; }
  // vrací kořen scény
  void setCamera(SoPerspectiveCamera * camera) { this->Camera=camera; }
  // metoda pro nastaveni používané kamery, důležité pro billlboarding
  SgObject * getSgo(int index)  { return sgoList[index]; }
  // vrací ukazatel na objekt scény podle zadaného indexu
  int getSgosNum() const  { return sgoList.getLength(); }
  // vrací počet objektů ve scéně
  SgObject * getActive() { return this->Active; }
  // vrací ukazatel na aktivní objekt

  // metody pro přidávání a odebírání objektů ze scény
  void appendSgo(SgObject *sgo);    // přidá objekt do seznamu
  void removeSgo(SgObject *sgo);    // odstraní objekt ze seznamu
  void removeSgo(int index);
  void deleteSgo(int index);        // odstraní objekt ze seznamu
  void deleteSgo(SgObject *sgo);    // a zruší jeho instanci
  void deteleAllSgos();             // odstraní a zruší všechny objekty
```

Algoritmy pro detekci a vyhodnocování kolizí jsem záměrně uvedl na konci třídy, protože zaslouží podrobně popsat. Některým algoritmům dokonce je vyčleněna samostatná kapitola.

```
private:
  SgoManager::CollidingType checkCollision();
  // detekuje kolize všech objektů ve scéně
```

```
  SgoManager::CollidingType checkCollisionWith(SgObject * sgo, int index);
  // detekuje kolizi objektu s ostatními objekty
  SoIntersectionDetectionAction * ida;
  // třída knihovny Open Inventor, pro detekci kolizi
  static SoIntersectionDetectionAction::Resp intersectionCb(void *closure,
      const SoIntersectingPrimitive*p1, const SoIntersectingPrimitive*p2);
  // funkce Open Inventoru „callback“ pro detekci kolizi
  void resolveCollisionExplozion(const double time, const double dt);
  // vyhodnocuje zachycenou kolizi, explozi
  void resolveCollisionSphere();
  // vyhodnocuje zachycenou kolizi, jako srážku koulí
  void resolveCollisionBody();
  // vyhodnocuje zachycenou kolizi, pomocí fyzikálního modelu
  void workIntersectionPoints();
  // eliminuje nesprávné kolizní body
  void workIntersectionVertexs();
  // odstraní duplikátní trojúhelníky
  SbVec3f getIntersectionPoint();
  // spočítá bod kolize
  void computeNormal();
  // spočítá normálu kolize

public:
  void timeStep(const double time, const double dt);
  // obsluhuje jednotlivé snímky

};
```

Rozbor metod zahájím poslední jmenovanou funkcí timeStep, která obsluhuje všechny ostatní funkce v každém snímku. Současně prochází celý seznam objektů scény a volá jejich funkce timeStep, aby každý objekt provedl integraci v čase. Pokud se ve scéně nachází běžící animace (exploze), tak ji obslouží, tzn. zavolá funkci timeStep pro běžící animaci. Dále následuje samotná detekce kolize voláním funkce checkCollision a na základě odezvy této funkce, která vrací typ kolize, je volána vyhodnocovací funkce. Vyhodnocení kolize je možné třemi způsoby - explozí jednoho z objektů, jednoduchým odrazem na principu koulí nebo na základě složitého fyzikálního modelu. Vyhodnocovací funkce pro explozi a jednoduchý odraz je možné volat okamžitě, ale pro fyzikální odraz je nutné přepočítat určité hodnoty, proto jsou před samotnou vyhodnocovací funkcí volány ještě následující funkce workIntersectionPoints, getIntersectionPoint, workIntersectionVertexs, které též objasním níže.

6.1. checkCollision a checkCollisionWith

Funkce checkCollision prochází všechny objekty scény a postupně pro každý objekt scény volá funkci checkCollisionWith, která hledá kolizi daného objektu s každým dalším objektem scény.

Pro detekci kolize je potřeba pojít kombinace(permutace) všech objektů scény, tedy každý s každým. Algoritmus lze optimalizovat na kombinační funkci permutace bez opakování, protože nám stačí detekovat kolizi jen od jednoho tělesa. Protože funkce permutují celý seznam objektů, je ve funkci mimo jiné přidán výpočet gravitační síly, který také potřebuje permutačně projít celý seznam

objektů. Výpočet gravitační síly je proveden podle vzorce F=G*(m1*m2/R), kde G je gravitační konstanta, m1 a m2 jsou hmotnosti těles a R je vzájemná vzdálenost těles[literatura]. Působící gravitační síla je k jednomu tělesu přičtena a od druhého odečtena, protože procházíme bez opakování.

Aby nebyla detekce kolize tak náročná, je prostor rozdělen na jednotlivé sféry, v našem konkrétním případě tyto sféry tvoří sluneční soustavy. Každé slunce tvoří jednu sférickou oblast. Dále každý objekt má svůj sférický obal, aby se neustále neprováděla detekce na úrovni modelů (trojúhelníků). S těmito optimalizacemi probíhá detekce kolize jen u objektů ve stejné sluneční soustavě, nebo všech objektů mimo všechny sluneční soustavy, a to na úrovni detekce koulí. Při nalezení kolize obalů je na základě kolidujících těles rozhodnuto o druhu kolize, dále o případném pokračování detekce kolize na úrovni trojúhelníků.

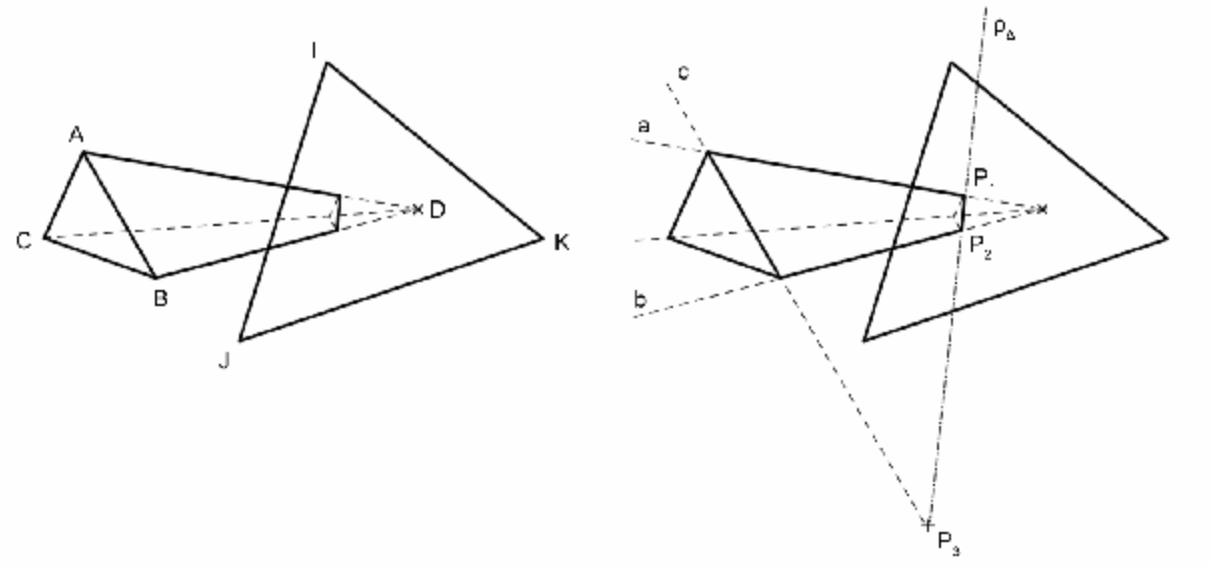
Konkrétně pro náš algoritmus:

- Testovaný objekt vstupující do funkce checkCollisionWith je testován s dalšími objekty.
- Pokud je testovaný objekt typu slunce, všem dalším objektům spadajícím do akčního rádius tohoto slunce je přidělen identifikátor této sluneční soustavy.
- Pro testované objekty libovolného typu je spočítána kolize se všemi dalšími objekty ve stejné sluneční soustavě.
- Při nalezení kolize jsou uloženy indexy kolidujících těles a je rozhodnuto, na základě typu objektů, o druhu kolize: jeden z objektů je slunce = exploze, jeden z objektů je planeta = jednoduchý odraz, vše ostatní mimo kamery = fyzikální odraz.

Podrobnější vyhodnocení kolize na úrovni trojúhelníků nastává jen v případě řešení kolize na základě fyzikálního modelu, protože u exploze stačí detekce na úrovni koule. Navíc exploze může nastat v našem případě jen při srážce stíhačky se sluncem, kde je detekce na úrovni koule dostačující, stejně jako u jednoduchého odrazu. V případě podrobnější detekce se vytvoří skrytá scéna, do které se umístí kolidující modely. Na tuto skrytou scénu je aplikována funkce Open Inventoru SoIntersectionDetectionAction, která při kolizi na úrovni trojúhelníků modelu vyvolá „callback“ funkci intersectionCb, která kolizi obslouží potřebným způsobem. Po skončení detekce je skrytá scéna vyprázdněna a schována pro další použití.

6.2. intersectionCB

Tato funkce je vyvolána při zjištěné kolizi a jejími parametry jsou dvě tělesa kolizních objektů a seznam dvojic jejich trojúhelníků u nichž doškol ke kolizi. Pro tyto dvojice trojúhelníků hledám jejich vzájemné průsečíky, které uložím do seznamu kolizních bodů. Dále si uložím seznam všech trojúhelníků k pozdějšímu zpracování.



Na obrázku je zachycen nejjednodušší případ, kdy několik trojúhelníků prvního tělesa pronikne do jednoho trojúhelníku druhého tělesa. Funkce intersectionCB vyhodnotí jako kolizní dvojice tyto troj. (ABD,IJK), (ACD,IJK), (CBD,IJK). Z těchto dvojic vypočítám průsečíky např. pro dvojici (ABD,IJK). Zvolím jako rovinu troj. IJK, a vrcholy tvořící troj. ABD vytvoří tři přímky protínající rovinu, pro tyto tři přímky poté vypočítám průsečíky s rovinou. Pak u zvolené dvojice „prohodím role“ a rovinu bude tvořit troj. ABD a vrcholy troj. IJK budou tvořit přímky. Takto získám celkem 6 průsečíků pro každou dvojici trojúhelníků. Nás ovšem zajímají jen průsečíky našich kolidujících těles, konkrétně body P1 a P2. Proto se ostatní průsečíky eliminují, více viz. funkce workIntersectionPoints.

Složitější případy, kdy pronikne více troj. jednoho tělesa do více troj. druhého tělesa se řeší stejným algoritmem.

6.3. Jednoduchý odraz - resolveCollisionSphere

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

7. Exploze a třída Billboard

7.1. Exploze - resolveCollisionExplozion

Kolize, která končí explozí je vyhodnocována touto funkcí, která menší z kolizních těles nechá explodovat. První věc ošetřená před samotnou explozí je odebrání explodujícího tělesa ze seznamu objektů scény, případně přesměrování kamery na jiný objekt, pokud byla kamera upnuta na explodovaný objekt. Následně se do scény umístí animovaný billboard s explozí na původní místo objektu. Správce scény (sgoManager) má svůj seznam již existujících billboardů (existuje instance třídy billboard), které mohou být právě používány, 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 k dispozici, vytvoří správce novou instanci, kterou inicializuje a umístí na konec seznamu billboardů. Na tomto billboardu opět spustí animaci.

7.2. Třída Billboard

Tato třída je věnována metodě billboarding popsané v teoretické části. Ve výpisu hlavičky třídy uvádím jen důležité metody a nastavovací funkce (properties) již neuvádím.

```
class Billboard{
private:
      SoSeparator * Manager;
      SoSeparator * Root;
      SoRotation  * Rotation;
      SoTranslation * Translation;
      SoCoordinate3 * Coordinate;
      SoIndexedTriangleStripSet * TriangleStripSet;
      SoTexture2 * Texture;
      SoTextureCoordinate2 * TextureCoordinate;
      // instance objektů knihovny Open Inventor
      SoPerspectiveCamera * Camera;
      // ukazatel na používanou kameru
      float Dimension;  // rozměr billboardu
      float FPS;        // rychlost animace
      double lastTime;  // poslední čas výměny snímku animace
      int TextureDimension;   //rozměr textury
      int Frame;        // snímek animace
      bool Used;  // určuje zda je billboard použitý nebo volný
      void updateDimension(float dimension);
      // upravuje rozměry billboardu
      void updateTextureCoordinate(int frame);
      // upravuje texturovací souřadnice
public:
      Billboard();
      Billboard(float dimension, char * file);
      // konstruktory, inicializují defaultní data
      bool isFree() { return !this->Used; }
      bool isUsed() { return this->Used; }
```

```
        // metody poskytují informace zda je objekt používán nebo volný
        void useBillboard();
        void useBillboard(SbVec3f pos);
        // metody znovu spustí animaci na billboardu
        void timeStep(double time, double dt);
        // metoda zajišťuje animaci v každém snímku
};
```

Podrobně vysvětlit si zaslouží metoda useBillboard a timeStep. Začneme jednoduší metodou a to useBillboard, která nastaví billboard jako použitý a nastaví první zobrazovaný frame animace. Případně metoda s parametrem nastavuje rovnou novou pozici billboardu a nemusí se již nastavovat voláním příslušné funkce.

Hlavní metodou, ve které se provádí samotná animace, je metoda timeStep, která je vyvolána v každém snímku. Metoda v první řadě upravuje natočení billboardu ke kameře, a to právě pomocí ukazatele na použitou kameru, pomocí které nastaví orientaci billboardu. Orientace billboardu je provedena tak, že orientace z transformační matice kamery je přiřazena do transformační matice billboardu s následným otočením o 180°, protože chceme, aby billboard byl otočený ke kameře a ne od kamery.

Animace je optimalizována, a proto není načítán každý snímek animace zvlášť, ale všechny snímky animace jsou umístěny v jedné velké textuře, u které se mění jen texturovací souřadnice. Animace je prováděna s určitou rychlostí, kterou určuje atribut FPS. Když nastane čas k výměně snímku animace, dojde k přemapování texturovacích souřadnic pomocí funkce updateTextureCoordinate. Po provedení animace (zobrazení všech snímků) se billboard sám uvolní. A také se sám uvolní ze scény, tzn. již se dále nezobrazuje.

8. Fyzikální odraz

Fyzikální obraz byl nejsložitější částí projektu. Samotná teorie fyzikálního odrazu je rozebrána v teoretické části. Shromáždění a pochopení této teorie zabralo největší podíl času věnovaného teoretické přípravě. Fyzikální odraz je realizován ve funkci resolveCollisionBody, která je implementovaná podle uvedených vzorců.

Samotná implementace fyzikálního odrazu není složitá, problém ovšem nastal při zajišťování potřebných fyzikálních veličin. U veličin jako je rychlost pozice nebo samotná kolidující tělesa, nebyl problém, protože tyto funkce jsou obsaženy buď v příslušných strukturách, nebo v objektu samotném. Problémové veličiny proberu a naznačím možná řešení postupně.

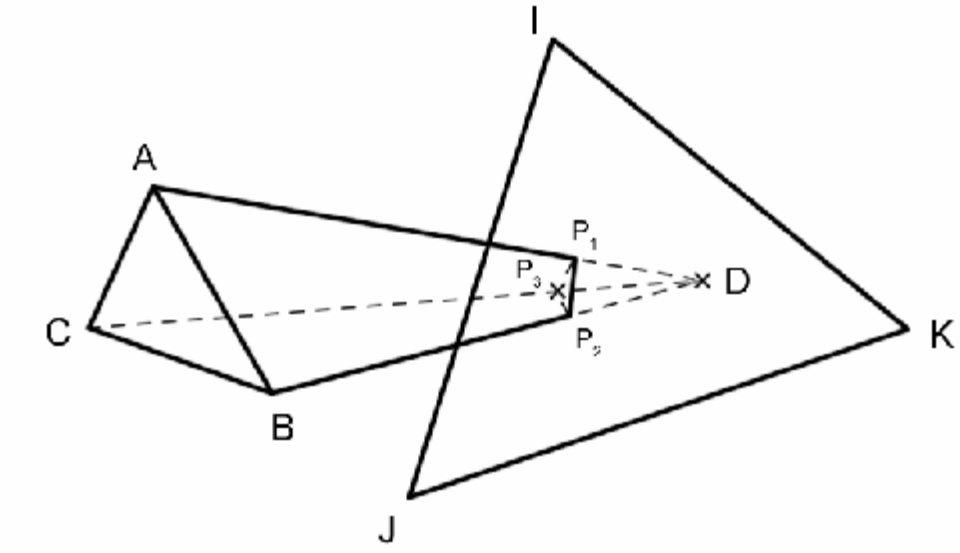
8.1. Výpočet bodu kolize

Na první pohled se může zdá otázka výpočtu bodu kolize jako jednoduchá, pravdou je, že patří k těm složitějším výpočtům v projektu. Vzhledem k tomu, že se pohybujeme na rovině diskrétních funkcí a ne spojitých, jak by bylo pro fyzikální model potřeba, vzniká mnoho problémů. První z nich

je samotná detekce kolize, při které dochází ne ke kolizi, ale k průniku těles, právě v důsledku diskrétního posunu těchto těles. Při průniku těles nám právě zanikne kolizní bod a normála, které se následně musejí vypočítat z objemu, který vznikl průnikem.

Začneme u funkce Open Inventoru SoIntersectionDetectionAction, resp. intersectionCb, která detekuje kolizi dvou těles a vrací všechny kolidující trojúhelníky. Zároveň vypočítá všechny průsečíky těchto trojúhelníků. Tyto průsečíky musíme eliminovat jen na průsečíky tvořící přesně kolizní objem, resp. kolizní těleso. K tomu slouží funkce workIntersectonPoints.

workIntersectonPoints eliminuje body na základě předpokladu, že bod, který je ve výčtu uveden vícekrát, je právě hledaný kolizní bod. Například průsečík P1 nalezneme jak pro dvojici troj. (ABD,IJK), tak pro dvojici troj. (ACD,IJK), obdobně pro bod P2 dvojice troj. (ABD,IJK) a dvojice troj. (CBD,IJK) viz. obrázek.



Z takto vypočítaného kolizního tělesa pomocí funkce **getIntersectionPoint** vypočítáme kolizní bod. Funkce vypočítá kolizní bod tak, že najde těžiště kolizního tělesa. Algoritmus pro výpočet těžiště pracuje následujícím způsobem:

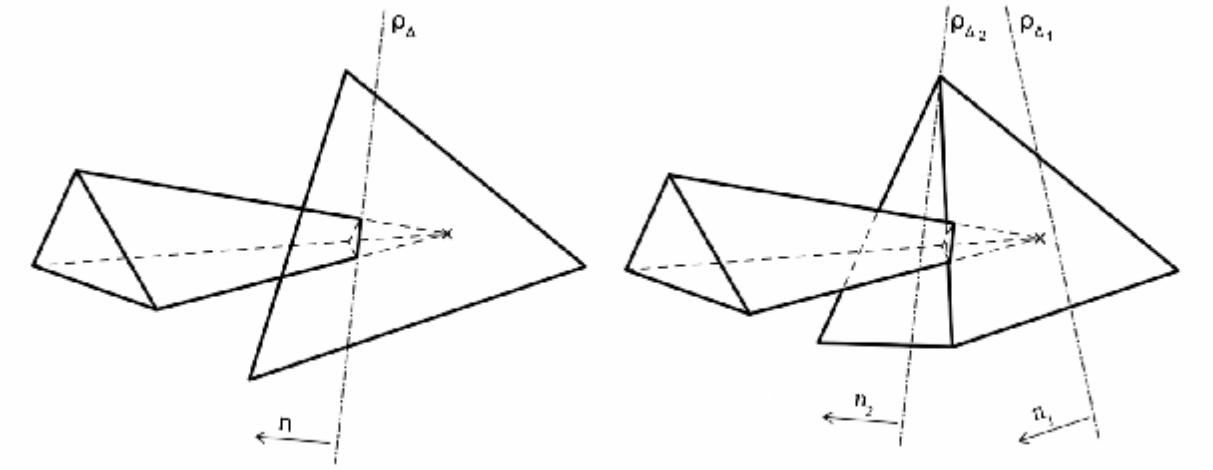
- Prochází se všechny vrcholy kolizního tělesa
- Sčítáme jejich polohové vektory
- Výsledný vektor (sumu poloh) vydělíme počtem vrcholů, čímž získáme pozici těžiště

8.2. Výpočet normály kolize

Výpočet normály kolize je zatížen podobnými problémy jako výpočet bodu kolize. Před samotným výpočtem normály, která se počítá na základě kolidujících trojúhelníků, je zapotřebí trojúhelníky uložené funkcí intersectionCb upravit, a to pomocí funkce workIntersectionVertexs.

Protože funkce intersectionCb hledá dvojice kolidujících trojúhelníků, mohou se v seznamu trojúhelníků každého objektu objevit trojúhelníky stejné, které jsou eliminovány právě funkcí **workIntersectionVertexs.** Po eliminaci stejných trojúhelníků můžeme přistoupit k výpočtu samotné normály.

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 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.



Při jednoduché srážce proniká několik trojúhelníků jednoho tělesa do jednoho trojúhelníku druhého. Tuto srážku lze vyhodnotit tak, že normálu vypočteme z osamoceného trojúhelníku. Při složité srážce proniká několik trojúhelníků jednoho tělesa do několika trojúhelníků druhého tělesa. Zde není jednoznačné, které těleso do kterého narazilo, proto se pro výpočet normály zvolí těleso s menším počtem trojúhelníků a vypočítají se normály pro všechny tyto trojúhelníky, které se ve výsledku zprůměrňují. Normály zprůměrňujeme tak, že vektory sečteme a výsledný vektor normalizujeme.

Závěr

K zadanému tématu knihovna pro detekci kolizí mezi objekty scény 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 ročníkovém projektu byl implementován algoritmus na detekci kolizí, který v rámci použitých prostředků poskytuje pro jednoduché vyhodnocení kolize dostačující data. Dále byly implementovány tři druhy algoritmů pro vyhodnocování kolizí, vyhodnocení kolize explozí, jednoduchým odrazem a vyhodnocení podle fyzikálních zákonů. První dva algoritmy poskytují uspokojivé výsledky pro libovolná tělesa scény. Nejsložitější algoritmus pro vyhodnocování kolizí na základě fyzikálních zákonů je naimplementován ve funkční podobě, ale vzhledem k možnostem detekce kolizí nemusí fungovat pro veškeré objekty korektně. Kvůli provádění složitých matematických operací dochází při výpočtu k nepřesnostem, které mohou ovlivnit funkčnost.

V ročníkovém projektu nebyl implementován algoritmus na výpočet těžiště a algoritmus na výpočet matice momentu setrvačnosti. Algoritmy pro výpočet těchto dvou veličin jsou komplikované a nebyly součástí zadání. V projektu je těžiště bráno jako střed použitého modelu, což je ve většině případů dostačující. Matice momentu setrvačnosti byla nahrazena matici pro jednoduchý kvádr, což je pro většinu těles nedostačující, a z tohoto důvodu může při vyhodnocování odrazu docházet k nepřesnostem a anomáliím.

Na tento ročníkový projekt bude navazovat diplomová práce, ve které budou pravděpodobně doimplementovány chybějící algoritmy a implementovány další algoritmy dle konkrétního zadání a směru diplomové práce.

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

Literatura

Seriál Grafická knihovna Open Inventro

http://www.root.cz/serialy/open-inventor/

Vše o programování 3D grafiky pod knihovnou OpenGL

http://nehe.ceskehry.cz/

http://nehe.gamedev.net/

Informační server o programování

http://www.builder.cz/

Moment setrvačnosti a systémy částic

http://kwon3d.com/theory/moi/moi.html

Dynamika pevných objektů

http://www.d6.com/users/checker/dynamics.htm

D.Halliday, R.Resnick, J.Walker: Fyzika. Vysoké učení technické v Brně, VUTIUM