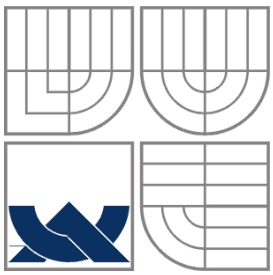
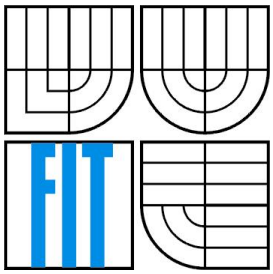


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

KNIHOVNA PRO REAL-TIME SIMULACI 3D PROSTORU

REAL-TIME SIMULATION LIBRARY IN 3D SPACE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — TOMÁŠ BENNA
AUTHOR

VEDOUCÍ PRÁCE — ING. MARTIN HRUBÝ, Ph.D.
SUPERVISOR

BRNO 2007

Zadání diplomové práce/5798/2006/xbenna00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2006/2007

Zadání diplomové práce

Řešitel: **Benna Tomáš**
Obor: Výpočetní technika a informatika
Téma: **Knihovna pro real-time simulaci 3D prostoru**
Kategorie: Modelování a simulace

Pokyny:

1. Navažte na výsledky svého semestrálního projektu, kde byly navrženy algoritmy simulace 3D prostoru se zadanými fyzikálními parametry a obsaženými 3D hmotnými tělesy.
2. Navrhněte efektivní knihovnu simulačních funkcí pro realistickou simulaci kolizí tuhých těles v 3D prostorových scénách. Sadu funkcí navrhněte jako knihovnu snadno použitelnou v simulačních aplikacích. Zaměřte se na úroveň abstrakce knihovny.
3. Navrženou knihovnu implementujte s ohledem na přenositelnost.
4. Implementujte sadu demonstračních příkladů pro komplexní ukázku schopností vaší knihovny.
5. Implementujte jeden netriviální příklad prostorového modelu.

Literatura:

- dokumentace OpenGL
- dle pokynů vedoucího

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: **Hrubý Martin, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 22. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. **Pan**

Jméno a příjmení: **Tomáš Benna**
Id studenta: 21254
Bytem: Bezručova 86, 747 21 Kravaře
Narozen: 02. 12. 1981, Opava
(dále jen "autor")

a

2. **Vysoké učení technické v Brně**

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
diplomová práce

Název VŠKP: Knihovna pro real-time simulaci 3D prostoru
Vedoucí/školitel VŠKP: Hrubý Martin, Ing., Ph.D.
Ústav: Ústav inteligentních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnožením.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☒ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

.....	.....
Nabyvatel	Autor

Abstrakt

Práce popisuje návrh a implementaci systému pro real-time simulaci fyzikálního 3D prostoru, fungující jako nadstavbu ke grafickému enginu. Tento systém obsluhuje zpracování simulace rigidních těles chovajících se dle fyzikálních zákonů včetně detekce a řešení kolize. V textu je obsažen teoretický úvod do této problematiky, návrh a implementace systému jako multiplatformní knihovny a popis jejího uživatelského rozhraní. Mimo tuto knihovnu jsou výsledkem této práce i ukázkové aplikace, které nám znázorňují funkcionalitu navrhnutého simulačního modelu.

Klíčová slova

Fyzikální simulace, real-time simulace, detekce kolizí, řešení kolizí, multikolize, řešení multikolize, polohový stav tělesa, fyzikální solver, rigidní těleso, lokální kolize, bod kolize, backtransforming.

Abstract

This thesis describes design and implementation of 3D space physical real-time simulation system as add-on to graphical engine. System contains processing of rigid solid simulation, collision detection and response, which are provided by physical laws. Document contains theoretical introduction of this course of study, design and implementation as multiplatform library and description of user interfaces. Output of this thesis, except library, is also example applications, which demonstrate functionality of designed simulation model.

Keywords

Physical simulation, real-time simulation, collision detection and response, multicollision, multicollision resolving, solid location state, physical solver, rigid solid, local collision, collision point, backtransforming.

Citace

Tomáš Benna: *Knihovna pro real-time simulaci 3D prostoru*, diplomová práce, Brno, FIT VUT v Brně, 2007

Knihovna pro real-time simulaci 3D prostoru

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením ing. Martina Hrubého Ph.D. a že jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Tomáš Benna
17. května 2007

Poděkování

Rád bych zde poděkoval zejména svým rodičům, že mi umožnili studovat na této škole a svému vedoucímu za jeho odborné znalosti, poskytnutou důvěru a čas při řešení této práce.

Obsah

1 Úvod

Mým úkolem je navrhnout a implementovat knihovnu, která dokáže rychle a relativně přesně simulovat 3D prostor v reálném čase za pomoci základních fyzikálních zákonů a znalosti tuhých těles.

Motivací pro vývoj této diplomové práce byla částečně zajímavost problematiky, která je v aktuální době často diskutovaným předmětem. Zejména v herním průmyslu, kde fyzikální simulace hraje podstatnou roli v konkurenčním boji. Dalším motivačním faktorem vývoje je imponující grafické znázornění výsledků modelovaného světa.

Jelikož je fyzikální simulace populární v herních aplikacích, jak již bylo řečeno, časem se cíl použití tohoto projekt zaměřil zejména na hry, podle čehož jsou také definována jeho omezení, o kterých si povíme v následujících kapitolách. Tato práce je implementována ve formě knihovny pro vývojáře, která řídí pohyb grafických těles za pomoci takzvaných fyzikálních obálek a určuje tak data pro grafickou reprezentaci scény. Obsahuje jistou znalost prostoru a jeho fyzikální simulace, funguje tedy jako nadstavba nad grafickou knihovnu.

Fyzikální simulace jako celek je velmi rozsáhlý obor obsahující mnoho problematických součástí. V této diplomové práci si nejprve popíšeme základní pojmy, rozebereme si teoretické části problematiky a ukážeme si implementované řešení, které je našim cílem.

1.1 Konvence použité v dokumentu

Nejprve si popišme jisté konvence použité v této diplomové práci. Ty se zejména týkají označování a formátování terminologických výrazů a metodiky popisu algoritmů pomocí matematické formy. Pro popis bibliografické citace je použita norma ČSN ISO 690. Veškeré ostatní zde nezmíněné konvence jsou přebírány dle standardních.

1.1.1 Terminologie

Slovo nebo skupina slov vyjadřující termín začínají velkým písmenem a jsou formátována kurzívou (např. termín *Rigidní tělesa*). Stejným formátováním můžou být upravena i slova, které mají v textu vyjadřovat určitý význam plynoucí z kontextu. Terminologická slova jsou ihned po jejich definici formátována jako normální text.

1.1.2 Matematika a algoritmy

Z důvodů přehlednosti a srozumitelnosti pro čtenáře nemluvící českým jazykem jsem se rozhodl v této práci znázornit algoritmy s využitím základní matematiky. To zejména pomocí množinových operací, pro které upřesním níže definované výjimky.

Klíčová slova vyjadřující chování algoritmu formátujme tučným písmem. Mezi takovéto klíčová slova patří podmínky či cykly definované standardním způsobem (např. podmínka $\boldsymbol{if}(B)\boldsymbol{then}\ E\ \boldsymbol{else}\ E$). Písmeno B vyjadřuje booleovský výraz, E expression, V proměnnou, N číslo a M množinu. V algoritmech lze definovat strukturované bloky, neboli množinu příkazů (oddělených čárkou či odřádkováním) za pomoci složených závorek (např. $\boldsymbol{if}(B)\boldsymbol{then}\{E,E\}\boldsymbol{else}\ \{E\}$). Nyní si ukažme seznam použitých klíčových slov v algoritmech, jejichž význam je dán podobně jako v programovacím jazyce C či Pascal.

$\boldsymbol{if}(B)\boldsymbol{then}\ E\ \boldsymbol{else}\ E$

$\boldsymbol{repeat}\ E\ \boldsymbol{until}(B)$

$\boldsymbol{for}\ V = N\ \boldsymbol{to}\ N\ \boldsymbol{do}\ E$

$\boldsymbol{while}(B)E$

$\boldsymbol{foreach}(V\ \boldsymbol{from}\ M)E$

Dále si definujme operátor pro zvětšení množiny X o množinu Y

$$X \cup= Y \overset{def}{\iff} X \coloneqq X \cup Y,$$

kde operátor $\coloneqq$ vyjadřuje přiřazení. Podobně si definujme operátor $\cap=$.

1.2 Návaznost na Ročníkový a Semestrální projekt

Tato diplomová práce navazuje na semestrální projekt v kombinaci se zkušenostmi z ročníkového projektu.

Obsahem ročníkového projektu bylo seznámit se s problematikou fyzikální simulace a navrhnout pokusnou aplikaci, která mi vcelku ukázala a naučila, jak řešit problémy související s takovouto simulací. Byl zde definován základní model fyzikální simulace těles, včetně navrhnutí metod funkčnosti a optimalizace. V důsledku některých drobných chyb a nevyhovujícímu návrhu implementace ročníkového projektu, jsem se rozhodl navrhnout a implementovat cílovou simulační knihovnu znovu. Zvýšíme tak škálu možností a ukážeme si, jak lze knihovnu jednoduše aplikovat do praxe a jaké možnosti nám nový návrh poskytuje. Toto bylo cílem následného semestrálního projektu. Zde byla částečně navrhnuta výsledná knihovna dle zkušeností z ročníkového projektu. Byl vytvořen souborový formát definice fyzikální scény spolu s aplikací editoru a aplikací simulace navrhnutého souboru.

2 Zaměření této práce

V dnešní době panuje na trhu s herním průmyslem velká konkurence. Vývojáři her tedy musejí své produkty obohatit o větší hratelnost. Tu mohou v našem případě získat za pomoci dodání "dalšího rozměru", fyzikální simulace. To byl také jeden z hlavních důvodu zaměření na aplikaci do her. S tím také souvisí kritéria a důrazy kladené na cílový produkt, o těch si povíme v následném textu. Zaměříme se tedy na historický význam této problematiky a následně kriteria z toho vyplívající.

2.1 Historie fyzikální simulace v hrách

Co se historie týče, dá se říci, že doba nedávná byla milníkem čtvrté generaci her. Tou první byly textové hry, dále hry s 2D grafikou, třetí generace byla nejvýznamnější, 3D grafika. Aktuální čtvrtá je tedy 3D fyzikální simulace. Jako první revoluční hrou v tomto ohledu můžeme označit Half-Life 2, která využívala, dá se říci i proslavila, komerční knihovnu *Havok*. Přímým konkurentem *Havoku* je *AGEIA PhysX*, známá spíše pro svou první hardwarově akcelerovanou fyzikální kartu. V neposlední řadě se tomuto žánru také začínají věnovat grafické giganty jako ATI a nVidia, které se také snaží prosadit své řešení na akcelerovaných grafických kartách.

2.2 Kritéria návrhu a implementace

S růstem výpočetní rychlosti osobních počítačů vznikla možnost aplikace fyzikální simulace. V dnešní době je to velmi moderní metoda, jak zvýšit hratelnost hry a přiblížit tak hráči realitu. Proto jsem se rozhodl navrhnout tento projekt spíše jako fyzikální *Engine* (jež lze definovat jako komponenta pro řešení určitého problému) do her. Díky tomu si také můžeme vytyčit kritéria a cíle implementace. Máme zde tedy jistá primární a vedlejší kritéria neboli vlastnosti, které musí tento projekt splňovat.

2.2.1 Primární kritéria

Primární vlastností implementovaných algoritmů musí být jejich rychlost, tj. nízká výpočetní složitost, to i na úkor přesnosti simulace. Nutnost optimalizace už od počátku projektu na globální úrovni. Rozvrhnout si všechny možnosti budoucího návrhu a počítat s nimi už na počátku implementace. Na rozdíl od komerčních produktů jako je *Havok* nebo *AGEIA PhysX*, které spíše využívají hromadného zpracování dat za pomoci externích karet (grafická nebo fyzikální), systém této knihovny je založen na komplikovanějším a inteligentnějším modelu. Zmíněné komerční produkty

mají k dispozici širší škálu hardwarových zdrojů (samostatný procesor či samostatnou hardwarovou jednotku), kdežto tato knihovna výpočetní jednotku sdílí. Jedním z mých cílů je tedy dokázat, že real-time simulace fyziky nemusí být závislá na určitém hardwaru či výpočetní výkonnosti.

2.2.2 Vedlejší kritéria

Další vlastností projektu musí být jeho univerzálnost, dynamičnost a jednoduchá aplikace v kombinaci s grafickým enginem. *Univerzálnost* je pojem, který si lze představit jako možnost snadno vylepšovat a upravovat funkčnost a rozsah cílové fyzikální simulace. Přidávat do systému například vlastní tělesa se speciálním chováním. *Dynamičnost* v tomto kontextu znamená možnost konfigurace parametrů systému během fyzikální simulace. Toto si můžeme představit jako změnu parametru hustoty prostředí, či přidání jiných fyzikálních sil. Konkrétní příklady v textu této dokumentace budou jasněji znázorňovat myšlenku dynamičnosti fyzikální simulace. *Jednoduchost aplikace* vyjadřuje nízkou náročnost použití cílové knihovny. Mít jasně definované komunikační kanály bez složitých konfigurací. Knihovna by tedy měla být snadno použitelná i pro uživatele nepříliš rozumějící této problematice.

3 Základní pojmy použité v této práci

Nadefinujme si některé pojmy související s touto problematikou. Některé části této kapitoly budou popsány velmi stručně, protože nejsou přímou součástí této diplomové práce. Nejprve si popišme, co je to *Rigidní těleso*, co je to *Simulace* včetně návaznosti s touto diplomovou prací a v neposlední řadě si vysvětlíme pojem *Fyzikální solver* (zdroj viz (1), (2)).

3.1 Rigidní těleso

Mějme hmotné těleso s konstantním hmotným středem, s konstantním rozložením hustoty, s konstantní hmotností a s konstantním povrchem bez deformací. Takovéto těleso nazýváme *Rigidní* (zdroj viz (1)). Fyzikální simulace je aplikována zrovna na tato tělesa, protože představují ideál a také protože jejich simulace je nejjednodušší. Rigidní tělesa jsou také rozdělena na idealizované podtypy, jako např. koule, boxy, kvádry, válce, plochy atd. Kombinačním spojením těchto primitiv dokážeme vytvořit složitější tělesa. Metodiku a postupy jak tyto tělesa spojovat se budeme ještě zabývat.

Existuje ještě obecnější popis povrchu tělesa, za pomoci kterého bychom všechny stávající mohly nahradit. Jedná se o univerzální popis za pomoci trojúhelníkových polygonů. Objekt tvořený pouze z takovýchto polygonů můžeme nazvat jako *Polygonální objekt*. Každý trojúhelníkový polygon má své relativní souřadnice tří bodů, pomocí těch se dá vypočítat hmotný střed této plochy. Kombinací všech hmotných středů trojúhelníků polygonálního tělesa lze následně vypočítat hmotný střed celého tělesa. Jelikož je těleso rigidní, tedy má konstantní rozložení hmotnosti, výpočet takového hmotného středu není příliš složité. Jedinou nevýhodou polygonálního objektu je jeho výpočetní náročnost. Ani ne tak náročnost výpočtu pohybu a fyzikální simulace, ale detekce kolize, jelikož se kolize musí testovat na každém samostatném trojúhelníkovém polygonu. Proto zde máme právě ty obecnější tělesa jako koule a kvádry, u kterých není výpočet kolize tak časově náročný.

3.2 Simulace

Cílem *Simulace* (zdroj viz (2)) je analýza chování systému za pomoci navrženého modelu. Tohoto lze dosáhnout stručně řečeno pomocí určení počátečního stavu systému a následně cyklickému řešení modelu.

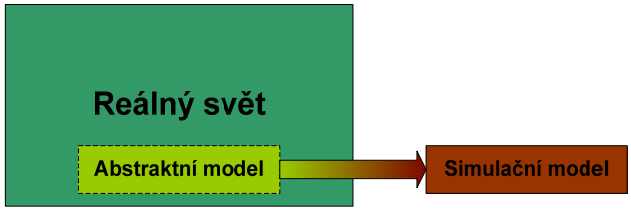
V našem případě počátečními stavy simulace jsou geografické a fyzikální vlastnosti rigidních těles v simulační scéně. Hlavním vstupním parametrem běhu (cyklickém řešení) simulace je relativní

časový krok Δt. Výstupními daty jsou pro nás pouze ty, které jsou potřebné ke grafickému znázornění, tedy pouze geografické stavy všech objektů v simulační scéně.

3.2.1 Model

Model lze definovat jako jakési zjednodušení reálného a příliš komplikovaného systému (například světa), přesněji vyseparováním jen pro nás důležitých parametrů jako ohraničenou část. Takto slovně definovaný model můžeme pojmenovat jako *Abstraktní*. Dalším typem modelu je model *Simulační*, který je definován jako abstraktní model zapsaný formou programu. Graficky si lze abstraktní a simulační model představit podobně jako na obr. 3.1.

obr. 3.1, vztah abstraktního a simulačního modelu



3.2.2 Typy simulačního modelu

Rozlišujeme tři základní typy simulačního modelu - diskrétní, spojitý a kombinovaný. Stručně řečeno *Diskrétní simulace* se týká činnosti řízené událostmi, kdežto spojitá simulace se týká zejména kontinuálních akcí a činností po sobě navazujících. *Spojitá simulace* je mnohdy řešením pro komplikovanější matematicky založené simulace, diskrétní se používá spíše pro popis nepříliš matematicky komplikovaných jevů. Posledním typem je *Kombinovaná simulace*, která jak název napovídá, obsahuje kombinaci diskrétní a spojité simulace. Tato diplomová práce je založena právě na kombinované simulaci. Pro upřesnění je kupříkladu spojitá část simulace pohyb těles v prostoru, tzv. simulační krok, a diskrétní část simulace změna stavu (např. kolize s jiným tělesem).

3.2.3 Validace modelu

Validace modelu je proces, ve kterém se snažíme dokázat, že opravdu pracujeme s modelem zdrojového systému, tedy fyziky reálného světa. Takovéto ověřování platnosti či přesnosti lze v našem případě dokázat velmi těžko, protože modelujeme komplexnější systém. Validaci lze částečně dokázat pouze na jednoduchých experimentech simulace. Například fyzikální vrh hmotného objektu je velmi jednoduché vypočítat a dokázat tak validaci. Ovšem při komplexnějších experimentech, kde mnoho hmotných objektů mezi sebou koliduje, je takřka nemožné platnost simulovaného modelu určit. V takových případech lze s omezenou přesností validaci experimentu ověřit pomocí oka pozorovatele, jak se mu simulovaný svět jeví ve srovnání se světem reálným. Důležitou vlastností této diplomové práce je dojem uživatele ze simulované scény, tedy reálně vypadající chování těles je důležitější, nežli přesný matematický výpočet simulace.

3.2.4 Real-time simulace

Real-time je speciálním typem simulace u které není tak důležitá přesnost, ale výkon a nízká výpočetní náročnost. Jak již sám název napovídá, simulace musí probíhat v reálném čase a také musí simulovat reálný čas. Tedy to co simuluje, musí časově odpovídat realitě. V následujících kapitolách si povíme tedy jak dosáhnout tohoto typu simulace.

3.3 Fyzikální solver

Fyzikální solver (zdroj viz (1)) se dá popsat jako systém, který zpracovává fyzikální simulaci a řeší případné interakce či kolize. Obsahuje všechny důležité vlastnosti, které jsou nutné pro správnou simulaci reálného prostředí. Záleží také na aplikaci solveru. Můžeme mít rychlou, ale bohužel nepřesnou simulaci např. v hrách, nebo přesnou, ale pomalou, takzvanou *Inženýrskou simulaci*. Proto je vhodné dopředu si určit primární cíle, a podle toho některé nedostatky přehlížet.

Každý fyzikální solveru by měl obsahovat několik základních vlastností, které si nyní stručně popíšeme.

3.3.1 Parametrizování prostoru

Parametrem ve fyzikálním solveru máme na mysli soubor nekonstantních vlastností, které přímo ovlivňují průběh fyzikální simulace.

Fyzikální prostor bychom si měli dostatečně nadefinovat podle vlastních parametrů. Bylo by celkem nepraktické, kdyby fyzikální solver uměl simulovat pouze jedno prostředí. Kouzlo simulace spočívá v definování různých vstupních parametrů (např. simulace ve vesmíru, ve vzduchu, pod vodou nebo s různými fyzikálními silami). Proto by tento systém měl být dostatečně univerzální a dynamický. Musíme mít ale také na paměti, že větší škála parametrizování prostoru je přímo úměrná se složitostí simulace. Proto je nutné si předem určit a omezit dynamičnost celého systému.

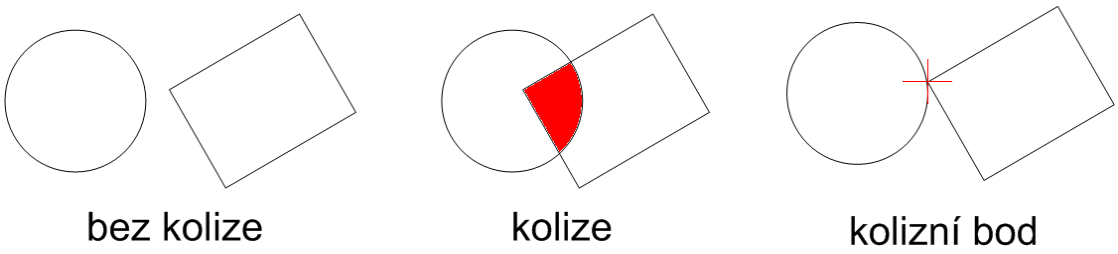
3.3.2 Pohybové rovnice

K výpočtu pohybu těles prostorem nám bohatě postačuje středoškolská fyzika, a to zejména dynamika a kinematika hmotného bodu a tuhého tělesa. Sem patří Newtonovy zákony, zákony zachování energie, hybnosti a další. Diferenciální rovnice pro určení chování tělesa jsou sice přesnější, ale výpočetně náročnější.

3.3.3 Detekce kolize

Kolizi bychom mohli definovat jako stav dvou hmotných těles, které mají společný průnik jejich hmoty. *Čas kolize* je okamžik, kdy tento průnik obsahuje pouze jeden bod na povrchu obou těles. Pokud by se kolize nedetekovaly, porušovaly by se fyzikální zákony hmotných těles.

obr. 3.2, znázornění pojmu Kolize



Detekce kolizí je přímou součástí fyzikální simulace a nezbytnou pro změnu stavu objektů. Samotná detekce kolizí je kapitola sama o sobě. Její matematické odvození je velmi složité a mnohdy i výpočetně náročnější nežli samotná simulace, proto je vhodné se velmi soustředit na optimalizace detekce kolizí. Zde také existuje problém řešení *Multikolizí*, tj. pokud spolu koliduje více objektů najednou v jednom časovém kroku, což také není triviální záležitostí. Problémem detekce a řešení kolize a řešení multikolize se budeme věnovat později v kapitole 7 Detekce kolizí.

3.3.4 Reakce na kolize

Pokud detekujeme kolizi, je nutné ji nasimulovat, respektive vypočítat odraz či přenos energie. S tímto také souvisí tak zvané *Handlování událostí*, jakoby vnější reakce na kolizi (např. při nárazu přehrání zvuku atd.). Blíže si toto téma popíšeme v konkrétní implementaci.

4 Analýza fyzikální simulace

V této kapitole se budeme zabývat analýzou a řešením některých základních problémů fyzikální simulace. Popíšeme si fyzikální a matematické vztahy, ze kterých tato diplomová práce vychází a které jsou předpokladem ke zdárnému splnění zadání této práce. Dále si zde definujeme pojmy související s touto problematikou a pokusíme se jemně přiblížit navrhnuté algoritmy pro lepší řešení určitých problémů (zdroj viz (1)).

4.1 Fyzikální simulace

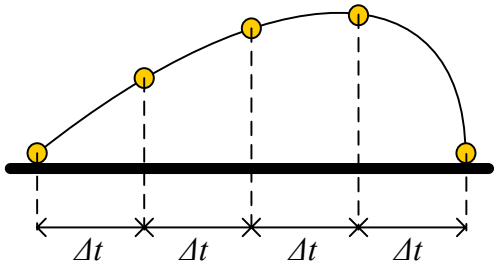
Jak již bylo řečeno, je zde využito kombinované real-time simulace. Tento typ simulace je založen na cyklickém volání simulačního kroku pro určitý relativní čas Δt.

Jestliže Δt bude mít konstantní velikost, dosáhneme tak plynulé a stabilní simulace, která bude nepřímo úměrná velikosti Δt (obr. 4.1, alg. 4.1). Pro nejpřesnější simulaci je tedy nutné nastavit hodnotu relativního kroku na nejmenší možnou hodnotu. Tedy Δt se musí limitně blížit k nule. Ovšem s takovouto konfigurací bychom nic nenasimulovali, protože pro simulaci jakéhokoli časového úseku je zapotřebí nekonečného množství simulačních kroků.

alg. 4.1, simulace daná konstantním krokem

```
Δt = Const
while(simulation is running)
{
        Simulate(Δt)
}
```

obr. 4.1, simulace s pevným časovým krokem



Pokud použijeme velikost časového kroku adaptivní k relativnímu času na cílovém počítači, dosáhneme tak real-time simulace. Relativní časový krok je tedy dán rozdílem uplynulé doby od minulého simulačního kroku (alg. 4.2).

alg. 4.2, simulace daná konstantním krokem

$t_{last} = Time()$

$\boldsymbol{while}(simulation\ is\ running)$

{

$t_{act} = Time()$

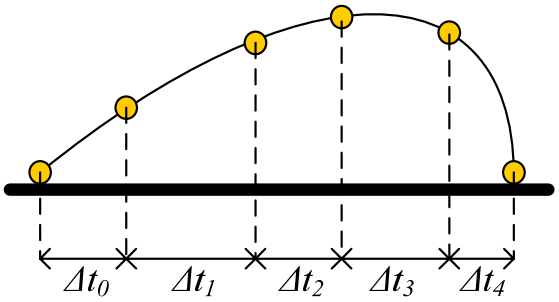
$\Delta t = t_{act} - t_{last}$

$Simulate(\Delta t)$

$t_{last} = t_{act}$

}

obr. 4.2, kroky real-time simulace



Kupříkladu mějme jednoduchý šikmý vrh (viz. obr. 4.2), kde je celková real-time simulace dána pěti kroky z počátečního stavu (čas 0) postupující v časech Δt_0 až Δt_4. Jak je z obrázku patrné, relativní časové kroky se mění v závislosti na lokální časové ose.

4.2 Problémy fyzikální simulace

Simulace fyzikálního prostředí může teoreticky vypadat jednoduše, bohužel tomu tak není. Reálný svět kolem nás je tvořen elementárními objekty, jako jsou atomy. Kdybychom opravdu chtěli simulovat pravý svět, musely bychom paralelně simulovat každý atom, což je i s dnešní výpočetní technikou zdaleka nemožné. Musíme se tedy spokojit s mnohem většími elementy jako například rigidními tělesy, což jsou celky složené z atomů.

Existují jistá chyby a nepřesnosti fyzikální simulace, které si zde stručně popíšeme v následujícím textu (zdroj viz (1)).

4.2.1 Chyba definice modelu

Jestliže převedeme fyzikální svět na matematický model pomocí diferenciálních rovnic, pak si vlastně realitu zkreslujeme a zjednodušujeme. Tento problém nás ovšem zas tak netíží, protože se spíše týká přesné, neboli inženýrské simulace.

4.2.2 Nepřesnost numerické metody

Kvůli rychlosti simulace aproximujeme diferenciální rovnice, například Taylorovým polynomem, čímž dosáhneme jen přiblížení skutečného průběhu reálné funkce.

S tímto také souvisí chyba při zaokrouhlování, a to použitím datových typů, které přibližováním k jejich maximu či minimu nemohou přesně reprezentovat výsledné číslo a zaokrouhlí jej (float, double).

4.2.3 Nepřesnost daná velkým časovým krokem

Při simulaci objekty neproudí prostorem, ale skáčou z bodu do bodu pomocí krokového času. Pokud ovšem tento krok je velký, dochází k nepřesnosti simulace. Simulace se vzdaluje od ideálního průběhu, pro přesnost krok simulace Δt by se měl limitně blížit k nule.

Dále dochází zejména k nepřesnosti výpočtu kolizí. Například mějme dva objekty v prostoru. První stojí na místě a druhý letí přímo proti němu. Při velkém časovém kroku, tj. při velkém skoku, nám metoda detekce kolizi nenalezne, což znamená, že druhý objekt ten první přeskočí. Ovšem tento problém lze vyřešit i návrhem lepší metody detekce kolize.

4.2.4 Nepřesnost daná proměnným časovým krokem

Tento problém je dán spíše nestabilitou časového kroku real-time simulace zapříčiněnou vnějšími vlivy, jako nestabilita FPS (frames per second) u grafického vykreslování (např. pokud je renderována scéna s mnoha objekty, FPS je nižší, a pokud je kamera zrovna natočena na prázdnou scénu, FPS je vyšší, a tím i kratší časový krok). Mějme dva různé simulační systémy (dva počítače, na kterých simulace proběhne), kde první je výkonnější než druhý. Aby simulace proběhly na obou stejně, je nutné na druhém systému zvětšit simulační krok. Postupným průběhem simulace pak zjistíme, že každý systém se chová jinak.

S tímto problémem se také pojí síťová komunikace. Například mějme síťovou hru více počítačů, kde každý má jiný výkon a tím pádem i různý časový krok. Jak tedy bude simulace probíhat, když se každý systém chová jinak?

5 Matematické vztahy

Zde si popíšeme jednotlivé matematické vztahy a operace použité v této práci (zdroj viz (3), (4), (5), (6)). Zejména operace nad základními prvky 3D prostoru a maticemi. Dále si popíšeme metodu *Backtransforming*, kterou jsem navrhl speciálně pro urychlení a ulehčení detekce kolize.

5.1 Vektor

Pohyb hmotného bodu vzhledem k souřadnicovému systému popisujeme pomocí *Polohového vektoru* $\vec{r}$, jehož počátek je v počátku souřadnicového systému. Takovýto vektor je definován složkami, jejichž počet je roven rozměru souřadnicového systému. Tedy pro 3D systém je vektor $\vec{r}$ dán trojící [*x,y,z*]. Dále už jen pro 3D vektory.

Pohyb bodu v souřadnicovém systému je dán funkcí

$$\vec{r} = [fx(t), fy(t), fz(t)], \tag{5.1}$$

kde *fx(), fy(), fz()* jsou funkce času .pro jednotlivé složky vektoru $\vec{r}$.

Triviální operace, což jsou základní operace vektorů jako sčítání, odčítání a násobení konstantou, zde nebudeme popisovat, soustředíme se spíše na komplikovanější operace a jejich využití.

Velikost vektoru označována jako $|\vec{r}|$ je dána vztahem

$$|\vec{r}| = \sqrt{x^2 + y^2 + z^2}, \tag{5.2}$$

kde *x*, *y* a *z* jsou jednotlivé složky vektoru $\vec{r}$.

Skalární součin vektorů neboli *Dot product*, jehož skalární výsledek je využíván například pro výpočet úhlu mezi vektory, je definován vztahem

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z = |\vec{a}||\vec{b}| cos\varphi, \tag{5.3}$$

kde vektory $\vec{a}$ a $\vec{b}$ jsou dány složkami $[a_x, a_y, a_z]$ a $[b_x, b_y, b_z]$, φ vyjadřuje úhel mezi vektory.

Vektorový součin neboli *Cross product*, jehož výsledkem je vektor kolmý k ploše tvořené dvěma vektory, je dán vztahem

$$\vec{a} \times \vec{b} = [(a_y b_z - a_z b_y), (a_z b_x - a_x b_z), (a_x b_y - a_y b_x)], \tag{5.4}$$

kde vektory $\vec{a}$ a $\vec{b}$ jsou dány složkami $[a_x, a_y, a_z]$ a $[b_x, b_y, b_z]$.

5.2 Rovina

Rovina je dána normálovým vektorem $\overrightarrow{n_P}$, bodem na rovině $\overrightarrow{p_P}$ a vzdáleností od počátku souřadnicového systému *d*. Obecná rovnice roviny je tedy dána vztahem

$$n_{Px}p_{Px} + n_{Py}p_{Py} + n_{Pz}p_{Pz} - d = 0, \tag{5.5}$$

pomocí znalostí skalárního součinu vektorů je tato rovnice ekvivalentní s rovnicí

$$\overrightarrow{n_P} \cdot \overrightarrow{p_P} = d. \tag{5.6}$$

5.2.1 Vzájemná poloha polopřímky a roviny

Vektorová reprezentace polopřímky je definována

$$\vec{w} = \vec{s} + \vec{u}w, \tag{5.7}$$

kde $\vec{s}$ je počáteční bod, $\vec{u}$ směrový vektor (normalizovaný) a w délka polopřímky od počátečního bodu.

Pokud polopřímka protne rovinu, musí platit následující rovnice:

$$(\overrightarrow{n_P} \cdot \vec{s}) + (\overrightarrow{n_P} \cdot \vec{u})w = d, \tag{5.8}$$

ze které vyjádříme vzdálenost *w*

$$w = \frac{d - (\overrightarrow{n_P} \cdot \vec{s})}{\overrightarrow{n_P} \cdot \vec{u}}, \tag{5.9}$$

dosadíme d pomocí $\overrightarrow{n_P} \cdot \vec{w} = d$ a po úpravě dostaneme

$$w = \frac{\overrightarrow{n_P} \cdot (\vec{w} - \vec{s})}{\overrightarrow{n_P} \cdot \vec{u}}, \tag{5.10}$$

kde w vyjadřuje přesnou vzdálenost od počátečního bodu polopřímky $\vec{s}$ ke společnému bodu polopřímky a roviny. Pokud w vyjádříme, výsledný bod má tvar $\vec{q} = \vec{s} + \vec{u}w$.

Detekce průniku polopřímky a roviny je jeden z důležitých operací pro optimalizaci či výpočet kolizního bodu tělesem s rovinou.

5.3 Matice

Matice je základním prvkem pro reprezentace rotační transformace vektoru. Jelikož pracujeme v 3D prostoru, využíváme matici 3×3, která je označena hranatýma závorkami a definována

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix},$$

kde $a_{11} \ldots a_{33}$ jsou jednotlivé hodnoty matice.

Determinant je speciální operace nad maticí, výsledkem této operace je skalár.

$$|[A]| = a_{11}(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{23}a_{31}) + a_{13}(a_{21}a_{32} - a_{22}a_{31}) \tag{5.11}$$

Determinant lze chápat jako objem obecného *n*-rozměrného rovnoběžníku, v našem případě jej ale používáme pro výpočet inverzní matice.

Určitou operací nad maticí $[A]$ dostaneme *Inverzní matici* k původní, tedy $[A]^{-1}$. Pro takovou matici platí rovnice $[A] \cdot [A]^{-1} = [A]^{-1} \cdot [A] = 1$, kde 1 vyjadřuje jednotkovou matici. Takovouto inverzní matici lze vypočítat řešením následující matice

$$\left(\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array}\right),$$

kde se snažíme dostat jednotkovou matici na levou stranu a výsledná pravá strana je výsledná inverzní matice. Inverzní matici $[A]^{-1}$ k matici $[A]$ využíváme zejména pro takzvanou operaci *Backtransforming*, kterou si popíšeme v následujícím textu.

Matice se využívá k mnoha transformacím jako translace, rotace, scaling (měřítko) a další. Pro naše účely je potřebná zejména rotační matice.

$$[Rot](\alpha, \beta, \gamma) = \begin{bmatrix} cos\beta\ cos\gamma & -cos\beta\ sin\gamma & sin\beta \\ sin\alpha\ sin\beta\ cos\gamma + cos\alpha\ sin\gamma & -sin\alpha\ sin\beta\ sin\gamma + cos\alpha\ cos\gamma & -sin\alpha\ cos\beta \\ -cos\alpha\ sin\beta\ cos\gamma + sin\alpha\ sin\gamma & cos\alpha\ sin\beta\ sin\gamma + sin\alpha\ cos\gamma & cos\alpha\ cos\beta \end{bmatrix}, \quad \textbf{(5.12)}$$

kde α, β, γ vyjadřují úhly rotací v jednotlivých osách souřadnicového systému.

5.3.1 Matice a vektor

Mezi nejdůležitější matematické operace nad maticí, které se využívá, je násobení matice a vektoru. Je to jakési aplikování vektoru na matici a výsledkem je opět vektor, tedy vektor transformovaný dle matice. Takovéto násobení je definováno vztahem

$$[A] \cdot \vec{V} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} a_{11}v_1 + a_{12}v_2 + a_{13}v_3 \\ a_{21}v_1 + a_{22}v_2 + a_{23}v_3 \\ a_{31}v_1 + a_{32}v_2 + a_{33}v_3 \end{bmatrix}, \quad \textbf{(5.13)}$$

kde a_{11} ... a_{33} jsou jednotlivé hodnoty matice a v_1 ... v_3 jsou jednotlivé hodnoty vektoru.

5.4 Backtransforming

Backtransforming je speciální metodika pro urychlené a přesné detekce kolize objektů v 3D prostoru. Využívá rotační matice, její inverzní matice, násobení vektoru maticí a sčítaní vektorů.

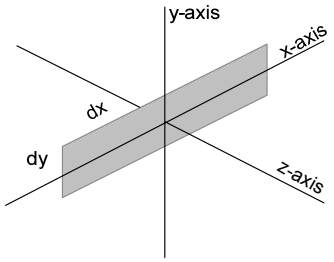
Mějme 3D objekt definovaný svým vektorem pozice v prostoru $\vec{P} = [p_1, p_2, p_3]$ a rotačním vektorem $\vec{R} = [\alpha, \beta, \gamma]$, kde α, β, γ jsou úhly rotace objektu. Pomocí rotačního vektoru $\vec{R}$ si vytvoříme rotační matici $[R]$

$$[R] = [Rot]\left(\vec{R}\right) = [Rot](\alpha, \beta, \gamma). \quad \textbf{(5.14)}$$

K této rotační matici $[R]$ objektu vytvoříme inverzní matici $[R]^{-1}$. Nyní máme vytvořené potřebné matice a můžeme pokročit ke smyslu Backtransformingu.

Mějme například 3D objekt plochy určené svou šířkou dx, výškou dy, zmíněným vektorem pozice $\vec{P}$ a vektorem rotace $\vec{R}$. Takováto plocha nám v 3D scéně vytváří kupříkladu zeď, podlahu či určitou překážku. Ideálními parametry takovéto plochy v 3D prostoru jsou vektor pozice $\vec{P} = [0,0,0]$ a vektor rotace $\vec{R} = [0,0,0]$. Při této konfiguraci můžeme přesně dle jednoduchých porovnávacích operací určit, zda jakýkoli bod v prostoru leží či neleží na ploše, jelikož plochu určují pouze parametry šířky dx, výšky dy, jak je tomu na obr. 5.1. Takovéto ideální parametry ale každá plocha či objekt v prostoru nemá. Zde přichází na řadu metoda Backtransforming, která pomocí inverzní rotační matice $[R]^{-1}$ převede jakýkoli bod v prostoru do této ideální pozice naší plochy. Kupříkladu mějme bod v prostoru daný vektorem $\vec{Q}$. Následujícím vzorcem převedeme bod do souřadnicového systému naší ideální plochy

obr. 5.1, ideální poloha plochy



$$\overrightarrow{Q'} = [R]^{-1} \cdot (\vec{Q} - \vec{P}). \quad \textbf{(5.15)}$$

Na výsledném vektoru $\overrightarrow{Q'}$ už za pomoci jednoduchých porovnávacích operacích zjistíme, zda bod na ploše leží či nikoli.

Metodika Backtransformingu se netýká pouze převodu bodu do souřadnicového systému ideální polohy objektu, ale také zpětné transformace bodu z ideální polohy do prostoru. Například mějme námi definovanou plochu. Bod, který určuje horní roh plochy je dán vektorem s ideální pozicí plochy $\overrightarrow{U'} = [\frac{dx}{2}, \frac{dy}{2}, 0]$, tento bod transformujeme do reálné pozice $\vec{U}$ v 3D prostoru pomocí následujícího vzorce

$$\vec{U} = \left([R] \cdot \overrightarrow{U'}\right) + \vec{P}. \quad \textbf{(5.16)}$$

První operace Backtransforming daná vzorcem (5.15) je stručně řečeno převod z 3D na 2D a operace daná vzorcem (5.16) je převod z 2D na 3D. Tato metodika je velmi užitečná zejména pro rychlou detekci kolize, kde využíváme znalosti tělesa a ideální pozice v prostoru. O použití Backtransformingu se zmíníme v následujících kapitolách, kde konkrétní příklady upřesní funkčnost a potenciál této metody.

6 Fyzikální vztahy

V této kapitole si rozebereme použité fyzikální vztahy a zákony ze kterých tato práce vychází (zdroj viz (6), (5)). Řekneme si něco o hmotném těžišti rigidního tělesa, rozložíme si fyzikální síly, zákony energie, hybnosti a pohybu hmotného bodu.

6.1 Těžiště rigidního tělesa

Pro *Hmotný střed*, tedy těžiště rigidního tělesa platí obecný vztah

$$\vec{r}^* = \frac{\sum_{i=0}^{n} m_i \vec{r_i}}{\sum_{i=0}^{n} m_i}, \tag{6.1}$$

kde hmotné těleso je rozděleno do n částic, kde každá i-tá částice má svoji hmotnost m_i a polohu danou vektorem $\vec{r_i}$.

V této práci se zejména zaměřujeme na ideální a obecná tělesa, nazývaná také jako *Primitivní tělesa*. Mezi takovéto tělesa patří koule, krychle, box. U takovýchto těles víme přesně, kde se vektor hmotného středu nachází. U zmíněných primitivních těles se hmotný střed nachází uprostřed tělesa. Pokud bychom vytvářeli složitější těleso z těles primitivních, využijeme vzorce (6.1), kde každá i-tá částice vyjadřuje jednotlivé primitivní těleso.

Existuje obecnější popis tělesa za pomoci trojúhelníkových polygonů. Takovéto těleso nazýváme polygonální. Výpočet hmotného středu u takovéhoto tělesa nemusí být triviální záležitostí, nelze totiž použít obecný vztah (6.1), protože těleso není rozloženo na menší částice (pokud nepočítáme jednotlivé atomy tvořící těleso). Výpočet hmotného středu polygonálního rigidního tělesa je funkcí obsahu ploch a normály všech trojúhelníků tvořící tento objekt.

6.2 Fyzikální síly

Nerovnoměrnost pohybu dynamických těles je dána zejména fyzikálními silami, které na tento hmotný objekt působí. Dle *Druhého Newtonova zákona* je vektor zrychlení funkcí hmotnosti tělesa a všech sil na něj působící, tedy

$$m\vec{a} = \sum \vec{F}, \tag{6.2}$$

kde *m* je hmotnost tělesa, vektor $\vec{a}$ jeho zrychlení a $\sum \vec{F}$ je součet všech vektorů fyzikálních sil na něj působících. Po jednoduché úpravě dostaneme výsledný vzorec pro výpočet zrychlení hmotného tělesa

$$\vec{a} = \frac{\sum \vec{F}}{m}. \tag{6.3}$$

Mezi takovéto fyzikální síly můžeme zařadit odpor prostředí, tření, různé gravitační a magnetické síly. Pomocí těchto sil lze vytvořit efekt výbuchu, který je jakousi inverzní bodovou gravitační silou.

Nyní si můžeme jednotlivě popsat některé z těchto sil, jež jsou v této práci použity.

6.2.1 Síla odporu prostředí

Odpor prostředí je síla definována vztahem

$$\overrightarrow{F_A} = -\frac{1}{2} C\rho S\vec{v}^2, \tag{6.4}$$

kde C je součinitel odporu (např. pro kouli je $C=0.5$), ρ představuje hustotu prostředí (např. pro vzduch $\rho=1.276$), S je obsah průřezu tělesa kolmý na směr pohybu a vektor $\vec{v}^2$ reprezentuje aktuální rychlost objektu. Abychom neztratili směr v 3D prostoru, pak $\vec{v}^2 = \frac{\vec{v}}{|\vec{v}|^3}$. Odpor prostředí si můžeme jednoduše představit jako negativní sílu působící na letící objekt v prostředí s nenulovou hustotou, jako vzduch či voda.

6.2.2 Plošná gravitační síla

Další použitou fyzikální silou je *Plošná gravitace*. Je to vlastně jednoduchá gravitace daná vztahem

$$\overrightarrow{F_G} = m\vec{g}, \tag{6.5}$$

kde m vyjadřuje hmotnost tělesa a $\vec{g}$ vektor plochy vytvářející gravitační pole. Pro gravitaci jak jí známe z reálného světa, tedy platí $\vec{g} = [0, -g, 0]$, kde g je gravitační konstanta země.

6.2.3 Bodová gravitační síla

Bodová gravitace je dle *Newtonova gravitačního zákona* dána vzorcem

$$\overrightarrow{F_P} = \kappa \frac{m_1 m_2 \vec{r}}{|\vec{r}|^3}, \tag{6.6}$$

kde κ je gravitační konstanta, $\vec{r}$ polohový vektor mezi tělesy, m_1 a m_2 hmotnost jednotlivých těles. Tento vzorec je bohužel pro naše účely nepoužitelný, protože vzájemná síla mezi dvěma tělesy je tak malá, že jsme tento zákon upravili na jakýsi gravitační bod, který ovlivňuje všechny hmotné tělesa v okolí. Výsledný vzorec má tvar

$$\overrightarrow{F_P} = \frac{Gm\vec{r}}{|\vec{r}|^3}, \tag{6.7}$$

kde G určuje bezrozměrnou intenzitu gravitačního pole zadanou uživatelem a m hmotnost cílového tělesa, který je touto bodovou gravitační sílou ovlivňován. Oproti plošné gravitaci má bodová navíc svoji pozici v prostoru, tedy $\vec{r}$ určuje polohový vektor mezi pozicí bodu gravitace a cílovým hmotným tělesem. Pomocí této gravitační síly už lze jednoduše simulovat například pohyb družic na oběžné dráze či pohyb planet. Zajímavou aplikací této bodové gravitační síly jsou exploze. Pokud intenzita gravitačního pole má na určitý časový moment (časovou délku exploze) zápornou a

dostatečně velikou hodnotu, síla těleso naopak bude odpuzovat a odmrští jej co nejdále od pozice této gravitační síly.

6.3 Pohyb hmotného bodu

Fyzikální simulace této práce je založena na kinematice a dynamice hmotného bodu. Hmotný bod pro nás vyjadřuje rigidní těleso a veškerý pohyb je funkcí času t. Takovéto těleso má svůj polohový vektor $\overrightarrow{s_0}$ a vektor počáteční rychlosti $\overrightarrow{v_0}$.

Rychlost tělesa je definována vztahem

$$\vec{v} = \int \vec{a}\, dt = \vec{a}t + \overrightarrow{v_0}, \tag{6.8}$$

kde vektor zrychlení $\vec{a}$ vypočítáme pomocí *Druhého Newtonova zákona* ze vztahu (6.3), popsaného v následujících kapitolách.

Aktuální polohový vektor tělesa lze vypočítat ze vztahu

$$\vec{s} = \int \vec{v}\, dt = \frac{1}{2}\vec{a}t^2 + \overrightarrow{v_0}t + \overrightarrow{s_0}, \tag{6.9}$$

Jak lze vidět, neznámým parametrem je pohybový čas t. Jelikož simulace probíhá cyklicky pro určitý časový úsek Δt, platí tedy pro inkrement pohybu tělesa $\overrightarrow{\Delta s}$ vztah

$$\overrightarrow{\Delta s} = \frac{1}{2}\vec{a}\Delta t^2 + \overrightarrow{v_0}\Delta t. \tag{6.10}$$

6.4 Energie a hybnost

Pohybový stav hmotného bodu charakterizuje veličina *Hybnost* hmotného bodu, definována vztahem

$$\vec{p} = m\vec{v}. \tag{6.11}$$

Kinetická energie pohybujícího se hmotného bodu je

$$E_k = \frac{1}{2}m\vec{v}^2. \tag{6.12}$$

kde *m* je hmotnost a $\vec{v}$ je vektor rychlosti hmotného bodu.

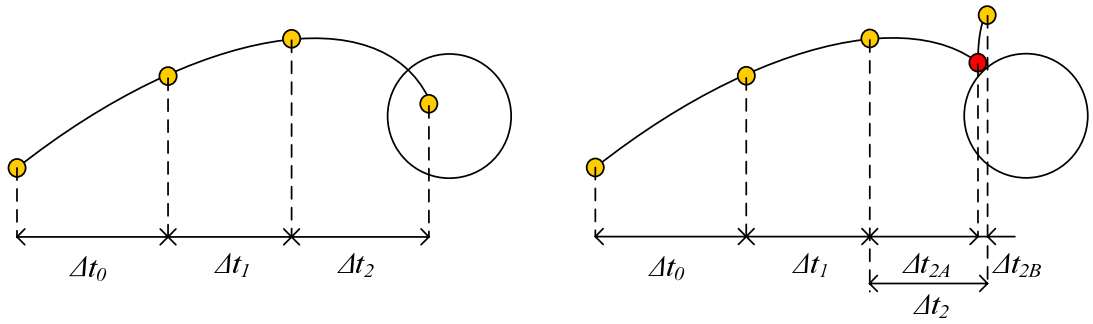
Těchto zákonů dynamiky hmotného bodu se využívá při výpočtu odrazu srážky dvou hmotných těles.

7 Detekce kolizí

Detekce kolizí je jednou z nejkomplikovanějších částí simulace fyzikálních těles (zdroj viz (3), (7), (8), (9), (10)). Detekce a řešení kolizí je právě ta část, která dělá simulaci částečně diskrétní, mění totiž stavy a pohyb jednotlivých těles z důvodu srážky. Režie této operace má největší podíl na celkové režii simulace, proto je nutné této části věnovat mnoho času a promyslet možné optimalizace. Nejprve si popišme co vlastně detekce kolize je.

Kolizi můžeme definovat jako společný průnik objemu dvou nebo více hmotných těles mezi sebou. Což je dle fyzikálních zákonů jak je známe nemožné. Proto tento stav těles musíme včas detekovat a určit přesný časový okamžik, kdy tělesa mají mezi sebou společný pouze jednobodový průnik, tzv. *Kolizní bod*.

obr. 7.1, simulace bez detekce kolize (vlevo), simulace s detekcí kolize (vpravo)



Na obr. 7.1 vlevo lze vidět průběh tří kroků simulace, kde v posledním kroku dojde k průniku těles. Detekce kolize musí najít bod kolize, jako tomu je na obr. 7.1 vpravo v čase Δt_{2A}. Výsledný odraz je vypočítán pro časový úsek Δt_{2B}, aby se dokončil celý simulační krok Δt_2.

Jak je z předchozího textu jasné, to co při detekci hledáme je čas kolize. Nazvěme si ho t_K. Samotný bod kolize je funkcí t_K. Tento čas stačí pouze aplikovat do pohybových rovnic a dostaneme výsledný bod. Existují jisté metodiky, které nám dovolí vypočítat bod t_K pomocí bodu kolize. Ty jsou dané matematickým odvozením pohybu těles, ovšem tato metodika lze použít jen ve speciálních případech. Kupříkladu pokud se náš hmotný bod pohybuje pouze po přímce, tedy při rovnoměrném přímočarém pohybu, kde poloha je dána vztahem

$$\vec{s} = \int \vec{v}\, dt = \vec{v}t + \overrightarrow{s_0} \tag{7.1}$$

a čas tedy vyjádříme jako

$$t = \frac{\vec{s} - \overrightarrow{s_0}}{\vec{v}}. \tag{7.2}$$

Pokud do vektoru $\vec{s}$ dosadíme matematicky vypočítaný bod kolize, pak čas nárazu lehce vypočítáme. Jak již bylo řečeno, analytické vyjádření bodu kolize jde pouze pro jednoduché pohybové rovnice. Existují ale i jiné, obecnější metodiky, jak nalézt bod kolize. Tou nejzákladnější je

Metoda půlení intervalů (zdroj viz (4)). Není příliš rychlá ani přesná, ale její implementace je vcelku jednoduchá a dá se snadno optimalizovat.

7.1 Metoda půlení intervalů

Nechť délka simulačního kroku je dána časem Δt a nechť simulační interval I je definován dvojící $\langle I_S, I_E \rangle$. Pokud je detekován průnik těles (kolize) v čase Δt, pak bod kolize leží v časovém intervalu $I = \langle 0, \Delta t \rangle$. Definujme časový bod t_H, který má hodnotu přesně v půli intervalu I. Tedy $t_H = \frac{I_S + I_E}{2}$. Proveďme simulační skok pro čas t_H. Pokud je detekován průnik těles, nechť $I_E = t_H$. V opačném případě kdy průnik těles je nulový, zjisti přesnost kolizního bodu. Jestliže přesnost nebude dostačující, nechť $I_S = t_H$. V dalším kroku opakuj simulační skok pro $t_H = \frac{I_S + I_E}{2}$, dále otestuj průnik těles a řiď se již definovanými pokyny. Opakuj, dokud nebude přesnost dostačující, výsledný kolizní čas je v t_H a tedy i kolizní bod je dán funkcí $\vec{s}(t_H)$. Přesnost kolizního bodu lze určit několika způsoby. Tou nejzákladnější je minimální časový skok dt. Pokud je rozdíl hodnoty t_H a hodnoty v minulém výpočtu t_H menší než dt, tedy až $\Delta t_H < dt$ přesnost je dostačující. Další přesnost lze určit rozdílem vzdálenosti v dvou po sobě jdoucích krocích, kde tato vzdálenost bude menší než definovaná minimální vzdálenost ds, tedy až $\Delta s < ds$. Jednou z přesností, která je spíše dána z důvodu ochrany proti zacyklení, je určení maximálního počtu interakcí půlení intervalu.

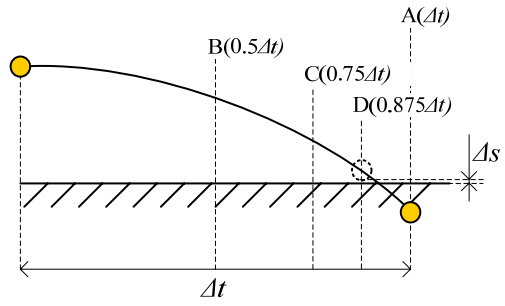
Pro lepší pochopení popíšeme algoritmus půlení intervalu v pseudo-matematickém jazyce (alg. 7.1). Mějme dvě tělesa A, B s geografickou reprezentací objemu V_A, V_B a pohybovými funkcemi $fs_A(t)$, $fs_B(t)$.

alg. 7.1, metoda půlení intervalů

$fs_A(\Delta t)$
$fs_B(\Delta t)$
$\boldsymbol{if}(V_A \cap V_B)\boldsymbol{then}\{$
    $I_S = 0$
    $I_E = \Delta t$
    $\boldsymbol{repeat}\ \{$
        $t_H = \frac{I_S + I_E}{2}$
        $fs_A(t_H)$
        $fs_B(t_H)$
        $\boldsymbol{if}(V_A \cap V_B)\ \boldsymbol{then}\ I_E = t_H$
        $\boldsymbol{else}\ \{$
            $Set\ \Delta s, \Delta t_H$
            $I_S = t_H$
        $\}$
    $\}\boldsymbol{until}((\Delta s < ds)\ OR\ (\Delta t_H < dt)\ OR\ MaxInteracitons())$
$\}$

Pro ilustraci si můžeme metodu půlení intervalů prohlédnout na obr. 7.2. Simulace kroku je v časovém intervalu Δt. Metoda nalezla přibližný bod kolize ve čtyřech po sobě jdoucích časových krocích A, B, C a D. Kde v kroku D byla nalezena dostačující přesnost $\Delta s < ds$.

obr. 7.2, nalezení bodu kolize



Existují určité optimalizace téhle metody založené zejména na definování přesnějšího počátečního intervalu $I = \langle I_S, I_E \rangle$. Pokud rozsah tohoto intervalu zmenšíme, počet interakcí půlení intervalu bude menší. Takováto optimalizace je ovšem závislá na konkrétním užití. Nelze ji tedy vždy obecně použít.

Mějme podobnou situaci jako na obr. 7.2, kde kulička letí prostorem rovnoměrně zrychleným přímočarým pohybem (rovnice (6.9)). V takovéto situaci nelze jednoduše analyticky vypočítat přesný bod kolize, ale lze ho odhadnout s určitou přesností. To docílíme převodem rovnoměrně zrychleného přímočarého pohybu na pohyb rovnoměrně přímočarý (rovnice (7.1)), ale pouze a jen v případě, když velikost vektoru zrychlení bude větší než 1 ($|\vec{a}| > 1$, z důvodu odhadu nekolizního počátečního času). Pomocí rovnice (7.1) můžeme jednoduše analyticky vypočítat čas kolize, který použijeme jako odhad, tedy počáteční interval I_S.

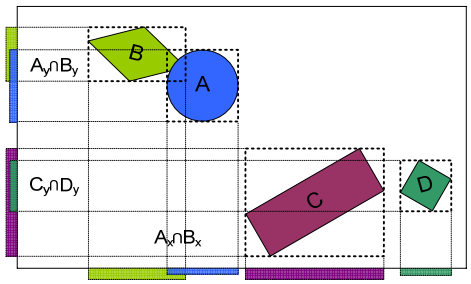
7.2 Metody a optimalizace detekce kolize

Operace detekce kolizí se provádí pouze pro dvě tělesa mezi sebou. Simulace pouze dvou těles by nám byla asi k ničemu, pokud je jich v systému více než jen dvě, musíme tedy určitým chováním otestovat kolizi mezi všemi. Mějme tedy n těles a všechny musí otestovat kolizi mezi sebou, tj. $\frac{n^2-n}{2}$ testu kolize. Což nám dává exponenciální složitost. Takováto složitost je pro naše účely naprosto nepřípustná, proto musíme do systému zavést určité optimalizace. Nejprve si popišme některé klasické metodiky optimalizace a navrženou metodiku optimalizace této práce, a dále výhody a nevýhody jednotlivých metodik včetně aplikace.

7.2.1 BoundingBox

Jednou z metodik, které se soustřeďuje zejména na optimalizací prvotního testu kolize, se jmenuje *Obalování těles* (zdroj viz (8)). Využívá takzvaných *Boundignboxů*, kterými se složitější těleso obalí do jakýchsi krabicových obálek. Nejprve dojde k rychlému testu kolize obálek těles (BoundingBox), když všechny rozměry mají společný průnik, volá se pro obalovaná tělesa přesná detekce kolize. 2D ukázka obalovaných těles lze vidět na obr. 7.3. Vytvoření obálky tělesa je vcelku jednoduchá operace, hledáme totiž jen maximální a minimální hodnoty všech hran tělesa. Z těchto hodnot vytvoříme dva vektory určující krabicovou obálku. Výhodou této metodiky je její jednoduchost a nízká režie rychlého testu kolize krabic, využívá pouze operace porovnání větší či menší. Výhodou jednoduchých algoritmu je jejich potenciál v hardwarové implementaci, takže se tato metodika dá využívat v takzvaných fyzikálních kartách. Používání BoundingBoxů se zejména hodí v simulacích menšího počtu těles, je totiž stále závislá na exponenciální složitosti dané počtem těles $\frac{n^2-n}{2}$. Existují ale vylepšení, které tuto složitost zmenšují, čímž se tato metodika vystavuje do popředí a proto je to jedna z nejpoužívanějších a vcelku rychlých testu detekce kolize. Nesmíme zapomenout na to, že obalování těles nedetekuje přesné kolize těles, ale pouze a jen průniky jejich obalů. Dá se aplikovat pouze do prvotního testu průniku. Přesnou detekci nám musí obstarat jiné, komplikovanější a mnohdy náročnější algoritmy. Detekce kolize obalovaných těles nám pouze separuje jen ty tělesa, u kterých je kolize možná. Kupříkladu jak je tomu na obr. 7.3 u těles s označením *A*, *B*.

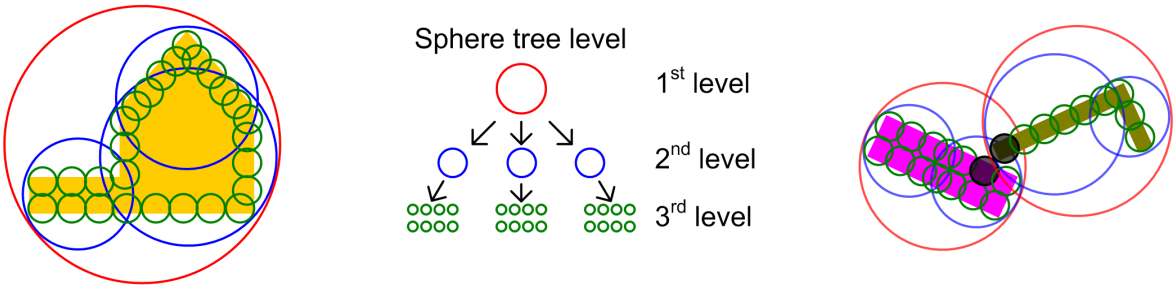
obr. 7.3, tělesa obalená do BoundingBoxů



7.2.2 Sphere-Tree

Další metoda částečně vycházející z obalování těles se nazývá *Sphere-Tree* (zdroj viz (11), (12)). Oproti BoundignBoxům, kde se využívá krabic, používá Sphere-Tree pouze koulí. Podobně jako do krabic obalí geometrické těleso do obalových koulí, ovšem obalování tělesa má více úrovní přesnosti. Čím vyšší úroveň, tím větší přesnost geometrického popisu tvaru tělesa. Úrovně přesnosti jsou ale navzájem propojené stromem, tedy každá obalová koule nižší úrovně má pod sebou přesnější popis vyšší úrovně. Nižší úroveň obsahuje vyšší jak je tomu na obr. 7.4b.

obr. 7.4(a,b,c), metoda detekce kolize Sphere-Tree

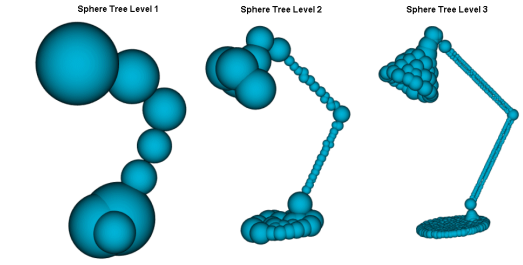


Ukázka komplikovanějšího geometrického tělesa popsaného pomocí této metody lze vidět na obr. 7.4a, Obalová koule první úrovně určuje celkový objem geometrického tělesa, stejně jak je tomu u BoundingBoxů. Druhá úroveň popisuje základní útvary, ze kterých se těleso skládá a tato úroveň v sobě obsahuje poslední množinu koulí, které už nejsou obálky, ale popisují geometrický povrch tělesa. Právě na té nejnižší úrovni Sphere-Tree koule popisují povrch a tím detekují relativně přesnou kolizi s jiným tělesem (obr. 7.4c). Přesnost detekce kolizního bodu a tedy i přesnost času je dána hloubkou zanoření Sphere-Tree. Algoritmus detekce kolize je u této metody vcelku jednoduchý, využívá se zde rekurzivního sestupu ve stromu obalovaných těles. Nejprve se zavolá test detekce kolize pro nejnižší úroveň, tedy test porovnání průniku dvou koulí. Pokud test projde, rekurzivně se zavolá test kolize pro všechny obalové koule nižší úrovně. Po výsledném rekurzivním sestupu dostaneme přesné koule nejvyšší úrovně, které mezi sebou kolidují (mají společný průnik). Jedinou náročnou operací je zde test průniku dvou koulí, který se provádí pro všechny v nižší úrovni. Tento velmi častý test se dá vypočítat vzorcem

$$d = |\overrightarrow{s_A} - \overrightarrow{s_B}| - (r_A + r_B), \tag{7.3}$$

pokud d je záporné, kolize nastala. Vektory $\overrightarrow{s_A}$, $\overrightarrow{s_B}$ zde vyjadřují polohou středu v souřadnicovém systému koule A a B. Hodnoty r_A, r_B jsou jejich geometrické poloměry. Výpočeně náročnou operací je zde velikost rozdílu polohových vektorů $\overrightarrow{s_A}$, a $\overrightarrow{s_B}$, který je dán vztahem (5.2). Obsahuje totiž několik mocnin, což je velmi pomalá operace. Na druhou stranu, jelikož je vztah (7.3) jediným geometrickým testem kolize, lze tuto metodu využít v hardwarové implementaci. Například integrovat jej do grafických akcelerátorů. Tím by se celá tato metoda podstatně zrychlila.

obr. 7.5, ukázka tělesa obalené do Sphere-Tree



Sphere-Tree je dle mého názoru jedna z nejlepších metod detekce kolize, už jen pro její jednoduchost a možnost optimalizací. Přesnost ovšem není tak závratná, ale pro aplikaci do her dostačující. Lze ji zlepšit na úkor rychlosti, stačí pouze zvýšit úroveň zanoření obalového stromu. Na obr. 7.5 (zdroj viz (12)) je zobrazena ukázka rozložení stromu komplikovanějšího 3D tělesa na třech úrovních přesnosti. Jednou z největších výhod této metody je popsání jakéhokoli povrchu tělesa, což zvyšuje její použitelnost. Na druhou stranu problémem může být vytvoření stromu geometrického tělesa, to bude vyžadovat složitější algoritmus. Stávající nevýhodou Sphere-Tree, stejně jako u BoundingBoxů je její počáteční testovací exponenciální složitost $\frac{n^2-n}{2}$ (test kolize každý s každým), kde n vyjadřuje počet těles v testované scéně.

7.2.3 BSP-Tree

Jednou z nejpoužívanějších metod se nazývá *BSP-Tree* (*BSP Strom*, zdroj viz (13)). Celá scéna je rozdělena do geograficky separovaných částí, které dle lokace vyjadřují jednotlivé uzly BSP stromu. V každém podprostorů je přibližně stejný počet těles. Test kolize se pro vybrané těleso zjišťuje pouze pro tělesa stejného podprostoru. Tímto dosáhneme snížení exponenciální složitosti testu kolize každý s každým na polynomiální.

7.2.4 Tile sets

Podobně jako BSP stromy pracuje mnou navržená optimalizace detekce kolize *Tile sets* (*Podprostorové množiny*). Definujme si třírozměrný prostor jako box $\overrightarrow{W}$ daný

$$\overrightarrow{W} = [dx, dy, dz], \tag{7.4}$$

kde dx, dy a dz vyjadřují šířku, výšku a hloubku. Takovýto omezený prostor lze rozdělit na $[nx, ny, nz]$ podprostorů (tiles), přičemž jeden podprostor má velikost

$$\overrightarrow{w_i} = \left[\frac{dx}{nx}, \frac{dy}{ny}, \frac{dz}{nz}\right]. \tag{7.5}$$

Prostor $\overrightarrow{W}$ jsme tedy rozdělily na konečný počet q podprostorů $\overrightarrow{w_i}$, kde $i < q$. Pokud objemové znázornění tělesa V leží v prostoru $\overrightarrow{W}$, leží také v konečném počtu m podprostorů. Existuje tedy množina podprostorů M_V, ve které každý prvek podprostoru $\overrightarrow{w_i}$ obsahuje těleso V,

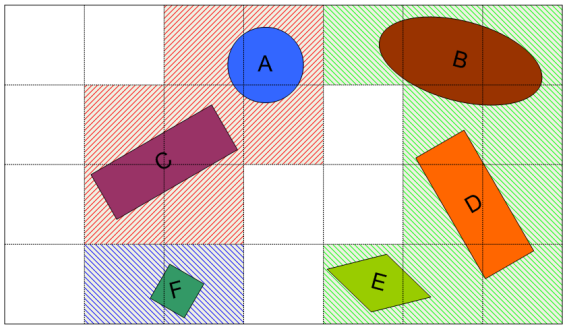
$$M_V = \{\overrightarrow{w} | (\overrightarrow{w} \in \overrightarrow{W}) \wedge (V \in \overrightarrow{w})\} \tag{7.6}$$

pak pro množinu T_V danou všemi tělesy podprostorů $\overrightarrow{w}$ z množiny M_V, platí *lokální kolize*,

$$T_V = \{v | (v \in \overrightarrow{w}) \wedge (\overrightarrow{w} \in M_V) \wedge (v \in T)\}, \tag{7.7}$$

kde T je množina všech těles. Lokální kolize je vlastně test detekce kolize každý s každým v množině T_V. Stručně řečeno hledáme tělesa v bezprostřední blízkosti dané podprostory, u kterých je možná kolize. Tělesa v této množině testují kolize mezi sebou, tj. pouze v rozsahu množiny T_V. Tímto jednoduše dosáhneme snížení exponenciální složitosti testu kolize každý s každým na polynomiální. Na obr. 7.6 je názorně ukázáno rozložení množin těles danou polohou v podprostorech dvojdimenzionálního prostoru. Detekce je tedy rozdělena na tři množiny. První obsahuje prvky A a C, druhá pouze F a třetí obsahuje B,D a E. Testu kolize se tedy provede $\frac{1^2-2}{2}+\frac{2^2-2}{2}+\frac{3^2-2}{2}=4$, kdežto u klasických metod $\frac{6^2-2}{2}=15$. Někdo by mohl podotknout, že těleso B nemusí testovat kolizi s E, protože nemají společný podprostor. Opak je ale pravdou, zatím se nezmiňujeme o problému *Multikolize*, kde například po vyřešení kolize mezi tělesy B a E může dojít ke kolizi s tělesem D. Tedy v jednom simulačním kroku nastane více kolizí v závislosti na kolizích jiných těles. Tuto problematiku a řešení si popíšeme v následujících kapitolách.

obr. 7.6, rozdělení těles v prostoru pomocí metody podprostorových množin



Technicky vzato, každý podprostor má list těles, které leží vně, ten se mění podle pohybu dynamických těles během simulace. V této metodě se také využívá BoundingBoxů jako obalových těles pro detekci podprostoru, ale více až v konkrétní implementaci podprostorových množin.

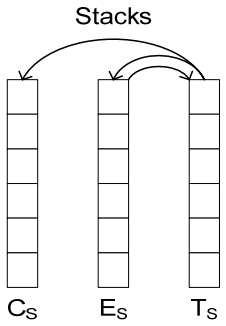
Tato metoda není příliš rychlá kvůli režii podprostorů, zvláště v simulačních scénách s malým počtem těles. Ovšem je vhodná na rozsáhlé simulační scény s velkým počtem dynamických a statických těles. Statická tělesa totiž v jakémkoli počtu nemají téměř žádnou režii při testu kolize. Mějme například počet dynamických těles ve scéně n_D a počet statických těles n_S. Ve většině případů má aplikace n_S mnohem větší hodnotu než n_D. Složitost testu kolize každý s každým například u BoundingBoxů je $s_{BB}=\frac{(n_D+n_S)^2-(n_D+n_S)}{2}$, kdežto u metody Podprostorových množin statisticky polynomiální (nelze určit přesný výraz složitosti, závisí na rozložení dynamických objektů ve scéně). Důležité ale je, že složitost je funkcí pouze dynamických těles, tedy $s_{PM}=f(n_D)$., u klasických metod je to $s=f(n_D+n_S)$.

7.2.5 Určení množiny lokální kolize dle metody Tile sets

Detekce množiny lokální kolize pomocí metody podprostorových množin lze jednoduše implementovat za pomoci tří zásobníků (viz. obr. 7.7). Zásobník T_S slouží k uchování identifikátorů podprostorů (*Tile Stack*). E_S slouží jako expanze dynamických těles, zde jsou uchovávána pouze

dynamická tělesa (*Expand Stack*). Poslední zásobník C_S uchovává tělesa lokální kolize (*Collision Stack*), zde také najdeme výsledek algoritmu, tedy hledanou množinu. Na obr. 7.7 lze také vidět směrové šipky určující komunikaci jednotlivých zásobníků mezi sebou. Zde si můžeme všimnout, že zásobník C_S nijak nekomunikuje s ostatními, pouze získává data z T_S, je to tedy výstupní zásobník.

obr. 7.7, rozložení zásobníků při algoritmu podprostorových množin



Mějme množinu dynamických těles D a množinu statických těles S. Nechť každé dynamické, statické těleso i podprostor lze označit jako *použitý* (v grafické reprezentaci označujeme šedou barvou). U_D je tedy množina použitých dynamických těles, U_S množina všech použitých statických těles a U_T množina použitých podprostorů. Každý zásobník (kromě C_S) dodržuje určité chování. Pokud E_S není prázdný, zpracovává položku, tedy dynamické těleso E na vrcholu zásobníku. Nalezne množinu R_E všech jeho podprostorů ve kterých leží a vloží všechny nepoužité do zásobníku T_S. Následně označí prvky množiny nalezených podprostorů označí za použité. Toto chování lze popsat následovně

$$E_S(E) \overset{def}{\Longleftrightarrow} \big(T_S \cup\!= (R_E \setminus U_T)\big), (U_T \cup\!= R_E). \tag{7.8}$$

Zásobník T_S při neprázdnosti zpracovává podprostor T na vrcholu zásobníku. Zjistí množinu těles R_T které leží vně a vloží všechny nepoužité do zásobníku C_S. Nepoužitá dynamická tělesa z R_T ještě vloží do zásobníku E_S. Označí statická i dynamická tělesa vkládaná do zásobníku E_S za použitá. Toto chování lze popsat následovně

$$T_S(T) \overset{def}{\Longleftrightarrow} \big(C_S \cup\!= \big(R_T \setminus (U_D \cup U_S)\big)\big), \big(E_S \cup\!= \big(D \cap (R_T \setminus U_D)\big)\big), (U_S \cup\!= (R_T \cap S)), (U_D \cup\!= (R_T \cap D)). \tag{7.9}$$

Hledaná množina v C_S je kompletní, pokud zásobníky E_S a T_S jsou prázdné, tedy platí

$$End() \overset{def}{\Longleftrightarrow} (E_S \equiv \emptyset) \wedge (T_S \equiv \emptyset). \tag{7.10}$$

Pro spuštění vyhledávání je nutné inicializovat zásobníky a množiny. Zásobníky E_S a C_S musí být zaplněny jediným dynamickým objektem Q a množina statických použitých těles musí být prázdná z důvodu detekce jiných lokálních kolizí (bude vysvětleno později), tedy

$$Start(Q) \overset{def}{\Longleftrightarrow} (E_S = \{Q\}), (C_S = \{Q\}), (U_S = \emptyset), (U_D = \{Q\}). \tag{7.11}$$

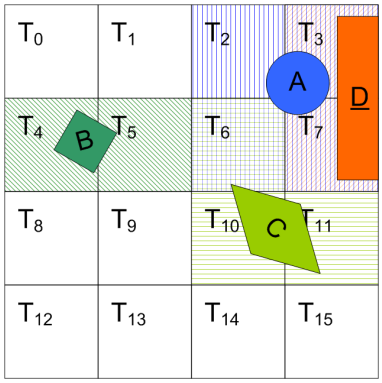
Nyní si můžeme pomocí těchto definicí zásobníků algoritmus implementovat pomocí pseudo-matematického jazyka (alg. 7.2).

alg. 7.2, detekce lokální kolize pomocí metody podprostorových množin

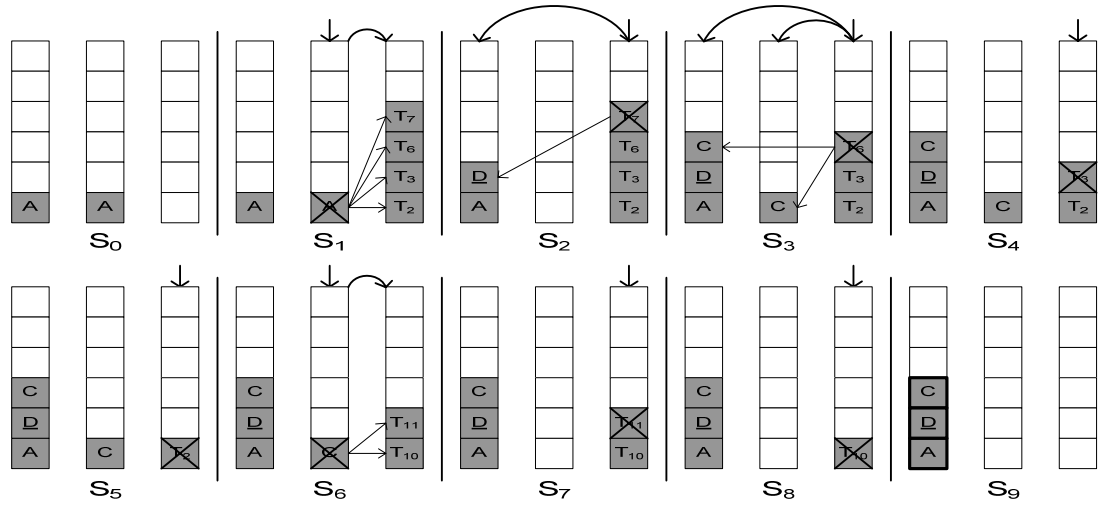
```
U_D = U_T = T_S = ∅ // Inicializace použitých množin.
foreach(Q from D) { // Cyklus přes všechna dynamická tělesa.
    if(Q ∉ U_D) { // Pokud je dynamické těleso nepoužité, proveď.
        Start(Q) // Provedení inicializace vyhledávání lokální kolize dle vztahu (7.11).
        repeat {
            while(E_S ≡ ∅){E_S(E)} // Opakuj vztah (7.8) dokud není zásobník E_S prázdný.
            while(T_S ≡ ∅){T_S(T)} // Opakuj vztah (7.9) dokud není zásobník T_S prázdný.
        }until(End()) // Opakuj dokud neplatí vztah (7.10).
    }
    SolveLocalCollision(C_S) // Zde máme cílovou množinu C_S, můžeme na ní řešit lokální kolizi.
}
```

Chování a průběh algoritmu si ukážeme na konkrétním příkladě, který je znázorněn na obr. 7.8. T_0 až T_{15} označují podprostory (tile) a na obrázku máme také tři dynamická tělesa ***A***, ***B***, ***C*** a jedno statické těleso ***D***. Dle alg. 7.2 cyklicky jdeme přes všechny tělesa ve scéně. Inicializujeme tedy zásobníky podle tohoto prvního tělesa, tak jak je tomu ve stavu S_0 na obr. 7.9. Nyní začněme zpracovávat zásobník E_S, ve stavu S_1 se tedy expandují podprostory z tělesa A do zásobníku T_S, včetně jejich označení za použité. Jelikož zásobník E_S je následně prázdný, začne se zpracovávat zásobníku T_S. Ve stavu S_2 je v podprostoru nalezeno nepoužité statické těleso ***D***, které se následně označí a přidá do C_S zásobníku. Stav S_3 nalezne těleso ***C*** a po označení přidá do C_S a E_S zásobníku jelikož je dynamické. Stav S_4 a S_5 pouze zpracovává do prázdnosti zásobník T_S. Dle alg. 7.2 se pokračuje dále, až dosáhneme prázdných zásobníků E_S a T_S. Tento stav S_9 nám identifikuje konec detekce, nyní můžeme na cílovou množinu lokální kolize aplikovat řešení kolizí (např. algoritmus řešení multikolize viz kapitola 7.5 Multikolize).

obr. 7.8, rozložení těles ve scéně při ukázce algoritmu podprostorových množin

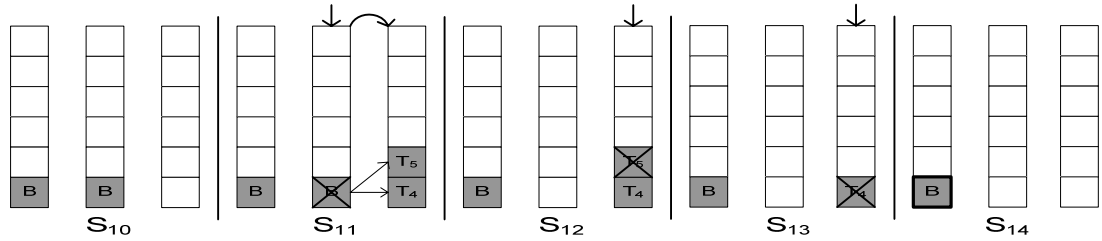


obr. 7.9, stavy zásobníku při hledání lokální kolizi tělesa A v obr. 7.8



Jelikož v cílové scéně jsou čtyři tělesa a my jsme zatím detekovali jen tři, alg. 7.2 nám nalezne lokální kolize pro zbývající. Ukázku dokončení algoritmu detekce můžeme vidět na obr. 7.10.

obr. 7.10, stavy zásobníku při hledání lokální kolizi tělesa B v obr. 7.8



Algoritmus detekce lokálních kolizí je vcelku jednoduchý a výpočetně nenáročný. Povšimněme si, že expanze dalších podprostorů lokální kolize je dána pouze dynamickými tělesy, což dokazuje nezávislost složitosti metody podprostorových množin na počtu statických těles.

7.3 Test kolize

Test kolize je matematická operace mezi dvěma tělesy pomáhající nalezení přibližného bodu kolize při použití metody půlení intervalů. Test kolize nám přímo nenalezne tento bod, ale jeho určité vlastnosti jsou identifikátorem přesnosti nalezení. Tuto operaci bychom mohly definovat jako funkci vlastnosti průniku dvou těles. Tyto tělesa buď průnik nemají, tím pádem nemají ani kolizi, nebo ho mají a jeho velikost nepřímo úměrně určuje vzdálenost od kolizního bodu. Je nutné podotknout, že tento test kolize je nejčastější operací nad celou fyzikální simulací. Proto je nutné složitost této operace optimalizovat. Použití již popsaných optimalizací detekce kolize jako např. metoda nalezení podprostorových množin pouze minimalizuje počet použití testů kolize, ne její složitost.

Nyní si popíšeme použité metody testu kolize primitivních těles mezi sebou. A to nejprve nejjednodušší test kolize mezi koulí a koulí. Dalším testem bude test kolize koule s plochou, kde popíši mnou navržený algoritmus rychlé detekce za pomoci metody backtransformingu.

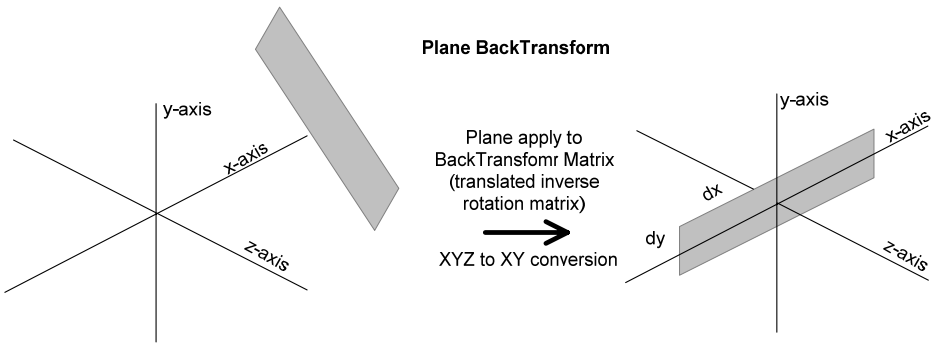
7.3.1 Koule vs. Koule

Test kolize mezi dvěma koulemi je díky geometrické ideálnosti koule velmi jednoduchý. Pouze vypočítáme vzdálenost mezi koulemi a jeho hodnotou zjistíme, zda koule mají mezi sebou průnik či nikoli. K tomuto testu stačí využít vztahu (7.3).

7.3.2 Koule vs. Plocha

Definujme *plochu* jako rovinu, danou svým normálovým vektorem (rotací) a hranicemi šířky dx a výšky dy. Takovouto ohraničenou rovinu si můžeme představit jako obdélník o rozměrech dx a dy. Jelikož plocha má svoji rotaci a rozměry, nelze tak jednoduše zjistit kolizi mezi koulí a plochou v obecném prostoru. Existují sice jisté metody, které nám analyticky dokážou vypočítat vzdálenost mezi plochou a koulí, tím získáme informaci o průniku koule s rovinou plochy, ovšem analytické zjištění ohraničení roviny plochou (rohy) nelze tak jednoduše zjistit. Proto byla vyvinuta metoda *Backtransforming*, popsaná v dřívějších kapitolách. Ta využívá ideální polohy plochy v souřadnicovém systému. Transformace plochy dle Backtransformingu lze vidět na obr. 7.11.

obr. 7.11, backtransforming plochy



Pokud aplikujeme transformaci danou vztahem (5.15) na polohový vektor koule K, $\overrightarrow{s_K}$, pak

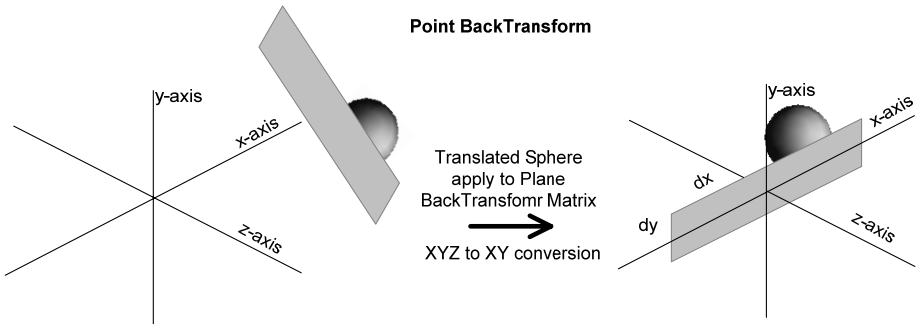
$$\overrightarrow{s_K}' = [R_P]^{-1} \cdot (\overrightarrow{s_K} - \overrightarrow{s_P}), \tag{7.12}$$

kde $\overrightarrow{s_P}$ vyjadřuje polohový vektor plochy a $[R_P]^{-1}$ je inverzní rotační matice plochy P. Dle definice Backtransformingu dostanu kouli K transformací na souřadnici koule K' v $\overrightarrow{s_K}'$. Tento polohový vektor se týká ideálního stavu plochy P, ten nazývejme P'. Jeho polohový vektor $\overrightarrow{s_P}' = [0,0,0]$, nemá tedy žádnou rotaci, tak jak je tomu na obr. 5.1. Prvotní test kolize průnikem transformované koule K' je pouze testem absolutní hodnoty z-ové souřadnice

$$Test\ (|s_{Kz}'| \leq r_K), \tag{7.13}$$

kde r_K je poloměr koule K, tedy i K'. Pro ilustraci na obr. 7.12 můžeme vidět jak se vlastně Backtransforming koule K (vlevo) transformuje na K' (vpravo). Musíme si ještě ujasnit, že plocha P (vlevo) se netransformuje do P', ale my její ideální postavení známe, tedy transformujeme pouze kouli K.

obr. 7.12, backtransforming bodu, resp. středu koule

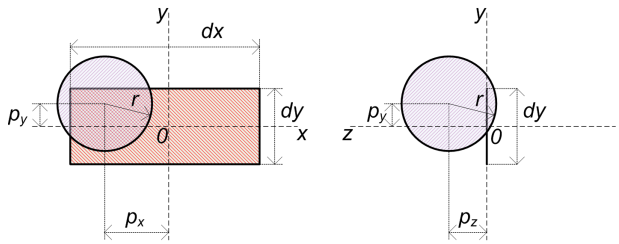


Pokud koule K' po testu z-ové souřadnice průnik má, musíme zjistit, zda neleží mimo obdélník plochy. To zjistíme tentokrát testem x-ové a y-ové souřadnice $\overrightarrow{s_K'}$ k rozměrům plochy dx a dy plochy

$$Test\ \left(\left(|s_{Kx}'| \leq \frac{dx}{2} + r_K\right)\ AND\ \left(|s_{Ky}'| \leq \frac{dy}{2} + r_K\right)\right). \tag{7.14}$$

První dva testy (7.13) a (7.14) jsou znázorněny na obr. 7.13. Zde hodnoty p_x, p_y vyjadřují podmínku (7.14) a hodnota p_z podmínku (7.13). Existuje ale ještě jeden speciální případ vzájemné kolidující polohy plochy s koulí, který musíme otestovat. Pokud alespoň jeden rozměr koule K' (x nebo y) je mimo rozsah rozměrů plochy P' (dx nebo dy) a z-ová souřadnice vykazuje jasný průnik, je možné že koule K' koliduje s rohem plochy P'. Zde je nutné otestovat vzdálenost mezi středem koule $\overrightarrow{s_K'}$ a rohem plochy vykazující tento speciální případ.

obr. 7.13, ukázka kolize plochy a koule



$$\begin{aligned} &\boldsymbol{if}\left(|s_{Kx}'| > \frac{dx}{2}\right)\boldsymbol{then}\ Test\ ((\sqrt{\left(|s_{Kx}'| - \frac{dx}{2}\right)^2 + |s_{Kz}'|^2} < r_K) \\ &\boldsymbol{if}\left(|s_{Ky}'| > \frac{dy}{2}\right)\boldsymbol{then}\ Test\ ((\sqrt{\left(|s_{Ky}'| - \frac{dy}{2}\right)^2 + |s_{Kz}'|^2} < r_K) \end{aligned} \tag{7.15}$$

Pro shrnutí si zopakujeme stručně postup testu kolize koule s plochou. Nejprve aplikujeme na kouli Backtransforming pomocí vztahu (7.12). Dále už jen postupně testujeme podmínky z (7.13), (7.14) a (7.15). Při prvním selhání podmínky $Test$ víme, že koule s plochou nekoliduje. V opačném případě, při všech pozitivních $Test$ech kolize existuje.

Výhodou Backtransformingu v tomto případě je zpětná transformace detekovaného bodu kolize. Pokud při hledání testu kolize vypočítáme přibližný bod kolize v systému ideální plochy, aplikujeme na tento bod zpětný Backtransforming (dle vztahu (5.16)) a dostaneme jeho přesnou polohu v reálném prostoru.

7.4 Nalezení času kolize

Čas kolize lze definovat jako okamžik, kdy dvě tělesa mají mezi sebou průnik ve formě jediného společného bodu (*Kolizní bod*). Takovýto bod lze v určitých případech vypočítat analytickou cestou za pomoci řešení rovnic. Ovšem většinou je nutné najít čas kolize obecnou cestou. V našem případě lze využít metody půlení intervalů definovanou v dřívějších kapitolách. Analytickou cestou dosáhneme výpočtu přesné pozice kolizního bodu, ale obecnou metodou půlení intervalů dosáhneme pouze přibližnou polohu kolizního bodu (dle definované přesnosti), což nám do celého systému vnořuje nepřesnosti, se kterými je nutno počítat.

Metoda půlení intervalů je závislá na evaluační funkci času, která vrací přesnost daného výpočtu v aktuálním kroku půlení intervalu. Nechť tato evaluační funkce je označena jako $f_E(t)$, pro naše využití je definována jako

$$f_E(t) = f_P(f_K(f_S(t))), \tag{7.16}$$

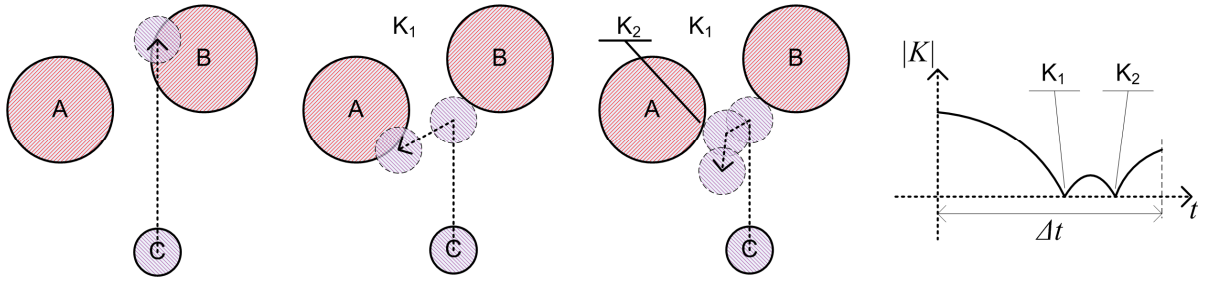
kde f_S vyjadřuje pohybovou funkci testovaných těles. Funkce f_K testuje kolizi v bodech dané f_S, zde se využívá testu kolizí dle těles (např. koule vs. koule, koule vs. plocha). Funkce f_P vypočítává přesnost kolize, ta je kupříkladu vzdálenost objektů mezi sebou (kladná hodnota přesnosti) nebo hloubka průniku (záporná hodnota přesnosti). Kladnost či zápornost výsledné funkce $f_E(t)$ určuje, jakým směrem se bude interval v příštím kroku půlit. Cyklicky tedy řešíme půlení intervalu dle výsledků $f_E(t)$ až dosáhneme požadované přesnosti. Musíme si ovšem dát pozor na to, aby metoda půlení intervalů neskončila v záporné hodnotě přesnosti. Záporná hodnota totiž určuje hloubku průniku, nemůžeme tedy najít bod kolize v objemovém průniku těles. Tento stav by mohl mít v určitých situacích za následek splývání těles v budoucích simulačních krocích, což je dle fyzikálních zákonů nemožné. Tento problém si ještě v textu popíšeme, protože je to jeden z častých problémů fyzikální simulace rigidních těles.

7.5 Multikolize

Mějme množinu těles lokální kolize T_V a mějme délku simulačního kroku Δt. Existuje stav, kdy koliduje alespoň jedno těleso z T_V vícekrát v časovém rozmezí Δt, tedy s několika tělesy najednou. Takovéto situaci říkáme *Multikolize*. Čím větší časové rozmezí Δt, tím větší pravděpodobnost multikolize. Tuto situaci lze vidět na obr. 7.14c. Graf na obr. 7.14d zobrazuje časy kolizí funkcí

vzdálenosti kolize $|K|$ stavu obr. 7.14c, kde simulační scéna obsahuje dvě statická tělesa A, B a jedno dynamické těleso pohybující se ve směru šipky. Rozložení na obr. 7.14a znázorňuje, jak by se simulovalo bez detekce kolizí. Na obr. 7.14b je zobrazen stav při standardnímu řešení kolize bez využití problematiky multikolize.

obr. 7.14a	*obr. 7.14b*	*obr. 7.14c*	*obr. 7.14d*



Dalším situací multikolize je když tělesa které v simulačním kroku Δt kolizi neměli, kolizi postupně dostanou následkem dřívější kolize. Jinými slovy řečeno, když jedno těleso svým nárazem ovlivní druhé, to může následkem narazit do třetího. To vše v jednom simulačním kroku Δt.

Tento problém lze jednoduše vyřešit, ale nejprve si definujme některé použité funkce. Mějme statické těleso S_j patřící do množiny statických těles S v množině lokální kolize T_V a dynamické těleso D_i patřící do množiny dynamických těles D v množině lokální kolize T_V. Nechť $fs_i(t)$ je funkce pohybu dynamického tělesa D_i dle fyzikálních vztahů a jejím parametrem je simulační čas. Pojmenujme ji tedy funkcí skoku i-tého dynamického tělesa z množiny lokální kolize T_V. V této množině se také vyskytují statická tělesa, tedy musí platit

$$(\forall D_i)\big((D_i \in T_V) \wedge (D_i \in D)\big) \wedge \big(\forall S_j\big)\Big(\big(S_j \in T_V\big) \wedge \big(S_j \in S\big)\Big) \wedge \big(T_V = (D \cup S)\big). \quad \textbf{(7.17)}$$

Nechť $f_C(P_D, P_T)$ je funkce detekce času kolize mezi tělesy, kde P_D je dynamické těleso z množiny D, P_T je těleso z množiny T_V a P_D není stejné těleso jako P_T, tedy platí

$$(\exists P_D)(P_D \in D) \wedge (\exists P_T)(P_T \in T_V) \wedge (P_D \neq P_T)\,. \quad \textbf{(7.18)}$$

Funkce $f_C(P_D, P_T)$ vrací kolizní strukturu K_k, která obsahuje informace o kolizi (jako např. čas kolize). Definujme ještě funkci $fk(K_k)$, která řeší kolizi objektů K_k, kupříkladu vypočítá nové vektory směru zapříčiněné odrazem (kolizí). Kolize K_k je prvkem množiny K, což je množina kolizí. Nyní si algoritmus řešení multikolizí popíšeme pomocí pseudo-matematického jazyka (alg. 7.3).

alg. 7.3, řešení multikolizí

$\boldsymbol{repeat}$ {

$\boldsymbol{foreach}(D_i \boldsymbol{from}\ D)\ fs_i(\Delta t)$ // Proveď skok všech dynamických těles podle hodnoty Δt.

$K = \left\{ f_C\left((\forall d \in D), \left(\forall w \in (T_V \setminus \{d\})\right)\right)\right\}$ // Nalezni množinu všech kolizí.

$\boldsymbol{if}(K \neq \emptyset)\ \boldsymbol{then}$ {

$K_k = MinCollisionTime(K)$ // Nalezni kolizi s nejmenším časem.

$t_{min} = GetCollisionTime(K_k)$ // Vytáhni z kolizní struktury čas kolize.

$\boldsymbol{foreach}(D_i \boldsymbol{from}\ D)\ fs_i\left(-(\Delta t - t_{min})\right)$ // Reverzní skok do času nejmenší kolize.

$fk(K_k)$ // Vyřeš kolizi s nejmenším časem.

$\Delta t = \Delta t - t_{min}$ // Posuňme čas pro další skok.

}

}$\boldsymbol{until}(K \equiv \emptyset)$

Slovně by se dal tento algoritmus řešení multikolize popsat asi jako cyklické řešení časově nejmladších kolizí. Po vyřešení takovéto kolize provedeme skok pro zbytek simulačního časového kroku Δt a znovu testujeme existenci kolize. Při existence opět nalezneme časově nejmladší a tu vyřešíme. Takto pokračujeme až do stavu, kdy žádná kolize neexistuje. Mechanismus tohoto inkrementálního řešení kolizí na množině lokální kolize je sice vcelku pomalý, ale nutný z důvodů změny stavů těles po vyřešení kolize.

Algoritmus je samozřejmě možné optimalizovat, to si ale ukážeme až v konkrétní implementaci. Funkci tohoto algoritmu si můžeme jednoduše představit jako posloupnost na obr. 7.14a, obr. 7.14b a obr. 7.14c. Pomocí tohoto algoritmu docílíme přesné detekce a řešení kolize v delším časovém úseku Δt. Mnoho systému pro simulaci fyziky tuto situaci multikolize vůbec neřeší, proto se stávají vcelku nepřesné. Jak je vidět, algoritmus je velmi přesný, protože vypočítává změny v pohybu těles dle kolize přesně v čas, kdy nastanou.

Existuje zde ovšem problém zacyklení algoritmu, který není zapříčiněn chybou algoritmu, spíše určitým stavem těles. Pokud máme například mnoho dynamických těles v těsné blízkosti mezi sebou, vzniká mnoho interakcí mezi tělesy při řešení multikolize. Můžeme se tak dostat do stavu, kdy čas další kolize t_{min} dle algoritmu bude mít hodnotu blížící se nebo rovnu nule. V tom případě je zapotřebí vyřešit nekonečně mnoho kolizí, což zapříčiní zacyklení. Tuto skutečnost si můžeme prakticky představit také jako ležící těleso na podlaze při klasické gravitační síle, kdy je těleso táhnuto svou váhou. Ležící těleso totiž nekonečněkrát koliduje s podlahou a tím dojde k zacyklení. Tato nežádoucí stav lze ale odstranit zavedením polohových stavů tělesa, kde stav určuje například pohyb v prostoru, valení na podlaze nebo stav stání (těleso leží na podlaze). V těchto konkrétních situacích můžeme s jistotou určit co je a co není kolize a také stav tělesa nám napomůže k určení jeho pohybových rovnic. Problematiku *Stavů tělesa* si popíšeme v příštích kapitolách.

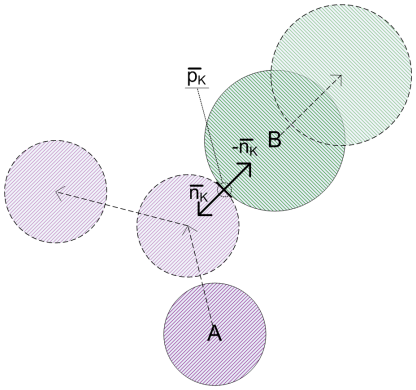
Jsou zde ještě jiné metody jak ochránit systém od zacyklení multikolizního algoritmu. A to za pomoci určení maximálního počtu interakcí algoritmu řešení multikolize, podobně jak je tomu v algoritmu metody půlení intervalů. Toto řešení nám ale může situaci i znepříjemnit, protože je možné dosáhnout stavu kdy tělesa při nedokončeném řešení multikolize začnou mít společný průnik. Problém nastane, až se bude v dalším simulačním kroku testovat kolize takovýchto dvou těles se společným průnikem. Nebudeme totiž moci metodou půlení intervalů vypočítat bod kolize, jelikož časový interval pro výpočet bude mít vždy společný průnik. Metoda půlení intervalů nám tedy nalezne čas kolize v nule a to nám zapříčiní zacyklení algoritmu řešení multikolize. Je totiž zapotřebí nekonečně mnoho interakcí k vyřešení takovéto kolize. Proto je nutné navrhnout ještě jiný způsob ochrany zacyklení multikolizního algoritmu, nebo ochrany pro detekci průniku těles.

7.6 Řešení kolizí

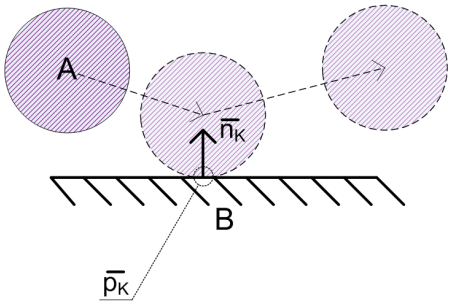
Nalezením bodu či času kolize pro diskrétní změnu ještě zcela nevyřeší problematiku kolizí. Dalším netriviálním problémem kolizí je jejich řešení (zdroj viz (7), (14), (8)), tedy výpočet změny parametrů zapříčiněné srážkou s jiným tělesem. Jelikož máme tělesa rozdělena na statická a dynamická, je nutné rozdělovat řešení kolizí na typy. Detekci kolize používáme pouze pro dynamické těleso, které může mít srážku jak s dynamickým, tak se statickým tělesem. Prvním typem řešení kolize je tedy dynamické vs. dynamické těleso (obr. 7.15). Při takovémto typu dochází k přenosu energií a hybnosti a výsledné vyřešení této kolize je vcelku složitější než kolize druhého typu. Druhým typem řešení kolize je dynamické vs. statické těleso (obr. 7.16). Zde energii nepředáváme, ale např. jen tlumíme odraz dynamického tělesa dle parametrů obou těles. Řešení kolizí také závisí na aktuálním polohovém stavu tělesa, zda prostorem letí nebo je v poloze klidové (více viz kapitola 8.1 Polohový stav tělesa).

Nejprve bychom si měli určit klíčové parametry pro řešení kolize, definujme si je vcelku jako *Kolizní strukturu*. Jelikož kolize se týká dvou těles, struktura tedy musí obsahovat znalosti, nebo alespoň odkazy na obě tyto tělesa a jejich fyzikální stav (rychlost, moment setrvačnosti, hmotný střed atd.). Dalším důležitým parametrem kolizní struktury je souřadnice bodu kolize $\overrightarrow{p_K}$, která určuje společný bod těchto těles. Posledním potřebným parametrem je normálový vektor bodu kolize $\overrightarrow{n_K}$. Ten lze definovat jako jednotkový vektor, určený směrem či úhlem srážky těles. Jeho směr je také dán pořadím kolidujících těles. Pokud normálový vektor bodu kolize z tělesa A do B je $\overrightarrow{n_K}$, pak normálový vektor bodu kolize z tělesa B do A je roven $-\overrightarrow{n_K}$ (vektorovému otočení $\overrightarrow{n_K}$), stejně jak je znázorněno na obr. 7.15 a obr. 7.16.

obr. 7.15, řešení kolize s dynamickým tělesem



obr. 7.16, řešení kolize se statickým tělesem



Jelikož potřebnou strukturu detekované kolize již máme popsánu, můžeme si tedy ukázat, jak lze jednotlivé typy kolizí těles řešit. Existuje také různý pohled na řešení takovýchto situací, řešit kolizi lze pomocí výpočtu jednoduchého odrazu bez rotací, který je matematicky i výpočetně nenáročný, ovšem nepopisuje přesné chování fyzikálního tělesa. Pro takovéto je nutné využít hlubších znalosti fyziky, např. momentů setrvačnosti, impuls síly atd. Základní roli zde hraje hmotný střed tělesa a bod kolize, který určuje páku pro výpočet impulsu síly. Přesný výpočet odrazu ovšem překračuje rámec této diplomové práce, popíšeme si tedy pouze jednotlivé typy jednoduchého řešení bez rotací a momentu setrvačnosti tělesa. Tyto typy řešení kolize jsou také implementovány pro kulatá tělesa v cílové knihovně.

7.6.1 Řešení kolize dynamické koule se statickým tělesem

Tuto situaci popisuje obr. 7.16, kde dynamické těleso koule A narazí do statického tělesa plochy B v bodu $\overrightarrow{p_K}$. Jednoduchý odraz (bez rotací) lze vypočítat pomocí základního zákona odrazu, tedy úhel odrazu je roven úhlu dopadu (zdroj viz (7)). Pro výsledný vektor rychlosti $\overrightarrow{v_{od}}$ tělesa A tedy platí

$$\overrightarrow{v_{od}} = 2\overrightarrow{n_K}\left(\overrightarrow{v_p} \cdot \overrightarrow{n_K}\right) + \overrightarrow{v_p}, \tag{7.19}$$

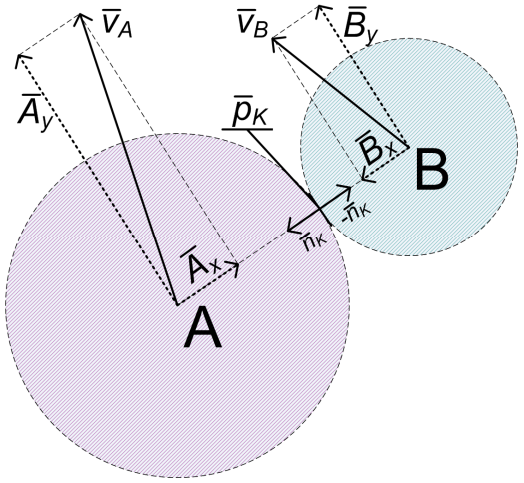
kde $\overrightarrow{n_K}$ je normálový vektor kolize z A do B a $\overrightarrow{v_p}$ je vektor rychlosti tělesa A v bodě kolize $\overrightarrow{p_K}$.

7.6.2 Řešení kolize dynamických koulí mezi sebou

Tuto situaci popisuje obr. 7.15, kde dynamické těleso koule A narazí do dynamického tělesa koule B v klidové poloze, a to v bodu $\overrightarrow{p_K}$. Zde je využito vztahů hybnosti (6.11) a kinetické energie (6.12) hmotného bodu, respektive zákona zachování hybnosti a zákona zachování energie. Pomocí těchto rovnic můžeme vyjádřit výpočet nových rychlostí dvou dynamických těles po srážce. Tento postup byl převzat ze zdroje (7) a byl mnou ovšem poupraven (optimalizován).

Základem pro vyřešení takovéto kolize je kolizní plocha, která je dána výřezem středů těles přes kolizní bod $\overrightarrow{p_K}$. Mějme tedy dynamické tělesa koule A a B o hmotnosti m_A a m_B, které kolidují v bodě $\overrightarrow{p_K}$ při vektorech rychlosti $\overrightarrow{v_A}$ a $\overrightarrow{v_B}$. Tento kolizní bod $\overrightarrow{p_K}$ má normálový vektor $\overrightarrow{n_K}$ (z tělesa A

obr. 7.17, řešení kolize dvou dynamických koulí



do B) a $-\overrightarrow{n_K}$ (z tělesa B do A). Takovouto situaci lze vidět na obr. 7.17. Nejprve je nutné vypočítat průmětny vektoru rychlosti $\overrightarrow{v_A}$ pro relativní osu danou středy koulí. Vektory průmětny $\overrightarrow{A_x}$ a $\overrightarrow{A_y}$ lze vypočítat pomocí vztahů

$$\overrightarrow{A_x} = -\overrightarrow{n_K}\left((-\overrightarrow{n_K}) \cdot \overrightarrow{v_A}\right), \qquad \textbf{(7.20)}$$

$$\overrightarrow{A_y} = \overrightarrow{v_A} - \overrightarrow{A_x}. \qquad \textbf{(7.21)}$$

Podobně také i pro průmětny vektoru rychlosti $\overrightarrow{v_B}$ platí

$$\overrightarrow{B_x} = \overrightarrow{n_K}(\overrightarrow{n_K} \cdot \overrightarrow{v_B}), \qquad \textbf{(7.22)}$$

$$\overrightarrow{B_y} = \overrightarrow{v_B} - \overrightarrow{B_x}. \qquad \textbf{(7.23)}$$

Nyní již máme dostatek potřebných veličin na samotný výpočet nových vektorů rychlostí, které jsou odvozeny zejména ze zákonu zachování hybnosti. Pro výsledné vektory $\overrightarrow{v_A}'$ a $\overrightarrow{v_B}'$ tedy platí

$$\overrightarrow{v_A}' = \frac{\left((\overrightarrow{A_x} \cdot m_A) + (\overrightarrow{B_x} \cdot m_B)\right) - m_B(\overrightarrow{A_x} - \overrightarrow{B_x})}{m_A + m_B} + \overrightarrow{A_y}, \qquad \textbf{(7.24)}$$

$$\overrightarrow{v_B}' = \frac{\left((\overrightarrow{A_x} \cdot m_A) + (\overrightarrow{B_x} \cdot m_B)\right) - m_A(\overrightarrow{B_x} - \overrightarrow{A_x})}{m_A + m_B} + \overrightarrow{B_y}. \qquad \textbf{(7.25)}$$

8 Problémy simulace těles

Reálný svět, jak ho známe, si pro naše účely idealizujeme, což u svých výhod obsahuje i podstatné nevýhody. V této kapitole se tedy zaměříme na problémy modelování tělesa a jeho simulace. Nejprve si něco povíme o polohovém stavu tělesa, který je dán neobecností fyzikálního chování dle simulačního modelu.

8.1 Polohový stav tělesa

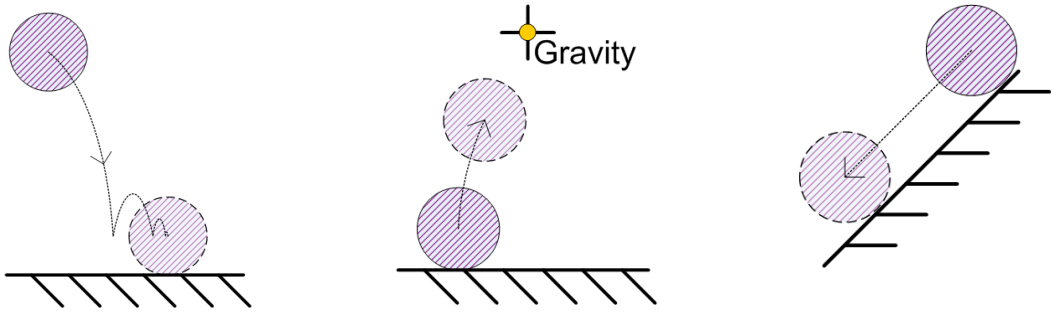
Jelikož je nereálné simulovat těleso na atomární úrovni, jako skupinu elementárních částic, vytváříme si tedy zjednodušenou verzi ve formě modelu tělesa. To nám poskytuje mnoho výhod, ale také i nevýhod. Výhodou je zejména aplikace fyzikálních zákonů na těleso jako celek, ovšem tyto zákony jsou určeny pouze pro speciální, separované stavy či chování. Za takové stavy můžeme považovat např. let hmotného tělesa nebo válení. Nevýhodou tohoto modelu je tedy komplikovanější popis chování dle stavu tělesa. Stavy jsou určeny zejména kolizí, např. pokud těleso nepřetržitě koliduje s jiným tělesem ve formě stejného bodu, či stejné roviny, jeho stav přechází na klidový. Pokud nebudeme dodržovat chování dle stavu, můžeme v určitých případech porušovat základní fyzikální zákony našeho světa. Kupříkladu může dojít ke sloučení objemů těles (těleso proletí jiným). Proto je nutné se touto problematikou zabývat, abychom mohli věrohodně popsat chování těles ve srovnání s realitou a zlepšit tak výsledný dojem ze simulace.

Nejprve si definujeme základní polohové stavy těles, mezi které patří let prostorem, válení či klouzání a klidová poloha, pro kterou lze také definovat problém závislosti na jiném tělesu. Popišme si tedy tyto stavy.

8.1.1 Stav letu

Mějme těleso v prostoru, které nemá žádnou kolizi s jiným tělesem. Takové těleso je ve *Stavu letu*. Pro tento stav platí fyzikální zákony ve formě pohybových rovnic (viz kapitola 6.3 Pohyb hmotného bodu a kapitola 6.4 Energie a hybnost). Při kolizi s jiným tělesem může dojít k odrazu, kde je nadále ve stavu letu. Ovšem může dojít k situaci, kdy po vyřešení kolize (výpočtu odrazu) opět dojde ke kolizi ve stejném bodě a rozdíl časů mezi takovýmito kolizemi je nepatrně malá. Dojde tedy ke změně stavu tělesa na *stav klidový*. Tuto situaci si lze představit jako padající kuličku na podlahu, která se dostává do stavu ležící na podlaze v klidu (viz obr. 8.1). V tomto novém stavu se už těleso chová dle jiných fyzikálních vztahů nežli při letu (viz kapitola 8.1.2 Stav klidu a závislosti).

obr. 8.1, stav letu→stav klidu *obr. 8.2, stav klidu→stav letu* *obr. 8.3, stav klidu→stav valení*



8.1.2 Stav klidu a závislosti

Stav klidu nám vyjadřuje stav tělesa neměnící svou polohu a závisející na tělese, na kterém leží. Stav klidu se také indikuje pomocí konstantní množiny kolizních bodů. Pokud totiž leží jedno těleso na druhém, obsahují mezi sebou konečný počet společných bodů na jejich povrchu. Chování takového tělesa je v podstatě jednoduché, ovšem složitější je určení přechodu do jiného stavu, které může být zapříčiněno například událostí.

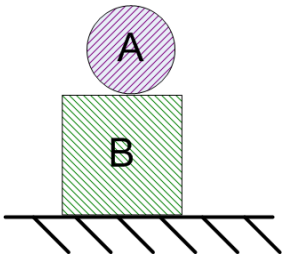
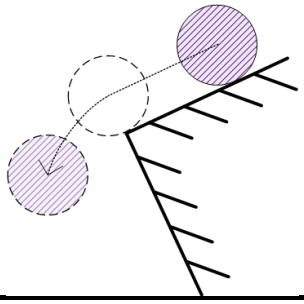
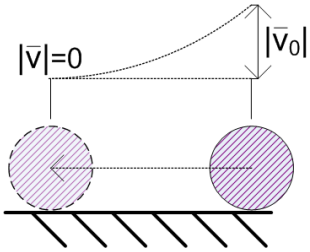
Pokud je těleso v klidovém stavu a při výpočtu pohybu dle pohybových rovnic má kolizi s jiným tělesem než na kterém leží či nemá kolizi vůbec, těleso se dostává do *Stavu letu*. Přechod ze stavu klidu do stavu letu si lze představit jako ležící těleso na statické podlaze při klasické gravitační síle (gravitace na povrchu země), kde nad tělesem vznikne jiná fyzikální síla (např. magnetická), která přitahuje ležící těleso (viz obr. 8.2). Při takovém pohybu může těleso narazit i do jiného. Je nutné podotknout, že takovýto převod stavu je zapříčiněn událostí změny fyzikálních sil. Tuto znalost lze využít při přepočtu stavů všech těles, kterou nám indikuje událost systému.

Pokud je těleso v klidovém stavu a při výpočtu pohybu dle jeho stavových pohybových rovnic má nekolmou kolizi (normála kolize není rovnoběžná s vektorem rychlosti pohybu) s tělesem na kterým leží, dostává se těleso do stavu *Válení či klouzání* (viz kapitola 8.1.3 Stav válení či klouzání). Takovouto situaci si lze představit jako těleso ležící na šikmé ploše v klidu při klasické gravitační síle, kde se těleso začne válet nebo klouzat po šikmé ploše (viz obr. 8.3). Změna stavu také může být zapříčiněna událostí systému (např. gravitačních sil).

Zde je také nutné definovat *Stav závislosti*, a to jako speciální stav klidu, který je dán závislostí na jiném tělese. Tuto situaci si můžeme představit jako tělesa ležící na sobě a zároveň na statické podlaze (např. komín z krychlí). V takovém stavu je těleso A závislé na jiném tělese B, na kterém leží (viz obr. 8.4). Změnu parametrů či stavu tělesa B nám zapříčiní událost pro přepočítání stavu tělesa A. Situaci si můžeme představit jako komín krychlí, do kterého narazí jiné těleso a komín se začne bortit. Tělesa závislá na tělesech odhozených kolizí je nutné přepočítat pro jejich nové stavy. Závislost tělesa A na tělese B je ale oboustranná, jelikož těleso B je ovlivňováno např. hmotností

tělesa *A*. Proto je nutné se problémem stavové závislosti těles věnovat více, ovšem tohle není tak zásadním cílem této práce.

Stav klidu lze také využít při optimalizaci detekce kolize těles. Jelikož těleso v tomto stavu nemá rychlost, nemá ani pohybovou energii a nemůže způsobit kolizi. Můžeme k němu tedy při detekci lokální kolize přistupovat jako ke statickému tělesu, čímž omezíme jeho test kolize se všemi ostatními dynamickými a statickými tělesy.

obr. 8.4, závislost těles	*obr. 8.5, stav válení→stav letu*	*obr. 8.6, stav válení→stav klidu*
		

8.1.3 Stav válení či klouzání

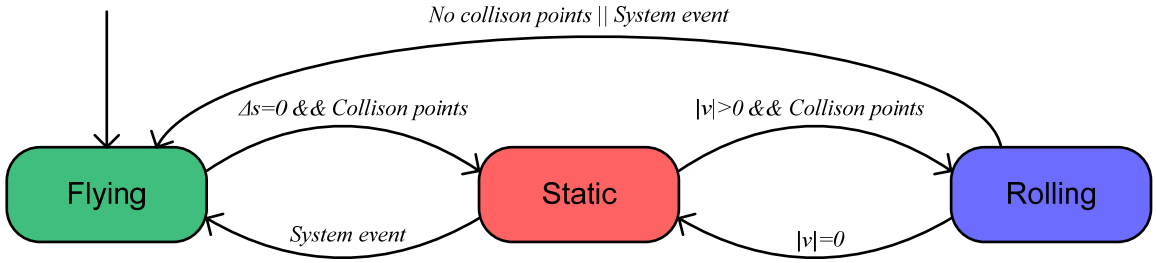
Stav válení či klouzání lze definovat jako speciální stav mezi stavem letu a klidu. Při tomto stavu je těleso v pohybu ale i přesto má množinu konečného počtu společných bodů kolize. Stav válení nebo klouzání je zapříčiněn geometrickými vlastnostmi tělesa. Kupříkladu koule se bude válet, kdežto krychle se může klouzat. Při takovémto stavu se pohyb těles určuje jinými fyzikálními rovnicemi nežli u stavu letu, zde se totiž počítá s třením.

Pokud je těleso ve stavu válení (či klouzání) a výpočet pohybu dle pohybových rovnic nám nalezne kolizi v jiném tělese, než na kterém se válíme, nebo nenalezne kolizi vůbec, pak se stav tělesa mění na let. Tato situace může být zapříčiněna buď událostí systému (např. přidání fyzikální síly), nebo opuštěním dráhy povrchu tělesa, po kterém se válí (viz obr. 8.5).

Pokud je těleso ve stavu válení (či klouzání) a velikost rychlosti pohybu dle válení je nulová, nebo výpočet pohybu dle pohybových rovnic nám nalezne kolizi s tělesem, na kterém se válí, kde vektor rychlosti pohybu je rovnoběžný s vektorem normály kolize, pak se stav tělesa mění na klidový. Tuto situaci si lze představit jako zastavení válení tělesa zapříčiněné třením (viz obr. 8.6).

Zjednodušený model stavů tělesa lze vidět na obr. 8.7 v reprezentaci stavového automatu. Simulace a řešení stavů těles je netriviální záležitostí a je to také hlavním nevýhodou tohoto typu modelování těles. Poskytuje nám ovšem určitou pozitivní vlastnost, kde stav tělesa identifikuje jeho speciální pohyb, pro které známe elementární fyzikální vztahy. Výhodou tedy je nižší výpočetní náročnost ve speciálních stavech, není totiž potřeba pro všechny tělesa vypočítat všechny druhy pohybu, ale pouze jen speciální pro daný stav.

obr. 8.7, zjednodušený stavový automat tělesa během simulace



8.2 Geometrie tělesa

Problémem simulace těles, co se týče geometrického povrchu, je jejich obecný popis. Mějme speciální tělesa, nazvěme je primitivní, která svými ideálními parametry ulehčují simulaci a znalost chování. Mezi takovéto primitivní tělesa patří koule, kvádr, válec a jiné. Tyto ideální tělesa nám ovšem nepopíší jakýkoli obecný povrch. Existují tedy určité metodiky, jak se tomuto obecnému povrchu přiblížit za pomocí primitivních těles. Obecného geometrického povrch lze také dosáhnout pomocí spojení skupiny trojúhelníkových polygonů, které tvoří celkový objem tělesa. Nyní si tyto dvě metodiky stručně popišme.

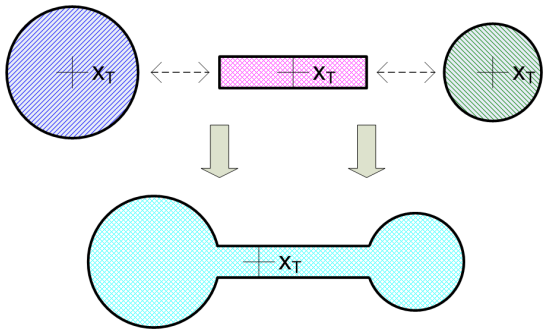
8.2.1 Skládání pomocí primitivních těles

Mějme množinu primitivních těles, kde každé nám představuje ideální tvar daný konečným počtem geometrických parametrů. Takovéto tělesa dokážeme vcelku jednoduše simulovat a detekovat na nich kolize. Jelikož potřebujeme popsat tvar obecného tělesa, můžeme využít těchto primitivních těles pro jeho složení. Jedno primitivní těleso spojíme se druhým pomocí společných bodů, společné přímky či společné roviny. Je sice možné spojit primitiva pomocí společného objemu, ale to by nám narušilo fyzikální parametry výsledného tělesa, respektive výpočtu hmotného středu. Jestliže máme takto spojená tělesa, hmotný střed je dán dle vzorce (6.1) průměrnou hodnotou hmotného středu všech primitiv těleso tvořící. Výpočet hmotného středu tělesa složeného z primitiv majících průnik, by byl vcelku složitější. Tato metoda skládání těles není tak výhodná pouze z hlediska fyzikální simulace, ale zejména z hlediska detekce kolize, kde využíváme ideálního tvaru těchto primitiv a kolizi jednoduše nalezneme. V této práci jsme se již setkali s podobnou reprezentací obecného tělesa v kapitole 7.2.2 Sphere-Tree, kde metoda využívala složení obecného tělesa pouze z koulí. Pro ukázku lze na obr. 8.8 vidět jednoduché těleso složené z tří primitivních těles, kde bod x_T je hmotným středem každého z těles.

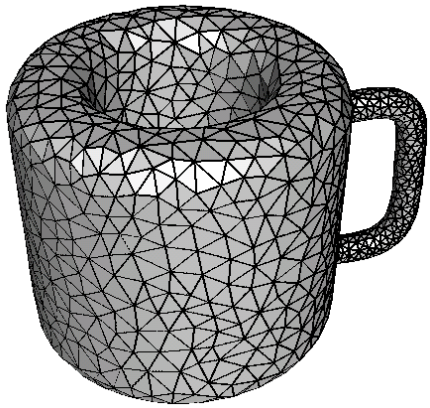
8.2.2 Vytvoření polygonální tělesa

Tato metoda využívá množinu orientovaných trojúhelníků (polygonů) pro popis povrchu obecného tělesa. U každého polygonu vytvořeného tělesa musí platit, že každá jejich strana je zároveň stranou vedlejšího polygonu. Jejich spojení tedy vytváří uzavřené těleso bez děr. Takto vytvořenému obecnému tělesu říkáme *Polygonální* (viz obr. 8.9, zdroj obrázku viz (15)). Dle obsahu a normálového vektoru každého polygonu tělesa lze vypočítat hmotný střed. Největší výhodou této metody je přesnost popisu obecného tělesa, problémem je spíše detekce kolize. Těleso je totiž natolik obecné, že je třeba testovat kolizi s každým polygonem vytvořeného tělesa. Test kolize s obecným trojúhelníkem je také vcelku náročnou operací, tím pádem tahle metoda není tak výhodná pro naše účely.

obr. 8.8, těleso vytvořené z primitiv



obr. 8.9, polygonální těleso



9 Návrh systému

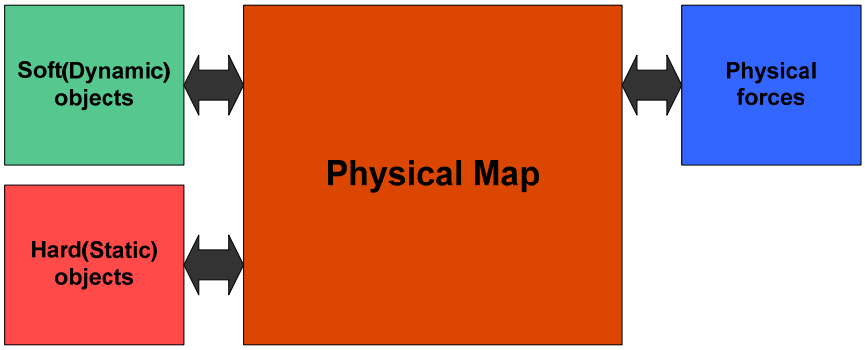
Návrh je nejdůležitějším aspektem každého rozsáhlejšího projektu. Proto je nutné se na něj zaměřit nejvíce a nepodceňovat ho. Dobrý návrh nám také pomůže vyřešit mnoho problémů, které se naskytnou při implementaci a při rozšíření. Naopak špatný návrh nám v průběhu implementace vytvoří mnoho problémů, které by mohli nakonec vést k neřešitelnému stavu. Je nutné si vytyčit vlastnosti celého systému, možné rozšíření a podle toho vytvořit adekvátní návrh, který nebude v rozporu s primárními cíli práce.

V následujících kapitolách si rozebereme návrh systému od základního rozčlenění modulů, až po jejich detailní architekturu.

9.1 Základní návrh modelu

Systém je rozdělen do několika různých částí. Na fyzikální prostor (*Physical map*), který je jádrem systému a řídí veškeré děje simulované scény včetně komunikace s uživatelem. Dále jsou to fyzikální tělesa rozdělená na dynamické (*Soft objects*) a statické (*Hard objects*). Nakonec samotné fyzikální síly (*Physical forces*), které ovlivňují dynamická tělesa. Hrubé rozvržení komunikace těchto samostatných celků knihovny lze vidět na obr. 9.1. Následovně si všechny části stručně popíšeme.

obr. 9.1, blokové schéma základního návrhu modelu



9.1.1 Fyzikální prostor

Fyzikální prostor nám reprezentuje jakousi mapu, neboli scénu, ve které se simulace zpracovává. Tento modul přebírá funkci fyzikálního solveru, definovaného v kapitole 3.3 Fyzikální solver. Aby mohly být fyzikální tělesa simulována, musí být všechna vloženy do tohoto prostoru. Tento prostor má konfigurovatelné vlastnosti, které by nám mohly dovolovat i simulaci jiného prostředí než v jakém žijeme. Například vzduchoprázdno, pod vodou či jiné. S takovýmto fyzikálním prostorem je úzce spjata i samotná simulace a její konfigurace.

Dalším úkolem této komponenty je řízení simulace, kolizí a komunikace s uživatelem neboli aplikací používající tuto knihovnu.

9.1.2 Fyzikální tělesa

Fyzikální tělesa lze rozdělit na dva typy, dynamické a statické. Dynamická tělesa jsou odvozena z reality. Můžeme s nimi například hýbat pomocí fyzikálních sil a určitým způsobem je tak ovlivňovat. Statické nejsou ovlivňovány fyzikálními silami ani dynamickými tělesy. Takovýto typ těles si lze představit jako pevná tělesa, např. podlaha, zeď, cokoli s čím nelze pohnout, pouze s tím může kolidovat dynamické těleso. V realitě neexistují statická tělesa, pouze dynamická, vše lze totiž ovlivnit. Ovšem z určitých režijních důvodů jsou zde definována statická.

Fyzikální tělesa reprezentují určité primitiva z reálného světa. Jednoduchá jako je koule, krychle, válec a složitější obecná tělesa. Tyto tělesa mají své fyzikální vlastnosti, např. rozměry, váhu, odrazovou konstantu atd.

Další vlastností systému je univerzálnost při rozšíření, jednoduché přidávání vlastních primitivních těles a jejich chování, použití už existujících těles a změna jejich chování či spojování těles do sebe. Rozšíření o další primitivní objekty je vcelku jednoduché a v žádném případě neovlivňuje běh systému.

Tělesa lze do fyzikálního prostoru přidávat dynamicky, tj. kdykoli během simulace lze vložit těleso do prostoru.

9.1.3 Fyzikální síly

Všechny dynamické objekty jsou ovlivňovány těmito silami, tedy např. dle druhého Newtonovým zákona.

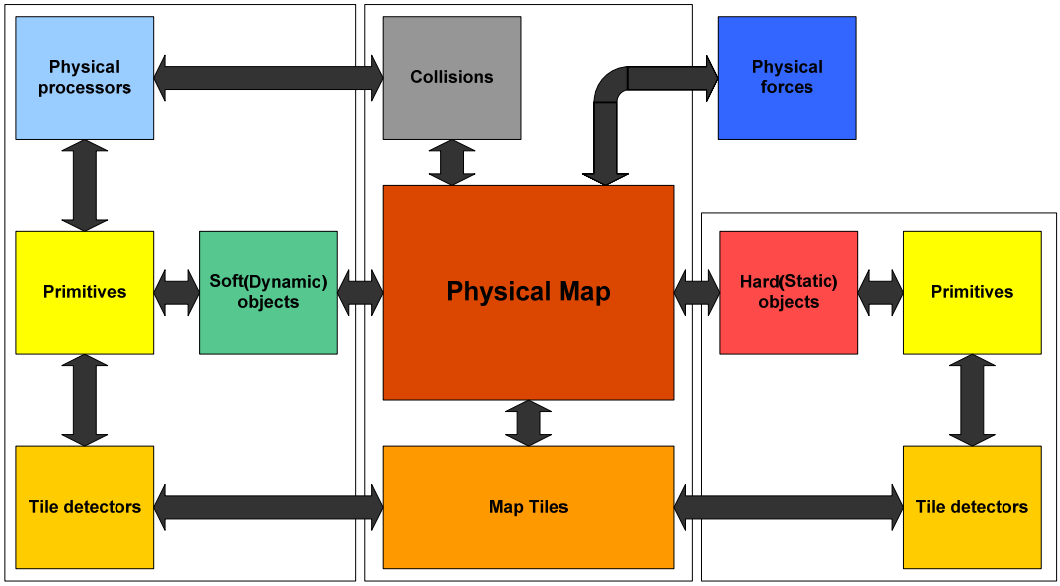
Systém musí přidávat a odebírat síly do prostoru dynamicky, a to kdykoli během simulace. Deklarace nových sil je implementačně jednoduché a nezávislé na prostoru či tělesech. Díky takovýmto vlastnostem můžeme vytvořit jakékoli gravitační či magnetické pole, exploze a jiné fyzikální efekty.

9.2 Detailní architektura modelu

Takováto komplexní fyzikální simulace včetně detekcí kolizí je vcelku složitý systém, je nutné jej tedy separovat na více částí. Mnohé také záleží na použitých metodách detekce kolize, řešení kolize, stavových automatech těles a samotných fyzikálních vztazích, ze kterých simulace vychází. Kompletní architekturu tohoto modelu lze vidět na obr. 9.2, ve kterém se rozčleňují moduly

základního návrhu na konkrétnější části a znázorňují jejich komunikace. V následujících kapitolách si tedy popíšeme funkci těchto částí.

obr. 9.2, blokové schéma architektury modelu



9.2.1 Architektura dynamického tělesa

Jelikož používáme mnou navrženou metodu detekce kolize pomocí podprostorových množin (Tile sets), musíme tedy mít modul, který zpracovává jeho funkce. Tento modul byl ještě separován na další dva, typu klient – server. Klienty zde pojmenujeme jako *Detektory podprostorů* (*Tile detectors*) a server jako *Prostorovou mapu* (*Map tiles*). Funkcí detektoru podprostoru je udržování množiny podprostorů, ve kterých těleso je a informování těchto podprostorů o existenci tělesa v nich pomocí prostorové mapy.

Detekce kolize a simulace fyziky je založena na *Primitivních tělesech* (*Primitives*). Je zde tedy modul *Fyzikální procesor* (*Physical processor*), který určuju konkrétní fyzikální chování primitivního tělesa. Jeho úkolem je také detekce a zpracovává kolize, má tedy přesnou znalost geometrie primitiva.

Instance primitiva, fyzikálního procesoru a detektoru podprostoru jsou obsluhovány univerzálním rozhraním *Dynamických těles* (*Soft objects*), které řídí stavový automat pohybu tělesa a komunikaci s jádrem modelu (*Physical map*). Strukturu a blokovou komunikaci tohoto dynamického tělesa lze vidět v ohraničené levé části obr. 9.2.

9.2.2 Architektura statického tělesa

Architektura statického tělesa je velmi podobná architektuře tělesa dynamického. Hlavním rozdílem je absence fyzikálního procesoru, který není u tohoto typu tělesa třeba. Také chování statického tělesa je poněkud odlišné od dynamického, a to zejména u detekování podprostorových množin, ke kterému

dochází pouze při inicializaci tělesa. Detekce podprostoru je také přesnější, oproti dynamickému, které přepočítává podprostory v každém simulačním kroku. Statické těleso pouze při inicializaci a díky větší přesnost zvýší i rychlost simulace. Znalost geometrie tělesa je také důležitým aspektem detekce přesných podprostorů, proto uvnitř statického tělesa nalezneme instanci jak detektoru podprostoru, tak primitivního tělesa. Blokové schéma statického tělesa lze vidět v ohraničené pravé části na obr. 9.2.

9.2.3 Architektura fyzikální mapy

Jak již bylo řečeno, *Fyzikální mapa* (*Physical map*) je jádrem celého systému. Jednou z jejich hlavních funkcí je detekce lokálních kolizí, obsahuje tedy prostorovou mapu jako server pro komunikaci s klienty těles (s detektory podprostoru). Fyzikální mapa je také řešitelem multikolizí a to pomocí *Struktur kolizí* (*Collisions*), kde je tento algoritmus popsán v kapitole 7.5 Multikolize. Tato komponenta je také hlavním rozhraním pro komunikaci s uživatelem (aplikací) a řízení simulace. Vnitřně zpracovává real-time i jiný druh simulace. Jedním z rozhraní komunikace je vkládání a odebírání těles do tohoto fyzikálního prostoru, takže v sobě obsahuje a řídí instance jednotlivých statických a dynamických těles. Tyto rozhraní a funkci fyzikální mapy si detailněji popíšeme v implementaci této komponenty, kde její blokové schéma lze vidět v ohraničené střední části na obr. 9.2.

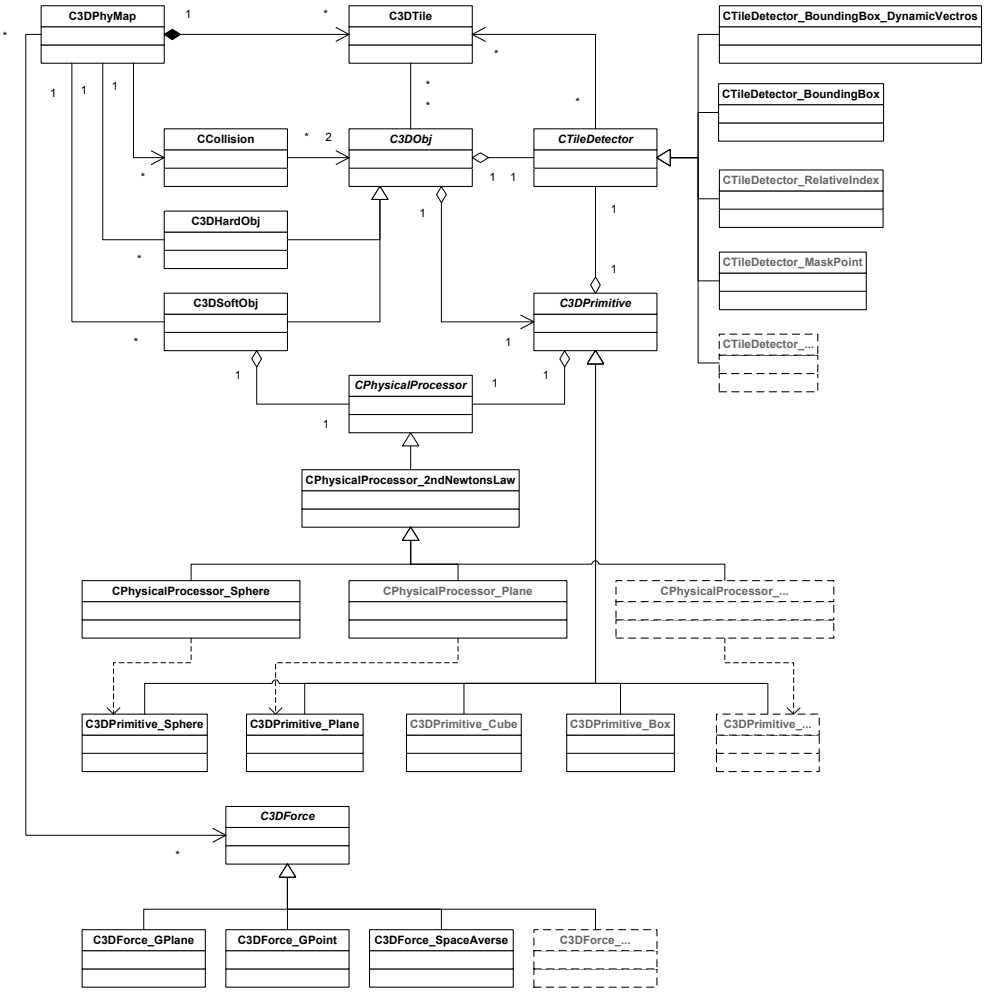
9.2.4 Architektura fyzikálních sil

Fyzikální síly (*Physical forces*) stále zůstávají samostatným modulem, jelikož jejich funkce není tak závislá na použitých metodách. Podobně jako instance těles jsou instance fyzikálních sil vkládány do fyzikální mapy, která řídí jejich komunikaci s dynamickými tělesy. Každá fyzikální síla totiž ovlivňuje každé dynamické těleso.

9.3 Objektový návrh modelu

V této kapitole se soustředíme na objektový návrh modelu systému, který si dále rozložíme. Tento cílový návrh lze vidět na obr. 9.3 v podobě standardního UML diagramu (zdroj viz (16)). Následně si stručně popíšeme jednotlivé třídy a jejich vztahy s ostatními. Konkrétní popsání včetně rozhraní bude k nahlédnutí v kapitole 10 Nástroje a forma implementace.

obr. 9.3, UML diagram simulačního modelu



Třída $C3DPhyMap$ zde vyjadřuje jádro systému a fyzikální mapu. Pouze jediná instante této třídy vytváří cílový simulační model a pomocí rozhraní této třídy můžeme simulaci konfigurovat. Jak lze na obrázku vidět, tato třída je majitelem třídy podprostorů, neboli prostorové mapy ($C3DTile$). $C3DPhyMap$ také udržuje seznamy všech statických těles ($C3DHardObj$), dynamických těles ($C3DSoftObj$) a fyzikálních sil ($C3DForce$). Jelikož jsme se v předchozích kapitolách dozvěděli, že modul fyzikálního prostoru řeší multikolize, obsahuje tato třída také struktury kolizí ($CCollision$) dané kolizemi dvojící těles.

Hlavní třídou tělesa je $C3DObj$, která definuje rozhraní pro komunikaci s fyzikální mapou, z této třídy dědí oba typy těles $C3DSoftObj$ a $C3DHardObj$. Oproti statickému tělesu, dynamické definuje speciální rozhraní pro detekci kolize a operace nad fyzikálním procesorem. $C3DObj$ ještě udržuje znalost geometrie tělesa za pomocí třídy primitivního objektu ($C3DPrimitive$). Komunikuje také s abstraktní třídou detektoru podprostoru ($CTileDetector$).

Abstraktní třída fyzikálního procesoru ($CPhysicalProcessor$) definuje rozhraní pro komunikaci s dynamickým tělesem a primitivním geometrickým objektem ($C3DPrimitive$). Tato

třída také definuje chování pohybu tělesa, detekuje a řeší kolize, je totiž hlavní třídou chování fyzikální simulace. Jako příklad zde máme třídu potomka, která implementuje chování fyziky dle Newtonových zákonů, z té dědí už konkrétní potomci závislí na geometrii tělesa. Jak lze na obrázku vidět, v systému si můžeme implementovat i jinou fyziku nežli je námi použitá Newtonova.

Třída primitivního tělesa ($C3DPrimitive$) je geometrickým znázorněním fyzického tělesa. Její potomci jsou už konkrétní primitivní tělesa jako například koule ($C3DPrimitive\_Sphere$) a plocha ($C3DPrimitive\_Plane$). $C3DPrimitive$ zde vyjadřuje rozhraní pro společnou komunikaci s $C3DObj$, $CPhysicalProcessor$ a $CTileDetector$.

Abstraktní třída detektoru podprostoru ($CTileDetector$) určuje komunikaci tělesa a jeho geometrie mezi prostorovou mapou danou třídou $C3DTile$. Každý potomek třídy detektoru je konkrétní metodou detekce podprostoru. Například je zde potomek využívající metody BoundingBox (viz kapitola 7.2.1 BoundingBox) pro detekci podprostorů s názvem $CTileDetector\_BoundingBox$. Další implementované metodiky si popíšeme v cílové implementaci této práce.

Poslední nepopsanou abstraktní třídou je třída fyzikálních sil ($C3DForce$), která definuje rozhraní komunikující s fyzikální mapou, jako fyzikální síly ovlivňující všechna dynamická tělesa v systému $C3DPhyMap$. Potomci $C3DForce$ určují konkrétní fyzikální sílu, kupříkladu plošná gravitace daná vektorem je implementována jako $C3DForce\_GPlane$.

Jak je z předchozího textu jasné, máme zde několik základních tříd definující rozhraní pro univerzální použití. Systém je tedy navrhnut tak, aby mohl být postupem času neustále vylepšován a takzvaně přihýbán k požadavkům uživatele tohoto systému. Navržení tohoto univerzálního modelu bylo docíleno pomocí předchozích zkušeností z ročníkového projektu.

10 Nástroje a forma implementace

Hlavním výsledným produktem této diplomové práce je implementovaný simulační systém ve formě platformě nezávislé C++ knihovny s pracovním názvem *PolyPhysics*. Kromě této knihovny bylo vytvořeno několik aplikací pro účely testování a navrhování simulačních scén. Veškeré implementované produkty této práce si popíšeme v následující kapitole a dále se soustředíme na implementaci cílové knihovny PolyPhysics.

10.1 Použité nástroje a související projekty

K implementaci této práce bylo využito programovacího jazyka C/C++ (zdroj viz (17)), .NET C++, XML, grafické knihovny OpenGL (Open Graphics Library), glEngine (OpenGL Engine), GLUT (OpenGL Utility Toolkit), generátoru dokumentace Doxygen a programovacího prostředí Microsoft Visual Studio 2005.

Během této diplomové práce, ročníkového a semestrálního projektu bylo vyvinuto několik projektů či aplikací, které napomohli k dovršení této práce. Nyní si je stručně popíšeme.

10.1.1 Knihovna glEngine

Tuto knihovnu jsem vytvořil za účelem grafické reprezentace simulované scény. Je to čistě C++ knihovna pro Windows fungující jako objektová nadstavba nad OpenGL. Její hlavní výhodou je vícekontextová reprezentace modelované scény, jinými slovy můžeme grafickou 3D scénu zobrazit ve více oknech, kde každé okno může mít jinou konfiguraci pohledu a zobrazení. Kupříkladu si takto vytvoříme pohled scény z několika směrů najednou, podobně jak je tomu v 3D CAD systémech (3D Computer Aided Design system). Pomocí takovéhoto zobrazení scény mohu přesněji validovat správnou funkci a chování modelu fyzikální simulace této práce.

Podobnou, ovšem platformě nezávislou knihovnou je zde GLUT, která je v našem případě oproti glEngine nevýhodná. Zejména kvůli absenci vícekontextovému renderování (obsaženo až v novějších verzích) a absenci samostatného renderovacího kroku. GLUT je ovšem použito při tvorbě testovacích aplikací knihovny PolyPhysics na nezávislé platformě.

10.1.2 Aplikace Map2XML

Tuto aplikaci jsem vytvořil za účelem vytváření fyzikální 3D scény pro simulaci ve formě souboru. Jinými slovy je to grafický editor fyzikální scény, který dokáže exportovat a importovat data ve formě XML dokumentu určitého formátu. Pomocí této aplikaci si můžeme navrhovat simulační scény, definovat vlastnosti jednotlivých těles a fyzikálních sil ve scéně, konfigurovat samotnou simulaci

knihovny PolyPhysics. Tato aplikace je naprogramována v C++ .NET 2.0 a využívá grafické knihovny glEngine pro zobrazení navržené scény. Jak již bylo řečeno, navrženou scénu lze importovat či exportovat do souboru XML. Fyzikální scéna v tomto souboru má jasně definovanou strukturu, kterou si teď zběžně popíšeme.

Nejprve si popišme základní bloky XML souboru.

```
<PolyPhysicsXml Version="1.0">
  <Map>
    <!-- blok konfigurace fyzikální mapy -->
  </Map>
  <Simulation>
    <!-- blok konfigurace fyzikální simulace -->
  </Simulation>
  <Primitives>
    <!-- blok těles fyzikální mapy -->
  </Primitives>
</PolyPhysicsXml>
```

Blok konfigurace fyzikální mapy by měl obsahovat dva parametry. Tím prvním je velikost simulační scény a druhým počet podprostorů v jednotlivých osách. Ukažme si to na příkladě.

```
  <Map>
    <Size X="10" Y="10" Z="10" />    <!-- velikosti fyzikální mapy -->
    <Tiles X="10" Y="10" Z="10" />   <!-- počet podprostoru v osách X,Y,Z -->
  </Map>
```

Blok konfigurace fyzikální simulace v sobě obsahuje parametr rychlosti simulace a zde také definujeme fyzikální síly, které ovlivňují dynamická tělesa v mapě. Ukažme si tedy na příkladu.

```
  <Simulation>
    <TimeSpeed F32="1" /> <!-- koeficient rychlosti simulace, F32 znamená float -->
    <!-- blok definice fyzikálních sil -->
    <Forces>
      <!-- bodová gravitační síla -->
      <GravityPoint UserID="18467">       <!-- identifikátor (číslo) síly -->
        <Translation X="5" Y="5" Z="5" /> <!-- pozice v prostoru -->
        <Gravity F32="50" />              <!-- konstanta síly gravitace -->
      </GravityPoint>
      <!-- plošná gravitační síla -->
      <GravityPlane UserID="29358">       <!-- identifikátor (číslo) síly -->
        <Translation X="5" Y="0" Z="5" /> <!-- pozice pouze pro grafickou reperzentaci -->
        <Gravity X="0" Y="-9,8" Z="0" />  <!-- vektor síly gravitace -->
      </GravityPlane>
    </Forces>
  </Simulation>
```

Blok těles fyzikální mapy v sobě obsahuje seznam těles, jejich fyzikální parametry a z jakých primitivních těles jsou tvořena. Popis všech parametrů je složitější, ukažme si vše na příkladu.

```
<Primitives>
  <!-- dynamické těleso -->
  <SoftObject UserID="26962">             <!-- identifikátor (číslo) tělesa -->
    <Translation X="5" Y="4" Z="1" />     <!-- pozice v prostoru -->
    <Rotation X="0" Y="0" Z="0" />        <!-- rotace tělesa -->
    <Bounce F32="1" />                    <!-- odrazová konstanta -->
    <Weight F32="1" />                    <!-- váha tělesa -->
    <InitialVelocity X="0" Y="0" Z="4" /> <!-- počáteční rychlost tělesa -->
    <!-- primitivní objekt typu koule -->
    <Sphere>
      <Radius F32="1" />                  <!-- poloměr koule -->
    </Sphere>
  </SoftObject>
  <!-- statické těleso -->
  <HardObject UserID="20537">             <!-- identifikátor (číslo) tělesa -->
    <Translation X="2" Y="7" Z="5" />     <!-- pozice v prostoru -->
    <Rotation X="1,5" Y="1" Z="1" />      <!-- rotace tělesa -->
    <Bounce F32="1" />                    <!-- pružnost či odrazová konstanta tělesa -->
    <!-- primitivní objekt typu obdelníkové plochy -->
    <Plane>
      <HalfSideX F32="1" />               <!-- poloviční velikost šířky plochy -->
      <HalfSideY F32="1,5" />             <!-- poloviční velikost výšky plochy -->
    </Plane>
  </HardObject>
</Primitives>
```

Takto vytvořený soubor dále můžeme použít v aplikaci PolyPhysics GLTest, kterou si následně popíšeme.

10.1.3 Aplikace GLTest

Tuto aplikaci jsem vytvořil za účelem testování fyzikální scény definovanou dle externího souboru ve formě XML vytvořeného v aplikaci Map2XML. Pomocí této aplikace mohu testovat vytvořené fyzikální scény a verifikovat tak průběh a chování fyzikální simulace knihovny PolyPhysics. GLTest je první aplikací využívající rozhraní simulace. Je naprogramována v C++ .NET 2.0 v kombinaci s glEngine a knihovnou PolyPhysics. Aplikace GLTest byla velkým pomocníkem při vývoji knihovny PolyPhysics a to zejména díky reprezentaci dějů probíhajících během simulace. Například nám graficky reprezentovala 3D simulovanou scénu včetně fyzikálních parametrů těles a kolizních bodů, řešených multikolizí a jiných statistických informací. Další užitečnou funkcí této aplikace je krokování simulace danou uživatelsky definovaným časovým krokem Δt, kde lze zkoumat chování a přesnost simulace danou velikostí Δt.

11 Knihovna PolyPhysics

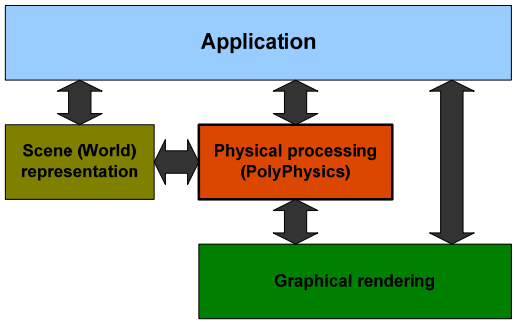
Hlavním produktem této diplomové práce je právě tato knihovna *PolyPhysics*, implementována pouze v programovacím jazyce C/C++ a je tedy platformě nezávislá. Nepoužívá žádných externích knihoven a obsahuje funkci fyzikálního solveru.

V této kapitole si nejprve povíme něco o celkové koncepci této knihovny a uživatelský návod, jak ji lze použít. Následně si zde popíšeme konkrétní vnitřní chování a komunikaci definovaných tříd mezi sebou. Jelikož projekt je napsán v jazyce C/C++, všechny ukázky programu budou také v tomto jazyce.

11.1 Koncepce a popis použití

Koncepce PolyPhysics spočívá v řízení scény dané aplikací a následné předání scény do modulu grafické reprezentace. Jednotlivé grafické tělesa jsou obalené do takzvané fyzikální obálky, která určuje pozici a rotaci při renderování v grafickém modulu. Funguje tedy jako nadstavba nad grafickým modulem, obsahuje totiž určitou znalost simulovaného světa. Blokové schéma tohoto slovního popisu lze vidět na obr. 11.1.

obr. 11.1, blokové schéma užití PolyPhysics



Knihovna PolyPhysics je navržena dle modelu definovaného v kapitole 9.3 Objektový návrh modelu. Obsahuje tedy přesné pojmenování a hierarchii tříd dle obr. 9.3. Máme zde několik základních tříd definující univerzální rozhraní pro komunikaci s ostatními. Mezi tyto základní třídy patří `C3DPhyMap` definující fyzikální mapu, `C3DObj` definující těleso rozdělené na `C3DSoftObj` pro dynamické a `C3DHardObj` pro statické, dále `C3DForce` definující fyzikální sílu, `CTileDetector` definující detektor podprostorů tělesa, `CPhysicalProcessor` definující fyzikální procesor (simulaci) a nakonec `C3DPrimitive` definující geometrii primitiva. Tato koncepce rozdělení základních tříd a rozhraní nám poskytuje obrovskou možnost rozšiřitelnosti této knihovny i na uživatelské úrovni, uživatel knihovny může vytvářet vlastní tělesa a síly, vlastní fyzikální zákony a chování i bez zdrojového kódu knihovny. Pouze vytvoří potomka z dané třídy a implementuje chování jeho rozhraní.

Nyní si zaveďme konvence spojené s pojmenováním entit ve zdrojových kódech a následně základní operace pro použití této knihovny.

11.1.1 Konvence rozhraní

Knihovna PolyPhysics je implementačně a modelově definovaná už jako druhá verze, tou první verzí byl ročníkový projekt. Každý objekt PolyPhysics před sebou tedy nosí namespace (pojmenování bloků v C++) $PP2$ (PolyPhysics verze 2).

Pojmenování tříd dědících ze základních tříd za sebou nosí postfix oddělený znakem podtržítka, který určuje konkrétní cílové použití. Pokud například máme základní třídu $C3DPrimitive$, název konkrétního použití pro kruh má tedy podobu $C3DPrimitive\_Sphere$.

Každá metoda rozhraní je definována třemi nebo dvěma částmi separovanýma znakem podtržítka. První část nazývanou prefix určuje kategorii metody. Druhá část určuje, čeho se tato metoda týká, tedy cílový objekt. Třetí část nazývanou postfix určuje operaci na cílový objekt. Pokud má metoda pouze dvě části (název metody obsahuje jedno podtržítko), druhá část vyjadřuje postfix, tedy operaci. Ukažme si toto rozdělení na příkladu.

```
PP2::C3DPhyMap::Sys_SoftObj_GetList(…);
PP2::C3DPhyMap::Sim_Init(…);
PP2::C3DPhyMap::Sim_Time_Get(…);
PP2::C3DObj::Geo_Translation_Get(…);
```

Kategorie jsou rozděleny do čtyř základních částí. Je to *Systém* (*Sys*), *Geografie* (*Geo*), *Fyzika* (*Phy*) a *Simulace* (*Sim*). Metody systému určují spíše operace nad formou implementace, operace identifikace či přístup na listy těles. Metody geografie určují operace nad znalostí prostoru a geometrie. Metody fyziky určují operace nad fyzikálními vlastnostmi objektů a sil. Poslední metody kategorie simulace určují operace a konfigurace samotné simulace.

11.1.2 Použití knihovny

Co se týče technické stránky, je zde nutné upozornit, že PolyPhysics je napsána jako single-threadová (používá pouze jedno vlákno procesu) knihovna. Veškeré konfigurace a operace musí být tedy volány pouze z tohoto jednoho threadu. Kupříkladu můžeme mít aplikaci používající tuto knihovnu, kde jeden thread bude zpracovávat fyzikální simulaci a druhý renderování simulované scény (typicky na multi-jádrovém procesoru). V jednodušším případě můžeme používat jeden thread jak na fyziku, tak na grafiku pomocí sekvenčního zpracování. Problematika threadů bude více jasná z konkrétních příkladů aplikace knihovny.

Jak již bylo řečeno v předchozích kapitolách, jádrem systému je třída $C3DPhyMap$, kde její instance vytváří cílový fyzikální solver. Nyní si tedy ukážeme jak tuto instanci vytvořit a inicializovat.

```
// deklarace pointeru na fyzikální mapu
PP2::C3DPhyMap *g_pPhyMap;

void CreatePhysicalMap()
{
  // vytvoření instance fyzikální mapy
  g_pPhyMap = new PP2::C3DPhyMap(
                        // float vektor velikosti mapy v metrech
                        F32POINT(10.0f,10.0f,10.0f),
                        // uint vektor počtu podprostorů na osy
                        UI32POINT(10,10,10),
                        // velikost paměti podprostorů (čistě pro optimalizaci)
                        TILE_MEMPOOL_SIZE);
  // inicializace simulace
  g_pPhyMap->Sim_Init();
}
```

Nejlépe před samotným cyklem simulace je vhodné vložit fyzikální tělesa a síly do mapy. Samozřejmě to můžeme provést i během simulace. Toto vkládání obstarávají metody systémové kategorie. Ukažme si příklad vložení dynamického tělesa s geometrií koule, statického tělesa s geometrií plochy a fyzikální sílu gravitační plochy dané vektorem.

```
void CreatePhysicalObjects()
{
  // vytvoření dynamického tělesa (koule uprostřed scény)
  PP2::C3DSoftObj *pSoftObj = new PP2::C3DSoftObj(
                    // vektor pozice v prostoru v metrech
                    &F32POINT(5.0f,5.0f,5.0f),
                    // vektor rotace tělesa v radiánech
                    &F32POINT(0.0f,0.0f,0.0f),
                    // definice fyzikálního stavu (váha a vektor počáteční rychlosti)
                    &PP2::CPhySoftState(1.0f,&F32POINT(0.0f,0.0f,0.0f)),
                    // vytvoření geometrie koule o poloměru 0.2
                    new PP2::C3DPrimitive_Sphere(0.2f) );
  // přenesení vytvořeného dynamického tělesa do fyzikální mapy
  g_pPhyMap->Sys_SoftObj_Add(&pSoftObj);

  // vytvoření statického tělesa (podlaha)
  PP2::C3DHardObj *pHardObj = new PP2::C3DHardObj(
                    // vektor pozice v prostoru v metrech
                    &F32POINT(5.0f,0.01f,5.0f),
                    // vektor rotace tělesa v radiánech
                    &F32POINT(1.5f*PI,0.0f,0.0f),
                    // definice fyzikálního stavu (zatím bez parametrů)
                    &PP2::CPhyHardState(),
                    // vytvoření geometrie plochy, kde poloviční
                    //   šířka je rovna 5.0f a poloviční výška je rovna 5.0f
                    new PP2::C3DPrimitive_Plane(5.0f,5.0f) );
  // přenesení vytvořeného statického tělesa do fyzikální mapy
  g_pPhyMap->Sys_HardObj_Add(&pHardObj);

  // vytvoření gravitační síly (zemská gravitace)
  PP2::C3DForce *pForce = new PP2::C3DForce_GPlane(
                    // vektor zrychlení gravitační plochy
                    &F32POINT(0.0f,-9.8f,0.0f) );
  // vložení fyzikální síly do mapy
  g_pPhyMap->Sys_Force_Add(pForce);
}
```

Nyní si ukážeme, jak by měl vypadat hlavní cyklus simulace. Na příkladě sekvenčně provádíme simulační krok a po něm vykreslíme simulovanou scénu.

```
void SimulationLoop()
{
  // prováděj cyklus, bRun je uživatelská proměnná
  while(bRun)
  {
    // volání metody real-time simulačního kroku
    g_pPhyMap->Sim_Step_Process();
    // simulační krok lze volat pro ne real-time simulaci, krok je dán velikostí dt
    //g_pPhyMap->Sim_Step_Process(dt);

    // vykreslení simulované scény
    RenderScene();
  }
}
```

RenderScene() je uživatelská funkce, která zjišťuje geografické vlastnosti těles ve fyzikální mapě a vykresluje je. Tunu funkci muže mít například následující podobu.

```
void RenderScene()
{
  PP2::C3DSoftObj *pSoft;
  // vytáhnutí listu dynamických těles z fyzikální mapy
  PP2::TSoftObjList *pSoftObjs = g_pPhyMap->Sys_SoftObj_GetList();
  // cyklus přes všechny prvky v pSoftObjs
  FOREACHp(pSoft,pSoftObjs);
    // funkce vykreslení tělesa
    GFX_Draw(
      pSoft->Geo_Translation_Get(),  // vektor pozice tělesa
      pSoft->Geo_Rotation_Get(),     // vektor rotace tělesa
      pSoft->Sys_Primitive_Get() );  // ukazatel na primitivní objekt
  _FOREACHp(pSoft,pSoftObjs);

  PP2::C3DSoftObj *pHard;
  // vytáhnutí listu statických těles z fyzikální mapy
  PP2::THardObjList *pHardObjs = g_pPhyMap->Sys_HardObj_GetList();
  // cyklus přes všechny prvky v pHardObjs
  FOREACHp(pHard,pHardObjs);
    // funkce vykreslení tělesa
    GFX_Draw(
      pHard->Geo_Translation_Get(),  // vektor pozice tělesa
      pHard->Geo_Rotation_Get(),     // vektor rotace tělesa
      pHard->Sys_Primitive_Get() );  // ukazatel na primitivní objekt
  _FOREACHp(pHard,pHardObjs);
}
```

Kombinace *FOREACHp* … *_FOREACHp* je makro definované v PolyPhysics jako cyklus přes všechny prvky v listu objektů. *GFX_Draw*() je uživatelsky definována funkce pro vykreslení grafické reprezentace tělesa. Nebudeme si zde ukazovat jak vykreslit 3D těleso, jelikož to není pro nás v tuto chvíli předmětné. Je ovšem nutné podotknout, že statická tělesa nemusíme neustále vykreslovat, jelikož jejich pozice ani rotace se nezmění, můžeme si je tedy při inicializaci připravit (podobně, jak je tomu u při vytváření listů v knihovně OpenGL).

Jak lze z předchozích příkladů vidět, použití knihovny PolyPhysics je velmi jednoduché. Za běhu simulace lze také mnoho vlastností fyzikální simulace měnit pomocí rozhraní třídy *C3DPhyMap*, ovšem všechny metody si zde popsat nemůžeme, jsou ale k nalezení v dokumentaci zdrojového kódu (viz Příloha 4.).

11.2 Vnitřní komunikace základních tříd

V této kapitole se zaměříme na rozhraní jednotlivých základních tříd. Přiblížíme si tak fungování knihovny PolyPhysics a ukážeme k jakým efektům lze díky univerzálnosti rozhraní dosáhnout. Popišme si tedy jednotlivě komunikační metody dle použitého blokového schématu z obr. 9.2.

11.2.1 Třída detektoru podprostorů

Úkolem této částečně abstraktní třídy $CTileDetector$ je komunikace s prostorem fyzikální mapy a udržování informace o podprostorech, ve kterých těleso leží a zároveň informovat podprostory mapy že v nich těleso leží. Instance třídy $CTileDetector$ je vytvářena v instanci primitivního tělesa, jelikož primitivní těleso určuje geometrii, určuje také metodiku a parametry detektoru podprostorů. Tato třída má vcelku jednoduchou sadu pěti virtuálních funkcí, které jsou definovány v souboru $polyTileDetectors.h$. Funkce $Init()$ slouží k inicializaci instance, ta je dána konkrétní implementaci potomka této třídy. Nejdůležitější funkcí je $New()$, která detekuje množinu podprostorů tělesa dle jeho pozice a informuje všechny podprostory o existenci tělesa v nich. Funkce $Get()$ vrací množinu podprostorů, ve kterých těleso leží (množinu vypočítanou v $New()$). Dále zde máme $Refresh()$, která by měla reinicializovat instanci dle změny stavu či rotace tělesa. Poslední funkcí této třídy je $Clear()$, která čistí existenci tělesa ve všech podprostorech vně ležící. Všechny tyto funkce jsou volány z instance tělesa $C3DObj$ (jak pro dynamické tak pro statické).

Při vytváření potomka této třídy je nejdůležitější implementace funkce $New()$, která vytváří komunikaci mezi prostorovou mapou a tělesem. Ve stávající knihovně PolyPhysics jsou implementovány zatím dvě metody detekce podprostorů, ale jsou již promyšleny další, jelikož každá metoda se hodí na jiný typ primitivního tělesa (kvůli rychlostním požadavkům). První metoda je založena na krabicovém obalování těles (popsané v kapitole 7.2.1 BoundingBox), kde detektor podprostoru $CTileDetector\_BoundingBox$ je určen statickou krabicovou obálkou. Je vhodná zejména u obalování primitiv koule. Druhá metoda je odvozena od první a spočívá v dynamickém určení krabicové obálky danou rohy tělesa, je vhodná např. pro primitiva plochy. Tato třída je pojmenována $CTileDetector\_BoundingBox\_DynamicVectros$. Zatím neimplementovaná metoda, fungující na principu seznamu relativních indexů podprostorů je $CTileDetector\_RelativeIndex$. Dále zde máme neimplementovanou metodu fungující na principu maskovacích bodů (body určují tvar tělesa), podle kterých jsou podprostory detekovány, se jmenuje $CTileDetector\_MaskPoint$.

Pomocí definovaného rozhraní detektoru podprostorů je jednoduché vytvořit vlastní detektor, což nám zvyšuje možnosti optimalizací a rozšíření knihovny PolyPhysics.

11.2.2 Třída primitivní geometrie tělesa

Úkolem této abstraktní třídy $C3DPrimitive$ je definice geometrie tělesa, vytvoření instance fyzikálního procesoru a instance detektoru podprostorů. Každý potomek této třídy definuje typ sám sebe pomocí struktury $SPrimitiveInfo$, kterou obdržíme zavoláním funkce $Sys\_Info\_Get()$. Důležité virtuální funkce této třídy jsou $Geo\_Surface\_Get()$, $Sys\_TileDetector\_Create()$ a $Sys\_PhysicalProcessor\_Create()$. Kde první vrací vypočítaný povrch tělesa (pro vypočet fyzikální síly tření vzduchu), druhá vytváří konkrétní instanci detektoru podprostorů této geometrie a poslední vytváří konkrétní instanci fyzikálního procesoru pro dynamické těleso této geometrie. Tyto funkce jsou volány z instancí tříd dynamického $C3DSoftObj$ a statického $C3DHardObj$ tělesa, kde funkce $Sys\_PhysicalProcessor\_Create()$ je volána pouze z dynamického tělesa.

Jak je z textu jasné, při vytváření nových potomků třídy $C3DPrimitive$ je nutné implementovat tyto virtuální funkce. V aktuální verzi je zcela implementována třída geometrie koule $C3DPrimitive\_Sphere$ a částečně (pouze pro statické těleso) implementována třída geometrie plochy $C3DPrimitive\_Plane$. Umístění definice těchto tříd je k nahlédnutí v hlavičkovém souboru $polyPrimitives.h$.

Pomocí tohoto rozhraní lze vytvářet vlastní primitivní tělesa a neustále tak vylepšovat možnosti knihovny PolyPhysics. Jedním z cílů bylo také vytvořit polygonální (popsané v kapitole 8.2.2 Vytvoření polygonální tělesa) těleso, které se bude chovat jako primitivní. Tím bychom zachovaly stávající koncept a bylo by možné navrhnout jakoukoli geometrii.

11.2.3 Třída fyzikálního procesoru

Abstraktní $CPhysicalProcessor$ je jednou z nejdůležitějších tříd v knihovně PolyPhysics. Jejím úkolem je detekovat kolizi, řešit kolizi a provádět samotnou simulaci tělesa. Konkrétní implementace, tedy potomek této třídy, je na míru šitá geometrii primitivního tělesa ji používající. Jak již bylo řečeno, instanci této třídy či jeho potomka vytváří instance primitivního tělesa a to pomocí definovaných virtuálních funkcí, které si postupně popíšeme. Funkce $Init()$ inicializuje instanci a je volána z instance třídy $C3DSoftObj$. Je nutné připomenout, že instanci fyzikálního procesoru může vlastnit pouze dynamické těleso, tedy $C3DSoftObj$. Nejdůležitější a nejkomplikovanější funkcí je zde $Sim\_Collision\_Detect()$, která detekuje kolizi a nastavuje kolizní strukturu. V implementacích této funkce v aktuální verzi knihovny PolyPhysics se zejména využívá metody detekce, definované v kapitole 7.1 Metoda půlení intervalů. Tato funkce je bohužel slabým místem knihovny PolyPhysics. Vyžaduje totiž detekci kolize typu geometrie primitivního tělesa s geometrií všech ostatních použitých těles. To nám vytváří jeden nepříjemný problém, pokud totiž přidáme nové primitivní těleso s novou geometrií, neodvozené od stávajících, je nutné do všech ostatních fyzikálních procesorů dodat detekci kolize s tímto novým primitivem. Tento nepříjemný efekt nelze nijak odstranit, pokud máme systém založený na primitivních objektech, je nutné vytvořit funkce detekce

každý s každým. Existuje sice řešení, a to ve formě definice např. polygonálního tělesa (viz kapitola 8.2.2 Vytvoření polygonální tělesa) nebo tělesa vytvořeného z primitiv (viz kapitola 8.2.1 Skládání pomocí primitivních těles), kde se počet kombinací detekce minimalizuje. Další funkce $Sim\_Collision\_Solve()$ řeší kolizi dle kolizní struktury, definované v kapitole 7.6 Řešení kolizí. Její implementace vypočítá nový vektor rychlosti, momenty a další fyzikální parametry tělesa. Funkce $Sim\_JumpTest\_Process()$ provádí testovací simulaci skoku (pohybu) tělesa dle jeho fyzikálních vlastností a sil na něj působících. Testovací skok je speciální verzí simulace zaměřená zejména na detekci kolize. Je zde vymyšlen určitá metoda simulace, která nejprve vytvoří testovací skoky všech těles a pokud nedojde ke kolizi, testovací skok se potvrdí za platný a těleso změní fyzikální stav dle této testovací simulace. O tomto problému si ještě povíme při implementaci řešení multikolizí.

V aktuální verzi knihovny PolyPhysics je implementován potomek této třídy plnící pouze úlohu simulace dle Newtonových zákonů, tedy $CPhysicalProcessor\_2ndNewtonsLaw$. Potomci této třídy teprve implementují konkrétní primitivní tělesa s funkcí detekce kolize dle geometrie primitiva. Plně implementovaná je pouze třída $CPhysicalProcessor\_Sphere$ vyjadřující fyzikální procesor koule. Konkrétně tedy simuluje dle Newtonových zákonů a detekuje kolizi pomocí metody půlení intervalů v kombinaci implementace testů kolize s koulí (viz kapitola 7.3.1 Koule vs. Koule) a s plochou (viz kapitola 7.3.2 Koule vs. Plocha). Umístění definice těchto tříd je k nahlédnutí v hlavičkovém souboru $polyPhysicalDetectors.h$.

Všechny tyto virtuální funkce třídy $CPhysicalProcessor$ jsou volány z instance $C3DSoftObj$. Rozhraní fyzikálního procesoru poskytuje opět možnosti rozšíření knihovny, například si zde můžeme vytvořit vlastní fyziku, vlastní chování a reakce simulace. Bohužel je zde jisté omezení z důvodu detekce kolize se všemi primitivními tělesy, jak již bylo zmíněno.

11.2.4 Třída tělesa

Třída $C3DObj$ vyjadřuje společné vlastnosti statického a dynamického tělesa, které z ní dědí. Jejím úkolem je spíše zobecnění, implementace společných funkcí a optimalizační pomůcky (obsahuje např. metody pro backtransforming, definovaný v kapitole 5.4 Backtransforming). Jednou ze základních funkcí je $Init()$, která inicializuje těleso dle instance fyzikální mapy. Jedno těleso může být pouze v jedné instanci fyzikálního procesoru. $Init()$ je tedy volána z fyzikálního procesoru. Nejpoužívanějšími funkcemi této třídy jsou $Geo\_Translation\_Get()$ a $Geo\_Rotation\_Get()$, o kterých jsme se již zmiňovali v použití knihovny PolyPhysics. Tyto dvě funkce nám vrací geografický stav těles ve fyzikálním prostoru a tyto stavy jsou nutnými parametry pro grafickou reprezentaci tělesa. Další skupinou funkcí této třídy jsou operace nad detekcí podprostorů tělesa, jsou to $Geo\_TilePos\_Process()$, $Geo\_TilePos\_RemoveFrom()$ a $Geo\_Tiles\_Get()$. První funkce říká instanci detektoru podprostoru, aby přepočítal a informoval podprostory. Další říká detektoru, aby vytáhl toto těleso ze všech podprostorů. Poslední funkce zjišťuje množinu podprostorů, ve kterých

těleso leží. Ostatní funkce rozhraní této třídy nejsou tak podstatné, proto si je zde nebudeme popisovat, jejich popis lze nalézt v dokumentaci zdrojového kódu knihovny PolyPhysics (viz Příloha 4.). Umístění definice této třídy je k nahlédnutí v hlavičkovém souboru $polyObjects.h$.

11.2.5 Třída statického tělesa

Třída $C3DHardObj$ je potomkem $C3DObj$ a neobsahuje zatím žádné podstatné funkce. V aktuální době je implementována pouze z důvodů separace statického od dynamického tělesa. Instance této třídy jsou vkládány do fyzikální mapy, aby definovaly pevná tělesa ovlivňující pohyb dynamických těles v prostoru. Umístění definice této třídy je k nahlédnutí v hlavičkovém souboru $polyObjects.h$.

11.2.6 Třída dynamického tělesa

Třída $C3DSoftObj$ je potomkem $C3DObj$ a oproti $C3DHardObj$ obsahuje funkce pro zpracování simulace fyziky. Jejím úkolem je komunikovat s fyzikální mapou a řídit chování tělesa dle stavů (viz kapitola 8.1 Polohový stav tělesa). Tato třída využívá instance detektoru podprostorů a instance fyzikálního procesoru pro plnění jeho funkce. Zde se také definují dva typy simulačních skoků, jeden je testovací (kvůli detekcím kolize fyzikálního procesoru) $Sim\_JumpTest\_Process()$ a funkce potvrzující platnost testovacího skoku, tedy aplikace skoku na geografický stav tělesa $Sim\_Jump\_Process()$. $Sim\_JumpTest\_ProcessSkip()$ je funkce reverze testovacího skoku, která vrátí testovací hodnoty do původního stavu (je využívána při řešení multikolizí, kdy je nutné resetovat testovací skok pro výpočet nového v důsledku kolize). Funkce $Geo\_Collision\_Get()$ se dle stavu tělesa ptá instance fyzikálního procesoru, zda existuje kolize s cílovým tělesem. Funkce $Sim\_Forces\_Process()$ vypočítává vektor fyzikální síly, který ovlivňuje toto těleso. Využívá k tomu rozhraní fyzikální mapy. Další funkcí simulace tělesa je $Sim\_Collision\_Solve()$, která dle stavu řeší kolizi a popřípadě stav tělesa mění. Poslední důležitou funkcí této třídy je $Geo\_TilePos\_Process()$, která přepisuje virtuální funkci $C3DObj::Geo\_TilePos\_Process()$ z důvodu přepočítání podprostorů pro testovací pozici (ne pro reálnou pozici tělesa, detekce podprostoru se bude provádět pro testový geografický stav).

Třída $C3DSoftObj$ nám vytváří jakýsi komunikační prvek mezi fyzikální mapou a konkrétní implementací fyzikálního procesoru s detektorem podprostorů primitiva. Instance této třídy je vkládána do fyzikální mapy aby mohla být simulována v cílovém prostředí. Umístění definice této třídy je k nahlédnutí v hlavičkovém souboru $polyObjects.h$.

11.2.7 Třída fyzikální síly

Třída $C3DForce$ je jednou z nejjednodušších, ale nejmocnějších částí knihovny PolyPhysics. Instance fyzikální síly vložené do fyzikální mapy nám totiž ovlivňuje všechny dynamická tělesa v tomto prostoru se pohybující. Definujeme zde pouze jednu virtuální funkci $F()$, která dle vstupního

dynamického tělesa vypočítá vektor fyzikální síly na těleso působící. Jednoduchost této funkce nám dává do rukou velikou moc, díky které docílíme zajímavých fyzikálních efektů simulovaných těles. Jednou z dalších výhod je dynamické konfigurace a přidávání sil do fyzikální mapy během simulace, můžeme tedy kupříkladu krátkodobě vytvořit silné gravitační pole, které bude simulovat silný výbuch.

V aktuální verzi knihovny PolyPhysics je implementována gravitační síla plochy daná vektorem, jako $C3DForce\_GPlane$. Instancí této třídy lze simulovat gravitační pole na povrchu země (viz kapitola 6.2.2 Plošná gravitační síla). Další implementovanou silou je $C3DForce\_GPoint$, která dle Newtonova gravitačního zákona vytváří gravitační bod (viz kapitola 6.2.3 Bodová gravitační síla). Poslední implementovanou silou vytvářející odpor prostředí daný povrchem tělesa při letu prostorem je $C3DForce\_SpaceAverse$ (viz kapitola 6.2.1 Síla odporu prostředí). Umístění definice těchto tříd je k nahlédnutí v hlavičkovém souboru $polyForces.h$.

Možnosti tohoto rozhraní jsou vcelku zajímavé, můžeme zde definovat určitou vlastnost dynamických těles, která bude ovlivňovat pouze speciální síla. Kupříkladu definujme vlastnost jako materiál, ze kterého těleso je (dřevo, železo, plast). Můžeme pak vytvořit sílu ovlivňující pouze určitý typ a simulovat tak třeba magnetickou sílu přitahující pouze železné předměty dohromady s gravitační (simulace levitace železných předmětů pomocí magnetického silového pole).

11.2.8 Třída podprostoru

$C3DTile$ je jednoduchou třídou definující podprostor a implementující základní funkce vložení tělesa do podprostoru $AddObj()$ a odebrání tělesa z podprostoru $RemoveObj()$. Tato třída je využívána zejména instancí detektoru podprostorů a fyzikální mapy. Definice této třídy je k nahlédnutí v hlavičkovém souboru $polyTiles.h$.

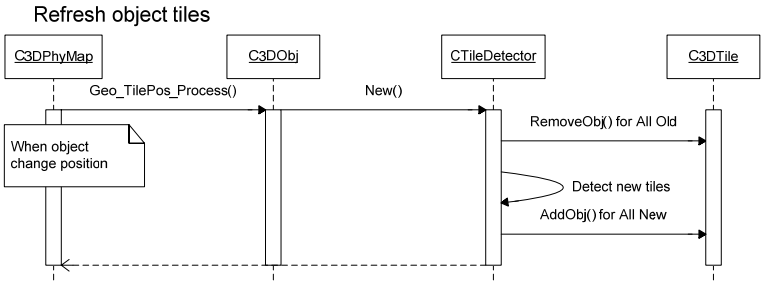
11.2.9 Třída fyzikálního prostoru

Jak již bylo řečeno, třída $C3DPhyMap$ je jádrem celé knihovny a hlavním rozhraní komunikace s uživatelem (aplikací). Používá rozhraní statických, dynamických těles a fyzikálních sil vně vnořené aby mohl řídit průběh simulace. Některé funkce této třídy již byly popsány při ukázkovém příkladě použití knihovny, proto si tedy popíšeme jen ty nejdůležitější. Hlavní funkce fyzikální mapy je zpracování simulačního kroku, který je dán funkcí $Sim\_Step\_Process()$. Ta implementuje nejsložitější algoritmus celé knihovny, který provádí simulační skoky všech dynamických těles, detekuje lokální kolize (viz kapitola 7.2.5 Určení množiny lokální kolize dle metody Tile sets) a řeší multikolize (viz kapitola 7.5 Multikolize). Tento algoritmus si ještě popíšeme. Jak již bylo řečeno, v simulačním kroku se nacházejí řešené kolize těles, funkce $Sim\_StepCollisions\_Get()$ nám vrátí zásobník kolizních struktur posledního simulačního kroku. Tento zásobník lze dobře použít při přehrání zvukových efektů kolize či řídit určitou událost danou

uživatelem knihovny. Dále zde máme funkce napomáhající či ovlivňující simulaci. $Sim\_Init()$ inicializuje simulaci, $Sim\_Pause\_Set()$ pozastaví či spustí zpracování simulace (pokud je simulace pozastavena, funkce $Sim\_Step\_Process()$ nic nezpracovává a ihned se vrací). Funkce $Phy\_Forces\_Process()$ vypočte vektor síly působící na těleso dané parametrem, tato funkce je využívána zejména instancemi třídy $C3DSoftObj$. Máme zde jistou sadu funkcí systémové kategorie, které nám přidávají a odebírají fyzikální síly, statická i dynamická tělesa z mapy a předávají listy těchto objektů. Jsou to funkce $Sys\_SoftObj\_Add()$, $Sys\_SoftObj\_Remove()$, $Sys\_HardObj\_Add()$, $Sys\_HardObj\_Remove()$, $Sys\_Force\_Add()$, $Sys\_Force\_Remove()$, $Sys\_SoftObj\_GetList()$, $Sys\_HardObj\_GetList()$ a $Sys\_Force\_GetList()$, kde již název napovídá typ operace. Umístění definice třídy $C3DPhyMap$ je k nahlédnutí v hlavičkovém souboru $polyPhysics.h$, který je také hlavním hlavičkovým souborem pro použití knihovny PolyPhysics, je ho tedy nutné do zdrojového kódu aplikace přidat pomocí direktivy $\#include$ "$polyPhysics.h$".

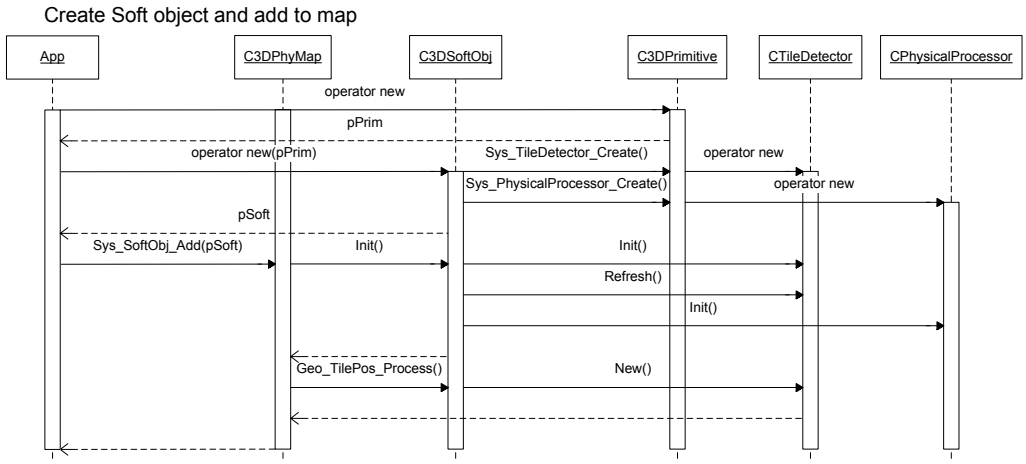
Jako příklady komunikace rozhraní knihovny si ukážeme několik sekvenčních diagramů znázorňující určitou operaci.

obr. 11.2, sekvenční diagram komunikace tříd při přepočítání množiny podprostorů tělesa



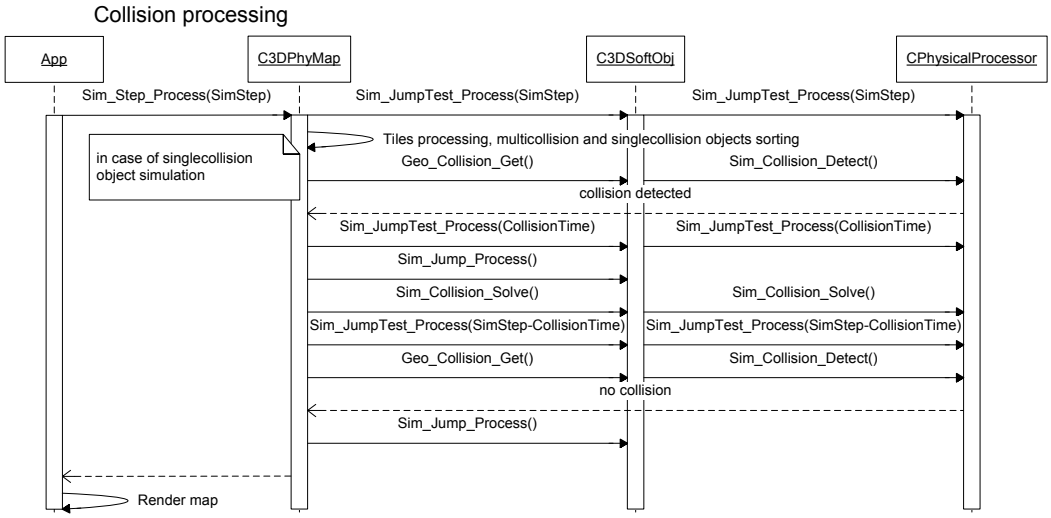
Na obr. 11.2 lze vidět sekvenční diagram spolupráce rozhraní tříd při přepočítání množiny podprostorů tělesa ve fyzikální mapě. Tato operace se provádí po výpočtu nového testovacího skoku tělesa a napomáhá k nalezení množiny lokálních kolizí.

obr. 11.3, sekvenční diagram komunikace tříd při vytvoření nového dynamického tělesa



Na obr. 11.3 lze vidět sekvenční diagram spolupráce rozhraní tříd při vytvoření nového dynamického tělesa dané třídou $C3DSoftObj$ v aplikaci používající knihovnu PolyPhysics. Instance tělesa je následně vložena do fyzikální mapy. Všimněme si, že instance primitiva tělesa je vytvořena jako první, jelikož je parametrem konstruktoru třídy $C3DSoftObj$.

obr. 11.4, sekvenční diagram komunikace tříd při zpracování simulačního kroku s kolizí

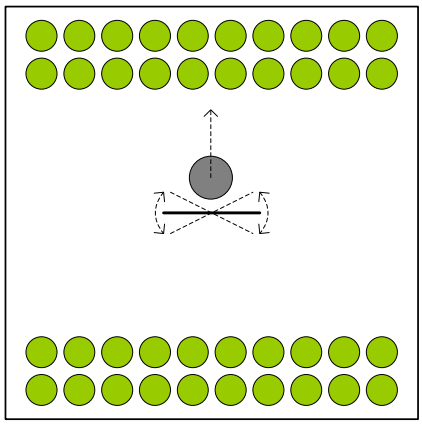


Na obr. 11.4 lze vidět sekvenční diagram spolupráce rozhraní tříd při zpracování simulačního kroku, kde máme scénu pouze s jedním dynamickým tělesem, které následně koliduje s jiným statickým. Kolize se vyřeší a dle metody řešení multikolizí se testuje kolize pro zbývající čas simulačního kroku. Algoritmus simulačního kroku je mnohem složitější, tento sekvenční diagram ukazuje pouze základní část a jeho funkci.

12 Demo použití knihovny PolyPhysics

Jedním ze zadání této diplomové práce je implementovat netriviální příklad prostorového modelu. Tento úkol byl již splněn za pomoci aplikací PolyPhysics GLTest, kde lze simulovat jakoukoli uživatelem definovanou scénu (pomocí aplikace Map2XML). S ohledem na přenositelnost byla ovšem vytvořena netriviální demo aplikace s použitím knihovny PolyPhysics ve formě hry. Tato hra s názvem *BallFun* je založena na klasické 2D hře *Arcanoid*, kde hráč kuličkou ničí zeď cihel. V našem případě je hra přenesena do 3D prostoru. Namísto cihliček ničíme koule letící v prostoru dle fyzikálních zákonů. Každá taková koule ale vykonává určité chování ovlivňující ostatní koule, například může vybuchnout, vytvořit silné gravitační pole nebo se z ní stane statické těleso. Tyto objekty ničíme pomocí železné kuličky řízené elektromagnetem, který ovládá hráč. Elektromagnet je tvořen plochou uprostřed scény, se kterou lze manipulovat a tím určovat směr odrazu železné kuličky. Oproti klasickému Arcanoidu je hrací pole zrcadleno, tedy hrajeme na dvě strany, podobně jak lze vidět obr. 12.1 v 2D reprezentaci.

obr. 12.1, 2D reprezentace hry BallFun



Hra BallFun je multiplatformě (Win32, UNIX) implementována v C/C++ v kombinaci s knihovnou PolyPhysics, GLUT a OpenGL. Ovládání je vcelku jednoduché. Železná koule je vymrštěna do prostoru a koliduje s ostatními tělesy, které následně s časovým odstupem provedou jejich funkci (výbuch, gravitaci, přeměna ve statické těleso či zmizí). Hráč dále může otáčet středovou odrazovou plochou, čímž může ovlivňovat směr odrazu železné koule. Tato plocha také obsahuje elektromagnetický bod, který může hráč kdykoli zapnout a tak přivolat zpět železnou kouli (dodat ji znovu rychlost a směr dle odrazu). Do prostoru hry je také vložena síla odporu prostředí, která brání neustálému pohybu těles.

Tato demo aplikace je ukázkou jednoduchosti použití knihovny PolyPhysics do cílového zaměření, tedy do her. Jsou zde předvedeny také dynamické a rozšiřující možnosti tohoto konceptu zejména u fyzikálních sil, kde si uživatel (aplikace používající knihovnu) implementuje vlastní komponenty (elektromagnetickou sílu, těleso ze železa). Bližší informace o této hře jsou v příloze této práce.

13 Závěr

Zadáním této diplomové práce bylo navrhnout a implementovat multiplatformní knihovnu pro real-time simulaci fyziky tuhých těles v 3D scénách. Implementovaná knihovna PolyPhysics vytváří nadstavbu nad grafickým enginem a z velké části tak ulehčuje programátorovi práci díky znalosti prostoru a chování objektů.

Dle zkušeností získaných z ročníkového projektu jsem navrhl univerzální a rozšířitelný simulační model a škálu teoretických poznatků a algoritmů, přispívajících k řešení problematiky fyzikální simulace. Mezi tyto mé příspěvky patří detekční metoda Backtransforming s detekcí kolize koule a plochy, optimalizační metoda podprostorových množin včetně algoritmu detekce lokální kolize, definice problému multikolize s algoritmem řešení a teoretický rozbor polohového stavu tělesa. Jako největší výhody této knihovny bych zdůraznil zejména její univerzálnost, rozšiřitelnost, jednoduchost aplikace a variabilitu založenou na modelu fyzikálních sil. Knihovna nám nejen dává možnost simulovat reálný svět, jak ho známe, ale také i experimentování s prakticky nerealizovatelným prostorem.

Jak lze z této práce usoudit, fyzikální simulace je vcelku rozsáhlé téma s řadou problémů s ní souvisejících. Nepodařilo se mi tedy zcela implementovat teoretický rozsah této práce a vytvořit tak plně funkční obecný model simulace fyziky. Mezi tyto chybějící části patří implementace chování dle polohového stavu tělesa, avšak model tělesa je na tuto skutečnost připraven. Mým osobním cílem bylo implementovat obecný model polygonálního tělesa a spojování primitivních geometrií. Tento cíl se mi bohužel z časových a znalostních důvodů nepodařilo splnit. V knihovně je tedy implementováno chování statického i dynamického primitiva koule a statického primitiva ohraničené plochy včetně přesných detekcí kolizí a jejich řešení. Díky rozšiřitelnosti knihovny je pouze otázkou času implementovat zbývající primitivní tělesa. Je nutné podotknout, že existují možné chyby chování simulace, které jsou zapříčiněné zejména chybějící implementací polohového stavu tělesa. Během implementace a návrhu bohužel z časových důvodů nedošlo ke konzultaci s odborníkem z oblasti fyziky, který by podstatně dopomohl k lepší funkčnosti knihovny PolyPhysics.

Díky této diplomové práci jsem si prohloubil znalosti problematiky fyzikální simulace, získal zkušenosti při návrhu univerzálního modelu a návrhu rozhraní při implementaci uživatelské knihovny. Získal jsem také mnoho zkušeností v technice objektově orientovaného programování, návrhu optimalizací a užívání knihovny OpenGL.

Literatura

1. **Filipovič, Jiří.** *Game Developers Session 2005.* [Online] prosinec 2005. URL: <http://video.fi.muni.cz/public/gds2005/SimulaceDynamikyRigidnichTeles.wmv>.

2. **Doc. Ing. Zdena Rábová, CSc.** Modelování a simulace. *FIT VUT.* [Online] prosinec 2005. URL: <https://www.fit.vutbr.cz/study/courses/MSI/private/Prednasky/welcome.html>.

3. **Baker, Martin.** Building a 3D world. *EuclideanSpace.* [Online] říjen 2006. URL: <http://www.euclideanspace.com/>.

4. **RNDr. Anežka Haluzíková, CSc.** *Numerické metody.* Brno : Vysoké učení technické v Čs. redakci VM MON, 1989. ISBN 80-214-0039-0.

5. **Sládková, Jarmila a kolektiv.** *Sbírka úloh z fyziky.* Brno : VUTIUM, 1998. ISBN 80-214-1216-X.

6. **Walker, D., Halliday, R. a Resnick, J.** *Fyzika.* Brno : VUTIUM, 2003. ISBN 81-214-1868-0.

7. **Christopoulos, Dimitrios.** Lekce 30 - Detekce kolizí. *CZ NeHe OpenGL.* [Online] únor 2005. URL: <http://nehe.ceskehry.cz/tut_30.php>.

8. **Ditchburn, Keith.** Collisions. *Toymaker.* [Online] únor 2005. URL: <http://www.toymaker.info/Games/html/collisions.html>.

9. **Nettle, Paul.** Generic Collision Detection for Games Using Ellipsoids. *Fluid Studios.* [Online] listopad 2005. URL: <http://www.fluidstudios.com/pub/FluidStudios/CollisionDetection/Fluid_Studios_Generic_Collision_Detection_for_Games_Using_Ellipsoids.pdf>.

10. **Minařík, Petr.** Detekce kolizí v DirectX. *Net-Mag.* [Online] prosinec 2006. URL: <http://programovani.net-mag.cz/?action=art&num=459>.

11. **Bradshaw, Gareth.** Sphere-Tree Construction Toolkit. *Interaction, Simulation and Graphics Lab.* [Online] listopad 2006. URL: <http://isg.cs.tcd.ie/spheretree/>.

12. **James, Doug.** BD-Tree: Output-Sensitive Collision Detection for Reduced Deformable Models. *Carnegie Mellon Graphics Lab.* [Online] prosinec 2006. URL: <http://graphics.cs.cmu.edu/projects/bdtree/>.

13. **Shimer, Carl.** BSP trees in 3D worlds. *Binary Space Partition Trees.* [Online] prosinec 2006. URL: <http://web.cs.wpi.edu/~matt/courses/cs563/talks/bsp/bsp.html>.

14. **Fauerby, Kasper.** Improved Collision detection and Response. *Peroxide Entertainment.* [Online] červenec 2005. URL: <http://www.peroxide.dk/papers/collision/collision.pdf>.

15. **Ray, Tathagata.** Ray's homepage. *Department of Computer Science and Engineering.* [Online] duben 2007. URL: <http://www.cse.ohio-state.edu/~rayt/>.

16. **Arlow, Jim a Neustadt, Ila.** *UML a unifikovaný proces vývoje aplikací.* místo neznámé : Computer Press, 2003. ISBN 80-7226-947-X.

17. **Schildt, Herbert.** *C++.* místo neznámé : Softpress, 2002. ISBN 80-86497-13-5.

Seznam příloh

Příloha 1. Obrázky a ovládání jednoduché hry BallFun

Příloha 2. Obrázky editoru fyzikální scény Map2XML včetně příkladu XML

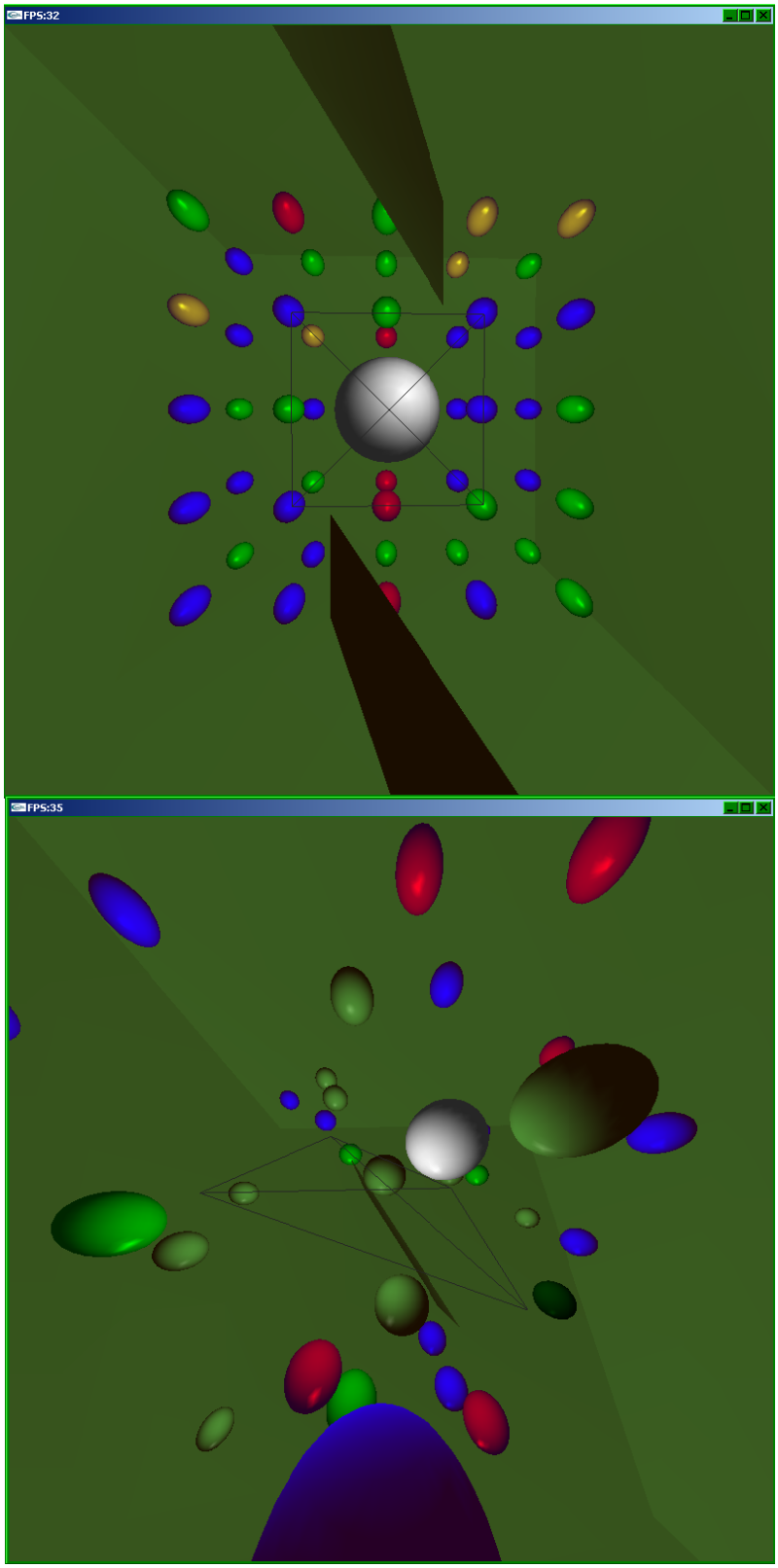
Příloha 3. Obrázky testovací aplikace GLTest

Příloha 4. CD obsahující:

- veškeré zdrojové kódy
- dokumentaci kódu (doxygen HTML Help)
- uživatelskou a technickou dokumentaci knihovny
- ostatní předlohy a materiály této diplomové práce

Příloha 1. Obrázky a ovládání jednoduché hry BallFun

Následující obrázky jsou ukázkové screenshoty ze hry BallFun.



Ovládání hry BallFun je vcelku jednoduché, je použita pouze počítačová myš se třemi tlačítky. Grafickou scénu si natáčíme držením pravého tlačítka myši. Levým tlačítkem myši pohybujeme s plochou vprostřed scény (odrazová deska) a při stlačení prostředního tlačítka zapneme sílu přitahující pouze bílou hrací kouli. V kombinaci natáčení odrazové desky a zapínaní síly pro bílou kouli můžeme manipulovat s herními kuličkami. Jsou čtyři typy (barev) kuliček, každá má jiné vlastnosti.

Modrá – 4 sekundy po nárazu s bílou koulí zmizí

Zelená – 4 sekundy po nárazu s bílou koulí se z ní stane statické těleso (bledozelená)

Žlutá – 3 sekundy po nárazu s bílou koulí vytvoří na zlomek sekundy gravitační bod a zmizí

Červená – 3 sekundy po nárazu s bílou koulí vybuchne (inverzní gravitační bod) a zmizí

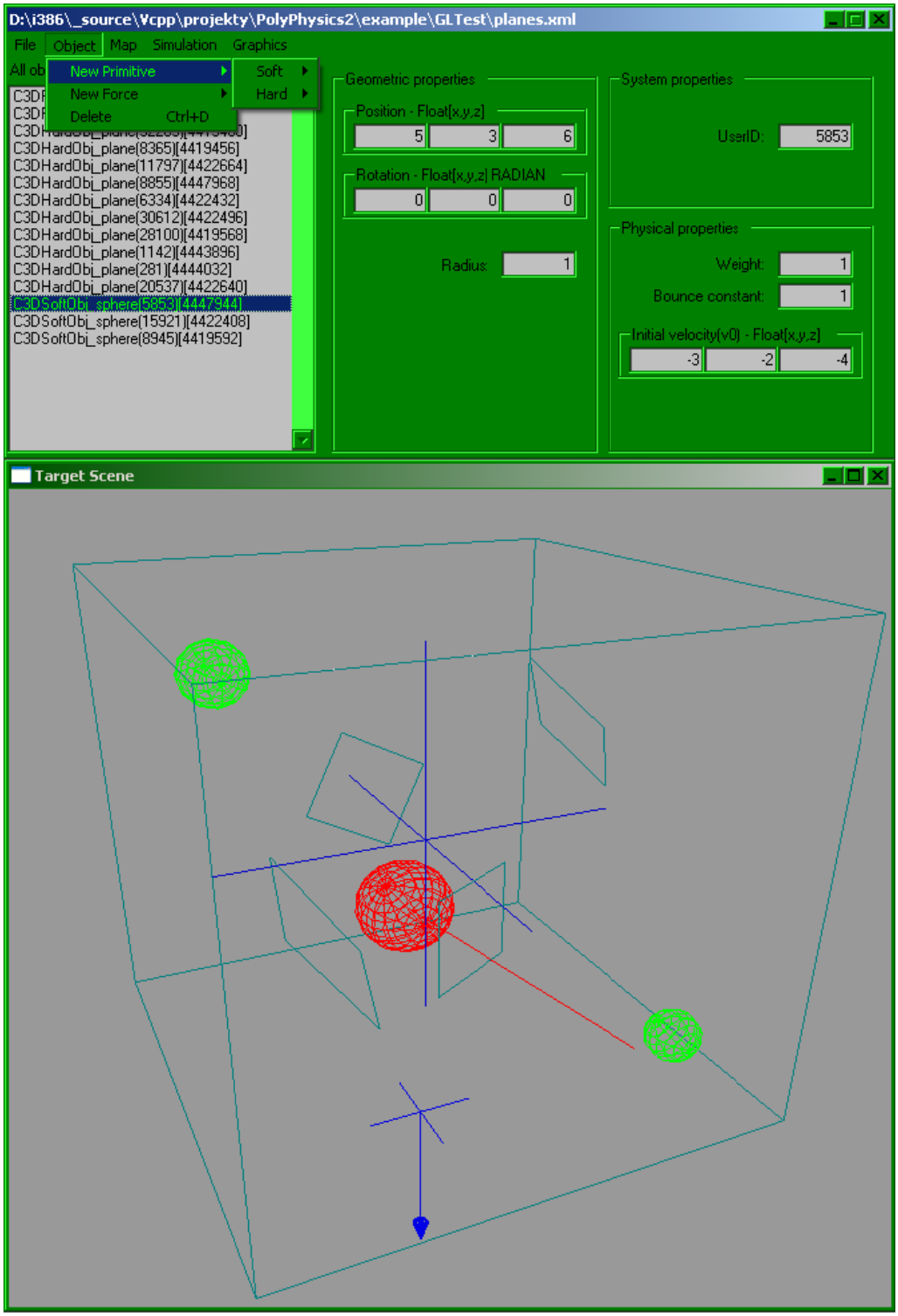
Příloha 2. Obrázky editoru fyzikální scény Map2XML včetně příkladu XML

Následující XML kód je ukázka definování složitější fyzikální scény.

```
<PolyPhysicsXml Version="1.0">
  <Map>
    <Size X="10" Y="10" Z="10" />
    <Tiles X="10" Y="10" Z="10" />
  </Map>
  <Simulation>
    <TimeSpeed F32="1" />
    <Forces>
      <GravityPlane UserID="9318">
        <Translation X="6" Y="0" Z="3" />
        <Gravity X="0" Y="-3" Z="0" />
      </GravityPlane>
      <GravityPoint UserID="7045">
        <Translation X="5" Y="5" Z="5" />
        <Gravity F32="8" />
      </GravityPoint>
    </Forces>
  </Simulation>
  <Primitives>
    <HardObject UserID="32285">
      <Translation X="10" Y="5" Z="5" />
      <Rotation X="0" Y="-1,570796" Z="0" />
      <Bounce F32="1" />
      <Plane>
        <HalfSideX F32="5" />
        <HalfSideY F32="5" />
      </Plane>
    </HardObject>
    <HardObject UserID="8365">
      <Translation X="0" Y="5" Z="5" />
      <Rotation X="0" Y="1,570796" Z="0" />
      <Bounce F32="1" />
      <Plane>
        <HalfSideX F32="5" />
        <HalfSideY F32="5" />
      </Plane>
    </HardObject>
    <HardObject UserID="11797">
      <Translation X="5" Y="5" Z="10" />
      <Rotation X="3,14159" Y="0" Z="0" />
      <Bounce F32="1" />
      <Plane>
        <HalfSideX F32="5" />
        <HalfSideY F32="5" />
      </Plane>
    </HardObject>
    <HardObject UserID="8855">
      <Translation X="5" Y="0" Z="5" />
      <Rotation X="-1,570796" Y="0" Z="0" />
      <Bounce F32="1" />
      <Plane>
        <HalfSideX F32="5" />
        <HalfSideY F32="5" />
      </Plane>
    </HardObject>
    <HardObject UserID="6334">
      <Translation X="5" Y="5" Z="0" />
      <Rotation X="0" Y="0" Z="0" />
      <Bounce F32="1" />
      <Plane>
        <HalfSideX F32="5" />
        <HalfSideY F32="5" />
      </Plane>
    </HardObject>
    <HardObject UserID="30612">
      <Translation X="5" Y="10" Z="5" />
      <Rotation X="1,570796" Y="0" Z="3,141593" />
      <Bounce F32="1" />
      <Plane>
        <HalfSideX F32="5" />
        <HalfSideY F32="5" />
      </Plane>
    </HardObject>
...
```

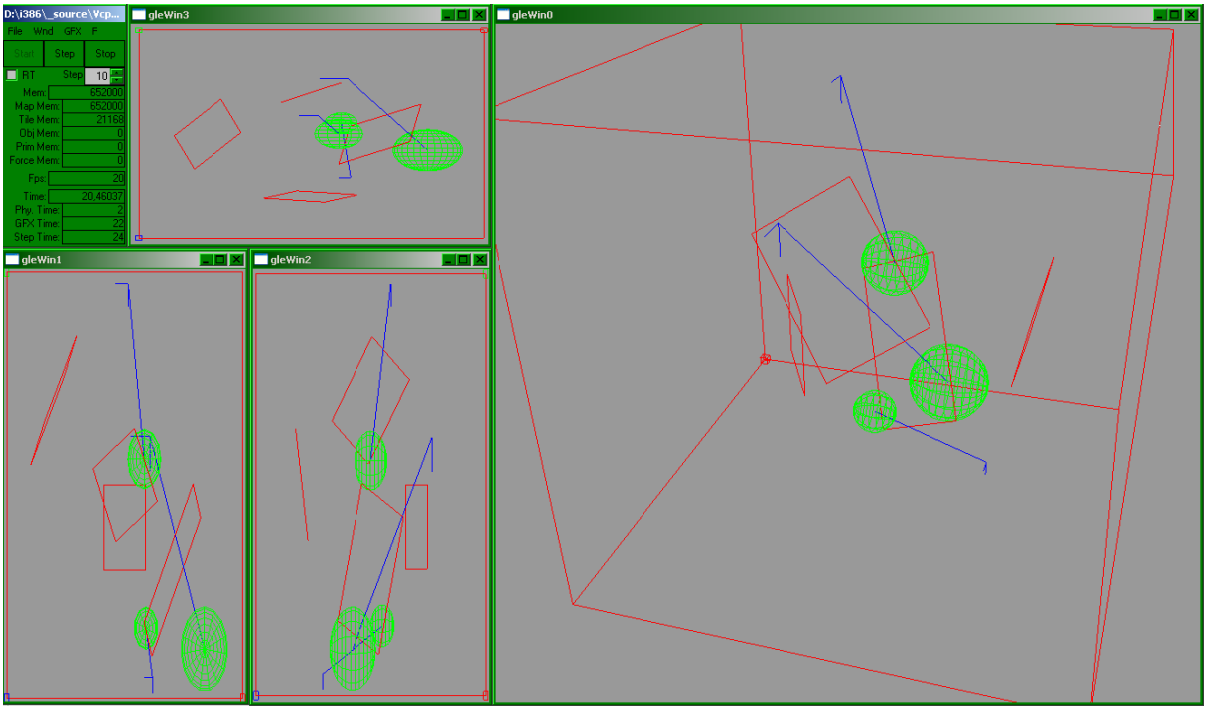
```
...
    <HardObject UserID="28100">
      <Translation X="5" Y="4" Z="3" />
      <Rotation X="0" Y="0,5" Z="0" />
      <Bounce F32="1" />
      <Plane>
        <HalfSideX F32="1" />
        <HalfSideY F32="1" />
      </Plane>
    </HardObject>
    <HardObject UserID="1142">
      <Translation X="5" Y="5" Z="8" />
      <Rotation X="0,2" Y="0" Z="0,5" />
      <Bounce F32="1" />
      <Plane>
        <HalfSideX F32="1" />
        <HalfSideY F32="1" />
      </Plane>
    </HardObject>
    <HardObject UserID="281">
      <Translation X="7" Y="3" Z="5" />
      <Rotation X="0,9" Y="1" Z="0,3" />
      <Bounce F32="1" />
      <Plane>
        <HalfSideX F32="2" />
        <HalfSideY F32="1" />
      </Plane>
    </HardObject>
    <HardObject UserID="20537">
      <Translation X="2" Y="7" Z="5" />
      <Rotation X="1,5" Y="1" Z="1" />
      <Bounce F32="1" />
      <Plane>
        <HalfSideX F32="1" />
        <HalfSideY F32="1,5" />
      </Plane>
    </HardObject>
    <SoftObject UserID="5853">
      <Translation X="5" Y="3" Z="6" />
      <Rotation X="0" Y="0" Z="0" />
      <Bounce F32="1" />
      <Weight F32="1" />
      <InitialVelocity X="-3" Y="-2" Z="-4" />
      <Sphere>
        <Radius F32="1" />
      </Sphere>
    </SoftObject>
    <SoftObject UserID="15921">
      <Translation X="2" Y="2" Z="1" />
      <Rotation X="0" Y="0" Z="0" />
      <Bounce F32="1" />
      <Weight F32="1" />
      <InitialVelocity X="0" Y="0" Z="0" />
      <Sphere>
        <Radius F32="0,5" />
      </Sphere>
    </SoftObject>
    <SoftObject UserID="8945">
      <Translation X="8" Y="8" Z="8" />
      <Rotation X="0" Y="0" Z="0" />
      <Bounce F32="1" />
      <Weight F32="1" />
      <InitialVelocity X="0" Y="0" Z="0" />
      <Sphere>
        <Radius F32="0,7" />
      </Sphere>
    </SoftObject>
  </Primitives>
</PolyPhysicsXml>
```

Následující obrázek znázorňuje grafickou reprezentaci XML souboru vytvořeného v editoru fyzikální scény Map2XML.



Příloha 3. Obrázky testovací aplikace GLTest

Následující obrázek znázorňuje screenshot aplikace GLTest při simulaci XML souboru popsaného výše.



Následující obrázek znázorňuje screenshot složitější fyzikální scény simulované v aplikaci GLTest. Modré šipky vyjadřují vektor rychlosti dynamického tělesa (zelené objekty) včetně vektorů rychlosti po srážce, fialové šipky vyjadřují normálový vektor kolize a žlutý hvězda bod kolize.