

Simulátor vrtulníku – Programový systém pro realizaci fyzikálního modelu

Dušan Krajčovič

Bakalářská práce
2005



Univerzita Tomáše Bati ve Zlíně
Fakulta technologická

Navrhňte a pomocí jazyka C/C++ a vhodných softwarových knihoven vytvořte programový systém realizující fyzikální model vrtulníku. Práce bude obsahovat tyto body:

- 1) Vytvořte rešerši na téma návrhu a tvorby simulátorů na PC, uveďte a srovnajte používané technologie, vývojové nástroje a softwarové knihovny a komponenty.
- 2) Navrhňte a vytvořte programový systém realizující fyzikální model simulátoru vrtulníku s těmito vlastnostmi:
 - a. Program bude vytvořen pro operační systém MS Windows
 - b. Navrhňte a realizujte softwarové rozhraní mezi hlavní aplikací a jednotlivými Plotony
 - c. Zajistěte, že aplikace bude rozšiřitelná pomocí systému pluginů zapouzdřující její funkčnost (simulace vnitřních a vnějších dějů, vizualizace, ovládání, atd.) Plutony budou tvořeny jednotlivými DLL soubory
- 3) Vytvořte dokumentaci ve formě UML diagramů popisujících vnitřní strukturu a činnost aplikace a jeho návaznost na jádro simulátoru.

Souhlasím s tím, že s výsledky mé práce může být naloženo podle uvážení vedoucího bakalářské práce a ředitele ústavu. V případě publikace budu uveden jako spoluautor.

Prohlašuji, že jsem na celé bakalářské práci pracoval samostatně a použitou literaturu jsem citoval.

Ve Zlíně, 03. 06. 2005

.....

podpis

RESUMÉ

Cílem této bakalářské práce je seznámení s dostupnými prostředky a knihovnami pro realizaci fyzikálního modelu helikoptéry a vytvoření vlastního funkčního modelu, který by bylo možné dále rozšiřovat. K tomu je nutná kvalitní dokumentace, a velmi dobře okomentovaný zdrojový kód. Ve své bakalářské práci se zmiňuji o základních postupech a možnostech při návrhu libovolné simulace a to především s použitím simulačních jazyků a knihoven. Nejdůležitější částí mé bakalářské práce je kapitola nazvaná „Fyzikální enginy“, která pojednává o principech práce se základními fyzikálními veličinami, této kapitole jsem věnoval velkou pozornost, protože jsem z ní čerpal při svém návrhu kontrolního systému. V experimentální části vysvětluji nejdůležitější části zdrojového kódu, přičemž jsem velmi podrobně zdokumentoval celý zdrojový kód tak, aby byl velmi přehledný.

Anotace ve světovém jazyce

The point of this bachelor work is apprise of available substances and libraries for the realization of physical model of the helicopter and to forme a functional model furthe. This model would be possible to expand further. It is necessary to have first-rate documentation as well, and to have very well commented source code. In my bachelor work I mention essential progresses and possibilities at suggestion of any simulation, simulation coding language and simulation libraries in the first place. The most important part of my bachelor work is chapter called "physical enginy" which handles about the work with a fundamental physical quantity. I have paid to this chapter big attention, because I have gathered of it during suggestion of the control system. I explain the most important parts of the source code in the experimental part, whereas I have documented the whole source code in the details in the way it would be very synoptical.

OBSAH

1 ÚVOD	7
2 MODELOVÁNÍ A SIMULACE	9
2.1 Metodologie modelování	9
2.1.1 Modelování	9
2.1.2 Experiment	9
2.1.3 Modely	10
2.2 Kybernetika a modelování	10
2.3 Matematický model	11
2.3.1 Fyzikální význam transformací matematického modelu	16
2.4 Počáteční a okrajové podmínky	16
3 SIMULACE	18
3.1 Identifikace	18
3.2 Simulační modely	21
4 SIMULAČNÍ JAZYKY	23
4.1 Sémantika simulačních jazyků	24
4.2 Třídění simulačních jazyků	25
4.3 Syntaxe simulačních jazyků	25
4.4 Prostředky simulačních jazyků	26
4.5 Jazyky pro simulaci spojitých systémů	28
4.6 Jazyky pro simulaci diskrétních systémů	29
4.7 Seznam významných simulačních jazyků	30
5 SIMULAČNÍ KNIHOVNY	31
5.1 C++Sim	31
5.2 SIMLIB	32
6 VISUÁLNÍ SIMULAČNÍ NÁSTROJE	34
7 FYZIKÁLNÍ ENGINE	36
7.1 Proč fyzikální engine	36
7.2 Základní návrh enginu	36
7.2.1 Hmota - hmotnost	37
7.2.2 Čas	37
7.2.3 Pozice	37
7.2.4 Rychlost	38
7.2.5 Lineární vektorová rychlost	38
7.2.6 Kvantitativní rychlost	39
7.2.7 Úhlová rychlost	39
7.2.8 Zrychlení	39
7.2.9 Lineární zrychlení	40
7.2.10 Úhlové zrychlení	40
7.2.11 Těžiště	40
7.2.12 Síly	41
7.2.13 Lineární síly	41
7.2.14 Úhlová síla	42
7.2.15 Výsledný efekt síly	43
7.2.16 Gravitační interakce	43
7.2.17 Gravitace na zemi a na jiných planetách	44
7.2.18 Simulace projektilů	45
7.2.19 Tření	46
7.3 The Open Dynamic Engine	46

7.3.1	Vlastnosti ODE	46
7.4	Tokamak	47
7.4.1	Základní třídy	47
7.5	Newton.....	48
7.6	NovodeX.....	49
8	EXPERIMENTÁLNÍ ČÁST	50
9	ZÁVĚR.....	58
	SEZNAM POUŽITÉ LITERATURY	59
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK	61
	SEZNAM OBRÁZKŮ	62

1 ÚVOD

Tři cesty vedou k poznání : cesta zamyšlení – ta je nejušlechtilější, cesta napodobování – ta je nejlehčí a cesta pokusů – ta je nejtrpčí

Konfucius (okolo 551 až 479 př. n. l)

Již od počátku vzniku prvního počítače se člověk snaží modelovat a simulovat reálné fyzikální děje a procesy probíhající kolem něj. K tomuto kroku ho vede hned několik faktorů, simulace jsou většinou mnohonásobně levnější a to jak finančně tak i materiálně, lze simulovat děje které by byly v příliž zdoluhavé či nebezpečné a nebo pro svou obtížnost těžce realizovatelné. Toto je jen malý výčet vlastností proč se simulace těší tak velké pozornosti. I přes neustále se rychle rozvíjející se trend v oblasti hardwaru a stále se zvyšující se výkon jednotlivých komponent, nejsme schopni naprosto přesně simulovat velké dynamické systémy, jakým je například helikoptéra, ve kterých probíhá obrovské množství dějů a procesů. Proto je nutné si hned na začátku položit otázku, „jaký fyzikální děj chceme simulovat, který je pro nás důležitý a zajímavý a které naopak můžeme zanedbat“. Veškeré simulace jsou omezeny na simulování reálných procesů, a pro správné vytvoření modelu musí být přesně popsány prostorové, časové, fyzikální a technické procesy. Pro vytvoření simulátoru neexistuje konkrétní metodika, je nutná podrobná znalost procesů a souvislostí v systému a ani to v mnoha případech nestačí, protože přiblížení k realitě je doprovázeno množstvím chyb, a to jak v analýze tak i při samotném modelování.

Tato bakalářská práce si dává za úkol vytvoření reálného dynamického modelu leteckého simulátoru. To přesně znamená, že se budeme snažit simulovat mechanické síly působící na helikoptéru. I když nejcharakterističtější „pohybem“ helikoptéry je samovolné visení, což se může z pohledu logiky zdát jako ten nejjednodušší pohyb, protože nekonáme žádný pohyb, z hlediska realizovatelnosti je to vlastně ten nejobtížnější, důvod je zcela prostý, je nutné aby výslednice všech sil působících na helikoptéru byla nulová

Dynamický model bude koncipován co nejobecněji aby byl použitelný pro co nejširší škálu simulátorů snažící se simulovat pohyblivé objekty. Simulátor bude napsán v jazyce C/C++ a bude využívat knihovnu MFC (Microsoft Foundation Class). Výsledná aplikace bude využívat principů objektového programování a bude složena z jednotlivých modulů DLL (Dynamic Link Library) , které budou reprezentovat fyzické objekty simulovaného modelu a fyzikální procesy probíhající a působící na model.

Simulace je metoda při níž se snažíme reálný objekt nahradit počítačovým modelem. Cílem je, aby se model choval stejně jako sledovaný objekt. Na vytvořeném modelu se pak dá

experimentovat takovým způsobem, který by v realitě nebyl vůbec možný nebo pouze za velmi nevýhodných podmínek (dlouhý čas, vysoké náklady).

2 MODELOVÁNÍ A SIMULACE

Modelování jako specifická forma experimentu umožňuje člověku poznávat různé objektivní zákonitosti přírody. Jeho velmi rychlý rozvoj způsobuje postupnou nepřehlednost v oboru. Proto se stává důležitým pojem „metodologie modelování“

2.1 Metodologie modelování

Velmi důležitým a obtížným úkolem metodologie modelování je nalezení společného hlediska pro rozdělení modelů, které by umožnily obecnou klasifikaci. Velmi slibnými se zdají hlediska informační a podobnostní.

2.1.1 Modelování

Modelování představuje jeden z nejobecnějších způsobů zobrazení vnějšího světa, prováděného ke zkoumání v něm existujících zákonitostí. „Modelování je experimentální informační proces, při němž se zkoumanému systému jednoznačně přiřazuje podle určitých kritérií jiný systém, fyzický nebo abstraktní, nazývaný model. Různé druhy modelování souvisejí s různými formami podobnosti a analogie. Přitom podobnost lze chápat jako jednoznačné vzájemné přiřazení mezi různými systémy v jejich struktuře, vlastnostech a chování. Modelování lze podle podobnosti dělit na fyzikální, matematické, kybernetické. Fyzikální podobnost vyjadřuje podobnost mezi systémy a procesy stejné fyzikální podstaty a zahrnuje vedle geometrické podobnosti parametrů a stavových veličin systému. Matematická podobnost vyjadřuje podobnost mezi systémy a procesy majícími stejný matematický popis. Jde-li o fyzikálně odlišné systémy a procesy, nazývá se analogie. Kybernetická podobnost vyjadřuje matematickou podobnost ve vnějším chování systémů. Kybernetické modelování používá tzv. model černé skříňky. Tento model nám neříká žádné informace o vnitřní struktuře, ale jenom o vnějším chování.

2.1.2 Experiment

Experiment je nadřazený pojem pro modelování. Je to cílevědomá praktická činnost směřující k rozvoji poznání. Experimenty dělíme:

- fyzický
- myšlenkový
- vědecký
- výrobní

- výukový
- aj.

Nejdůležitějším z nich je experiment vědecký, který vychází z měření a modelování. Přitom měření se rozumí informační proces, jehož podstatou je snímání, převod a přenos informace z objektu k uživateli. Důležitou vlastností experimentu je jeho opakovatelnost za stejných podmínek. Modelový experiment se využívá ke zjišťování informace o zkoumaném objektu. Experimentální technika zahrnuje měřicí, výpočetní a řídicí techniku. Integroující disciplínou ve vztahu k vyšetřování a řízení složitých systémů je technická kybernetika. Experimentální údaje jsou symbolicky vyjádřené výsledky experimentu. Plán experimentu zahrnuje algoritmus experimentu, postup práce, charakter a velikost změn proměnných veličin aj. Algoritmem experimentu se rozumí soubor jednoznačných předpisů, jejichž splnění vede k uskutečnění cíle experimentu. Počítač má v experimentu tři základní role: hromadně zpracovává experimentální data, slouží jako modelový prostředek a řídí proces experimentování.

2.1.3 Modely

Podle charakteru modelového systému se modely rozdělují na abstraktní a fyzické. Získávání informací je u obou modelů odlišný. Fyzický model dynamického systému použitý k simulaci se nazývá simulační model. Abstraktní modely jsou objektivní z hlediska svého obsahu, tzn. Že vyjadřují zákonitosti reálného světa a popisují jej. Mezi abstraktní modely patří především matematické modely a modely vyjádřené programovacími jazyky, vývojovými diagramy aj.

2.2 Kybernetika a modelování

„Kybernetika je teorie modelů ve vědě bez výrazných hranic mezi fyzikálními, biologickými a ekonomickými procesy.“ (S. Bear) Kybernetika výrazně přispěla ve zdokonalení tradičních metod modelování a zavedením počítačového modelování. Obohacení spočívá v ovlivnění způsobu myšlení, ve vysoce rozvinuté abstrakci, ve schopnosti řešit kvalitativně vyšší úlohy optimalizace a řízení složitých dynamických systémů s využitím teorie umožňuje syntézu poznatků z různých oborů, vzájemné ovlivňování oborů a tím i jejich urychlený rozvoj. Důležitým pojmem v kybernetice a modelování je teorie systémů.

Teorie systémů

Se zabývá obecnými otázkami matematického modelování. Základním nástrojem v teorii systémů je abstrakce, modelování a analogie. Z teorie systémů vyplývá pojem systém.

Systémem se rozumí množina prvků, spojených navzájem i s okolím interakcemi, jež vytvářejí vlastnosti umožňující splnění účelových i cílových funkcí. Vnitřní uspořádání systému a vzájemné vazby vnitřních složek se nazývá strukturou.

Formální popis systému ve tvaru $S(X, R)$, kde

$X = \{x_1, x_2, \dots, x_n\}$ množina prvků

$R = \{r_1, r_2, \dots, r_n\}$ množina vztahů mezi prvky

Obecně se dá říct, že každý prvek systému je sám systémem a označuje se jako subsystém. Tím jsme tedy schopni získat celou systému hierarchii. Pro modelování a simulace složitých systémů má hierarchický přístup nejen metodologický, ale i praktický význam.

2.3 Matematický model

Definice: „Matematický model je matematické vyjádření podstatných vlastností existujícího nebo konstruovaného systému, který popisuje znalosti o systému v použitelné formě.“

Matematický model je složen ze stavů procesů, které jsou charakterizovány stavovými veličinami jakými jsou tlak, teplota, množství či koncentrace, atd. Tedy pod pojmem matematický model procesů či zařízení si představujeme matematické vztahy mezi těmito stavovými veličinami v závislosti na čase. Matematický model vždy představuje zjednodušenou abstrakci studované reality. Tomuto procesu se také říká dynamický proces. V matematických modelech se většinou matematické vztahy vyjadřují pomocí diferenciálních rovnic a jejich soustav nebo pomocí nelineárních a lineárních rovnic a jejich soustav. Pro účely automatizace se také často používá přenosových funkcí. Tyto rovnice a vztahy se však stávají matematickým modelem, neboť neexistuje jednoznačné přiřazení ke konkrétnímu procesu nebo jevu. Matematický model může vzniknout jako výsledek přímého zkoumání a pozorování, nebo dedukcí, kdy model vzniká jako dílčí případ obecnějšího popisu

Samotné vytváření matematických modelů můžeme rozdělit do dvou základních kategorií, stochastické nebo deterministické modely.

Stochastický model

Předpokladem stochastického modelu je přesná definice vstupů a výstupů, přičemž samotný model si představujeme jako „černou skříňku“. Stochastické modely mají danou strukturu. Strukturou rozumíme řád a zvolený typ diferenciální či diferenční rovnice (lineární, nelineární diferenciální rovnice, typ nelinearity atd.), nebo soustavu těchto rovnic, obrazový nebo diskrétní přenos se zvolenými stupni polynomů v čitateli a jmenovateli, nebo přenosovými maticemi zvoleného rozměru. Stochastické modely představují soustavy rovnic a algebraické vztahy, které obsahují konstanty - parametry těchto rovnic a vztahů.

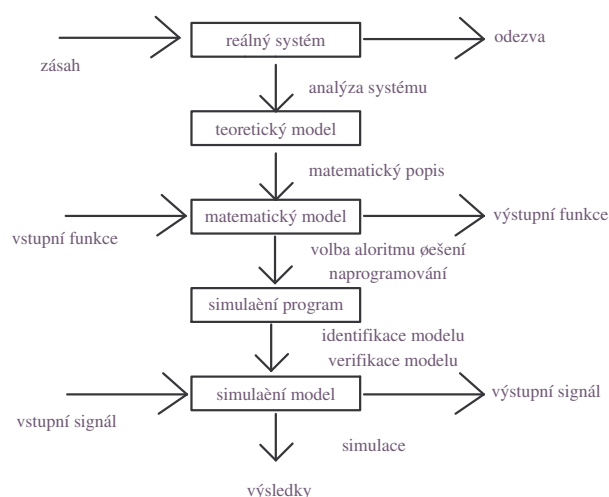
Postup vytváření modelu:

- matematický popis zvolíme podle předpokládaného chování systému, popis obsahuje parametry neznámých funkcí.
- Na vstup přivádíme v krátkých, pravidelných intervalech proměnlivý signál a hodnoty na vstupu a na výstupu měříme tak dlouho dokud nezískáme dostatečný popis systému
- Naměřené hodnoty zpracujeme tak, abychom získali parametry neznámých funkcí v matematickém popisu

Tohoto postupu používáme tehdy, když nejsme schopni odvodit deterministický model.

Deterministický model

Deterministický model získáváme z analýzy a matematického popisu skutečné podstaty děje. Každý děj v modelu si řídí podle určitých fyzikálně – chemických zákonů, přičemž každý tento zákon lze matematicky vyjádřit. Deterministické modely představují funkční závislosti mezi vstupním a výstupním signálem. Toto matematické vyjádření je podstatné pro sestavení deterministického modelu. Ve většině aplikací je ovšem toto vyjádření příliš složité a nebo samotný děj neznáme natolik podrobně. Proto se ve většině aplikací omezuje na popis pouze těch dějů, které vystihují podstatné vlastnosti a děje, pro které byla aplikace určena. Ostatní děje můžeme zanedbat, jelikož kritériem použitelnosti modelu je souhlas s realitou ve vlastnostech pro které byl model navržen. Výsledná závislost se vyjadřuje buď graficky nebo pomocí tabulky, která číselně popisuje závislost. Konstanty – parametry jsou obsaženy ve výsledných funkčních závislostech, a lze je získat až po výpočtu pro daný model.



obrázek 1 Postup návrhu

Analýza systému

Představuje nenáročné matematické operace jednotkami nebo rozměry fyzikálních veličin, jejichž cílem je získání bezrozměrových kritérií nebo funkčních závislostí mezi působícími veličinami. Někdy lze určit i strukturu zobecněných proměnných. Cílem analýzy je tedy rozložení problému na dílčí fyzikální úlohy. Úplné pochopení do fyzikální podstaty problému však není možné z pouhé znalosti působících veličin a jejich rozměrů.

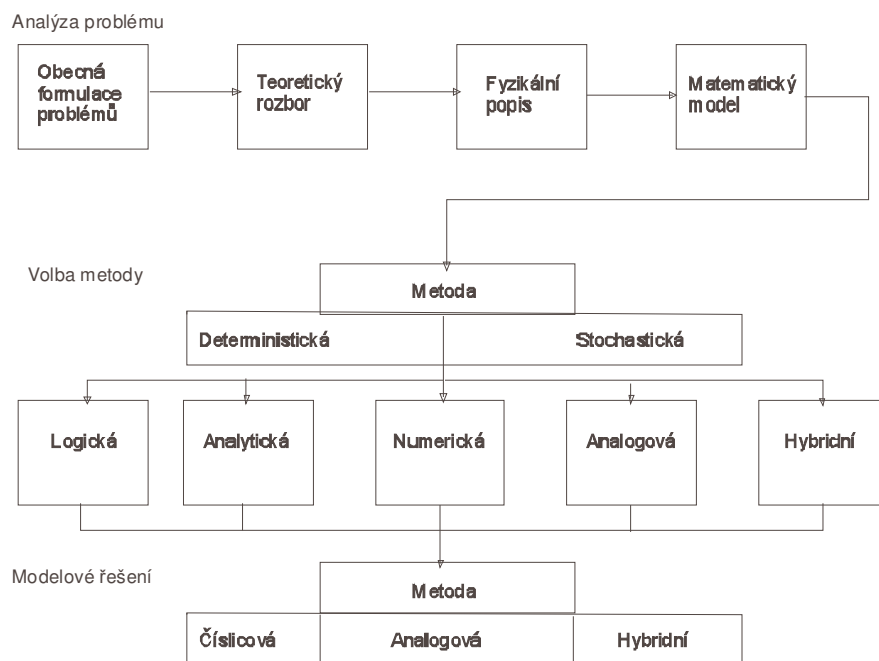
Analýza systému spočívá v následujících krocích

- specifikace dějů, které probíhají v procesu a určení jejich podstaty
- vymezení vlivů, které ovlivňují samotný průběh procesu
- určení veličin, které popisují samotný proces
- určení podstatných prvků, které budeme považovat za podstatné a které zanedbáme
- u každého prvku rozhodneme zda-li jej budeme uvažovat v celém rozsahu a ve všech souvislostech, a nebo použijeme některá zjednodušení

Výsledkem těchto kroků je teoretický model. Teoretický model sice nepopisuje zcela přesné skutečné děje, ale přináší nám jednoduché a přehledné řešení výsledných rovnic.

V posledním kroku analýzy systému jsme hovořili o zanedbávání a zjednodušování některých vztahů. Tento krok má zcela zásadní vliv na kvalitu výsledného modelu. Toto zjednodušování se řídí několika předpoklady.

- Složité systémy se rozdělují na menší subsystémy, s ohledem na jejich vzájemné souvislosti, které zpracováváme jednotlivě
- Zavádí se z idealizované formy(představy), např. ideální plyn, ideální kapalina
- Předpoklad nezávislosti materiálních vlastností na teplotě
- Předpoklad homogenních a isotropních materiálů
- Při současném průběhu rychlých a pomalých dějů předpokládáme u rychlých dějů dosažení rovnovážného nebo ustáleného stavu
- Ztráty jsou zanedbávány
- U nelineárních závislostí je snaha je linearizovat
- Zavádí se korekční koeficienty
- Ve výpočtech se používají empiricky zjištěné vztahy a závislosti mezi veličinami
- Volí se vhodné souřadnicové systémy, pro co nejjednodušší popis geometrického systému
- Odstraňují se závislosti sledovaných veličin na souřadnicích, zaváděním předpokladů



Obrázek 2 Schéma postupu při analýze problému, volbě metody a modelovém řešení

Na základě teoretického modelu se sestavuje matematický model. Cílem této fáze je vytvoření obecného matematického modelu procesu, přičemž využíváme fyzikálně-chemické zákony a vztahy. Může se ovšem stát že se nám nepodaří náš teoretický model popsat pomocí matematických vztahů, potom nám ovšem nezbývá než pozměnit náš teoretický model, nebo můžeme využít experimentálně zjištěných výsledků. Samotné sestavení matematického modelu spočívá:

- Výběr matematického popisu zákonitostí použitých v teoretickém modelu
- Vytvoření modelových rovnic, tzn. doplnění předpoklady a úprava matematických rovnic
- Určení počátečních a okrajových podmínek

Dalším krokem postupu je vytvoření simulačního programu.

- Zvolíme metody modelových rovnic
- Nalezení vhodného algoritmu řešení modelových rovnic pro řešení na počítači
- Sestavení výpočetního programu

Konečnou fází je vytvoření počítačového programu, který se obecně nazývá simulační model. Tato část postupu zahrnuje

- identifikace modelu, tj. nalezení neznámých parametrů modelu
- verifikace modelu, tj. ověření správnosti modelu pro jeho skutečné použití

Výsledkem tohoto postupu je vyhovující nebo nevyhovující model. Při vyhovujícím modelu můžeme samotný model prakticky používat, pokud je ovšem model nevyhovující je nutné navrhnout nový teoretický model, a s využitím získaných poznatků se snažíme najít chyby kterých jsme se dopustili při zjednodušování a zanedbávání parametrů. Celý postup je nutné zopakovat s novými hodnotami parametrů.

Hlavním úkolem celého tohoto procesu je důkladné ujasnění podstaty mechanismů studovaného děje.

2.3.1 Fyzikální význam transformací matematického modelu

Při transformaci matematického modelu se mění vlastní model vzhledem k prostoru nebo času, ale objevují se prvky podobnostní a informační. Fyzikální procesy popsané parciálními diferenciálními rovnicemi 2. řádu se např. Laplaceovou transformací převádí podobnost vnitřního procesu na podobnost ve vnějším chování. Podobně se obyčejné diferenciální rovnice převádějí díky transformaci na rovnice algebraické.

Konečné integrální transformace se vyznačují tím, že obsahují informaci o geometrickém transformací matematického modelu vede k ustředění hledané fyzikální veličiny v prostoru.

Diferenciální Taylorova transformace se používá při řešení úloh popsaných soustavami rovnic.

2.4 Počáteční a okrajové podmínky

Jak již bylo zmíněno, při popisu modelů se často využívají diferenciální rovnice, které ovšem musí být doplněny počátečními a okrajovými podmínkami. Formulace těchto podmínek je nedílnou součástí popisu každého matematického modelu. Pro každou nezávisle proměnnou potřebujeme tolik vzájemně nezávislých podmínek, jaký je nejvyšší v rovnicích se vyskytující řád derivace podle této proměnné.

Počáteční podmínka definuje situaci na počátku měření a okrajové podmínky se vyskytují tam, kde souřadnice vystupují jako nezávislé proměnné. V publikaci (1) jsou popsány 4 typy okrajových podmínek:

- 1. druhu -Dirichletova podmínka: „Hodnota závisle proměnné v místě x_0 je známou funkcí ostatních souřadnic a času.“
- 2.druhu - Neumannova podmínka: „Hodnota derivace závisle proměnné podle jedné souřadnice (např. podle x v bodě x_0) je známou funkcí ostatních souřadnic a času.“
- 3. druhu -Newtonova - Fourierova podmínka: „Hodnota lineární kombinace hodnoty závisle proměnné u v bodě x_0 a její derivace podle x v místě x_0 je známou funkcí ostatních souřadnic a času. Konstanty a, b jsou koeficienty lineární kombinace.“
- 4. druhu – Vyjadřuje podmínku rovnosti plošných toků energie na rozhraní mezi dvěma oblastmi v dokonalém styku, mající různé fyzikální vlastnosti.

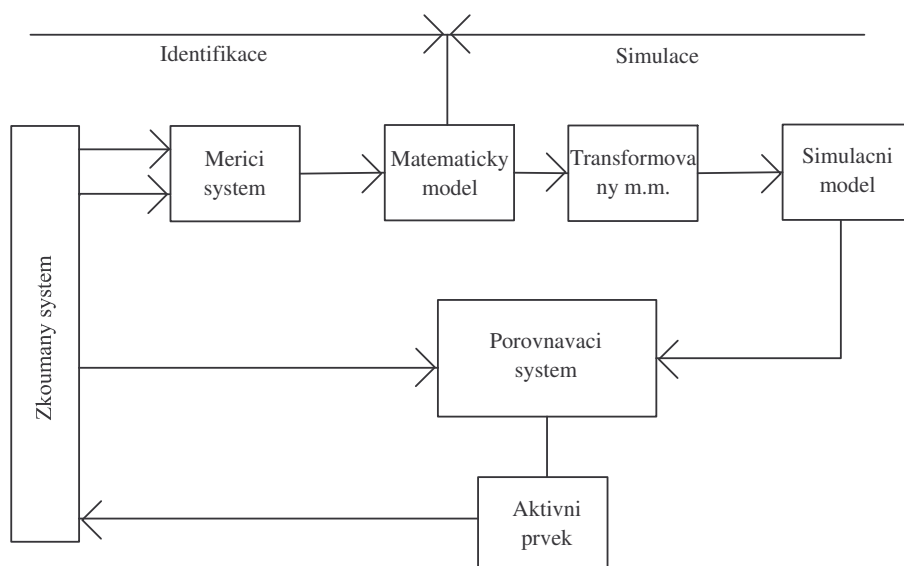
3 SIMULACE

Modelování dynamických systémů s přímým nebo nepřímým zpětným působením na zkoumaný objekt se nazývá simulací. Z definice vyplývá, že pojem simulace lze vždy nahradit obecnějším pojmem modelování, obráceně však nikoliv.

Pro účely matematického modelování rozumíme pojmem „simulace“ řešení rovnic matematického modelu. Je nutné poznamenat, že simulace není výpočet, jelikož se tyto dva pojmy navzájem zaměňují. Výpočet představuje pouze mechanický sled výpočetních operací, aniž by vyjadřoval vztah k řešenému procesu. Snažíme se nastavit vstupní veličiny tak abychom na výstupu získali požadované výstupní hodnoty. Při práci s dynamickými modely to obnáší řešení diferenciálních rovnic a jejich soustav. Analytického řešení se také využívá, ale většinou pouze u jednodušších aplikací. V dnešní době je k dispozici velké množství simulačních jazyků a knihoven a proto se většinou volí numerické řešení. Kvůli správné interpretaci výsledků je vhodné znát principy numerického řešení diferenciálních rovnic a o přesnosti výsledků.

3.1 Identifikace

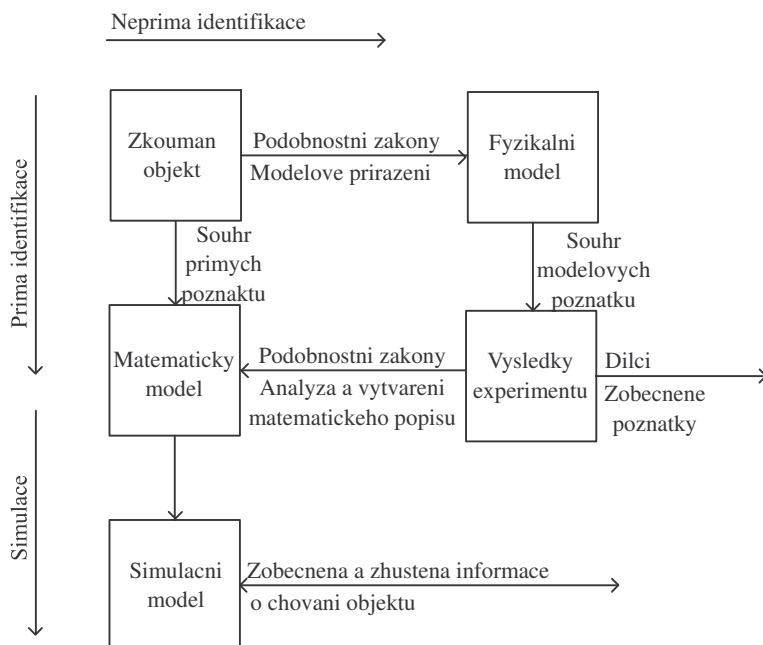
Při vyjadřování závislostí veličin v modelu, nebo vytváření samotného matematického modelu, se často používají konstanty – parametry, jejichž hodnotu předem neznáme, a proto je nutné je nějakým způsobem určit. Tomuto určování konstant říkáme identifikace systému. Identifikace je experimentální způsob určování podstatných charakteristik systémů a procesu umožňující sestavení matematického modelu. Charakteristikami se systému se rozumí zejména různé fyzikální vlastnosti, struktura a parametry systému a v něm probíhajícího procesu. Identifikaci systému si můžeme také představit jako proces definování systému na objekty. Cílem identifikace je rozložení modelu na samostatné objekty, s kterými se poté pracuje samostatně. Na jednotlivých objektech uvažujeme pouze ty veličiny, které nás zajímají a nebo jsou významné pro simulaci.



Obrázek 3 Identifikace a simulace v procesu poznání zkoumaného systému

Podstatné informace o chování systému jsou získány přímým měřením pomocí měřícího systému. Cílem procesu identifikace je formulace matematického modelu. Matematický model se transformuje do tvaru umožňujícího vytvořit zvolenými technickými prostředky model simulační.

K vytvoření matematického modelu existují v podstatě dvě cesty. Přímá a nepřímá identifikace.



Obrázek 4 Schéma postupu při přímé a nepřímé identifikaci a vytváření matematického a simulačního modelu.

Přímá identifikace vychází ze souhrnu poznatků o chování zkoumaného objektu získaných jednak identifikací, jednak jako výsledek rozvoje poznatků v příslušném oboru. Ve většině případech se informace získávají nepřímo na fyzikálním modelu. Výsledky experimentu jsou vyjadřovány nejčastěji ve tvaru fenomenologického popisu fyzikálního modelu, a v důsledku modelového přiřazení, i popisu zkoumaného objektu.

Celý tento proces se provádí proto, že není možné zkoumat celou objektivní realitu modelu v celé své složitosti, a proto se omezuje pouze na části – objekty.

Identifikace vycházející z matematicko – fyzikální analýzy se označuje jako analytická identifikace, identifikace vycházející z měření na systému se nazývá experimentální identifikace

Pomocí experimentální identifikace můžeme získat:

- Funkční závislost mezi vstupním a výstupním signálem, nebo průběh dynamické charakteristiky
- Koeficienty – parametry modelu

Pokud chceme dosáhnout souhlasu ODR s empirickými daty je nejlepší volbou pro stanovení konstant metoda optimalizace. Z matematiky jsou známy metody aproximace empirických hodnot pomocí metody nejmenších čtverců.

Optimalizace je matematický postup, při kterém hledáme nejlepší variantu řešení. Ve skutečnosti ovšem hledáme extrém funkce. Této funkci se často také nazývá kritérium optimalizace. Kritérium optimalizace musíme předem určit z matematického modelu takovým způsobem abychom po optimalizaci byli schopni z množiny řešení vybrat to nejlepší, které je pro nás v nějakém smyslu nejlepší. Kritérium nám pomáhá vybrat nejlepší možné řešení. Pro vyřešení optimalizace matematického modelu je nutné upřesňovat model a kritérium podle výsledku. Kritérium nastavujeme pomocí optimalizačních proměnných, které jsme si zvolili z matematického popisu.

Je zřejmé, že model reálné situace je vždy zjednodušen. Již v této fázi se můžou vyskytnout problémy s nepřesným popisem reálného děje.

Cílem optimalizace je nalezení extrému, buď maxima nebo minima, což je závislé na úloze kterou řešíme. Extrém hledáme v množině optimalizačních proměnných, přičemž v této množině může být více extrému, v takovém případě hovoříme o několika lokálních extrémech a jenom jednom

globálním extrému, který je tak jediným správným řešením, které hledáme. Již zmíněné lokální extrémy představují značný problém při hledání globálního extrému, protože v něm může optimalizační postup uvíznout, a to znamená že nejsme schopni nalézt globální extrém. K vyřešení tohoto problému můžeme použít buď Gaussovu metodu nebo metodu polyedrického hledání. U Gaussovy se používají „nástřely“, které představují počáteční hodnoty každé optimalizace, pokud tedy uvízneme v některém lokálním extrému, zvolíme prostě jiný nástřel.

U polyedrického hledání spočívá princip v určení směru hledání n -rozměrného prostoru. Tyto body se volí ve vrcholu pravidelného $n+1$ - rozměrného polyedru.

Rozhodovací situace

Je to moment při kterém je potřeba se rozhodnout. Existuje řadu dějů, které mají matematický model popsaný staticky, popsaný pouze algebraickými vztahy a veličinami, které jsou časově nezávislé. Práce se staticky modely je mnohonásobně jednodušší a proto se i dynamicky popsané modely často zjednodušují, tak aby vznikl děj statický. Řešení statických optimalizačních problémů, u nichž čas nevystupuje jako nezávisle proměnná se nazývá matematické programování.

Jsou-li vztahy mezi veličinami a kritériem lineární, dostaneme problém lineárního programování.

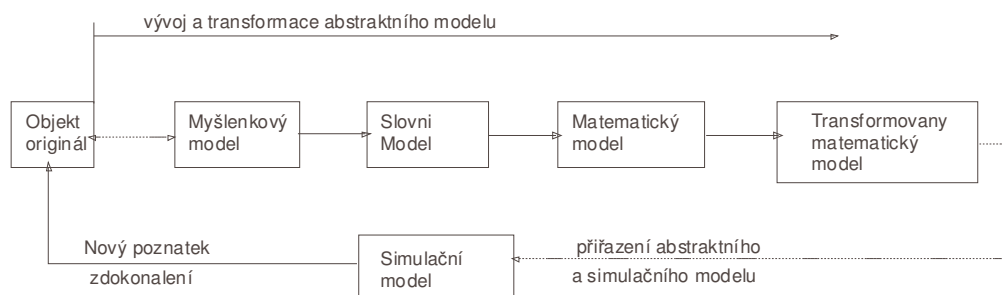
Obecně vede tento typ úlohy na problém nelineárního programování.

Jedná-li se o model dynamického systému, mluvíme pak o problému dynamické optimalizace. Protože pro statické úlohy nelineárního programování byla vypracována řada účinných numerických metod, snažíme se převést úlohy dynamické optimalizace na statický problém.

Pomocí diferenciálních rovnic se dají řešit úlohy dynamické optimalizace, pomocí kterých dostáváme rekurentní vztahy vhodné pro numerický výpočet.

3.2 Simulační modely

Mezi základní pojmy modelování je nutné si povšimnout procesu transformace abstraktního modelu na model matematický a simulační a uplatnění simulačních modelů.



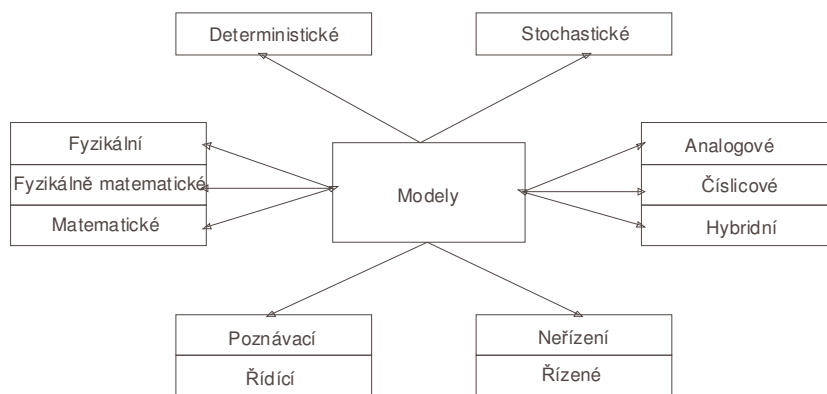
Obrázek 5 : Vývoj abstraktního modelu při vytváření simulačního modelu

Jako první krokem při vývoji abstraktního modelu je myšlenkový model jako odraz zkoumaného světa. Dalším krokem je model slovní a především model matematický. Po volbě metody algoritmu a dalších transformací včetně simulačního prostředku vzniká simulační model. Zpětná cesta (čárkovaná) zdůrazňuje cíl simulace směřující ke zpřesnění a zdokonalení objektu.

Rozdělení modelů

Modely lze rozdělit do několika hledisek:

- Podle charakteru procesu
- Podle hlediskové podobnosti
- Podle způsobu zpracování modelové informace
- Podle hlediskového účelu modelu
- Podle hlediska řízení modelového procesu



Obrázek 6 Rozdělení modelů podle charakteru modelového procesu

Postupem doby práce s počítačovými simulacemi se ukázalo, že by bylo mnohem výhodnější vytvořit speciální programovací jazyk, který by přinášel rychlý a snadný způsob pro tvorbu simulačních programů, než používání klasických programovacích jazyků (C, Fortran, Angol, Assembler). Takto vzniklé programovací jazyky sloužící pro tvorbu počítačových simulací se nazývají simulační jazyky. Cílem těchto simulačních jazyků je poskytovat různé prostředky, umožňující popis simulačního programu v takové formě, aby připomínala popis simulovaného systému a to tak aby malé změny v popisu systému odpovídala malým změnám v popisu simulačního programu. Simulační jazyk musí poskytovat vhodný popis vlastního simulátoru, ale i situace, v níž se nacházejí během pokusu – ustálený stav. Obvykle se při programování simulačního programu programují i podaplikace zpracovávající získané data během simulace a jejich reprezentace na výstupní zařízení.

4.1 Sémantika simulačních jazyků

Abychom pochopili logickou návaznost simulačních jazyků, pokusíme se tyto jazyky charakterizovat obecně.

Sémantika jazyka vyjadřuje smysl textů, které v něm byly utvořeny podle správných syntaktických postupů. Každý simulační jazyk používá kompilátor pro překlad napsaného zdrojového kódu, který je srozumitelný člověku – programátorovy do formy srozumitelné počítači, většinou ve formě spustitelného souboru. Každý kompilátor má přesně definovanou sémantiku, která se musí zcela striktně dodržovat aby kompilátor byl schopný naprogramovaný zdrojový kód přeložit.

Sémantika simulačního jazyka je charakterizována dvěma faktory:

algoritmy simulačního jazyka operují s dynamickými systémy, dynamické systémy lze v tomto případě chápat jako objekty, které v průběhu simulace vznikají a zanikají, ale jejich nejdůležitější charakteristikou je, že se v průběhu simulace mění.

Triviální algoritmy, oproti klasickým programovacím jazykům mají tyto algoritmy v simulačních jazycích zcela specifické postavení. Jejich úkolem je pouze vytvořit dynamický systém a poté již pouze sledují hodnoty parametrů nebo atributů, nebo jen ukončují jeho existenci.

Závěrem tohoto odstavce je nutné zdůraznit princip simulačních jazyků. Účelem je tedy formulovat text do takové podoby, která by charakterizovala objekty do množiny dynamických systémů. Tyto dynamické systémy jsou ovšem pouze abstraktní a existuje jen v našem vědomí.

4.2 Třídění simulačních jazyků

Třídění simulačních jazyků se provádí v závislosti na dynamických systémech.

- Intuitivní třídění simulačních jazyků
- Permanentní prvky, tyto prvky existují po celou dobu běhu simulace, tyto prvky se taky můžou nazývat aktivitami.
- Přechodné prvky, entity, tyto prvky vstupující, resp. Vystupující ze systému do okolí v průběhu simulace. Těmto prvkům říkáme též transakcemi.

V závislosti na tomto intuitivním třídění dělíme dynamické systémy na systémy A, T, AT. Kde A znamenají aktivity, a T transakce. Podle tohoto dělení dělíme i simulační jazyky na typu stejného typu (A, T, AT). Toto dělení má ovšem význam až po určitém zjednodušení systému, protože prakticky všechny systémy v obecném tvaru jsou typu AT, což znamená že obsahují jak aktivity, tak i transakce. Aby tedy rozdělení simulačních jazyků bylo efektivní musíme provést vhodnou abstrakci a některé prvky vyloučit, čímž získáme simulační model jiného, většinou jednoduššího tvaru.

Dalším kritériem, podle kterého se simulační jazyky dělí, jsou typy charakteristik se kterými simulační jazyk pracuje:

- Jazyky pro spojitou simulaci
- Jazyky pro kombinovanou diskrétně spojitou simulaci
- Jazyky pro diskrétní simulaci

4.3 Syntaxe simulačních jazyků

Simulační jazyky musí umožňovat vhodným způsobem popis dynamických systémů. Přitom ovšem nevystačíme s klasickými matematickými nástroji, k tomu se někdy využívají diferenciální rovnice a operátory. Klasické matematické nástroje totiž nestudují tak složité systémy jako simulace a nemá prostředky pro strukturování složitých informací.

Simulace zavádí zcela nový druh popisu, tzv. procesy, které slouží k popisu složitých dějů a to tak, že tyto složité děje popíšeme několika vzájemně se ovlivňujícími se jednoduššími ději - procesy. Popis děje v dynamickém systému, který vystupuje v simulačním programu, se strukturuje na několik procesů.

I když si pomocí procesů značně ulehčíme práci, je toto rozdělení dějů na procesy u velmi složitých simulací nedostatečné, proto se simulační jazyky snaží najít v jednotlivých procesech najít podobnost a podle této podobnosti definovat obecné třídy, které obsahují tzv. typické děje. Díky těmto třídám jsme schopni v systému vytvořit objekty, aniž bychom museli deklarovat jejich děje a charakteristiky.

Hlavní program vytvořený v nějakém simulačním jazyku můžeme degenerovat na dvě části. Úsekem počátečních podmínek a koncový úsek. Z názvu těchto úseků je patrné kdy se které část programu použije, buďto na začátku simulace nebo na jejím konci.

Je nutné si uvědomit co všechno lze považovat za simulační jazyky a co již je pouze simulační program. Simulační programy jsou aplikace ze kterých lze udělat simulátor pouhým dosazením dat. Tyto aplikace jsou ve většině případech naprogramovány pomocí klasických programovacích jazyků s využitím některých simulačních programovacích knihoven. Navíc tyto aplikace jsou většinou vytvořeny pouze pro konkrétní problém a nebo velmi úzce specifikované typy dynamických systémů.

4.4 Prostředky simulačních jazyků

Zde si uvedeme některé základní prostředky simulačních jazyků, které se vyskytují téměř ve všech simulačních jazycích, ale v klasických programovacích je najdeme jen stěží.

Plánovací příkazy

Každý simulační jazyk obsahuje prostředky pro provádění výpočtů zapsaných v textu simulačního jazyka. Jako příklad uvedu nutnost přerušit sekvenční běh simulace v určitém okamžiku a po určité době nebo po splnění určité podmínky opět započat výpočet. Existují tři druhy plánovacích systémů.

-Interrogativní, je to plánovací systém řízen příkazy, které čekají na splnění podmínky. Nejčastěji se používá konstrukce „WaitUntil“. Příkaz čekání na splnění podmínky se nazývá interogativním plánovacím příkazem. Tento způsob je sice nejobecnější, vychází z něj ostatní plánovací systémy řízení, ale zároveň s tím je i nejpomalejší, protože je nutné provádět značné množství zbytečných instrukcí.

-Imperativní, je to plánovací systém řízen příkazy, které čekají po určitou dobu. Tyto příkazy mají většinou tvar typu „Watt“, „Hold“, „Delay“... Tyto příkazy jsou speciálními tvary interogativních příkazů.

-Automatový, jsou to spíše parametry časového zpoždění, který mají všechny prvky simulovaného systému stejný. Tzn. že v určitém okamžiku se všechny prvky zaktivizují nebo naopak deaktivují, vzhledem k časovému běhu simulace

Prostředky pro řešení konfliktních situací

Simulační jazyky musí poskytovat prostředky pro definování výpočtů v momentech, kdy je nutné ve stejném okamžiku provést výpočet různých prvků. Těmto okamžikům se říká konfliktní situace. Tento problém se řeší tak, že se musí provést výpočet aniž by došlo ke změně simultánního času.

Konfliktní situace nastávají v několika situacích, například při dopravním zpoždění, nebo když některé děje trvají jen velmi krátkou dobu, takže se tato doba považuje za nulovou nebo dokonce pokud určitý děj se v simulovaném systému vůbec nevyskytuje, tato konfliktní situace nastává v simulačních jazycích, které uvažují čas jako celočíselnou proměnnou.

Pro odstranění těchto situací se nejčastěji využívá přidělování priorit jednotlivým prvkům. Díky těmto prioritám můžeme nejprve vyřešit ty děje, které jsou podstatnější pro naši simulaci.

Prostředky pro generování a likvidaci transakcí

Každý simulační jazyk, který podporuje práci s transakcemi, se musí vypořádat s jejich generováním.

- Dělení transakcí, jedna transakce způsobí vznik nové transakce stejné třídy s podobnými parametry. Tento proces je obdobný jako když se dělí buňky.
- Generátor transakcí, představa je taková, že máme k dispozici objekt (zdroj), který má za úkol generovat transakce.

Prostředky pro zahájení simulace a opakování simulačních pokusů

Simulační jazyky musí umožňovat definovat počátek simulace popřípadě jeho opakování. Nejčastěji se pro začátek simulace používá nejjednodušší způsob, a to je prostě započítí vykonávání aktivit. Při tomto způsobu sice vznikají často konfliktní situace, ale tento problém řeší priority. Jiný způsob začátku simulace se využívá pro simulaci spojitých systémů, v těchto simulacích se provádí automaticky pro kompilaci projektu. U diskretních systémů jsou některé aktivity blokovány a

neodblokuje se do té doby, dokud nebude splněna některá z jejich vnitřních podmínek. Tohle jsou ovšem pouze obecné způsoby začátku simulací. U konkrétních simulačních jazyků se můžou vyskytnout speciální způsoby, které překračují rámec této práce. Pro podrobnější obeznámení s problematikou odkazují na publikaci [3].

4.5 Jazyky pro simulaci spojitých systémů

Tyto jazyky obsahují většinou pouze aktivity. Tyto jazyky lze využívat v řadě různých aplikačních oblastí, např. v oblasti mechanických soustav (Dyana, Super Sceptree, FmsSim), v oblasti elektronických soustavách (NET, TRAC, ECAP) nebo v oblasti chemických simulací (DigSim, ChemCSMP, SimComp)

Jazyky blokově orientované. Popis bloku má tvar $CALL(n, p_1, p_2, \dots, p_n)$

kde: n – identifikátor

$p_1 \dots p_n$ – jsou parametry, vstupní reference, a indexy výstupních parametrů.

Jazyky spojově orientované. Většinu textu v něm napsané tvoří popisy spojů. Obecný tvar není možné napsat, protože spojitý popis je pro každý simulovaný objekt jedinečný.

Jazyky simulace analogové výpočetní techniky. Když bylo v historii dosaženo dostatečného výkonu číslicových počítačů, začaly se tyto počítače používat k vědecko-technickým výpočtům. Mezi základní výpočty patřily soustavy obyčejných diferenciálních rovnic.

Praxe ovšem ukázala, simulační jazyky musí obsahovat prostředky nejen pro blízký popis analogových počítačů, ale musí být vybaveny i prostředky algoritmů klasických jazyků. Simulační jazyky s tímto vybavením můžeme rozdělit do třech hlavních skupin.

- Prvky simulujícího systému musí být nějak pojmenovány, a to buď číslem (id) nebo jednoduchým identifikátorem. Každý blok je popsán v jednom řádku, kde je uvedeno jméno (index) bloku, jeho funkce a jeho vstupní parametry. Někdy jsou parametry reprezentovány konstantami, které připomínají potenciometry v analogové technice. Simulační jazyky této skupiny jsou využitelné pouze pro ty odborníky, kteří jsou zvyklí na práci s analogovými počítači.
- Texty těchto simulačních jazyků připomínají tradiční programovací jazyky algoritmického typu. Odborníci využívající tyto simulační jazyky nemusejí vědět nic

o analogové technice nebo o jeho metodice. Zajímavou myšlenku, kterou využívají tyto jazyky pro úpravu diferenciálních rovnic, je že se vyšší derivace nahradí obyčejnými proměnnými, a poté se rovnice řeší jako rovnice s více neznámými.

- Simulační jazyky této skupiny umožňují napsat diferenciální rovnice bez přepisu na integrální tvar.

Zdrojový kód spojitých simulací má vždy určitou strukturu. Tato struktura obsahuje úsek počátečních podmínek, úsek hlavní a úsek koncový. U většiny simulačních jazyků jsou úseky počátečních podmínek a koncové různě omezeny a standardizovány. Naopak hlavní úsek není nijak omezen ani standardizován, to ovšem vede v některých případech ke špatným výsledkům, tyto chyby jsou většinou způsobeny špatným řazením instrukcí, proto je většina simulačních jazyků vybavena automatickým řazením instrukcí, a to proto aby se programátor nemusel starat o toto řazení. V praxi se vyskytuje mnoho způsobů realizace tohoto řazení. Mezi tyto způsoby patří „řazení hlavního úseku se provede vždy“, nebo je u některých simulačních jazyků možné rozdělit hlavní úsek na podúseky a definovat, které podúseky se mají být řazeny a které ne. Pokročilejší jazyky umožňují dokonce řadit celé podúseky. Některé jazyky, tvoří ovšem výjimku, nemají možnost automatického řazení.

4.6 Jazyky pro simulaci diskrétních systémů

Hlavním požadavkem na diskrétní systémy je, že musí obsahovat jak aktivity tak i transakce a musí umožňovat popis interakcí mezi aktivitami a transakcemi.

Všechny diskrétní simulační jazyky rozdělujeme podle způsobu popisu systému a jeho prvků. Podle tohoto hlediska dělíme simulační jazyky pro diskrétní systémy na

- Blokově orientované
- Instrukčně orientované

Jednodušší možností pro méně zkušené programátory jsou blokově orientované simulační jazyky, které umožňují prostředky pro formulaci modelů i pro popis dějů. V dnešní době jsou všechny blokově orientované jazyky jsou téhož typu a jsou odvozeny od jazyku GPSS.

Instrukčně orientované jazyky předpokládají určité zkušenosti s jazykem FORTRAN nebo ALGOL. Instrukčně orientované jazyky vycházejí z několika rozdílných jazykových struktur. Mezi hlavního představitele těchto obecných jazyků je jazyk SIMSCRIPT a jeho druhá verze SIMSCRIPTII, který je ovšem značně odlišný od svého předchůdce.

4.7 Seznam významných simulačních jazyku

SimPy – Simulační jazyk vybudovaný na jazyku Python. Je objektově orientovaný a je určen pro programování události v diskrétních systémech.

JiST – Je to vysoce výkonný engine pro simulace diskrétních událostí běžící na standartu Java virtual machine.

ACSL Sim – jazyk pro simulace spojitých systémů.

SLAM II – vizuální programovací jazyk.

GPSS/H – obecný účelový simulační systém. Jazyk pro diskrétní simulace.

Ptolemy Project – jedná se o vědecký projekt, který má za úkol implementovat vysokou podporu pro signální procesy, komunikaci a real-time řízení.

DSDS – simulátor pro diskrétní události.

MODSIM III – jazyk pro simulaci dvou objektů a procesů.

Simscript – podrobně probrán na předcházejících stránkách.

Simple I – podporuje modelování diskrétních a spojitých systémů.

SEMoLa – je výjimečný tím, že se jedná o neproceduární jazyk, pro modelování spojitých událostí.

Pasion – jazyk vytvořený v Pascalu. Tento jazyk má proces/událostní strukturu.

WinSAAM – modelovací jazyk složený ze dvou jiných jazyků. Využívá se především pro biologické procesy.

MathCore – jedná se o jazyk který rozšiřuje možnosti programu Mathematica.

5 SIMULAČNÍ KNIHOVNY

C++Sim – knihovna napsaná v jazyce C++, pro simulaci události v dynamických systémech

SimLib – knihovna vyvíjená na VUT v Brně, psána v jazyce C.

JavaSim – obdoba knihovny C++Sim, napsaná pro programovací jazyk Java

DSOL – knihovna napsaná v jazyce Java, šířitelná pod záštitou projektu Open Source. Knihovna se používá simulaci událostí ve spojitých tak i diskrétních systémech

SimKit – balíček, komponenta (package), který slouží pro vytváření událostí pro diskrétní systémy.

NAG – velice známá a užívaná knihovna matematických funkcí a algoritmů napsaných v jazyce C

Základní informace o této knihovně, nebo spíše souboru knihoven lze najít v publikaci (26).

NetLib – další knihovna obsahující matematické funkce a algoritmy

SSS – knihovna v jazyce C, pro simulace událostí v diskrétních systémech. Jedná se o jednodušší knihovnu, velice vhodná pro inspirování pro tvorbu vlastní knihovny.

BaseSim – Soubor knihoven rozšiřující programovací prostředí Borland Delphi o řadu vizuálních komponent sloužících pro simulaci diskrétních systémů

HighMAST – Komerčně distribuované knihovny a pracovní prostředí pro nejrůznější simulace.

Knihovna je napsaná v programovacím jazyku C# na architektuře .Net.

DEx – Velmi zajímavý projekt, pracující na tvorbě experimentálních nástrojů pro simulace. Důraz je kladen na rychlost, flexibilitu a spolupráci jazyka C++.

5.1 C++Sim

C++Sim byla vyvinuta jako nástroj pro výzkumné účely projektu Arjuna (22), Arjuna poskytuje nástroje pro vývoj „chyby-dovolující“ distribuované aplikace, používající atomární akce a replikace na objekty. Jednou částí vyvíjenou v rámci projektu Arjuna je i Replica Management System, který byl navržen tak aby dovozoval dynamickou rekonfigurovatelnost základních skupin objektů v závislosti na změnách základních charakteristik systému. Protože Arjuna byla napsaná v jazyce C++, bylo rozhodnuto že simulační knihovna bude také napsaná v tomto jazyce, aby ji bylo snadné připojit do systému Arjuna.

Při vytváření C++Sim byli definovány následující požadavky

- Jednoduchost v učení a používání: rozhraní simulační knihovny musí být snadné k porozumění
- Správná abstrakce: C++ programátoři by neměli najít simulační vzory v konfliktech s programovými vzory prezentované v C++.
- Flexibilní, rozšířitelný: Mělo by být relativně jednoduché přidat novou funkci do systému.
- Efektivní: systém by měl být efektivní a měl by produkovat kvalitní simulační běh.
- Přenosný: systém by mělo být možné používat na různých počítačových architekturách a různých operačních systémech
- Spolupráce se SIMULOU: v době vývoje C++Sim se v hojné míře používal simulační jazyk SIMULA, takže vznikl požadavek na kompatibilitu
- Spolupráce s jinými knihovnami: Jak již bylo řečeno C++Sim je napsaný v jazyce C++, jeden z požadavků bylo aby byla možná spolupráce C++Sim se všemi jinými knihovnami napsané v C++.

Pro podrobnější popis odkazují na publikaci (23).

5.2 SIMLIB

Simulační knihovna SIMLIB je vyvíjena od roku 1990 na ústavu informatiky a výpočetní techniky FEI VUT Brno. Knihovna je implementována na počítačích třídy PC pod MS-DOS a pod operačním systémem Linux. Knihovna je přenositelná i na jiné platformy, vyžaduje však úpravu v modulu implementujícím procesy. SIMLIB poskytuje základní prostředky pro řízení simulace. Při tvorbě simulačních modelů a experimentování s nim lze používat různá integrovaná prostředí, která umožňují interaktivní tvorbu a ladění modelů.

Knihovna usnadňuje efektivní popis modelů přímo v jazyce C++, není tedy nutný překladač simulačního jazyka. Tato koncepce má své výhody i nevýhody. Je možné používat všech ostatních prostředků vytvořených v C++. Uživatel také není nijak omezován při případném doplňování prostředků knihovny. Za nevýhodu lze považovat nemožnost dodatečných syntaktických a

sémantických kontrol, které by se mohly provádět při použití překladač simulačního jazyka. Na straně uživatele se předpokládá základní znalost programování v jazyce C++.

Model je v SIMLIB chápán jako množina prvků (entit), které jsou spolu navzájem propojeny vazbami. Tyto vazby spolu s chováním prvků určují chování systému jako celku. Podobný přístup je podstatou objektově orientovaného programování, jehož principy se poprvé objevily šedesátých letech v jazyce SIMULA 67. Objektově orientovaný program je tvořen množinou objektů, které spolu navzájem komunikují – posílají si zprávy.

Rozdělení systému na jednotlivé objekty je závislé na účelu modelu. Mezi objekty modelu můžeme najít takové, které mají shodné podstatné charakteristiky a ty pak můžeme zařadit do jedné třídy objektů. Třída definuje vnitřní strukturu objektů, reakce objektů na vstupy a vlastní chování objektu v čase.

Objekty modelu provádějí určité akce jako odezvu na přijímané zprávy a současně provádějí jiné akce, které jsou nezávislé na přijímaných zprávách. Akce mění stav objektu, jenž je dán obsahem jeho vnitřních datových struktur. Každý objekt má definované akce, které realizuje když je vytvořen a když je zrušen. Tyto akce se nazývají konstruktory a destruktory. Stejně jako ostatní akce, související s popisem chování objektu, patří k takzvaným metodám. Přijetí zprávy odpovídá vyvolání metody a metoda pak popisuje akce, kterými má objekt na tuto zprávu reagovat. Mezi třídami objektů modelu lze nalézt jisté vzájemné vztahy. Tyto vztahy mají obvykle hierarchický charakter. Některé třídy popisují obecné vlastnosti objektů, jiné je více konkretizují. Toho lze výhodně využít při návrhu tříd modelu tak, aby třídy konkrétní mohly využít všeho, co již definovaly třídy obecné. Tato hierarchie tříd je definována relací dědičnosti. Třída může zdědit vlastnosti své bazové třídy, přidávat nové vlastnosti a případně modifikovat vlastnosti zděděné. Takto pojatá hierarchie tříd umožňuje v maximální míře využívat již existujících tříd, což zjednodušuje a zpřehledňuje implementaci modelů. Hierarchie tříd je v podstatě uspořádáním abstrakcí, obvykle od nejobecnějších po konkrétní.

6 VISUÁLNÍ SIMULAČNÍ NÁSTROJE

Extend – Simulační prostředí pro modelování, analyzování a optimalizování procesů. Tento nástroj je velice rozsáhlý a obsahuje řadu důležitých a kvalitních rysů, jejich přehled překračuje rámec této práce.

SIMUL8 – Nástroj, který velice snadno dovoluje pouhým vybíráním předdefinovaných nastavení a objektů vytvářet projekty simulující nejrůznější procesy.

Dymola – objektově orientovaný nástroj pro modelování a simulace spojitých simulací.

VisSim – nástroj pro tvorbu blokových diagramů k nelineárním dynamickým simulacím.

AweSim – poskytuje simulační engine k vytváření modelovým animacím.

Labview – nástroj vyvinutý speciálně pro experimentální a laboratorní simulace.

GoldSim – vizuální prostředí pro účely fyzikálních, matematických, finančních aj. simulací.

BuildSim – vizuální prostředí pro návrh, simulace a analýzu systémů. Toto prostředí obsahuje řadu dalších nástrojů, jako numerické analýzy, tvorba blokových diagramů, a mnoho dalších.

MicroSaint – hlavním účelem této aplikace je flexibilní simulace jiných aplikací.

EcosimPro – jedná se o simulátor spojitých systémů.

SimCreator – slouží k vytváření grafických simulací a modelů.

XJ Technologies – poskytuje rozsáhlou řadu nástrojů pro nejrůznější typy simulací.

Simprocess – je objektově orientovaný, procesy modelující a analyzující nástroj.

SansGUI – je to prostředí pro modelování a simulování vědeckých a inženýrských simulací, bez tvorby grafického rozhraní.

QX3D – jedná se o editor a kódový generátor ke tvorbě pomocných grafických zobrazení a animací konkrétních simulací.

SimCAD Pro – nástroj pro simulování a modelování rozsáhlých průmyslových procesů.

Vizuální simulační nástroje jsou v dnešní době velmi využívány díky svojí jednoduchosti, protože dovolují vytvářet simulace bez znalostí konkrétního programovací jazyka. To je dovoleno tím, že většina zmíněných nástrojů jsou objektově orientované a využívají moderní komponentovou

myšlenku, tzn. že jednotlivé procesy nebo objekty se vybírají z již definovaných šablon u kterých se nastaví konkrétní vlastnosti. V tomto projektu nebudeme využívat žádný zmíněný vizuální nástroj, protože vytváříme velmi rozsáhlý projekt, na který nejsou zmíněné nástroje navrženy, ale i přesto jsem je zde uvedl, protože tvoří důležitou oblast prostředků pro tvorbu simulací.

7 FYZIKÁLNÍ ENGINE

Vytvořit fyzikální engine není tak obtížné jak by se na první pohled mohlo zdát, samozřejmě záleží na množství fyzikálních dějů, které chceme fyzikálně popsat. V našem případě se budeme bavit o kinematické části fyziky, proto vždy když budu v tomto textu mluvit o fyzikálním enginu budu mluvit o částech programu, které umožňují matematicky vyjádřit kinetické síly působící na objekty (bude probráno později)

7.1 Proč fyzikální engine

Ať už se bavíme o tvorbě simulátorů nebo počítačových her, je nutné si uvědomit, že na objekty, které používáme v aplikaci, působí nejrůznější vnější síly, nebo samotné objekty působí navzájem svojí hmotností proti sobě, aby aplikace splňovali podmínku reálnosti, musí se objekty na které působí vnější síly chovat stejně jako v reálném světě. Proto je nutné do aplikací zabudovat prostředek, který bude schopný spravovat a vyhodnocovat vnější síly, kolize objektu a vše co s fyzikální kinetikou souvisí. Tento prostředek se nazývá fyzikální engine. Protože naprogramování kvalitního enginu je velmi komplikované a především nákladné, vytváří se obecné enginy, které se importují do každé aplikace ve které chceme fyzikální síly zakomponovat. Za zmínku stojí, že kvalitní fyzikální enginy světových výrobců stojí i řádově několik desítek milionů dolarů.

7.2 Základní návrh enginu

Základní vlastností fyzikálního enginu je aby byl flexibilní a aby byl použitelný pro co nejvíce druhů simulátorů a her. Musí být také dostatečně robustní a musí dávat co nejpřesnější výsledky. Navíc musí být dostatečně jednoduchý aby ho mohla používat co nejširší skupina programátorů. Jednoduchost musí také umožňovat snadné připojení do programu.

Ty nejjednodušší fyzikální enginy obsahují jenom jednu třídu, která nese informace o attributech a proměnných o objektu, musí umožňovat aplikovat síly působící na objekt a také musí detekovat kolize. V příloze 2. je ukázka nejjednoduššího způsobu definování této základní třídy. Je nutné si uvědomit, že standardní konstruktor nastavuje všechny atributy a proměnné na 0. nebo na NULL.

Základní fyzikální pojmy

Při programování fyzikálního enginu vystává velké množství problémů při realizaci fyzikálních podmínek. Hlavním problémem je, že většina lidí chápe fyzikální zákony poněkud zjednodušeně nebo

nezná všechny podmínky a zákony. Proto si zde osvětlíme základní fyzikální pojmy a matematicky je popíšeme.

7.2.1 Hmota - hmotnost

Každý objekt v reálném světě má určitou velikost a tudíž i hmotu. Hmota samo o sobě nám v kinetické pojetí fyziky nic neříká, proto se musí nějakým způsobem měřit, tyto naměřené výsledky nám udávají hmotnost. Fyzikální a chemické principy měření hmotnosti jsou mimo rozsah této práce. Jak bylo uvedeno dříve, mezi lidmi panují špatné představy o fyzikálních procesech a vlastnostech, v u hmotnosti je to například, že hmotnost je závislá na okolních podmínkách jako je gravitace země. Proto je nutné si uvědomit, že hmotnost je nezávislá na gravitaci, proto je jedno zda měříme objekt na zemi nebo na měsíci, objekt má vždy stejnou hmotnost. Mezinárodní jednotkou pro hmotnost je kilogram [kg].

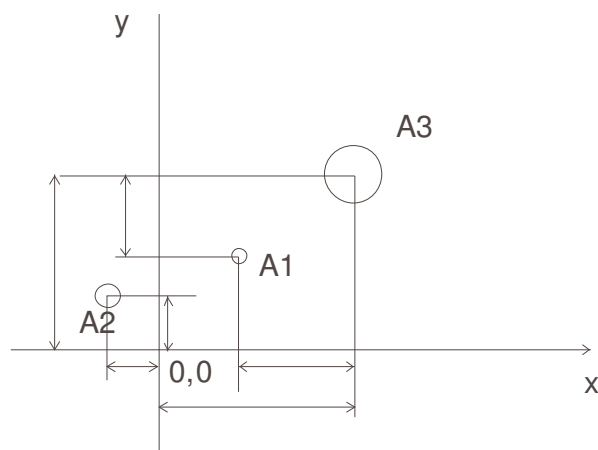
7.2.2 Čas

Pojem čas je ve fyzice velice zajímavý, protože všichni vědí co je to čas, neustále se používá, je to dokonce nej přesněji měřená veličina, ale zároveň tato veličina ve skutečnosti neexistuje, lidstvo si tuto veličinu vymyslelo aby bylo schopno nějakým způsobem změřit změnu stavu, zda už nějaká událost nastala či nenastala. K popsání času by bylo možné napsat stovky stránek, a stejně bychom nepostihly celou podstatu. Protože každý tuto veličinu nebo také můžeme říct vlastnost každý den používá, budeme dále předpokládat že všichni vědí co to čas je a co definuje. Mezinárodní jednotkou pro čas je sekunda [s].

7.2.3 Pozice

Každý objekt má svoji pozici. Tato definice je velmi strohá ale ve zjednodušené představě je přesná, protože každý objekt se pohybuje v nějakém souřadnicovém prostoru. Pozici lze vyjádřit zeměpisnou délkou, zeměpisnou šířkou, nebo v matematickém prostoru, např. Kartézským souřadnicovým systémem osou x, y, z. Existují i jiné způsoby popisu, ale v našem projektu budeme používat zmíněný kartézský systém.

U pozice je nutné si pamatovat pouze jedinou věc, pozice je relativní. Relativní vzdálenost určuje vzdálenost jednoho objektu od druhého nebo vzdálenost objektu od počátku souřadnicového systému, ve kterém objekt právě uvažujeme.



Obrázek 7 Relativní pozice objektů v prostoru

Na obrázku je znázorněna že pozice objektu A3 k počátku je jiná než pozice A3 k A1, pozice je relativní. Obecně se pozice počítá následovně.

$$\vec{PoziceAkB} = \vec{PoziceB} - \vec{PoziceA}$$

Rovnice 1

7.2.4 Rychlost

Pojem rychlosti je dalším problematickým pojmem ve fyzice, protože spousta lidí nerozlišuje mezi vektorově vyjádřenou rychlostí a kvantitativně vyjádřenou rychlostí. Pro správnou funkčnost fyzikálního enginu je nutné tyto pojmy zcela striktně rozlišovat.

7.2.5 Lineární vektorová rychlost

Lineární rychlost vyjadřuje změnu pozice během časové periody. Tato změna je reprezentována jako vektor, který říká o kolik pohybových jednotek se objekt posunul za kolik jednotek času. Díky vektorovému vyjádření jsme schopni určit ve které ose došlo o jakou změnu. Lineární vektorovou rychlost lze vyjádřit:

$$\vec{Rychlost}(\text{čas}) = \vec{Position}'(\text{čas})$$

Rovnice 2

Jak je vidět z uvedené rovnice je lineární vektorová rychlost závislá na čase, z toho ovšem vyplývá, že tak jako pozice je i lineární rychlost relativní. Lineární vektorová rychlost je měřena v metrech za sekundu [m/s]

7.2.6 Kvantitativní rychlost

Jestliže známe vektorovou rychlost na co potřebujeme kvantitativní rychlost? Vektorová rychlost nám udává rychlost v souřadnicovém systému v osách x, y, z, ale v praxi potřebujeme znát skutečnou rychlost objektu ve směru, ať už je jakýkoliv, ve kterém se pohybuje. Tuto skutečnou rychlost nám vyjadřuje kvantitativní rychlost.

7.2.7 Úhlová rychlost

Úhlová rychlost nám udává změnu orientace objektu v časové periodě. Pro lepší pochopení uvedu příklad na letícím disku. Letící disk se pohybuje určitým směrem určitou rychlostí, této rychlosti se říká vektorová (směrová) rychlost, bylo probráno v předcházejícím odstavci, a zároveň se otáčí kolem své osy také určitou rychlostí, této rychlosti se říká úhlová rychlost.

Úhlová rychlost se spočítá:

$$U_{hlova}Rychlost = \frac{(AktualniUhel - PredchoziUhel)}{(AktualniCas - PredchoziCas)}$$

Rovnice 3 Výpočet úhlové rychlosti

Komentář [MB1]: Opravte si formátování textů ve vzorcích. (zkontrolujte si všechny vzorce. Také bývá dobrým zvykem vzorce číslovat – na pravé straně stránky vedle vzorce)

Úhlová rychlost je skalární veličina, protože v 2D prostoru udává orientaci úhel, a v 3D prostoru je úhlová rychlost vyjádřena 3 úhly. Úhlová rychlost se měří v radiánech za sekundu [rad/s].

7.2.8 Zrychlení

Zrychlení je velice důležitý pojem, protože nás bude velmi často zajímat změny pohybu objektu v závislosti na změně působících sil. Zrychlení je závislé na rychlosti, a proto se tak jako rychlost dělí na dva typy.

7.2.9 Lineární zrychlení

Lineární zrychlení je změna rychlosti v závislosti na čase. Jestliže objekt zrychlí o 10 m/s, změní se jeho aktuální rychlost o plus 10 za 1 sekundu. Zrychlení se spočítá ze vztahů:

$$\vec{Zrychleni}(cas) = \vec{Rychlost}'(cas)$$

Rovnice 4 Výpočet zrychlení z rychlosti

$$\vec{Zrychleni} = \frac{(\vec{AktuaIniRychlost} - \vec{PredchoziRychlost})}{(AktuaIniCas - PredchoziCas)}$$

Rovnice 5 Výpočet zrychlení z aktuálního a předešle hodnoty rychlosti a času

7.2.10 Úhlové zrychlení

Úhlové zrychlení je změna úhlové rychlosti během časové periody. Jedná se o obdobnou věc jako je lineární zrychlení, ale aplikované na úhlovou rychlost. Úhlovou rychlost určíme:

$$UhloveZrychleni = \frac{(\vec{AktuaIniUhlovaRychlost} - \vec{PredchoziUhlovaRychlost})}{(AktuaIniCas - PredchoziCas)}$$

Rovnice 6 Výpočet úhlového zrychlení

7.2.11 Těžiště

Těžiště je bod objektu ve kterém se veškerá hmota rozložená v objektu rovná nule. Tento bod se dá celkem snadno najít nebo spočítat pouze za podmínek, kdy na těleso nepůsobí žádné vnější síly.

Těžiště se spočítá:

$$\vec{Teziste} = \frac{\sum (\vec{Pozice}_i * hmotnost_i)}{\sum hmotnost_i}$$

Rovnice 7 Výpočet těžiště

Tento vzorec znamená, že se sečtou všechny hmotnosti všech bodů v objektu vynásobené jejich pozicí a vydělené celkovou hmotností objektu. K nalezení naprosto přesné polohy těžiště se používá několik málo integrací a dalších zajímavých výpočtů.

7.2.12 Síly

Síly jsou základním a nejvíce používaným pojmem ve fyzice. Síly působící na objekt můžeme rozdělit na dvě skupiny, lineární síly, a úhlová .

7.2.13 Lineární síly

Lineární síly ovlivňují pouze změny polohy těžiště v prostoru. Aby nedošlo k nedorozumění, těžiště má stále stejnou polohu vzhledem k objektu, ale lineární síly se působí vždy na těžiště a tím mění polohu celého objektu v prostoru. V momentě kdy aplikujeme lineární sílu na objekt, projeví se změny v podobě změny hodnot lineárního zrychlení a tím i tedy i lineární rychlost.

$$\vec{Rychlost} = \int \vec{Zrychleni} \Leftrightarrow \vec{Rychlost} = \vec{PocatecniRychlost} + \vec{Zrychleni} * Cas$$

Rovnice 8 Výpočet rychlosti

$$\vec{Pozice} = \iint \vec{Zrychleni} * Cas$$

$$\vec{Pozice} = \int (\vec{PocatecniRychlost} + \vec{Zrychleni} * Cas) * CasovaKonstanta$$

$$\vec{Pozice} = \vec{PocatecniPozice} + (\vec{PocatecniRychlost} * Cas) + \frac{1}{2} * (\vec{Zrychleni} * Cas)$$

Rovnice 9 Výpočet pozice

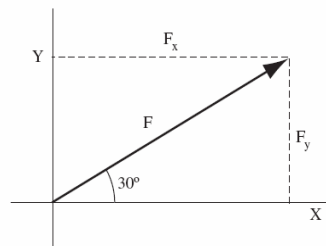
Snad každý si pamatuje vztah probíraný na středních školách který definoval pan Newton a nazval ho „druhý pohybový zákon“ :

$$\vec{Sila} = Hmotnost * \vec{Zrychleni}$$

Rovnice 10 Výpočet síly

Síly se měří v jednotkách Newton [N].

Síly jsou vektory, a protože se naše helikoptéra bude pohybovat v 3D prostoru, bude možné všechny vektory sil rozložit na tři složky podle os, x, y, z.



Rovnice 11 Rozklad sil ve 2D prostoru

7.2.14 Úhlová síla

Úhlová síla působí na objekt v bodě, který není roven těžišti. Důsledkem této síly je kroutivý moment, laicky řečeno, těleso začne rotovat, a na rozdíl od lineárního působení sil nikdy nezpůsobí pohyb tělesa jakýmkoliv směrem. Pro výpočet kroutivého momentu se používá vztahu:

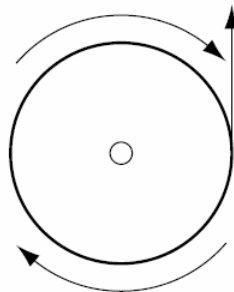
$$\vec{K}_{\text{kroutivyMoment}} = \vec{R}_{\text{rameno}} * \vec{S}_{\text{ila}}$$

Rovnice 12 Výpočet kroutivého momentu

kde:

$$\vec{R}_{\text{rameno}} = \vec{BodPusobeniUhloveSily} - \vec{Teziste}$$

Rovnice 13 Vypocet ramene



Rovnice 14 Kroutivý moment

Nyní když již jsme schopni vyjádřit kroutivý moment tělesa, musíme vnést do enginu další vlastnost, a tou je setrvačnost. Dle Newtonovy definice je setrvačnost odpor vůči změně pohybu.

Důvodem proč vůbec setrvačnost zavádíme je ten, že nám spolu s úhlovým zrychlením způsobují kroutivý moment.

$$KroutivMoment = UhloveZrychleni * Setvacnost$$

Rovnice 15 Vznik kroutivého momentu díky úhlovému zrychlení a setrvačnosti

7.2.15 Výsledný efekt síly

Nyní, když jsem si vysvětlili pojmy jako je lineární a úhlová síla, si řekneme co se ve skutečnosti děje s tělesem pokud na něj působíme nějakou z uvedených sil. To jestli se těleso začne pohybovat nebo rotovat závisí na několika parametrech, jako jsou, v kterém bodě objektu síla působí, zda je objekt pevně spojen s jiným objektem. Obecně působící síla v náhodném bodě tělesa způsobí jak posun – pohyb tělesa tak i jeho rotaci. Samotná rotace ovšem může vzniknout případně, když na objekt působí vnější síla, která uvede objekt do rotace, zatímco síla která má uhel mezi ramenem rovný 180, způsobí nulovou rotaci.

7.2.16 Gravitační interakce

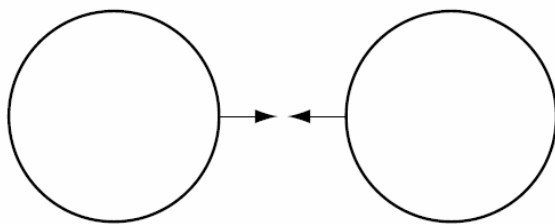
Tento pojem neznamena nic menšího než to, že se jakékoliv dvě tělesa navzájem ovlivňují. Pokud bychom uvažovali dvě tělesa ve vesmíru a odstranili bychom všechny odporové a rušivé síly, zjistili bychom, že se tělesa navzájem přitahují. Tento děj se obecně nazývá „Univerzální gravitační zákon“ a lze vyjádřit rovnicí:

$$GravitacniSila = GravitacniKons\ tan\ ta * \frac{(HmotnostTelesaA * HmotnostTelesaB)}{(VzdalenostAkB^2)}$$

rovnice 16 Výpočet gravitační síly

Rovnice (51) nám dává velikost gravitační síly, pokud bychom potřebovali přesný směr působení gravitační síly, musíme pracovat s vektory, a postup je následující:

- Získáme vzdálenost vektorů
- Normalizujeme vzdálenostní vektor
- Vynásobíme normalizovaný vektor velikosti



Obrázek 8 Vzájemná interakce těles

V rovnici (51) se používá pojem gravitační konstanta, tato konstanta je jedinečná pro každý prostor který uvažujeme, protože v reálném světě je přesně definované co je 1m, 1kg, nebo 1 sekunda, proto i při počítání s reálnými objekty je tato konstanta přesně definované, problém ovšem nastává při počítačových simulacích, kdy si každý programátor definuje jakým způsobem bude zobrazovat např. 1 metr proto je nutné u počítačových simulací tuto konstantu volit tak, aby vypočtené hodnoty odpovídali co nejlépe realitě.

7.2.17 Gravitace na zemi a na jiných planetách

Gravitační síla naší planety, se kterou se potýkáme prakticky neustále aniž bychom si to uvědomovali, samozřejmostí je i to, že tato gravitační síla působí na všechny objekty v našem okolí. Protože náplní této práce je vytvořit létající objekt, který se bude pohybovat v zemské atmosféře, bude zřejmě vhodné probrat tuto charakteristickou sílu poněkud podrobněji. Snad každý někdy slyšel, že země má gravitaci 9.8, co ž je ovšem jenom část pravdy. Pro výpočet gravitační síly země se používá vztah:

$$Sila = GravitačníKonstanta * HmotnostObjektu * \frac{HmotnostPlanety}{VzdálenostObjektuOdPlanety^2}$$

rovnice 17 Výpočet gravitační síly

Pro planetu zemi je gravitační konstanta rovna $6.67 \cdot 10^{-11}$. Vzdálenost objektu od planety se uvažuje jako vzdálenost těžišť.

Jak bylo uvedeno, země má gravitační sílu rovnu 9.8. Ukážeme si jak se tato hodnota spočítala, při výpočtu vycházíme z rovnice (52).

$$Sila = 6.67 * 10^{-11} * 1 * \frac{5.98 * 10^{24}}{(6.38 * 10^6)^2}$$

$$Sila = 9.799N$$

rovnice 18 Ukázka výpočtu gravitační síly na zemi

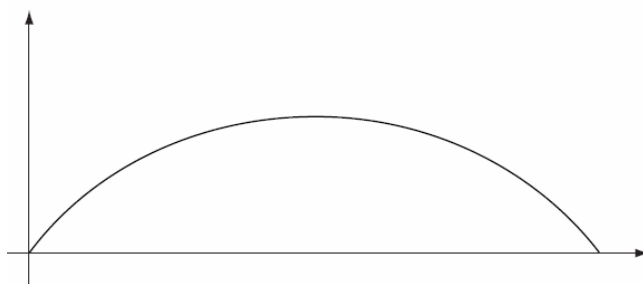
V ukázkovém příkladu uvažujeme hmotnost objektu jako 1kg. Nyní se nám podařilo získat působící sílu na všechny objekty na naší planetě, takže jsme schopni si vyjádřit tolik známé gravitační zrychlení.

$$Zrychleni = \frac{Sila}{hmotnost} = \frac{9.8}{1} = 9.8 \frac{m^2}{s}$$

rovnice 19 Ukázka výpočtu gravitačního zrychlení

7.2.18 Simulace projektilů

Posledním tématem vztaženému ke gravitaci je simulace projektilů. Za projektil budeme uvažovat objekty mající určitou hmotu – hmotnost, které na počátku udělíme rychlost a poté na tento objekt působí pouze jediná síla, a tou je síla gravitační. Tím se docílí toho, že se objekt pohybuje po určité trajektorii.



Obrázek 9 Trajektorie projektilu

Jak je vidět na obrázku, trajektorie projektilu má tvar paraboly. Důležitý předpoklad u projektilů u simulací je, že rychlost projektilu na konci je stejná jako je rychlost na počátku. Pro výpočty projektilu se uvažuje že nejvyšší bod vertikální rychlosti leží v bodě 0 a horizontální rychlost je po celou dobu konstantní, přičemž zrychlení je nulové.

Abychom mohli stanovit vertikální a horizontální rychlost projektilu, musíme znát počáteční rychlost, kterou si rozložíme na složky x,y.

Vertikální rychlost si můžeme vyjádřit ze dvou vztahů.

$$RychlostY(cas) = PocatecniRychlostY + Zrychleni * Cas \Leftrightarrow$$

$$RychlostY(cas) = PocatecniRychlost * \sin(\Theta) + (-9.8) * Cas$$

Rovnice 20 Výpočet vertikální rychlosti

Nyní když známe rychlost můžeme zjistit i pozici v libovolném čase trajektorie.

$$Pozice(cas) = PocatecniPozice + PocatecniRychlost * \sin(\Theta) * Cas + \frac{1}{2} * (-9.8) * Cas^2$$

Rovnice 21 Výpočet pozice v libovolném čase projektilu

Díky možnosti výpočtu rychlosti a pozice v libovolném čase můžeme používat v simulaci projektily. Další zajímavé prvky jsou zmíněny v publikaci (24).

7.2.19 Tření

Je to jedna z nejproblémovějších témat ve fyzikálních enginech. Problémem není samotný výpočet, ten je docela jednoduchý, problémem je provádění výpočtu na počítačích, protože výpočet je značně časově náročný.

Pod pojmem tření si můžeme představit odpor působící proti pohybovým změnám objektu. To způsobuje elektrické, mechanické nebo tepelné ztráty. Výpočty třecí síly v počítačích jsou značně komplikované a přesahují rámec této práce. Pro zájemce odkazuji na publikaci (24).

7.3 The Open Dynamic Engine

The Open Dynamic Engine (ODE) je volně šiřitelná, průmyslová knihovna pro simulaci spojených tuhých dynamických objektů. Je vhodný pro simulaci pozemních dopravních prostředků, „chodících“ živočichů, zkrátka pro pohyb objektů ve virtuálním prostředí. ODE je velice rychlé, flexibilní a rozsáhlé, samozřejmostí je podpora kolizí objektů. Ačkoliv je v současné době ODE ve verzi 0.5 dosahuje velmi kvalitních výsledků, protože jsou v něm aplikovány nejmodernější programovací techniky a prostředky, navíc je to velmi stabilní knihovna.

7.3.1 Vlastnosti ODE

Jak již bylo řečeno ODE je vhodný pro simulace spojených dynamických objektů. Samotné spoje objektů jsou vytvořeny v momentě kdy je vytvořeno spojení objektů a od této chvíle probíhají

výpočty. Knihovna je napsána velmi obecně a umožňuje používání rozsáhlé tvary objektů tak i typů spojů.

ODE je navržen tak aby byl používán v interaktivních real-time aplikacích. ODE využívá vysoce stabilní integrační nástroje, tím je dosažena velká stabilita a proto nedochází k fatálním chybám ani při špatně definovaných vlastnostech objektů. ODE má velmi kvalitně zpracované časování a s tím související časové funkce. Další zajímavou vlastností je existence C a C++ rozhraní, čímž se zjednodušuje integrování knihovny do aplikací.

7.4 Tokamak

Tokamak je fyzikální engin napsaný v jazyce C++ a je určen pro objektově orientované aplikace. Tento engine poskytuje dva základní typy tříd. Prvním typem tříd jsou „interfacy“, tyto třídy musí být vytvořeny a uvolněny během Tokamak API funkcí nebo v členských funkcích obecných Tokamak tříd. Druhou skupinou tříd jsou třídy „vnější“. Ty jsou tvořeny obyčejnými C++ třídami používané API „uživateli“. Snahou vývojářů tohoto enginu je udržovat počet virtuálních funkcí na minimu, přitom všechny funkce, vypadají vnitřně jako ne-virtuální.

7.4.1 Základní třídy

Jak bylo uvedeno výše, Tokamak poskytuje dva typy tříd. Vzhledem k autorovým doposud získaným poznatkům a zkušenostem, se jeví fyzikální engin Tokamak jako nejvhodnější volba pro tvorbu simulátoru. Proto se pokusíme tento engine probrat poněkud podrobněji.

Interface třídy

neRigidBody – tato třída poskytuje klasické metody pevného-tuhého tělesa.

neAnimatedBody – třída pro definici animovaných objektů, to jsou objekty, které nejsou fyzikálně podstatné pro simulaci, ale jsou vhodné pro lepší vizualizaci, je ovšem nutné detekovat kolize s jinými objekty.

neGeometry – tato třída zapouzdřuje většinu geometrických operací

neSimulator – třída, která slouží pro ovládání a kontrolu celé simulace

neSenzor – v simulacích je nutné zobrazovat nebo reagovat na získané výsledky, k tomu slouží tato třída, která získává informace z ostatních tříd, a poté s nimi provádí příslušné operace.

neJoint – třída která definuje metody pro práci s dvěma objekty, které spolu spolupracují.

neRigidBodyController – třída, která rozšiřuje ovládací možnosti třídy *neRigidBody*

neJointController – opět třída, která rozšiřuje možnosti třídy *neJoint*.

neCollisionTable – třída spravující kolize objektů

Obyčejné třídy, struktury a typy

neBodyType – výčtový typ, který definuje typy pevných objektů, používá se při kolizích.

neCollisionInfo – třída poskytující řadu užitečných informací o kolizích.

neSimulatorSizeInfo – zapouzdření informací o simulaci, jako jsou počty objektů atd...

neAllocatorAbstract – abstraktní třída pro alokaci objektů.

neAllocatorDefault – třída děděná z *neAllocatorAbstract*, poskytující jednoduchý paměťový manager.

nePerformanceReport – třída pro práci s výsledky a jejich následnou prezentací

neTriangle – okolní prostředí a terén je většinou kvůli rychlosti zobrazován mnoha trojúhelníky, tato třída slouží pro definici každého z nich.

neTriangleMesh – tato třída slouží pro celkovou správu terénu

neRigidBodyControllerCallback, *neJointControllerCallback* – simulace je prováděna v cyklech, tyto třídy spravují správné volání aktualizace objektů v simulaci

7.5 Newton

Fyzikální engine Newton je další velmi často užívaný, volně šiřitelný nástroj pro ovládání tuhých těles v real-time simulacích. Tento engine obsahuje rozsáhlou sadu tříd a struktur pro snadnou definici a správu rozličných objektů. Velice zajímavými jsou například třídy umožňující definici celého vozidla, zajímavé je to tím, že většina fyzikálních enginů umožňuje pouhou definici jednoduchých objektu, krychle, koule, trubka atd..., a teprve z těchto jednoduchých objektů se sestavují celky. Důležitou informací o tomto enginu je to, že se musí definovat dříve než je volána první funkce API.

Definování Newton enginu se provádí přes funkci `NewtonCreate(NewtonAllocMemory mallocFnt, NewtonFreeMemory mfreeFnt)`.

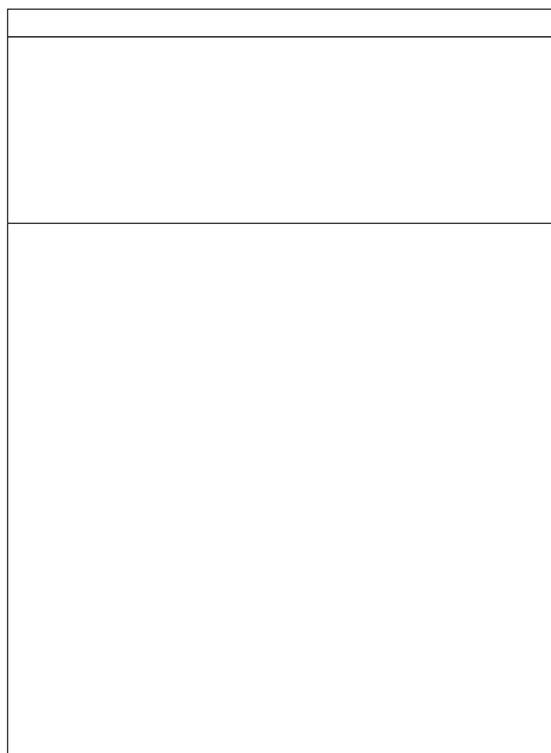
7.6 NovodeX

NovodeX je samostatný jednoduchý balíček funkcí a tříd pro správu tuhých těles v dynamických systémech. Tento balíček tvoří soubor knihoven SDK, a knihovní funkce a třídy, které nemají mezi sebou žádnou závislost, tzn. že NovodeX nevyužívá objektového způsobu programování.

8 EXPERIMENTÁLNÍ ČÁST

V této části se pokusím objasnit nejdůležitější myšlenky návrhu fyzikálního modelu helikoptéry. Mým úkolem bylo vymyslet hierarchickou strukturu objektů, které se budou používat v programu pro realizaci jednotlivých bloků helikoptéry. Dalším problémem bylo vytvoření komunikačních rozhraní, přes které komunikují objekty mezi sebou a rozhraní, které zprostředkovávají komunikaci mezi jednotlivými pluginy, přičemž pluginy vytváří celý tým programátorů. I když to v zadání není zcela konkrétně zmíněné mám za úkol vytvořit fyzikální engine, neboli systém který bude provádět a řídit veškeré vnitřní pochody v aplikaci, bude říkat kdy se má co přepočítat kdy vykreslit, nebo kdy se má reagovat na vstupní události. Velmi důležitou částí je načítání dat ze souborů, jde především o statické data, které je nutné znát již před spuštěním.

V aplikaci je nutné vytvořit systém, kterým budeme řídit běh aplikace, dále budeme tento systém využívat pro jednodušší přístup k často využívaným handlům. Tento systém budeme nazývat GameEngine. Jeho deklarace v nejjednodušší formě je na obrázku 10.



Obrázek 10 Deklarace GameEngine

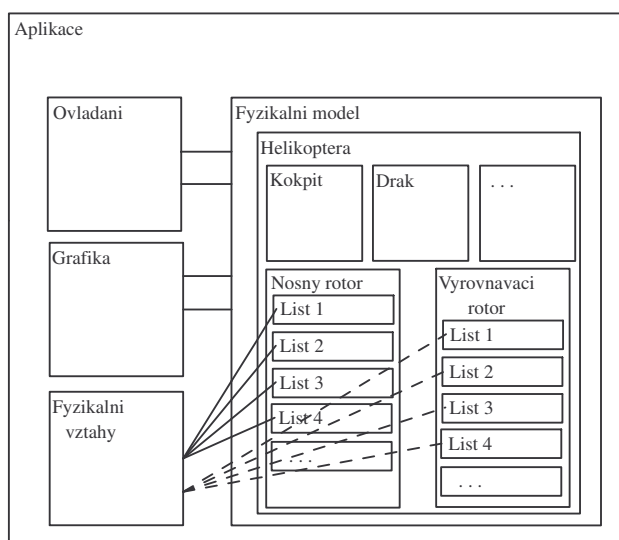
GameEngine

```
-static m_pGameEngine : GameEngine
-m_hInstance
-m_hWindow
-m_szWindowClass : wchar_t
-m_szTitle : wchar_t
-m_iFrameDelay : int
-m_bSleep : bool
-m_uiJoystickId : unsigned int
-m_rcJoystickTrip
-m_uiMidiPlayerId : unsigned int
+static GetEngine()() : GameEngine
+Initialize(int iCmdShow)() : bool
```

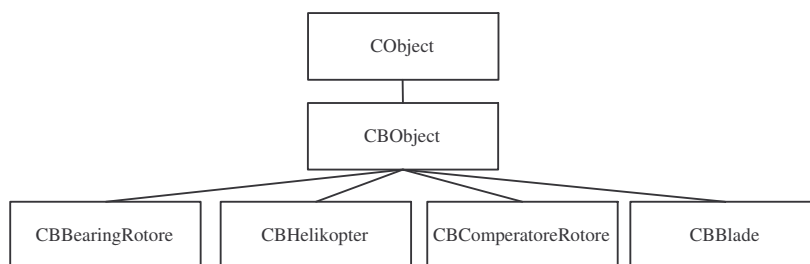
Hierarchie objektů

Vzhledem k tomu, že helikoptéra je příliš složitý stroj, a není v našich silách popsat helikoptéru do nejmenších součástek, vytvoříme si mnohem jednodušší představu o helikoptéře, která bude sestavena z nejdůležitějších částech helikoptéry, jsou to především ty části které vytváří síly potřebné pro let helikoptéry. Tyto části budou tvořeny objekty s požadovanými vlastnostmi. Aby daná představa byla funkční musí být vytvořené objekty v určitém spojení, tzn. objekty musí být schopné spolu komunikovat což přesně znamená že si musí mezi sebou předávat řadu informací, nejdůležitější informací je výsledná síla, kterou daný objekt vytváří.

Abychom dodrželi jeden ze základních požadavků na možné rozšiřování je každý objekt helikoptéry vytvořen jako samostatný DLL soubor, tím je dosaženo možnosti vyvíjení nových a nových objektů bez nutnosti změny celé aplikace, je nutné dodržovat pouze několik základních pravidel, jako je shodný název souboru, název třídy, popřípadě název exportovaných a importovaných funkcí.



Obrázek 11 Zjednodušená hierarchie



Obrázek 12 Hierarchie objektu

Z obrázku 14 je vidět, že všechny objekty reprezentující součásti helikoptéry jsou děděny od třídy CObject, tato třída obsahuje nejdůležitější vlastnosti, proměnné a funkce které jsou společné pro všechny objekty, jako je například těžiště, hmotnost, výsledná síla, atd. Tato třída je navíc děděna od třídy CObject, což umožňuje používat všechny námi definované objekty jako MFC-objekty, této vlastnosti využijeme především při práci se seznamy objektů, pro které nám knihovna MFC poskytuje kvalitní prostředky.

Při práci s programem je nutné si uvědomit, že každá námi definovaná třída, je uložena v samostatném DLL souboru, a aby jednotlivé třídy byly schopné mezi sebou komunikovat, je nutné provádět různá křížení odkazů na jiná DLL-ka. To má za následek, že prvotní kompilace se nemůže provádět jako celek, ale musí se postupovat postupně, a to od hierarchicky nejnižších tříd až po ty výše postavené, v praxi to znamená to, že se jako první musí zkompilevat třída CObject, poté všechny objekty které dědí nějaké vlastnosti od této třídy. Přitom je nutné kompilovat potřebné fyzikální vztahy, které jsou taktéž uloženy v samostatných DLL souborech. Pokud provedeme kompilaci u všech objektů a fyzikálních vztahů, můžeme provést kompilaci nejdůležitější třídy v aplikaci a tou je třída CBModel, veškeré kompilace musí proběhnout zcela v pořádku, až na několik upozornění, které upozorňují na nebezpečí straty dat při převodu z double na float, v praxi to znamená chybu v řádu $1 \cdot 10^{-6}$.

Komunikace s grafikou

V mé části programu se s grafickými prvky vůbec nepracuje, kontroly a řídím helikoptéru a program z technického hlediska, nikoliv vizuálního, ale i přesto musí docházet alespoň k minimální komunikaci mezi danými moduly. Jedná se o příkaz vykreslení, tato činnost musí být plně kontrolovatelná programátorem, aby se dala měnit frekvence překreslování na základě výkonnosti počítače, nebo volby definované uživatelem. Dalším důvodem pro komunikaci je neznalost přesné pozice v prostoru, jedná se především o výšku, která se používá v několika vzorcích. Tato neznalost

je dána tím, že okolí-svět ve kterém se pohybuje helikoptéra mi není známa, nic o ní nevím, takže místo přesné polohy v prostoru, komunikujeme pomocí relativní změny, vždy když dojde k překreslení obrazovky (50x za vteřinu) zjistí se o kolik metrů se helikoptéra posunula od posledního známého bodu, a protože známe čas a posun, je tato změna přičtena k předešlé poloze čímž dostáváme naši novou polohu.

Komunikace s ovládáním

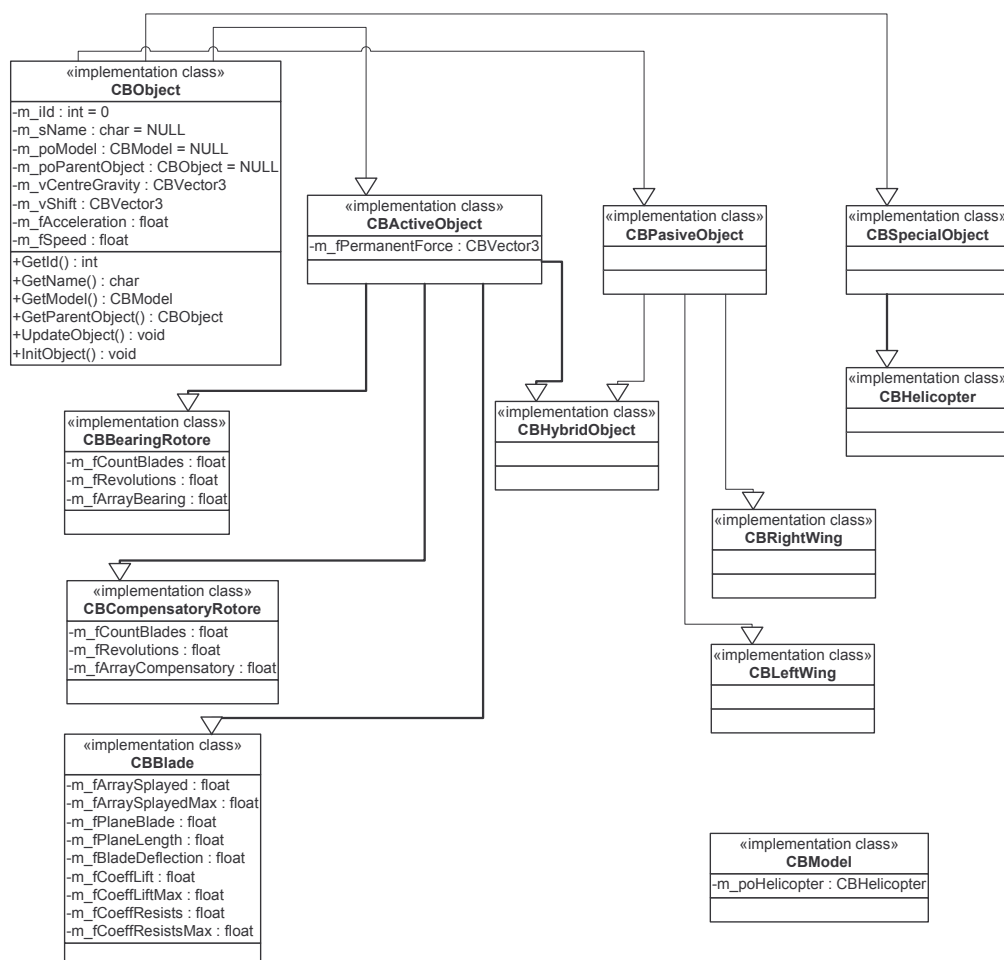
Komunikace s ovládáním je mnohem jednodušší protože ze strany aplikace nemusím předávat žádné informace, ovládání si pomocí API funkcí stav vstupních zařízení, z mého pohledu mě nezajímá jakým způsobem je toho dosažené, pouze v příslušném okamžiku provedu aktualizaci jednotlivých vlastností helikoptéry, jako je například úhel náběhu listů, náklon rotoru nebo otáčky nosného rotoru. Aktualizace je volána ze strany ovládání, kdy pomocí exportovaných funkcí, které jsou pojmenovány podle ovládacích prvků helikoptéry (levý, pravý knipl, levý, pravý pedál). Volání daných funkcí je v kompetenci ovládacího modulu.

CBModel

Zřejmě nejdůležitějším objektem v aplikaci je objekt této třídy. Na obrázku 13 je znázorněno velice zjednodušené schéma hierarchie objektů v aplikaci. Instance „Fyzikální model“ je vlastně objekt třídy CBModel. Úkolem tohoto objektu je zapouzdřovat stromovou hierarchii všech objektů helikoptéry, a zároveň umožňovat komunikaci mezi vstupem(ovládání) a výstupem(grafikou). Komunikace je dosaženo importováním a exportováním funkcí.

CBObject

Třída CBObject je předkem všech objektů helikoptéry, které budeme v naší aplikaci požívat. Při deklaraci jsem se nechal inspirovat z deklarace obdobné třídy ve fyzikálním enginu Tokamak. Deklarace třídy je uvedena v příloze 1.



Obrázek 13 Návrh dědičností hierarchie

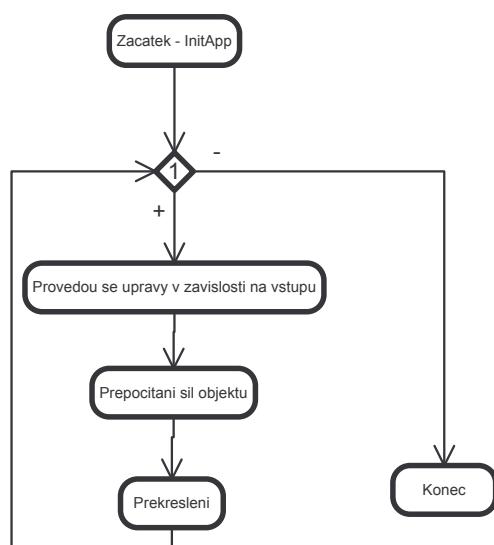
Přepočet sil

Nejvíce využívanou vlastností všech objektů helikoptéry, je výpočet a přepočet sil působících v jednotlivých objektech a předávání výslednici síly vždy svému rodiči až do okamžiku kdy je dosaženo vrcholu stromové hierarchie, čemuž odpovídá objekt CBHelicopter. Při průchodu stromem se využívá rekurzivního volání metody „UpdateForces()“. Tuto metodu dědí všechny třídy odvozené od třídy CObject, ovšem v případě nutnosti dovoluje syntaxe jazyka C++ přepisovat zděděnou metodu svoji vlastní. Aby výpočet sil byl správný je nutné nejprve spočítat výslednici v nejnižších vrstvách stromové hierarchie a poté teprve spočítat výslednice předků. Rekurse patří mezi základní programátorské techniky, a proto ji zde nebudu blíže popisovat.

Komentář [MB2]: Z obrázku 13 to vidět není... Z toho je pouze parné, že objekt CBHelicopter je odvozen od třídy CObject.

Funkce Run

Naše aplikace musí být schopná co nejrychleji přepočítávat síly a co nejrychleji překreslovat obrazovku. Proto musí být funkce provádějící tyto operace umístěny v místě, kde se budou po celou dobu běhu aplikace volat. K tomu jsme mohli použít časovač. Ten je ovšem značně nepřesný a pomalý; nepřesný z důvodu nízké priority a pomalý protože jeho nejkratší reakce je 50 ms. Proto jsem zmíněné funkce umístili do funkce Run, kterou jsme tím pádem museli přepsat. Funkce Run si můžeme představit jako klasickou nekonečnou smyčku, ale MFC poskytuje právě tuto funkci, která tuto smyčku nahrazuje. Tento způsob přináší řadu výhod a především snadnější manipulaci.



Obrázek 14Zjednodušené schéma Run funkce

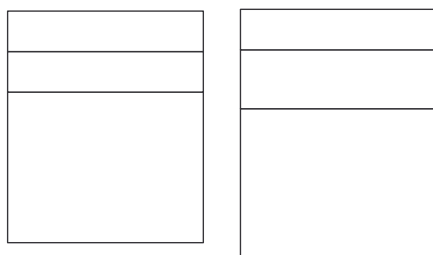
Časovače

V naší aplikaci jsem se rozhodl použít časovač multimediálního typu, který je součástí API rozhraní. Pro zjištění aktuálního času se využívá funkce `QueryPerformanceCounter((LARGE_INTEGER *)&);` je mnohonásobně přesnější než klasický časovač. Tento časovač využívám pro řízení programu ve funkci Run a to abych byl schopný řídit FPS (počet překreslení za sekundu), a pro výpočet časově závislých vztahů, například zrychlení.

CBVector3 a CBVector2

Tyto třídy jsem se rozhodl použít pro zjednodušení práce s vektory. Protože se bude neustále pracovat s trojrozměrným prostorem, je nutné se v něm orientovat, k tomu využívám os x, y, z.

Abych si nemusel neustále definovat 3 proměnné vytvořil jsem si tyto třídy, ve kterých jsem přetížil nejběžnější operátory pro práci se standardně definovanými datovými typy.



Načítání dat

Pro načítání statických dat jsem se rozhodl uchovávat potřebné data v XML souboru. Důvody pro použití tohoto typu souborů je především to, že se jedná o textový soubor který je snadno čitelný pro člověka, navíc se jedná o moderní způsob zpravování dat, který se bude v budoucnu jistě hodně rozšiřovat. Protože kompilátor C++ ve Visual Studiu nemá standardní podporu pro práci s XML soubory, museli jsme použít některou z mnoha volně šiřitelných knihoven. Po domluvě jsme se rozhodli použít knihovnu známou pod názvem EXpat. EXpat je poněkud specificky vytvořená knihovna, její použití nemusí být hned na první moment jasné. Především se jedná o systém pracující s CALLBACK funkcemi, a proto je nutné přizpůsobit způsob psaní funkcí, kdy nejsme schopni předem říct, která z předem definovaných funkcí bude volána, ale přitom musíme přesně vědět, kde se v danou chvíli nacházíme, abychom byli schopni načítat potřebné data. V nejjednodušším případě, se definují pouze 3 CALLBACK funkce. První se volá po nalezení elementu, a jmenuje se StartElementHandler, jejím opakem je ukončení elementu, EndElementHandler, pomocí těchto funkcí jsme schopni rozlišovat kde se zrovna v XML souboru nacházíme, další podstatnou funkcí, kterou musíme definovat se nazývá CharakterDataHandler, která je volána vždy když se při rozkladu XML souboru narazí na řetězec znaků, v našem případě se jedná o konkrétní data, která chceme získat, a jelikož teda víme v jakém elementu se nacházíme, a známe i data, která jsou v elementu uložena, jsme schopni provést inicializaci dat v naší aplikaci.

```
«implementation class»
class CBVector {
public:
    -m_fx : float
    -m_fy : float
    +CBVector() : <unspecified>
    +SetVector() : void
    +operator ==()
    +operator +()
    +operator -()
    +operator *()
    +operator /()
    +operator =()
```

```
«implementation class»
class CBVec {
public:
    -m_fx : float
    -m_fy : float
    -m_fz : float
    +CBVector() : <unspecified>
    +SetVector() : void
    +operator ==()
    +operator +()
    +operator -()
    +operator *()
    +operator /()
    +operator =()
```

```

- <Object>
  <CentreGravity>1.0,2.0,3.0</CentreGravity>
  <Mass>10000.0</Mass>
  <Inertia>0.0</Inertia>
  <StaticFrictionCoefficient>1.0</StaticFrictionCoefficient>
  <KineticFrictionCoefficient>1.0</KineticFrictionCoefficient>
  <CoefficientOfRestitution>10.0</CoefficientOfRestitution>
  <Position>0.0,0.0,0.0</Position>
  <LinearVelocity>0.0,0.0,0.0</LinearVelocity>
  <Orientation>0.0</Orientation>
  <AngularVelocity>0.0</AngularVelocity>
  <TotalForce>0.0,0.0,0.0</TotalForce>
  <TotalTorque>0.0</TotalTorque>
  <FrictionForce>0.0,0.0,0.0</FrictionForce>
  <TotalImpulse>0.0</TotalImpulse>
  <Shift>0.0,0.0,0.0</Shift>
  <Acceleration>0.0</Acceleration>
  <Speed>0.0</Speed>
</Object>

```

Obrázek 15 XML soubor

Komentář [MB3]: No, ta experimentální část se mi zdá celkem krátká. Snažte se víc prodat to, co jste reálně vytvořil a co jste ještě vytvořit chtěl. Můžete se rozvykládat (+ graficky popsat) o tom, jak je celá aplikace inicializována, jak jsou načítány DLLka, jak jsou jednotlivé objekty propojeny, volány, jaké jsou rozhraní mezi vaší částí aplikace a dalšími moduly (grafika, ovládání), jak se pracuje s eXpatem, atd. Prostě se trochu víc pochlubte ☺

9 ZÁVĚR

Největší část své bakalářské práce jsem se věnoval návrhu hierarchie objektů a komunikace mezi nimi. Poznatky jsem čerpal především z již existujících, volně šířitelných fyzikálních enginů, a tyto poznatky jsem upravil pro potřeby našeho simulátoru. Kvalita mého návrhu se ovšem prokáže až v pozdější části vývoje, kdy hierarchie objektů bude mnohem složitější a zpracování veškerých fyzikálních vztahů bude mnohem časově náročnější.

Komunikaci mezi jednotlivými objekty funguje na velice jednoduchém principu, kde jeden objekt má přes ukazatel přístup ke druhému a naopak. Komunikace mezi moduly, která se využívá mnohem častěji, protože se jedná o pluginově založenou aplikaci, probíhá pomocí importovaných a exportovaných funkcí, při návrhu jsem předpokládal, že může nastat situace, že jeden modul bude obsahovat příliš těchto funkcí a proto v případě výskytu této situace se definuje globální struktura, která bude zapouzdřovat importované a exportované funkce.

Při testování aplikace se nám podařilo zjistit, že při běhu aplikace dochází ke 100%-nímu využití procesoru, a to i přesto, že jsem ve funkci Run provedl několik změn, především jsem definoval periodu zobrazení FPS co nejmenší. Dalším změnou ve funkci Run je použití funkce OnIdle, která je volána při nečinnosti, ale ani jedna ze zmíněných změn nepřinesla žádaný efekt. Problém je způsoben zřejmě v kombinaci několika technologií, SDL – vstup, OpenGL – výstup, MFC – aplikační systém.

Závěrem bych rád sdělil, že se nejedná rozhodně o hotovou aplikaci, především z časových důvodů. Vývoj nového simulátoru v současné době trvá i několik let a to celým programátorským týmem, které mají mnohaleté zkušenosti s vývojem simulací. Navíc mají k dispozici nejnovější technologie, a enginy. Ty nejlepší fyzikální enginy jsou cenově naprosto nedostupné, jediná licence stojí přes 250 000\$.

SEZNAM POUŽITÉ LITERATURY

- (1) Zítek, Pavel : Simulace dynamických systémů, Státní nakladatelství technické literatury, Praha 1990
- (2) Petrová Růžena, Zítek Pavel: Matematické a simulační modely, ČVUT, 1996
- (3) Noskiewič, Petr : Simulace systémů, VŠB, Ostrava, 1996
- (4) Vavřín, Petr : Teorie dynamických systémů, VÚT Brno, 1989
- (5) A Clection of Modeling and Simulation Resources on Internet – <http://www.idsia.ch/~andrea/simtools.html>
- (6) Chalupa, Radek : 1001 tipů a triků pro Visual C++
- (7) Prošise, Heft : Programování ve Windows pomocí MFC, Computer Press, 2000
- (8) Arlow, J.: UML a unifikovaný proces vývoje aplikací, Computer Press, 2003
- (9) Balátě, J. Automatické řízení. BEN, Praha, 2003. ISBN 80-7300-020-2.
- (10) Prokop, R., Husták, P. a Prokopová, Z. Návrh a ladění spojitých regulátorů aneb co dokáže algebra v teorii řízení. Automatizace Vol. 12, 2002, Praha. ISBN 0005-125X.
- (11) Kruglinski J. Shepherd G. Wingo S Programujeme v Microsoft Visual C++, PRESS, Praha, ISBN 80-7226-362-5
- (12) Maňas, M. Teorie Her a optimální rozhodování, SNTL, Praha, 1974. ISBN 04-012-74
- (13) Peacock, L. a Richardson D. AH – 64, Londyn 1994, ISBN 28-072-94
- (14) Beneš, L. Československé vrtulníky známé a neznámé, Olomouc, 1998, ISBN
- (15) Mařík, V. Štěpánková, O. Katanský J. Umělá inteligence 2, Praha, 1997, ISBN 80-200-0504-8
- (16) Materiální část Mi-17 „HIP“ Let – 3 – 76
- (17) Rychlík, J. Programovací techniky, Kopp, České Budějovice, 1992, ISBN 80-901051-7-3
- (18) Modrlák O. Úvod do identifikace, Studijní materiály VSLIB
- (19) Hanuš B. – Alexik M. – Bíla J. – Kroc L. – Krupka Z. Teorie automatického řízení II. Skripta Liberec, 1985
- (20) Štěcha J. Optimální rozhodování a řízení, skripta, Praha, 1999
- (21) Vašek, V. Vašek L. Simulace systémů, Vysoké učení technické v Brně.

- (22) G. D. Parrington et al, "The Design and Implementation of Arjuna", Broadcast Project Technical Report, October 1994
- (23) The University, Newcastle upon Tyne, C++SIM user's guide, NE1 7RU, UK
- (24) Bruno Miguel Teixeira de Sousa, Game programming All in one, Premiér Press 2002
- (25) Russel Smith, Open Dynamic Engine, v0.5 User Guide, 2004
- (26) NAG C Library – Essential Introduction, Manual
- (27) Další zdroje na internetu

Citace literatury se řídí platnou normou ČSN 01 0197 (bibliografické citace). Podrobné informace o citování literatury v odborné práci (i elektronických zdrojů) najdete na www.knihovna.utb.cz/czech/citace.php.

Odkazy na literaturu v textu jsou vloženy do kulatých závorek a arabská čísla musí korespondovat s pramenem uvedeným v seznamu literatury.

Např.: (3), (18, 25, 33), (32-36, 42) ...

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ODE – Open Dynamic Engine

ODR – Obecné Diferenciální rovnice

DLL – Dynamic Linking Library

STL – „Standartně definované šablony“

MFC – Microsoft Foundation Class

API – „Windows rozhraní“

SDL – Simple Direct Library

FPS – „perioda překreslení obrazovky“

SEZNAM OBRÁZKŮ

Obrázek 1 Postup návrhu	13
Obrázek 2 Schéma postupu při analýze problému, volbě metody a modelovém řešení.....	15
Obrázek 3 Identifikace a simulace v procesu poznání zkoumaného systému	19
Obrázek 5 : Vývoj abstraktního modelu při vytváření simulačního modelu	22
Obrázek 6 Rozdělení modelů podle charakteru modelového procesu	22
Obrázek 7 Relativní pozice objektů v prostoru.....	38
Obrázek 8 Vzájemná interakce těles.....	44
Obrázek 9 Trajektorie projektilu.....	45
Obrázek 10 Deklarace GameEngine.....	50
Obrázek 11 Zjednodušená hierarchie.....	51
Obrázek 12 Hierarchie objektu	52
Obrázek 13 Návrh dědičností hierarchie.....	54
Obrázek 14 Zjednodušené schéma Run funkce	55
Obrázek 15 XML soubor	57

Příloha 1.: Deklarace třídy, která složila jako podklad pro třídu CObject

mrEntity
-m_kCenterOfMas : CBVector3 -m_fMass : double -m_fInertia : double -m_fStaticFrictionCoefficient : double -m_fKineticFrictionCoefficient : double -m_fCoefficientOfRestitutions : double -m_kPosition : CBVector3 -m_kLinearVelocity : CBVector3 -m_fOrientation : double -m_fAngularVelocity : double -m_kTotalForce : CBVector3 -m_fTotalTorgue : double -m_kFrictionForce : CBVector3 -m_fTotalImpulse : double
+Simulate(double)() : void +ApplyLinearForce(vector)() : void +ApplyTorgue(double)() : void +ApplyForce(vector, vector)() : void +ApplyFriction(double)() : void +HandleCollision(...)() : void +Set() : void +Get() : <unspecified>