České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská
Katedra matematiky

Modelování a animace 3D scény

diplomová práce

diplomant: MARTIN RÜCKER
školitel: RNDr.JOSEF PELIKÁN, KSVI MFF UK
školní rok: 1994/1995

Prohlášení:

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a použil pouze uvedené literatury.

V Praze dne 5.5.1995 Martin Rücker

Obsah

ÚVOD **1**

POČÍTAČOVÁ GRAFIKA - ANIMACE **2**

KONVENČNÍ ANIMACE: 2
JEDNODUCHÉ ANIMAČNÍ EFEKTY: 3
ROZDĚLENÍ ANIMACE 3
- Animace klíčových snímků 4
- Parametrická animace 4
- Algoritmická animace 4

ANIMAČNÍ JAZYKY 4
- Lineární seznam instrukcí: 4
- Vyšší programovací jazyky 5
- Grafické jazyky 5

METODY ŘÍZENÍ ANIMACE 7
- Úplné explicitní řízení 7
- Procedurální řízení 8
- Systémy založené na omezeních 8
- Snímání živého pohybu 8
- Herec 9
- Fyzikální simulace 9

ZÁVĚR **9**

PROGRAM MDL **10**

ZÁKLADNÍ DATOVÁ STRUKTURA PROGRAMU MDL-VEKTOROVÝ PROSTOR **10**

SKALÁR 10
VEKTOR 10
MATICE 11

VYŠŠÍ DATOVÉ STRUKTURY PROGRAMU MDL-TRANSFORMACE,OBJEKT A SCÉNA **11**

TRANSFORMACE 11
TĚLESO 11
SCÉNA 13

KONSTRUKCE OBJEKTŮ **14**

TĚLESA POLYHEDRÁLNÍHO TYPU 14
KUBICKÁ TĚLESA 15
- Bézierovy křivky 15
- Bézierovy plochy 16
- B-Spline křivky 16
- ###-Spline křivky 16
- ###-Spline plochy 17
- Kubické šablony 17

ZOBRAZOVACÍ METODY **17**

ZOBRAZENÍ DRÁTOVÉHO MODELU 18
ZOBRAZENÍ PROSTOROVÝM ŘEZEM 18
ZOBRAZENÍ PAMĚTÍ HLOUBKY 19
STÍNOVÁNÍ 20

TRANSFORMACE **22**

UŽIVATELSKÉ ROZHRANÍ **24**

GRAFICKÉ OKNO 25
TEXTOVÉ OKNO 27

MODIFIKACE SCÉNY **28**

JAZYK MDL **29**

ANIMACE V JAZYKU MDL 29
PROSTOR 30
Entity 31
Herec 32
Popis herce 33
Kamera 34
Editace tvarového stromu 35
ČINNOST 36
Časové procedury 37
Trajektorie 37

ZÁVĚR **38**

PŘÍLOHY **39**

POPIS JAZYKA MDL 39
Požadavky při implementaci jazyka 39
Zápis animace 39
Základní prvky jazyka 39
Znaky 39
Identifikátory 40
Komentáře 40
Klíčová slova 40
Výrazy 41
Podmínky 41
Příkazy 41
Jednoduché příkazy 42
Složený příkaz 43
Příkaz if 43
Příkaz while 43
Příkaz for 43
Operátory 44
Typy 44
Přehled elementárních typů 46
Číselné typy 46
Vektorové typy 46
Matice 47

PREPROCESOR A INTERPRETR 48

LITERATURA 51

OBRAZOVÁ PŘÍLOHA ..... **CHYBA! ZÁLOžKA NENÍ DEFINOVÁNA.**

Úvod

Animace je všeobecně považována za synonymum pohybu, avšak význam tohoto slova je však poněkud širší. Animace představuje libovolnou změnu, jejíž účinek lze spatřit. Budeme-li hovořit řečí počítačové grafiky, pak lze animovat polohu těles, barvu těles, průhlednost a texturu těles, jejich strukturu, osvětlení scény, pozice kamery a jiné vlastnosti.

Animace není využívána pouze v zábavním průmyslu a reklamě, ale také pro simulace v oblasti letecké a námořní dopravy, pro vědecké aplikace (např. pohyb atmosférických vrstev, v medicíně) a nově se začíná animace prosazovat ve formě stereoskopické vizualizace - virtuální reality.

Problém počítačové animace je velice komplexní, vyžaduje znalosti celého spektra oborů computer science. Jedná-li se navíc o vědeckou vizualizaci, pak se k těmto znalostem přidružuje ještě kompendium znalostí příslušného oboru. V takových případech je animace často generována simulací skutečných jevů a je početně velice náročná. Je to z toho důvodu, že se jedná o práci s velice rozsáhlými soubory dat, která mají být zpracována a zobrazena co nejrychleji. Tato kritéria se projevují v požadavcích na používaný software a hardware. Vědecká vizualizuace se využívá nejen při veřejné publikaci výsledků vědeckých prací, ale stále častěji i v samotném průběhu výzkumu.

Ve své diplomové práci jsem se zaměřil na algoritmický přístup k popisu animace. Cílem mé práce bylo vytvořit program s implementovaným animačním jazykem, který by umožňoval algoritmicky popisovat animaci. Při práci byl kladen hlavní důraz na dvě základní části problému, a to v první části navržení a implementace grafického systému umožňujícího reprezentovat a zobrazovat prostorová tělesa a v druhé částí navržení a implementace animačního jazyka. Řešení obou problémů je představeno v této diplomové práci v podobě programu MDL. Program umožňuje vytvářet, modifikovat a zobrazovat prostorové scény, jež je možno dále animovat pomocí jazyka MDL a výsledné sekvence ukládat do výstupních souborů.

Před vlastním popisem diplomové práce bych velice rád poděkoval svému školiteli RNDr. Josefu Pelikánovi ze obětavé vedení diplomové práce a výzkumného úkolu.

Počítačová grafika - animace

Konvenční animace využívá nedokonalosti lidského oka. Její princip spočívá v tvorbě dvourozměrných obrazů, které se v rychlém sledu střídají a díky anatomickým vlastnostem oka vytvářejí iluzi pohybu. Frekvence pro videorekordéry je 30 obrazů za vteřinu a pro klasický film 24 obrazů za vteřinu.

Užití počítačů v animaci se postupně vyvíjelo od vytváření středních snímků, přes vyplňování vykreslovaných oblastí, až k plně počítačové animaci. Vytváření klasické filmové sekvence je práce nesmírně časově náročná a zcela nekreativní. Je nezbytné rozkreslit jednotlivé snímky filmu, které se od sebe liší jen nepatrně. Nahrazení této práce byl jeden z prvních úkolů pro počítače. Ty dnes již běžně nahrazují kreslíře a mezi jednotlivé klíčové snímky doplňují ostatní, které by jinak museli být dokresleny ručně. Převrat v užití počítačů v grafice a zvláště v animaci přineslo až zobrazování prostorových modelů, což byl první a nejdůležitější krok k vytváření plně počítačových animací. V dnešní době se tato oblast stále rozvíjí a dosahuje mimořádných úspěchů.

Konvenční animace:

Pojmem **konvenční animace** rozumíme postup při vytváření kresleného filmu pomocí konvenční techniky (tj. bez užití počítačů). V tomto způsobu animace je ustálen následující postup [3]:

Navrhne se osnova filmu
Vytvoří se základní návrhy hlavních scén (**sketches**)
Nahraje se zvuková stopa
Přesně se stanoví klíčové scény (**key frames**)
Tvoří se zbývající obrazy mezi jednotlivými scénami (**inbetweening**)
Vytvoří se zkušební film (**pencil-test**)
Všechny obrazy zkušebního filmu se přenesou do buněk (**cells**). Při použití buněk se vytvářejí rozdílné vrstvy pro pozadí a popředí animace
Jednotlivé buňky se vybarví a nasnímají

Některé animační operace popsané ve výše zmíněném postupu lze automatizovat a užít k jejich provedení možností počítačů, a nahradit tak netvořivou a zdlouhavou práci. Především jde o stanovování středních snímků, vyplňování barevných ploch buněk a podobně. Aby se mohl obraz zpracovat, musí se nejprve přenést do formy dostupné počítači (většinou jde bitové mapy, nebo data ve vektorové formě). Obrazová data se do počítače přenáší pomocí speciálních vstupních zařízení, jako jsou **scanery** nebo **tablety**. Po přenosu do počítače se obraz očistí od případných přenosových šumů, filtruje se, zesiluje se kontrast, vyhledávají se hrany a podobně. V takto zpracované sekvenci je možno kombinovat pozadí a postavy, zvětšovat části obrazu a vystřižené části posunovat. Lze tedy využít všechny prostředky digitálního zpracování obrazu [3].

Výše popsaným způsobem lze zpracovat i klasické hrané filmy. Vkopírování, nebo vymazání určitých částí scény se stalo poměrně běžnou trikovou metodou. V poslední době se využívají dokonce trojrozměrné počítačové modely pro doplnění nahrané scény. U některých hraných filmů se triková část (vytvářená počítačem) stala dominantní částí fimu. Požadavky na nové speciální efekty byly dokonce u zrodu některých nových technik užívaných v počítačové grafice a animaci. Jako příklad lze uvést **morphing** - plynulý přechod tvaru tělesa do nového tvaru. Natočený film, reprezentovaný v počítači pomocí posloupnosti bitových map, dnes již nepostačuje nárokům na speciální efekty, proto byl navržen nový postup spojení filmu a počítačové postprodukce - **Motion Control** [4]. Během natáčení filmů počítač zaznamenává v reálném čase polohu a orientaci kamery

a dalších důležitých komponent scény. Tato data se pak využivají při vytváření trojrozměrných modelů. Jako příklad klasických filmů s počítačovými sekvencemi lze uvést filmy **The Abyss** (Propast) a **Tron**. Firmy specializující se na tuto oblast (jako třeba **ILM**) pracují nejčastěji s počítači **Silicon Graphics** a softwarovými produkty **Alias**, **Wavefront** a **SoftImage**.

Nyní se vrátíme k využití počítačů v konvenční animaci a v krátkosti si popíšeme vytváření středních snímků. Je to proces, který je zdánlivě velice jednoduchý, ale skrývá mnoho úskalí. Problémy nastávají především, klademe-li vysoké požadavky na realističnost nově vytvořených snímků. Nejjednodušším způsobem, jak nahradit chybějící obrazy je využití lineární interpolace (**lerping**). Princip této metody spočívá v tom, že se v klíčových obrazech naleznou důležité body (tzv. řídící body). Provede se párování těchto bodů mezi jednotlivými klíčovými snímky a chybějící řídící body (ve středních snímcích) se interpolují lineární interpolací. Tato metoda je velice jednoduchá, ale přináší mnoho problémů. Vezmeme-li např. šikmý vrh se třemi klíčovými body (interpolaci provádíme mezi nimi): počátek, vrchol trajektorie a místo dopadu, pak obraz vzniklý lineární interpolací neodpovídá skutečnosti. Je to tím, že metoda sice generuje spojitou trajektorii, ale její derivace jsou nespojité. Dalším problémem lineární interpolace je, že tato metoda je velice citlivá na výběr řídících vrcholů.

Je snaha nedostatky lineární interpolace odstranit použitím polynomiální interpolace vyššího stupně, kterou lze reprezentovat odpovídajícími křivkami [5]. Často se užívají parametrické kubické křivky [3]. Tyto křivky se zadávají pomocí posloupnosti řídících bodů a tím lze měnit vlastnosti přechodu mezi jednotlivými klíčovými pozicemi. Tyto křivky mají podstatně lepší vlastnosti na přechodech mezi jednotlivými klíčovými pozicemi, ale jsou početně náročnější. Vcelku se dá říci, že vyhovují požadavkům animace. Problémy vznikají pouze při interpolaci orientace těles v prostoru. Pro realističtější přechod od jedné orientace ke druhé se užívá reprezentace orientace pomocí kvaternionů [6].

Ve výše zmíněných postupech sice ovlivňujeme způsob nahrazení chybějících částí sekvence mezi jednotlivými klíčovými snímky, ale přitom podstatně neovlivňujeme změnu tvaru těles při pohybu. To umožňuje až postup reprezentace tělesa pomocí kostry (**skeleton**). Jednotlivé figury na scéně mohou být reprezentovány kostrou vytvořenou z bodů, které se mezi snímky pohybují po dané křivce. Zbytek objektu se pak "obalí" kolem kostry. Danému skeletonu je možno přiřadit mechanické vlastnosti a na generování snímků použít **inverzní kinematiky**. V tomto případě se již nejedná o klasické stanovení středních snímků, ale o pokročilou metodu plně počítačové animace.

Jednoduché animační efekty:

Jedním z nejjednodušších efektů je **periodická změna barvy** v nějaké oblasti. Tato změna vyvolá iluzi pohybu. Postup tvorby takového efektu je snadný. Definujeme paletu barev v takovém pořadí, v jakém chceme, aby za sebou následovaly. Každému elementu, který chceme zobrazovat přiřadíme určité číslo, které představuje barvu. Tato čísla peridicky měníme a tím vzniká iluze pohybu.

Dalším příkladem může být animace bitových map, která nalezla velké uplatnění v oblasti počítačových her a jednoduchých simulací. Každý pohybující se element (**sprite**) se skládá ze dvojice bitových polí. První pole obsahuje nakreslený objekt a druhé masku, která pomocí logických operací určí, který bit se zobrazí a který ne. Animace vzniká pohybem těchto elementů po obrazovce.

Rozdělení animace

Většina autorů [7] rozděluje prostorovou animaci do tři základních oddílů: **animace klíčových snímků, parametrická animace a algoritmická animace.**

Animace klíčových snímků

O tomto druhu animace jsme již mluvili v předcházející kapitole. Animační sekvence je udána posloupností klíčových snímků. Střední snímky jsou interpolovány automaticky animačním systémem. Všechny objekty v každém snímku jsou reprezentovány posloupností řídících vrcholů. Není-li odpovídající počet řídících vrcholů mezi jednotlivými klíčovými snímky stejný, musí systém určit způsob párování a tvorby nových řídících vrcholů, což je často velice komplikované.

Parametrická animace

Parametrická animace je založena na následující myšlence. Každá entita (objekt, kamera) na scéně je charakterizována množinou parametrů. Animátor vytváří klíčové snímky specifikací těchto parametrů. Sekvence se nevytváří interpolací polohy řídících vrcholů, ale interpolací parametrů. Jako příklad lze vzít translační pohyb tělesa popsaný parametrickými rovnicemi. Parametr, který tyto rovnice mění, je právě poloha tělesa.

Algoritmická animace

V tomto druhu animace je pohyb popsán seznamem transformací. Každá transformace je definovaná množinou parametrů, které se mohou měnit v průběhu animace podle přání animátora. V praxi se mění podle parametrických rovnic, nebo je změna výsledkem řešení soustavy pohybových rovnic. Protože cílem diplomové práce byla právě algoritmická animace (v podobě animačního jazyka), popíšeme podrobněji animační jazyky v následující kapitole.

Animační jazyky

Díky fyzikálním simulacím se rozvinul problém, jak co nejjednodušším způsobem popsat pohyb v prostoru resp. děj na animační scéně. K tomuto účelu byly vyvinuty animační jazyky. Je nezbytné poznamenat, že se vývoj v této oblasti vydal několika směry, které byly odrazem různého chápání popisu scény. Část jazyků se specializovala na modelování scén, což v samotných jazycích zanechalo stopy. Jiné jazyky se specializovaly čistě na popis animace.

Animační jazyky je možné rozdělit do tří základních skupin:

###lineární seznam instrukcí (**linear-list notation**)
###vyšší programovací jazyky (**general-purpose languages**)
###grafické jazyky (**graphical languages**)

Lineární seznam instrukcí:

V tomto způsobu popisu je celá sekvence rozdělena na jednotlivé snímky, které jsou očíslovány. Každá událost na scéně dostává číselný rozsah svého působení ve formě parametrů. Konkrétní událost na scéně se popíše například následujícím způsobem:

```
42, 53, B ROTATE "PALM", 1, 30
...
```

což znamená:"od snímku 42 do snímku 53 prováděj rotaci objektu PALM okolo osy 1 o 30 stupňů, kde míra rotace pro každý snímek je určena v tabulce B ". Každý povel má stanovenu metodu, jak interpolovat snímky mezi počátečním a koncovým stavem (v našem případě je to tabulka). Je jistě zcela zřejmé, že animace celé scény závisí na pořadí povelů (příkladem může být postupná rotace kolem os x a y).

Jako příklad jazyků patřících do kategorie lineárních seznamů můžeme uvést jazyk **Scefo** (SCEne FOrmat)[8], který má rysy lineárního seznamu, ale zároveň obsahuje skupinovou a objektovou hierarchii a abstrakci změn nazvanou action. Tento jazyk má některé další aspekty vyšších programovacích jazyků jako jsou například proměnné, řízení běhu programu atp., čímž se liší od standardního lineárního seznamu. Tento jazyk popisuje pouze scénu a neřídí vlastní zobrazování, k tomu účelu je nutno přidat k jazyku **Scefo** nějaký standardní zobrazovací program.

Vyšší programovací jazyky

Jiným způsobem definice animace je využití možností obecných vyšších programovacích jazyků [9][10][11]. Jednotlivé operace na scéně se definují pomocí standardních procedur, kterým se dodají potřebné parametry. Tyto procedury vytvářejí vlastní animaci. Často se užívají standardní vyšší programovací jazyky, které vytvářejí simulaci a animaci provádějí jako vedlejší efekt. Využití těchto jazyků k popisu scény a animace je nesmírně účinné, díky velkým schopnostem vyšších programovacích jazyků, avšak použití takové aplikace často vyžaduje jistou míru prográmátorských znalostí, což může například výtvarníkům působit jisté potíže.

Jako příklad může sloužit jazyk **ASAS** [9] založený na jazyce **LISP**. Zahrnuje tyto entity : vektory, barvy, polygony, objekty (množina polygonů), skupina (množina objektů), pozice kamery a osvětlení. Pozice kamery se skládá z její polohy a orientace. **ASAS** také zahrnuje velké množství geometrických transformací a operace s tělesy. Tyto operace berou těleso jako argument a vrací jeho přetransformovanou kopii. Příkladem transformací jsou: **up, down, left, right, zoom-in, zoom-out, forward a backward** (jejichž význam je patrný již z názvů). Uvedený fragment části tohoto jazyka ilustruje jeho strukturu (komentář je vždy za povelem):

```
(grasp my_cube): nastaví krychli aktivním objektem
(cw 0.05) : odrotuje aktivní objekt
(grasp camera) : nastaví kameru aktivním objektem
(right panning_speed ): pohyb doprava
```

Výhodou jazyka **ASAS** oproti jazykům typu lineárního seznamu je schopnost generovat procedurální objekty a animace . Tato schopnost je na úkor zvýšených nároků na animátora, který musí zvládat základy programování.

Grafické jazyky

Výše zmíněné textové jazyky jsou sice vhodné pro práci na scéně na té nejzákladnější úrovni, ale jejich velkou nevýhodou je chybějící možnost přímé optické kontroly při definici animace. Pokud je k dispozici preview animace v reálném čase (společně s rychlým zobrazením uložených dat), pak tato nevýhoda není patrná. Tato varianta je poměrně ojedinělá z důvodů velkých nároků na hardware.

Grafické jazyky popisují animaci jednodušší cestou, v daleko větší míře se zde uplatňuje intuitivní přístup, což je výhodou pokud chceme, aby animace vypadala tak, jak si představujeme, a nezáleží nám přitom na tom, aby se objekty pohybovaly podle předem předepsaných rovnic.

První práce v této oblasti pochází od **Baeckera** [12], který zavedl P-křivky v animačním systému **GENESYS**. P-křivka je parametrickou reprezentací pohybu objektu (nebo jiného atributu). Animátor graficky popisuje cestu objektu jako funkci času (pomocí spline křivek) tak, že zadá projekce trajektorie do jednotlivých hlavních rovin. Na jednotlivých průmětech i výsledné trajektorii se nanesou značky, které odpovídají stejnému časovému intervalu při pohybu objektu.

Diagramatický animační jazyk **DIAL** obsahuje některé rysy linenárního seznamu, odlišnost je ale v tom, že zobrazovací posloupnost událostí je popsáná paralelními řadami znaků. Vertikální čára značí iniciaci události a horizontální označuje trvání události. Jako příklad definice události v jazyce **DIAL** můžeme uvést translaci objektu:

```
% t1 translate "block" 1.0 7.0 15.3
```

V programu **DIAL** je možno napsat program asi tímto způsobem (komenář je před každým příkazem):

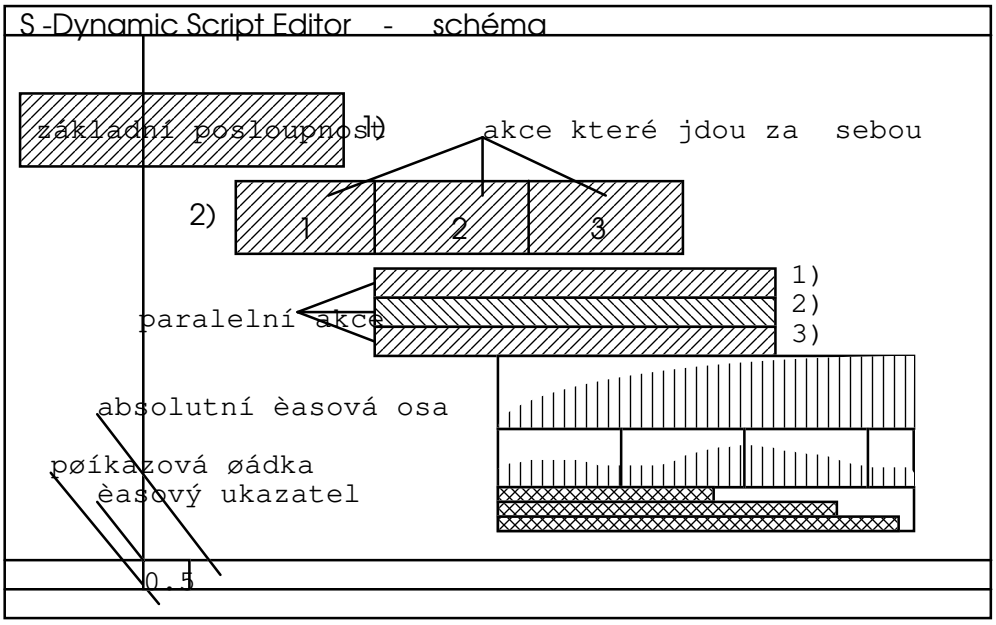
```
naète objekt ze souboru a nazve ho "BLOCK"
! getsurf "block.d" 5 5 "block"
definuje okno na xy rovine
! window -20 20 -20 20
definuje dvì události 1) translaci a 2)rotaci
% t1 translate "block" 10 0 0
% r1 rotate "block" 0 1 360
...
```

nyní můžeme popsat působení jednotlivých operací ve snímcích:

```
t1 I - - - - - - - I- - - - - - - - - - - -
r1 I- - - - - - - - - - - -
r1 I - - - - - - - - - - - - - - - - - - - - - -
```

Místo působení každé události je tedy specifikováno čárkami. Delší popisy scény se píší na další řádek, který označíme identifikátorem události.

S-DYNAMIC [10] se v grafickém přístupu dostal o krok dál a kombinuje výhody jazyka **DIAL** s výhodami **P-křivek**. Program je opatřen prostředím s okny. Přebírá popis z **DIALu**, ale navíc se místo vodorovných čárek užívá svislých čárek, které mění svou délku podle aktuální polohy v daném snímku. Čárky jsou pro každou událost jsou nakresleny v odpovídajícím okně, jehož poloha se mění v závislosti na počátku této události . Symbolické schéma dialogového okna je na následujícím obrázku:



Metody řízení animace

Řízení animace je nezávislé na konkrétním jazyce, který pro danou aplikaci použijeme. Míra řízení celého procesu se pohybuje od explicitně předepsaného průběhu, kde animátor musí popsat každou akci až po počítačem kompletně modelovaný průběh. V následujících kapitolách bude podán krátký přehled nejdůležitějších způsobů řízení. Mezi způsoby řízení patří [3]:

úplné explicitní řízení
procedurální řízení
systémy založené na omezeních
snímání živého pohybu
herec

Tyto metody často přecházejí jedna v druhou a mají mnoho společných rysů. Z následujícího stručného popisu jednotlivých přístupů bude patrné, že se tyto metody často neliší v základní myšlence, ale pouze ve způsobu její technické interpretace.

Úplné explicitní řízení

Tato metoda je nejjednodušší z metod řízení. Animátor zde popisuje vše, co je důležité pro běh animace, od operací (translace, rotace..) přes polohu těles až po druh interpolace.

Příkladem úplného explicitního řízení může být **BBOP**[14] systém, který poskytuje interaktivní druh řízení. Objektový model obsahuje hierarchicky spojené mnohostěny. Animátor může interaktivně zasahovat do každého spoje pomocí konkrétního vstupního zařízení, může také ovlivňovat v těchto bodech transformační matice. Tento způsob interakce se uplatňuje na klíčové snímky, ostatní se generují podle zadané interpolace. Tento způsob řízení sice na jedné straně umožňuje zasahovat do všech parametrů tvorby scény a animace, ale na druhé straně je celkem neefektivní, protože zde není umožněna přímá provázanost mezi jednotlivými událostmi.

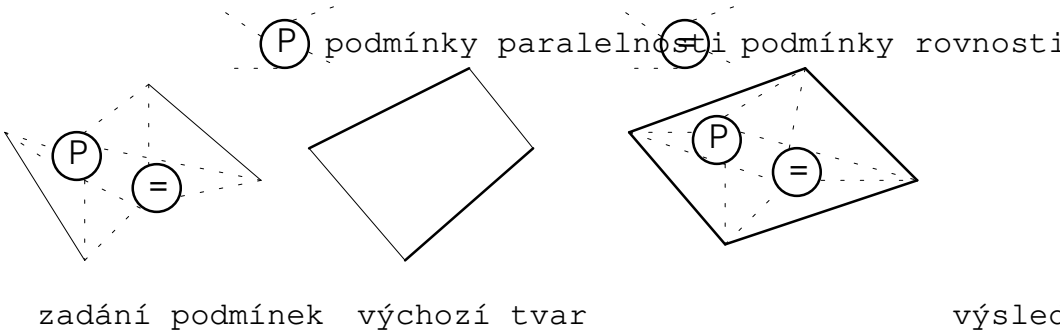
Procedurální řízení

Ve své podstatě se jedná o opak metody explicitního řízení. Jde o způsob vytváření animace pomocí vzájemného působení těles, která mění své vlastnosti v reakci na změnu vlastnosti určité entity na scéně. Vhodným příkladem by mohla být simulace pohybu trávy ve větru. Pro modelování zvolíme částicový systém. Částice vzduchu ovlivňují pohyb trávy a naopak. Tato úloha by byla metodou explicitního řízení prakticky nerealizovatelná. Procedurální řízení je často součástí jiných modelů .

Systémy založené na omezeních

Většina objektů v přírodě se nepohybuje po přímkách, protože jejich pohyb je ovlivňován jinými tělesy - je omezen souborem podmínek. Jednoduchým příkladem může být pohyb převodů, ozubených kol nebo pohyb hřídele.

Specifikace animační sekvence modelu pomocí podmínek je velice jednoduchá a názorná (často jednodušší než pomocí přímé metody). Uživatel definuje posloupnost podmínek (silové působení, vzájemné vazby), které má model splňovat. Prvním modelem tohoto typu byl **Sutherlandův Sketchpad system** [15], který umožňoval sestavovat prvky schématu tak, jak to dělají 2D programy dnes. Jednotlivé elementy systému (úsečky, kružnice, atp.) mohly být zatíženy bodovými podmínkami (např. “tato úsečka se může volně pohybovat, ale jeden konec je upevněn”), dále zde byly zavedeny různé podmínky na spojení (např. “tato úsečka musí být vždy spojena s daným bodem”) a konečně mohli být v systému zahrnuty omezující podmínky pro úhly mezi danými elementy. Tyto podmínky umožňovaly vytvořit objekty s požadovanými vlasnostmi pomocí metody relaxace. Jako příklad můžeme uvést čtyři úsečky v čtyřúhelníku, které zatížíme podmínkami spojů v jeho rozích, dále podmínkou, že každé dvě protější strany budou paralelní a stejné délky. Postup je znázorněn na obrázku:



Co je to metoda relaxace? Máme-li soubor podmínek a chceme-li, aby scéna těmto požadavkům vyhovovala, pak měníme možné vlastnosti nebo charakteristiky scény tak, aby její vlastnosti co nejlépe vyhovovaly podmínkám, to znamená, aby byly požadované vlastnosti splněny, nebo jim byly co nejblíže.

Snímání živého pohybu

V zásadě se při tomto druhu řízení aplikují dva přístupy. První spočívá v natočení a následné digitalizaci snímané scény. Při tomto postupu je možno například odstranit pozadí a nahradit jej jiným . Druhý přístup využívá pro specifikaci pohybu význačných bodů, které jsou snímány a převáděny do počítače. Techniky tohoto snímání jsou různé. Od malých světel až po datové rukavice a snímače polohy jednotlivých částí těla.

Herec

Tento způsob chápání řízení animace je typický pro animační jazyky objektově založené. Herec (**actor**) je procedura v programu, která předurčuje pro každý objekt jeho chování v dalším snímku. Tato procedura může vysílat a přijímat zprávy z programu a tím ovlivňovat chování ostatních herců..

Fyzikální simulace

Fyzikální simulace bývají často děleny stejným způsobem jako odpovídající oblasti fyziky. Místo o animaci se zde častěji mluví o **vizualizaci**.

Systém pro fyzikální simulaci by měl obsahovat výhody předcházejících způsobů řízení. Často obsahuje specializované celky, které "dohlížejí " například na kinematiku, dynamiku, deformaci tělesa a tření. Je jasné, že chování tělesa při fyzikální simulaci musí odpovídat realitě světa, tudíž musí být možnost působit na pohyb tělesa pomocí jiných těles. Aby systém mohl vyhovovat předchozímu požadavku, musí v něm být tato možnost implementována.

Obecně platí, že komplexní fyzikální simulace je jedna z nejtěžších možností aplikace počítačové animace resp. modelování.

Závěr

Ve výše zmíněných kapitolách jsme uvedli různé druhy popisu animace a způsoby jejího řízení. V následující části tohoto textu se budeme věnovat popisu autorského programu **MDL**. Animaci generovanou tímto programem lze v kontextu předcházejícího členění označit jako algoritmickou animaci s procedurálním řízením. Program sám ovšem nese silné rysy interaktivního přístupu k definici prostorové scény, což je především vlastnost grafickych jazyků. Hlavní snahou při tvorbě tohoto programu bylo vytvořit maximálně interaktivní část pro tvorbu a modifikaci scény a naopak ryze algoritmickou část pro popis animace.

Program MDL

Výsledkem práce na zadaném úkolu je program **MDL**. Tento program umožňuje interaktivním způsobem modelovat a animovat prostorovou scénu. Program vznikl podstatným přepracováním a rozšířením programu **Intro**, který byl výsledkem výzkumného úkolu [16].

Program **MDL** umožňuje interaktivně z vytvářet geometrických elementů prostorovou scénu. Tuto scénu je možné modifikovat, a poté ukládat na disk. Hotovou scénu je možno zobrazit různými zobrazovacími algoritmy a v různém rozlišení. Uživatelem zapsaný algoritmus v podobě textového souboru pak umožní celou scénu animovat. Výsledná animovaná sekvence je uložena do souboru a samostatně spustitelná. Celá komunikace s programem probíhá pomocí vlastního uživatelského rozhraní, které je jeho součástí.

Program byl napsán v objektově orientovaném jazyku **C ++,** protože koncepce objektového programovaní dobře odpovídá intuitivnímu chápání grafiky (např. tvorba složitějších objektů z elementárních)

Program byl kódován pomocí překladače **Borland C++ 3.1**. Je určen pro počítače třídy **PC** s procesorem řady **386** (a vyšší) a grafickou kartou **SVGA** (alespoň **1MB**) standardu **VESA**.

V následujících kapitolách postupně projdeme všemi tématickými oblastmi, které se týkají programu MDL. Nejprve se zmíníme o datových strukturách a způsobu uložení těles v programu, pak popíšeme jakým způsobem program tato tělesa transformuje a zobrazuje, nakonec popíšeme vlastní animační jazyk a jeho interpretaci.

Základní datová struktura programu MDL-Vektorový prostor

Základní datové třídy jsou odvozeny od abstraktního pojmu lineárního vektorového prostoru. Tento prostor je realizován množinou deklarací šablon [17] základních objektových typů. Šablony jsou dvouparametrické a jsou vzorem třídám reprezentujícím prvky prostoru. První parametr šablony určuje dimenzi prostoru a druhý typ složek jeho vektorů. V programu jsou použity instance základních integrálních typů, a to: **`int, long int, float`** a **`double.`** Nyní si podrobněji popíšeme jednotlivé typy šablon.

Skalár

Tato šablona, jak už název napovídá, představuje skalár. Každá konkrétní instance obsahuje hodnotu tohoto prvku a dále pak množinu základních algebraických operací (sčítání, násobení) a relací (porovnávání).

Vektor

Šablona generující vektorové třídy vytvoří podle hodnoty svého prvního parametru, pole jemu odpovídajícího rozměru, se složkami odpovídajícími druhému parametru.

Podobně jako skalár má i konkrétní instance vektorové třídy některé důležité metody. Jsou to především základní algebraické operace : sčítání, násobení skalárem, skalární součin, vektorový

součin, výpočet normy vektoru a jiné. Algoritmy těchto metod jsou totožné s matematickými postupy, a proto je zde nebudeme popisovat.

Matice

Posledním důležitým typem šablon jsou matice představující operátory v prostoru generovaném šablonou vektorů. V programu MDL jde o čtvercové matice řádu odpovídajícího prvnímu parametru šablony se složkami zadanými druhým parametrem. Mezi metody konkrétních instancí této šablony patří : sčítání matic, násobení skalárem, násobení matice a vektoru a další.

Vyšší datové struktury programu MDL- Transformace,Objekt a Scéna

Od základních datových typů popsaných v předcházející kapitole jsou odvozeny hierarchicky vyšší objektové třídy. Jsou to především třídy popisující jednotlivé **transformace, tělesa** na scéně a konečně celou **scénu**.

Transformace

Některé entity (okno, těleso a scéna) v programu MDL mohou obsahovat tabulky transformací, podle kterých se transformuje množina vektorů. Prvkem každé tabulky je právě instance třídy transformace.

Transformace je určena uspořádanou trojicí:

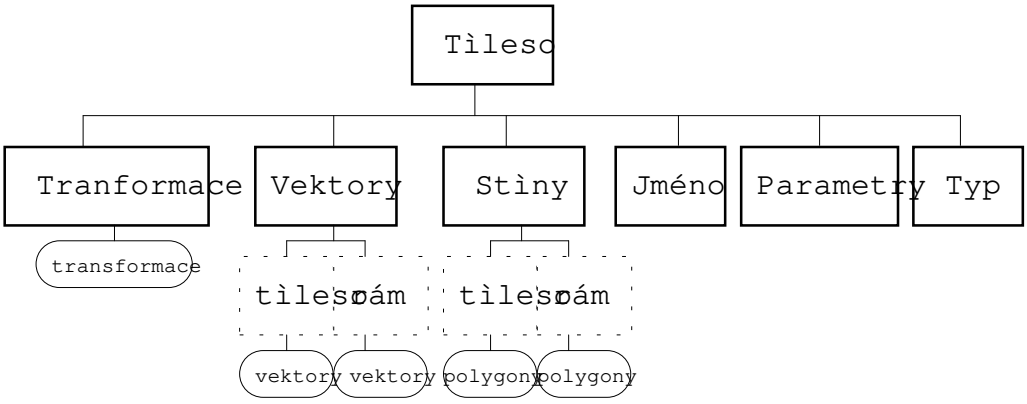
(typ, parametr,parametr)

Jak již napovídají názvy jednotlivých argumentů, typ určuje o jakou transformaci jde (např. rotace, translace nebo změna měřítka) a parametry určují parametry konkrétní transformace.

Těleso

První třídou mající intuitivní vztah k zobrazované scéně je třída představující těleso.V definici této třídy jsou obsaženy další objektové typy, jejichž význam jsme již vysvětlili v předcházejících kapitolách, a které takto nepřímo vysvětlíme. Společně s atributy této třídy zdůrazníme i některé její metody.

Schématicky lze tuto třídu znázornit stromovou strukturou, jejíž jednotlivé prvky následně vysvětlíme.



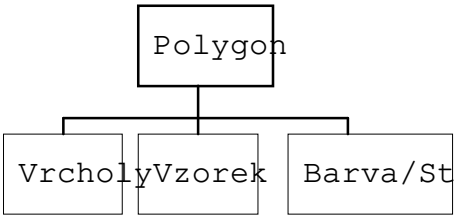
Na těleso je možno aplikovat transformace uložené v tabulce **Transformace**. Tabulka je realizována dynamickým polem transformací popsaných výše. Tyto transformace se aplikují v takovém pořadí, v jakém byly ukládány do tabulky. Způsob aplikace jednotlivých transformací závisí na způsobu reprezentace tělesa v programu. Tato reprezentace je uložena v častech třídy nazvané **Vektory** a **Stěny**.

V programu MDL jsou tělesa reprezentována dvojím způsobem, a to buď jako seznam polygonálních stěn, nebo pomocí kubických křivek a ploch. V obou případech je však třeba ukládat seznamy řídících vrcholů. Tyto seznamy jsou právě v části nazvané **Vektory**. U těles reprezentovaných seznamem polygonů jsou informace o polygonech uloženy v části nazvané **Stěny**. V případě, že těleso není tohoto typu, je tato část prázdná (až na stěny obalového tělesa).

Každému tělesu je pro snadnější a rychlejší manipulaci přiřazeno tzv. **obalové těleso**. Toto těleso je vytvořeno tak, aby všechny řídící vrcholy vlastního tělesa byly v tomto obalovém tělese obsaženy. Tímto tělesem je kvádr. Kvádr je přiřazen tělesu v době jeho vzniku. Jeho tvar a poloha se řídí všemi polohovými a tvarovými změnami samotného tělesa. Data obalového tělesa jsou uložena (stejně jako data vlastního tělesa) v částech třídy nazvané **Stěny** a **Vektory**.

Aby byl popis těchto částí úplný, je nutné zmínit se o tom, že seznamy vektorů (jak vlastního, tak i obalového tělesa) jsou zdvojeny. Nyní si podrobněji vysvětlíme důvod tohoto rozdělení. První seznam nazveme **maticí** a druhý **bází**. Tyto seznamy mají za úkol uchovávat nestrukturované pole vektorů, ze kterých se každé těleso skládá. Nestrukturované pole znamená, že vektory, které jsou v něm uloženy, následují za sebou bez jakékoliv prostorové souvislosti. Každý řídící vektor tělesa je prakticky zdvojen, jednou se vyskytuje v **matrici** a podruhé v **bázi,** a to ve stejném pořadí (tj. indexy si v obou případech odpovídají). Rozdíl mezi těmito seznamy je ovšem značný. **Matrice** slouží jako vzor pro posloupnost transformací a **báze** jako pole, které zaznamenává výsledky dílčích transformací a tyto výsledky dále předává jako vzor pro další transformace . V praxi je tato alternace seznamů využita tak, že před provedením posloupností transformací, se vektory v **bázi** aktualizují vektory z **matrice**. Jednotlivé transformace pak převádí vektory z **báze** opět do **báze**. Důvod tohoto zdvojení seznamu vektorů tělesa je zcela zřejmý. Můžeme si to ilustrovat na příkladě postupné rotace tělesa kolem jedné osy. Kdybychom vzali výsledek předcházející rotace jako vzor další rotace, pak bychom už při malém počtu pootočení obdrželi zdeformované těleso. Tomuto se lze vyhnout právě zdvojením seznamů vektorů. Necháme proběhnout žádané transformace a ty, jak již bylo řečeno, změní hodnoty vektorů v **bázi** na jiné a výsledek se opět uloží do **báze**. Nyní zobrazíme těleso (tj. využijeme hodnoty v **bázi**). Před novým vykreslovacím cyklem překopírujeme původní vektory z **matrice** do **báze** a činnost pokračuje dál stejným způsobem.

V předcházejícím odstavci jsme se zmínili, že některá tělesa jsou popsána seznamem polygonálních stěn. Každá stěna je určena posloupností vektorů - jejich řídících vrcholů. Tato názorná představa je v programu MDL realizována následovně. Všechny vektory, které se v tělese vyskytují, jsou paralelně uloženy jak v **matrici**, tak i v **bázi**. V obou případech odpovídají témuž vektoru tytéž indexy. Proto je možné každou stěnu jednoznačně popsat jako pole indexů vektorů, ze kterých se tato stěna skládá. A právě tuto funkci plní třída **Polygon**. Seznam všech stěn (polygonů) v tělese je uložen do dynamického pole, které se nachází v části třídy nazvané **Stěny**. Nyní se budeme podrobněji věnovat třídě



reprezentující polygon. Nejlépe vysvětlíme jednotlivé atributy této třídy na přiloženém schématu. Dynamické pole **Vrcholy** obsahuje indexy vektorů uložených v **matrici** (a zároveň v **bázi**), které jsou vrcholy tohoto polygonu. **Vzorek** označuje číslo stínovací matice (podrobněji v kapitole věnované zobrazování těles), se kterou bude polygon vyplněn. Atribut **Barva**/**Stín** identifikuje barvu polygonu resp. hodnotu konstatního stínu.

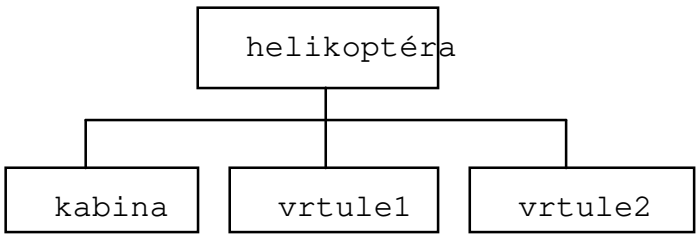
Dále budeme pokračovat ve výčtu zbývajících atributů tělesa. **Jméno** je devítipísmennou identifikací tělesa (používá se například u animace). Část nazvaná **Parametry** shrnuje pomocné parametry tělesa, jako jsou hodnoty beta parametrů kubických ploch a křivek. Posledním důležitým atributem tělesa je jeho **Typ**. Každý druh tělesa má svoje interní identifikační číslo reprezentované příslušným makrem. Tyto typy v podstatě odpovídají nabízeným geometrickým primitivům. Podle typu tělesa se rozhoduje, jaký druh vykreslování se zvolí, jak interpretovat data v instanci apod. Program MDL interně eviduje následující typy: *POLLINE* (lomená čára, speciálně bod nebo úsečka), *POLYGON* (polygon), *PRISM* (hranol), *PLANE* (rovina), *SPHERE* (polyhedrální aproximace koule), *PYRAMID* (jehlan), *CRTEMPL* (rotační šablona stahující krajní body k ose rotace), *ORTEMPL* (rotační šablona nestahující krajní body k ose rotace, např. prstenec), *SPIRAL* (spirála), TRANTEMPL (translační šablona), *BEZCUR* (Bézierova křivka), *BSPCUR* (B-Spline křivka), *BETCUR* (Beta-Spline křivka), *BEZSUR* (Bézierův povrch), *BSPSUR* (B-Spline povrch), *BETSUR* (Beta-Spline povrch), *SWPBET* (translační šablona z Beta-Spline křivky), *BETSURO* (Beta-Spline povrch z otevřené rotační šablony), *BETSURC* (Beta-Spline povrch z uzavřené rotační šablony) a TRBETSUR (translační šablona z Beta-Spline plátů).

Scéna

Hierarchicky nejvýše položenou třídou v datové struktuře programu MDL je třída představující scénu. Instance této třídy obsahuje informace o tělesech, které se nacházejí na scéně. Tyto informace jsou uloženy ve dvou strukturách, které nyní popíšeme.

První strukturou je **lineární seznam**, který uchovává všechna tělesa, která mohou být zobrazena na scéně. V tomto seznamu jsou jak uživatelem definovaná tělesa, tak tělesa užívaná programem (osový kříž, rovina a tělesový buffer).

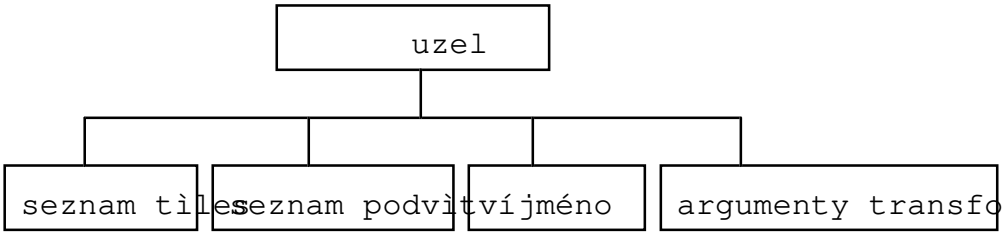
Podstatně komplikovanější je druhá struktura - **seznam tvarových stromů**. Lineární seznam sice poskytuje všechny informace o tělesech, ale pro účely animace je nepostačující resp. existují struktury podstatně výhodnější. Tyto struktury mohou totiž vyjadřovat i vzájemné vztahy těles, které jsou součástí jednoho celku. Příkladem takové struktury je acyklický graf - strom. Výhodnost ukládání některých dat o tělesech do této struktury vysvětlíme na následujícím příkladě. Naším úkolem je například modelovat pohyb helikoptéry, která je (pro jednoduchost) složena z kabiny a dvou vrtulí. Této scéně by odpovídal v programu MDL tento strom:



Je vidět, že tvorba stromu odpovídá členění tělesa na logicky nižší části. Kořen stromu představuje celé těleso, které se skládá ze všech svých elementů (podstromů). Tyto podstromy mohou být opět dále členěny na další elementy. Máme-li těleso takto uloženo, pak je pro nás mnohem jednodušší provádět požadované pohyby, protože můžeme provádět polohové transformace s jednotlivými částmi tvarového stromu. V našem případě to znamená, že aplikujeme rotace na větve tvarového stromu, kde jsou popsány vrtule. Tak zajistíme jejich nezávislé otáčení.

Bez ohledu na to, jaký pohyb vykonávají vrtule, můžeme zároveň měnit polohu celého vrtulníku (včetně vrtulí) tak, že aplikujeme translaci na kořen stromu. Kdybychom měli pouze lineární seznam těles (tj. kabina, vrtule1 a vrtule 2), pak by uživatel musel zadávat translaci a rotaci pro každé těleso zvlášť a v některých případech dokonce superponovat vlivy více pohybů.

Nyní si podrobněji popíšeme realizaci tvarového stromu v programu MDL. Základním prvkem stromu je uzel, který je reprezentován třídou popsanou na následujícím schématu.



Uzel obsahuje seznam odkazů na objekty, které tvoří tento uzel. Tyto odkazy vedou do výše zmíněného lineárního **seznamu tělesa** na scéně. Část objektu, kterou popisuje uzel je tvořena i jinými tělesy, na něž jsou odkazy uloženy v uzlech patřících některé jeho podvětvi. Uzel dále obsahuje **seznam podvětví**, **vlastní jméno** a **tabulky argumetů transformací** (podrobněji v popisu jazyka MDL).

Konstrukce objektů

V předcházející kapitole jsme si popsali způsob ukládání dat v programu MDL. Nyní vysvětlíme, jak jsou interpretována data uložená v jednotlivé instanci třídy **těleso**. Nebudeme popisovat konkrétní způsob konstrukce, protože je podrobně popsána v nápovědě, která je přítomna v programu.

V programu je užita hraniční reprezentace těles [3]. Všechna tělesa (resp. jejich reprezentace) lze v podstatě rozdělit do dvou skupin. V první skupině jsou tělesa **polyhedrálního typu** a ve druhé **tělesa reprezentovaná kubickými křivkami resp. bikubickými plochami** (tato tělesa budeme dále nazývat kubickými)**.** Jak bylo již dříve řečeno, jsou všechna tělesa instancemi třídy reprezentujícími těleso, a tudíž mají všechny informace uloženy ve stejných datových strukturách. Ovšem jejich reprezentace se liší. V následujících částech popíšeme podrobněji právě tuto reprezentaci.

Tělesa polyhedrálního typu

Nejjednoduššími objekty, jež jsou poskytovány uživateli programem MDL, jsou objekty polyhedrálního typu. Tyto objekty jsou tvořeny posloupností řídících vrcholů a stěn z nich složených. Těleso chápeme jako prostor uzavřený množinou jeho stěn (v případě prostorových těles), nebo jako množinu stěn (v případě samotných ploch). Tento způsob reprezentace je výhodný zejména pro svou jednoduchost . Při transformaci se například transformují pouze řídící vrcholy. Nevýhodou je pak relativně malá třída těles, kterou lze tímto způsobem reprezentovat.

Při konstrukci těles tohoto typu se užívají tři způsoby konstrukce, a to **jednoduché formy, přímé zadávání** a **šablony [3]**.

Konstrukce těles pomocí **jednoduchých forem** je založena na množině parametrických prototypů těles. Výsledné těleso se získá zadáním žádaných parametrů. Program poskytuje následující jednoduché formy: **koule a rovina**.

Přímé zadávání se užívá pouze při konstrukci **lomené čáry** (resp. bodu, čáry). Tímto způsobem se zadá vstupním zařízením posloupnost řídících vrcholů.

Posledním, ale nejčastěji užívaným způsobem, je zadávání pomocí **šablon**. Tímto způsobem se konstruuje prostorové těleso z polygonu, který se nazývá šablona. Těleso vznikne buď rotací, nebo translací šablony, pak mluvíme o tzv**. rotačních a translačních šablonách**. V programu MDL jsou implementovány následující šablony a jejich modifikace: **translační šablona, rotační šablona, spirála, hranol a jehlan.**

Kubická tělesa

Kubické těleso je popsáno posloupností (resp. maticí) řídích vektorů. které plně popisují tvar kubické křivky (resp. plochy). Kubické křivky se skládají z jednotlivých segmentů a kubické plochy z jednotlivých plátů. Na zadání jednoho segmentu kubické křivky jsou třeba 4 vektory, obdobně na zadání jednoho plátu kubické plochy je třeba matice 4x4 vektory.

Kubická tělesa (stejně jako polyhedrální tělesa) využívají k uložení dat instancí třídy popisující těleso. Nevyužívají však všech argumentů této instance. V **matrici** a **bázi** jsou uloženy seznamy řídících vektorů. V závisloti na typu tělesa se tyto seznamy reprezentují různým způsobem. Podrobně si je nyní vysvětlíme u jednotlivých typů kubických těles.

Bézierovy křivky

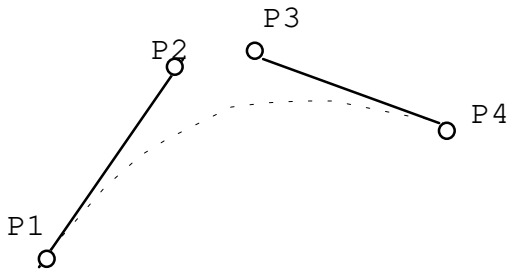
Bézierova křivka stupně **n** je definovaná posloupností **n+1** řídících bodů a její parametrické vyjádření má tvar

$$\mathbf{Q(t)} = \sum_{i=0}^{n} \mathbf{B_{i,n}(t).P_i} \qquad \text{,kde } \mathbf{B_{i,n}} = \binom{n}{i}.\mathbf{t^i(1-t)^{n-i}}$$

Kde **t** je z intervalu **###0,1###** a B jsou tzv. **Bernsteinovy polynomy** Pro kubické Bézierovy křivky, dostáváme tvar :

$$\mathbf{Q(t) = (1-t)^3P_1 + 3t(1-t)^2P_2 + 3t^2(1-t)P_3 + t^3P_4}$$

Kubické Bézierovy křivky jsou určeny krajními body **P1, P2** a tangentami v nich, které jsou vyjádřeny nepřímo, prostřednictvím pomocných bodů **P2,P3** (viz. obr.).



Zadáváme-li v programu MDL Bézierovu křivku, pak je tím myšleno, že zadáváme posloupnost čtveřic řídících bodů. Může se stát, že počet vrcholů neodpovídá počtu Bézirových úseků, potom se přebývající vrcholy vynechají. Všechny vrcholy jsou za sebou seřazeny tak, jak se zadávaly v lomené čáře definující řídící vrcholy. První segment využívá první až čtvrtý řídící vektor, druhý segment čtvrtý až sedmý atp. Každé dva sousední intervaly mají společný řídící bod (tj. poslední resp. první), ze způsobu konstrukce plyne, že křivka vytvořená z posloupností Bézierových křivek je spojitá. Spojitost prvních derivací není na přechodech zajištěna, ale lze ji snadno docílit volbou řídících bodů. Program MDL totiž umožňuje zobrazit derivace v krajních bodech, a tím umožňuje vizuálně kontrolovat derivace na přechodu.

Bézierovy plochy

Logickým rozšířením Bézierových křivek jsou plochy téhož jména. Na vytvoření jednoho elemetu plochy je nyní třeba matice 4x4 řídící body, které mají stejnou funkci jako řídící body u křivek. Element plochy má parametrické vyjádření ve tvaru:

$$\mathbf{Q(s,t)} = \sum_{i=0}^{3}\sum_{j=0}^{3} \mathbf{B_{i,3}(s).B_{j,3}(t).P_{i,j}}$$

Kde funkce **B** (jsou Bernsteinovy) jsou totožné s funkcemi u křivek a **t,s** jsou z intervalu **###0,1###**.

Všechny charakteristické vlastnosti Bézierových křivek se přenášejí na Bézierovy plochy. V matrici a bázi jsou lineárně seřazeny řádky sítě řídících vrcholů, vzniklé při konstrukci.

B-Spline křivky

Program MDL také umožňuje zadávat B-Spline křivky, které mají proti Bézierovým křivkám hlavně tu výhodu, že jsou spojité až do druhého řádu (včetně) [16].

Jeden úsek B-Spline křivky má následující parametrické vyjádření:

$$\mathbf{Q(t)} = \frac{\mathbf{(1-t)^3}}{\mathbf{6}}\mathbf{P_0} + \frac{\mathbf{3t^3-6t^2+4}}{\mathbf{6}}\mathbf{P_1} + \frac{\mathbf{-3t^3+3t^2+3t+1}}{\mathbf{6}}\mathbf{P_2} + \frac{\mathbf{t}}{\mathbf{6}}\mathbf{P_3}$$

$$\mathbf{0 \le t \le 1}$$

Tuto křivku zadáváme opět jako posloupnost řídících bodů. Opět se posloupnost řídících vrcholů zadáva lomenou čarou a je lineárně uložena jak v matrici, tak v bázi. Způsob přiřazení jednotlivých vrcholů k segmentům se však liší. Stejně jako u Bézierovy křivky každé čtyři po sobě jdoucí vrcholy zadávají jeden její úsek. Na rozdíl od výše zmíněného případu se dva po sobě následující segmenty téměř překrývají. Skládá-li se první čtveřice z bodů 1,2,3,4 , pak druhá z bodů 2,3,4,5.

Mezi prvním a druhým (stejně tak jako mezi předposledním a posledním) řídícím uzlem není křivka definována, což je obecná vlastnost B-Spline a ###-Spline křivek.

###-Spline křivky

Z hlediska vlastností a způsobu zadávání jsou ###-Spline křivky zcela totožné s B-Spline křivkami (pro speciální parametry). ###-Spline křivky však mají jednu podstatnou výhodu, jsou mnohem variabilnější. Obsahují dva parametry ###1, ###2, kterým se také říká **sklon** a **napětí**. Sklon nabývá kladných hodnot a ovlivňuje souměrnost křivky vzhledem k řadě definičních uzlů. Standardní hodnota sklonu je 1. Pro nižší sklon se křivka posouvá k počátečním bodům, pro sklon větší než 1 ke koncovým. Pro napětí rovno 0 se křivka chová jako B-Spline, jsou-li hodnoty vyšší pak body křivku více "přitahují" a jsou-li nižší, pak body křivku "odpuzují". Jeden úsek křivky má následující parametrický tvar:

$$\mathbf{Q(t)} = \sum_{i=0}^{3} \mathbf{Bet_i(t,\beta_1,\beta_2).P_i}$$

Kde **t** je z intervalu **###0,1###**, **P** jsou řídící body a **Bet** jsou váhové funkce tvaru:

$$\mathbf{Bet_0}(t,\beta_1,\beta_2) = \frac{2}{\delta}(1-t)^3,$$

$$\mathbf{Bet_1}(t,\beta_1,\beta_2) = \frac{1}{\delta}(2\beta_1^3 t(t^3-3t+3) + 2\beta_1^2(t^3-3t^2+2) + 2\beta_1(t^3-3t+2) + 2\beta_2(2t^3-3t^2+1)),$$

$$\mathbf{Bet_2}(t,\beta_1,\beta_2) = \frac{1}{\delta}(2\beta_1^2 t^2(-t+3) + 2\beta_1 t(-t^2+3) + \beta_2 t^2(-2t+3) + 2(-t^3+1)),$$

$$\mathbf{Bet_3}(t,\beta_1,\beta_2) = \frac{2}{\delta}t^3,$$

kde je $\delta = 2\beta_1^3 + 4\beta_1^2 + 4\beta_1 + \beta_2 + 2$

Beta-Spline plochy

Beta-Spline plochy jsou odvozeny podobně jako stejnojmenné křivky. Parametrické vyjádření jednoho úseku plochy je následující

$$\mathbf{Q}(s,t) = \sum_{i=0}^{3}\sum_{j=0}^{3} \mathbf{Bet_i}(s,\beta_{1s},\beta_{2s}).\mathbf{Bet_j}(t,\beta_{1t},\beta_{2t}).\mathbf{P_{ij}}$$

Kde funkce **Bet** jsou totožné s funkcemi definovanými u křivek, **s,t** jsou opět z intervalu **<0,1>**. Implementace těchto křivek v programu se od tohoto vyjádření liší pouze v tom, že koeficienty b jsou shodné v obou směrech, což je sice na újmu obecnosti, ale v praktických konstrukcích se požadavek na jejich různost neprojeví.

Kubické šablony

Stejných postupů jako u polyhedrálních šablon se užívá u kubických šablon. Rozdíl je pouze v tom, že vrcholy získané pomocí šablon nejsou chápány jako vrcholy polyhedru, ale jako řídící body kubických beta-Spline ploch.

Zobrazovací metody

V programu MDL jsou implementovány tři zobrazovací metody a několik jejich variant: Metoda **zobrazení drátového modelu (Wire-frame)**, metoda zobrazení **prostorovým řezem (Depth-clipping)** a metoda **paměti hloubky (Z-Buffer)**.

Tyto metody se od sebe podstatně liší, a to zejména kvalitou výsledného zobrazení, použitým algoritmem, rychlosti a paměťovou náročností.

Metoda zobrazení drátového modelu je velice rychlá, paměťově nenáročná a je určená pro běžnou práci se scénou. Má však podstatné nevýhody. Neumí odstranit skryté linky a nezobrazuje plochy.

Metoda prostorového řezu je už skutečnou zobrazovací metodou, protože odstraňuje skryté linky. Je paměťově nenáročná, ale ve srovnání s následující metodou až čtyřikrát pomalejší, neumožňuje (v dané implementaci) také jiné než konstantní stínování.

Poslední metodou je metoda paměti hloubky. Jde o velice rychlou, ale paměťově náročnou metodu. Tato metoda umožňuje práci s barvou, světelnou intezitou a texturami. Tato metoda se často užívá a existuje dokonce hardware, který je schopen ji užít.

Zobrazení drátového modelu

Jak již bylo řečeno mají všechna tělesa data uložena ve stejné třídě. Ale z předcházející kapitoly je zřejme, že interpretace těchto dat je různá. Proto bylo nutno rozdělit vykreslovací metody podle typu tělesa.

Nejsnadněji se vykreslují tělesa **polyhedrálního typu**, protože všechna potřebná data jsou již k dispozici, a tudíž není třeba další výpočtů. V konkrétní instanci polyhedrálního tělesa je uložen seznam polygonálních stěn tělesa. Každá polygonální stěna je složena z indexů jejich vektorů odkazující do **matrice** (resp. **báze**). Těleso, které prošlo prostorovými a pohledovými transformacemi má jejich výsledek uložen v **bázi**. Chceme-li těleso nyní vykreslit, tak pouze projdeme seznam polygonů, načteme odpovídající ztransformované vektory a polygony vykreslíme.

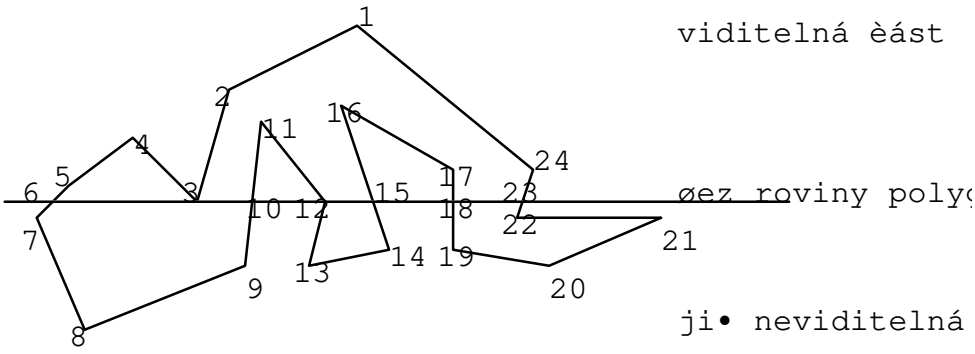
Složitější situace je u **kubických těles**, kde je nutno z řídících bodů křivek či ploch při každém vykreslení napočítat skutečné vrcholy polyhedru, který těleso aproximuje. Toto aproximační těleso je složeno pouze z úseček (v případě křivek), nebo trojúhelníků (v případě ploch), které se postupně vykreslují. Postup při získávání aproximačního tělesa je následující:

1. Získá se posloupnost vrcholů, která plně určuje segment křivky, nebo plochy
2. Podle konkrétních vzorců se spočte parametrické vyjádření segmentu křivky nebo plochy. Parametry v těchto rovnicích mohou nabývat hodnot od 0 do 1.
3. Výše zmíněný interval se rozdělí podle požadované přesnosti na daný počet stejných podintevalů. V bodech tohoto rozdělení se spočtou hodnoty udané parametrickými rovnicemi. Tyto hodnoty představují vrcholy aproximačního tělesa.
4. Z těchto bodů vytvoříme lomenou čáru resp. množinu trojúhelníků, která dobře popisuje odpovídající segment, či plát kubického tělesa.
5. Jde-li o poslední segment, pak skončíme, jinak pokračujeme opět na 1).

Výše popsaným způsobem se vytvoří aproximační síť resp. lomená čára. Funkční předpisy, podle kterých se jednotlivá tělesa vypočítávají jsou uvedeny v kapitole věnované konstrukci těles .

Zobrazení prostorovým řezem

Tento zobrazovací algoritmus je ve své podstatě triviální. Budeme-li vykreslovat odzadu (myslí se ve směru pohledu) jednotlivé řezy scény, pak při jemném kroku řezu budou bližší řezy zobrazeny později, což má za následek odstranění skrytých linek. Každý objekt ať polyhedrálního či jiného typů se postupně rozloží na polygony. Způsob tvorby aproximačního tělesa je popsán v předcházející kapitole. Při každém řezání tělesa se na každý polygon aplikuje algoritmus, který můžeme ve značně zjednodušené formě popsat následujícím způsobem na přiloženém příkladu :



1. Všechny vektory z polygonu se přenesou do dynamického pole ***p.***
2. Je-li polygon řezán rovinou, pak algoritmus pokračuje. Je-li polygon kolmý k ose pohledu, tak je celý vykreslen. V jiných případech algoritmus končí.
3. Řeže-li rovina některou úsečku definovanou dvěma po sobě jdoucími body pole ***p***, pak se mezi tyto vektory vloží řezný vektor (jsou to vektory později označené čísly: 6,10,12,15,18 a 23)
4. Je-li polygon bod, nebo úsečka, pak se vykreslí, a poté algoritmus končí, jinak pokračuje.
5. Z části, která je nad osou se nalezne nejvyšší vrchol (1)
6. Nyní se nalezne oblouk polygonu obsahující vrchol (1). Postupuje se tak, že od vrcholu (1) se postupuje doprava a doleva a naleznou se první vrcholy, které se dotýkají řezné úsečky(3 a 23). Tímto způsobem jsme nalezli řeznou úsečku, která je definovaná právě vrcholy (3) a (23).
7. Při pohledu na obrázek, je jasné, že definici oblouku obsahující vrchol (1) lze rozšířit doleva až k vrcholu (6). V bodě 7) se provádí právě takové rozšíření (pokud ho lze udělat) do obou směrů.
8. Nalezneme vrchol na řezné úsečce ((6),(23)) takový, že je nejblíže k vrcholu (6) a jdeme-li doleva, pak procházíme tímto vrcholem z části, která je za řeznou rovinou, do části, která je před ní. Tímto vrcholem je vrchol (10). Dáme ***index***=(10)
9. Do pomocného pole uložím všechny vrcholy, které projdu ve výše zmíněném směru od vrcholu (***index***), než opět narazím na řeznou úsečku (6)(23).
10. Z vrcholů, které jsou na cestě za vrcholem (***index***) a před pravým krajem (23), nalezneme vrchol na řezné úsečce ((6),(23)) takový, že je zprava nejblíže k vrcholu (index), a jdeme-li doleva, pak procházíme tímto vrcholem z části, která je za řeznou rovinou, do části, která je před ní. Je-li už nalezený vrchol pravým okrajem (23), pak jdu na 11), jinak přiřadím ***index***=(index nalezeného vektoru) a jdu zpět na 9).
11. Do pomocného pole přidám všechny zbylé vektory mezi pravým a levým okrajem (jdu-li stále doleva).
12. Pomocné pole reprezentuji jako polygon, a tak je také vykreslen.
13. Zbyl-li ještě nějaký vrchol nad řeznou rovinou, který nebyl zařazen to pomocného pole, pak celý postup aplikuji znova (tj. jdu na 5), ale pouze pro vrcholy, které ještě nebyly využity. Pomocné pole se smaže.

Z výše popsaného algoritmu je patrné, že není paměťově náročný, což je veliká výhoda, zvláště, bereme-li v úvahu, jak je program s daty jako celek rozsáhlý. Jsou patrné i ještě nezmíněnou nevýhodou, jakou je nepřesnost zobrazení, v případě že krok řezu je větší než jeden pixel. Zobrazované plochy se vykreslují buď danou barvou, nebo barvou a stínem, který se vytváří vzorkováním. Podrobněji v kapitole o stínování.

Zobrazení pamětí hloubky

Algoritmus paměti hloubky [1][3] je založen na následující myšlence. Nechť je naším úkolem vykreslit scénu do zadaného obdélníka. Předpokládáme dále, že jde o obdelník na výstupním

rastrovém zařizení. Vytvoříme dvojrozměrné pole o rozměrech okna (v pixelech [18]). Toto pole inicializujeme hodnotou nejmenšího čísla (v dané aritmetice). Postupně bereme jednotlivé polygony na scéně a rozkládáme je na pixely. Pro každý pixel z polygonu spočteme jeho hloubku. Je-li tato hodnota větší než hodnota uložená v poli na odpovídajícím místě, pak tento bod vykreslíme a jeho hloubku zapíšeme do pole. Není-li jeho hlobka větší, pak s tímto pixelem nic neprovádíme. Je jasné, že rozložím-li všechny polygony na scéně výše popsaným způsobem, pak bude scéna zobrazena korektně.

Algoritmus užitý v programu MDL lze rozepsat následovně:

výpočet probíhá v souřadném systému umístěném o obdélníku tak, že osa x je rovnoběžná s horizontální stranou obdelníka a osa y je rovnoběžná s vertikální stranou obdelníka. Osa z je kolmá k oběma zbývajícím osám a je orientována tak, že kladná část osy je blíže k pozorovateli.

1. Inicializuji pole nejmeší hodnotou v dané aritmetice
2. Vyberu těleso ze scény. Není-li již na scéně žádné těleso, pak končím zobrazování.
3. Je-li těleso vybrané v bodu 1) některé z kubických, pak vytvořím polyhedrální aproximaci segementu křivky (resp. plátu plochy). Vyčerpal-li jsem včechny segmenty (resp. platáty), jdu na 1)
4. Vezmu polygon z tělesa, nebo aproximace. Vyčerpal-li jsem všechny polygony z aproximace, pak jdu opět na bo 2). Vyčerpal-li jsem všechny polygony z tělesa, pak jdu na 1).
5. Zjistím maximální a minimální hodnotu souřadnice y, jakou nabývá některý pixel z tohoto polygonu .
6. Přes všechny hodnoty y provádím řádkový rozklad polygonu. Pro každý pixel řádkového rozkladu polygonu spočtu hodnotu jeho z-tové složky. Tuto hodnotu porovnám s hodnotou v poli na pozici (x,y), kde (x,y,z) jsou souřadnice bodu odpovídajícího pixelu. Je-li hodnota souřadnice tohoto bodu větší, pak se tato hodnota zapíše do pole a bod se vykreslí. Algoritmus řádkového rozkladu je popsán samostatně.
7. Jdu na 3).

Algoritmus řádkového rozkladu použitý v programu se liší při užití pro trojúhelníky a obecné polygony. Pro trojúhelníky probíhá rozklad následovně. Pro pevné y se z vrcholů trojúhelníka lineární interpolací určí průsečíky hran trojúhelníka s rovinou XZ. Dále se interpoluje mezi těmito dvěma body, čímž se získají všechny body trojúhelníka v daném rozkladu.

Podobně jako u trojúhelníků se vytváří řádkový rozklad i u polygonů. Rozdíl je pouze v tom, že postup hledání krajních bodů rozkladové linky probíhá jiným způsobem. Zjednodušeně lze říci, že vytvoříme množinu průniků roviny XZ s polygonem. To znamená, že projdeme všechny hrany polygonu a hledáme odpovídající průsečíky. Algoritmus je modifikován tak, že počet takto získaných bodů je sudý. Krajní body se uspořádají v ose X a následně se vytvoří dvojice za sebou jdoucích bodů. Při získávaní průsečíků se dbalo na to, aby spojnice později vytvořených dvojic obsahovala právě body patřící polygonu. Tímto způsobem dostaneme řádkový rozklad polygonu uložený v posloupností úseček (resp. jejich krajních bodů).

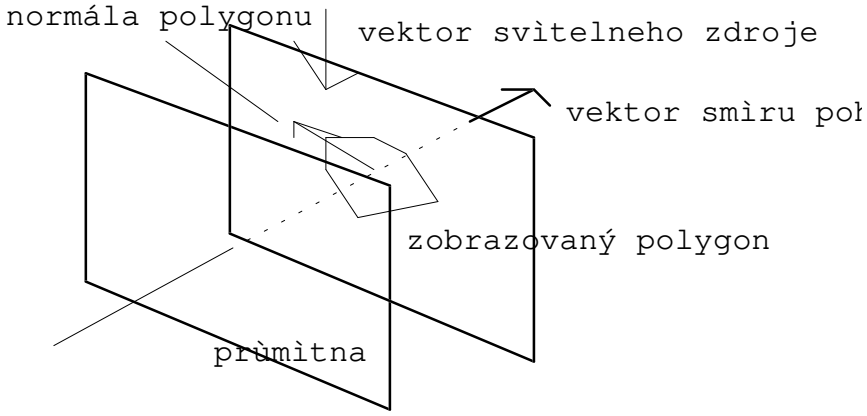
Na závěr této kapitoly by bylo užitečné zmínit se o realizaci tak rozsáhlého pole, jakým je hloubka celé scény. K realizaci tohoto pole se užívá rozšíření konvenční paměti - paměti XMS. Program MDL na počátku zobrazování scény zjistí jaké je dostupné množství této paměti. Je-li jí dostatek, pak v ní naalokuje pole z-bufferu. V konvenční paměti vytvoří buffer, do kterého se v případě potřeby přenášejí jednotlivé části z-buffer-u z XMS. Je-li ukončena práce s danou částí uloženou v bufferu, pak se tato část přenese zpět do XMS a do bufferu se nahraje nová část.

Stínování

Důležitým bodem na cestě k realistickému zobrazení scény je stínování. Úkolem stínování je snaha zobrazit těleso různými odstíny barev (nebo alespoň různými texturami) tak, aby se celkový vjem zobrazované scény blížil skutečnosti. Nejdůležitějšími faktory mající vliv na výsledný odstín elementu tělesa je přítomnost světelných zdrojů a poloha ostatních těles. Bereme-li při výpočtu v úvahu posledně zmíněný faktor, pak se jedná o tzv. vržené stíny [3].

V programu MDL jsou implementovány dvě základní metody stínování. Jsou to **kostatntní stínování** a **Gouraudovo stínování**. Aby se kvalita těchto druhů stínování projevila, je nutno mít k dispozici dostatečně velkou barevnou paletu. Tuto možnost však standardní VGA adaptery neposkytují. Proto byl program MDL byl vybaven driverem (od Jordan Hargraphix) adapterů SVGA standardu VESA [18][19]. Tento driver (resp. více driverů) umožňuje zobrazovat (v závislosti na dostupné paměti adapteru) 256 barev v rozlišeních 640x480, 800x600 a 1024x768. Pro práci se stíny byly vytvořeny dvě základní barevné palety. První obsahuje 256 odstínů šedi a druhá obsahuje za sebou seřazených 16 odstínů 16-ti základních barev poskytovaných adapterem VGA.

Konstantní stínování přidělí všem bodům vykreslovaného polygonu tentýž odstín, odtud adjektivum názvu. Jaká barva (resp. odstín) se zvolí pro daný polygon se rozhodne následujícím postupem: Pro daný polygon se z jeho řídících vrcholů vektorovým součinem spočte normála. Směřuje-li normála po směru pohledu (viz. obr.), pak program chápe tento polygon jako odvrácený. Protože polygon chápeme jako "velice slabou desku", je pochopitelné, že nebudeme zobrazovat odvrácenou stranu, ale naopak zobrazíme tento polygon jako přivrácený. To v řeči normál znamená, že otočíme orientaci normály. Tímto způsobem je možné zobrazovat vnitřky těles apod. Nyní již máme stanovenou normálu polygonu.

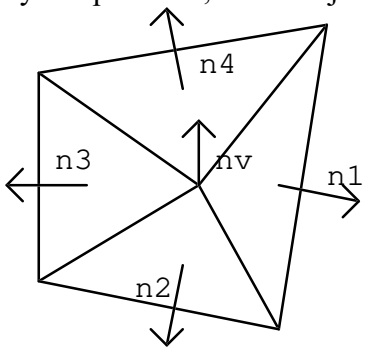


Na scéně přítomný světelný zdroj má svoji orientaci udanou směrovým vektorem jeho paprsků. Standardní norma tohoto vektoru je rovna jedné. Zdroj je umístěn v "nekonečnu" a světelné paprsky jsou rovnoběžné. Skalárním součinem vektorů světelného zdroje a normály polygonu dostaneme obecně číslo v intervalu od ###-1,1###, které je rovné kosinu úhlu, který tyto vektory svírají. Tedy tato hodnota určuje, jak je polygon přivrácen resp. odvrácen od zdroje. Na základě tohoto skalárního součinu přidělí program polygonu index v rozmezí (0,255). Tento index přímo odkazuje do palety (případ zobrazení v úrovních šedi), nebo je dále zpracován. Zobrazuje-li se scéna metodou prostorového řezu, pak je možné vystínovat těleso pouze v 16-ti barvách. Jednotlivé odstíny se simulují různými vzorky pro vyplňování polygonů. Na stínování tělesa je použito 17 vzorků (včetně plného a prázdného) .Vzorky byly nagenerovány pomocí metody popsané v [20]. Princip této metody spočívá v tom, že se do matice, která odpovídá počtem svých prvků počtu zobrazovaných úrovní světelných intenzit, které chceme aproximovat, rovnoměrně rozmístí číselná řada. U matice 4x4 jsou to tedy čísla od 1 do 16. Chceme-li potom zobrazit i-tý vzorek (kde

i je mezi 0 a 16) provedeme to tak, že v matici vzorků na všech místech, kde jsou čísla menší rovna i, nastavíme bit na 1. V programu MDL byla užita tato matice 4x4:

$$G=\begin{bmatrix} 1 & 9 & 3 & 11 \\ 13 & 5 & 15 & 7 \\ 4 & 12 & 2 & 10 \\ 16 & 8 & 14 & 6 \end{bmatrix}$$

Gouraudovo stínování [20] [3] (na rozdíl od konstantního stínování) určuje odstín resp. světelnou intenzitu každého bodu zobrazovaného polygonu. Toto zobrazení je určeno pro polyhedrální aproximaci křivých ploch. Tato metoda je založena na interpolaci jasu vycházející ze znalosti normály jednotlivých hraničních polygonů tělesa. Nejprve se určí normály ve vrcholech. Normálu ve vrcholu definujeme jako aritmetický průměr normál ploch, které tento vrchol sdílejí. Jsou-li plochy vyjádřeny pomocí rovnic, lze normálu určit přímo. Jako příklad lez uvést normálu vrcholu nv, který patří čtyřem plochám, které mají normály n1, n2, n3 a n4 (viz. obr.).




```
nv= ( n1 + n2 + n3 +
```

Pro takto získanou normálu dostaneme požadovaný odstín způsobem popsaným v konstantním stínování. Máme tedy určeny odstíny ve všech vrcholech polygonu. Postupen totožným s řádkovým rozkladem získáme pro každý bod polygonu jeho specifický odstín pomocí postupných lineárních interpolací. Pro hranoly a jehlany se provádí interpolace pouze u bočních stěn.

Výpočet Gouraudova stínování lze zobecnit tak, že se postupně neinterpolují jasové hodnoty v jednotlivých vrcholech, nýbrž jejich normály. Normála nese více informací než jas, toho se dá využít pro určení odlesků světelných zdrojů na tělese atp. Tento způsob stínování se nazývá **Phongovo stínování**[20][3] a není v programu zatím implementováno.

Transformace

V programu MDL se užívají transformace k dvojímu účelu. Buď jako projektivní transformace, nebo jako polohové transformace. Transformační metody lze rozdělit do dvou skupin. Transformace se vždy týkají tělesa a pracují tak, že převádí vektory z **báze** na nové vektory.

Program užívá tyto transformace: **rotace tělesa kolem počátku souřadné soustavy, translace tělesa, změna velikosti tělesa, dof (direction of flight) rotace, rovnoběžné a středové promítání.** Tyto metody lze samozřejmě kombinovat a vytvářet tak metody nové. Jako příklad transformace vzniklé spojením translace a rotace lze uvést rotaci tělesa kolem dané osy.

Obecnou transformaci lze zapsat v maticovém zápisu pomocí homogenních souřadnicí následujícím způsobem.

$$T=\begin{bmatrix} Axx & Axy & Axz & Tx \\ Ayx & Ayy & Ayz & Ty \\ Azx & Azy & Azz & Tz \\ Px & Py & Pz & 1 \end{bmatrix}$$

Kde koeficienty **Aij** odpovídají rotaci a změně měřítka, koeficienty **Ti** translaci a koeficienty **Pi** středovému promítání. Nyní si podrobněji objasníme, jaké koeficienty jsou zvoleny v programu.

Trojrozměrný vektor se transformuje násobením transformační maticí zleva tak, že se nejprve převede na homogenní vektor přidáním čtvrté složky, která je rovna jedné. Provede se transfomace a výsledný vektor se zpětně normalizuje tak, aby jeho poslední složka byla jedna. První tři složky homogenního vektoru se chápou jako výsledný trojrozměrný vektor. Jsou-li níže popsané transformační matice rozměru 3x3, znamená to, že danou transformaci lze zapsat maticí již v prostoru dimenze 3. Tyto transformace lze samozřejmě zapsat pomocí výše zmíněné homogenní matice T.

Rotace je zadaná vektorem rotace, který je metodě předán. Jeho jednotlivé složky mají význam úhlu otočení kolem odpovídající osy. Velikosti úhlů jsou udané ve stupních. Rotace kolem jednotlivých os je možné uvést v maticovém zápisu:

$$Rx=\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos x & -\sin x \\ 0 & \sin x & \cos x \end{bmatrix} \quad Ry=\begin{bmatrix} \cos y & 0 & \sin y \\ 0 & 1 & 0 \\ -\sin y & 0 & \cos y \end{bmatrix} \quad Rz=\begin{bmatrix} \cos z & -\sin z & 0 \\ \sin z & \cos z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Skládání rotací je sice asociativní, ale bohužel není obecně komutativní. To s sebou přináší celou řadu problémů. V programu MDL je transformace rotace chápaná jako postupná rotace kolem os Z,Y a X . Výsledný vektor pak získáme složením výše zapsaných matic.

Je-li třeba jiné pořadí rotací např. v jazyce MDL, pak musí uživatel postupně volit rotaci s vektorem rotace zadaným tak, že všechny jeho složky jsou nulové kromě složky odpovídající dané rotaci.

Dof rotace je speciálním případem rotace. Tato rotace se v animaci užívá častěji než výše popsaná rotace, protože uživatel může snadněji zadat požadovanou polohu tělesa. Tato rotace má jediný argument - vektor **dof**, který určuje směr "letu" tělesa. Transformace provede rotaci tělesa tak, aby směr osy x po transformaci byl totožný s vektorem **dof**. Aby byla transformace popsána jednoznačně je nutno dodat podmínku na novou polohu osy y. Touto podmínkou je, že nová osa y musí být kolmá na **dof** a na původní osu z. Použitá matice má ve sloupcovém podobě tvar:

$$Rdof=[\ |dof|\ ,\ |z \times dof|\ ,\ |dof \times (z \times dof)|\ ]$$

kde z je bazický vektor původní souřadné soustavy, přiřazený ose z.

Translace je zadaná vektorem posunutí. Metoda posune těleso o tento vektor. Má-li vektor posunutí tvar **(a,b,c),** pak má homogenní transformační matice tvar:

$$T=\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Změna měřítka násobí jednotlivé složky vektorů z tělesa koeficienty odpovídajících změn měřítka v jednotlivých osách. Tyto složky jsou uloženy ve vstupním vektoru. Aby vektor změny

měřítka mohl být složen z celých čísel, a zaroveň se neztratila možnost vyjadřovat změnu ve zlomcích, jsou složky tohoto vektoru chápány jako procentuální zvětšení nebo zmenšení tělesa. Složky tohoto vektoru mohou být tedy i záporné, To je v případě, že těleso zmenšujeme. Je-li zmenšení větší než 100%, pak se těleso přetransformuje podle středu souřadné soustavy. Pak se zvětšujícím se koeficientem zmenšení těleso zvětšuje.

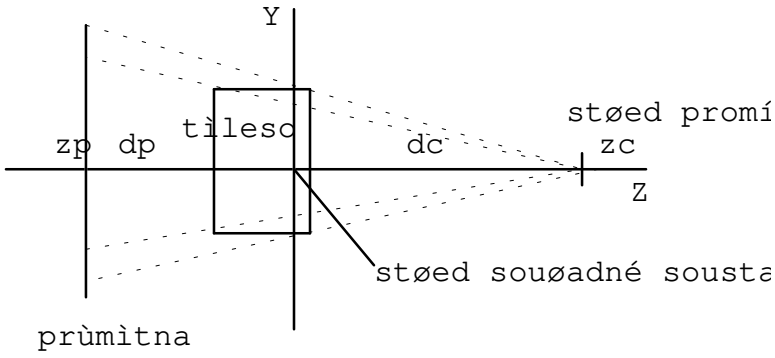
Transformační matice má tvar:

$$Z=\begin{bmatrix} mx & 0 & 0 \\ 0 & my & 0 \\ 0 & 0 & mz \end{bmatrix}$$

Rovnoběžné promítání je metoda zobrazení těles, kdy promítací paprsky jsou rovnoběžné. Rovnoběžné promítání se dále dělí na pravoúhlé a kosoúhlé. V programu bylo užito pravoúhlé rovnoběžné promítání zadané transformační maricí:

$$T=\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Středové promítání (perspektivní) všechny promítací paprsky procházejí jedním bodem - středem promítání. Tato projekce odpovídá vnímání světa. Úsečky, které jsou na scéně dál jsou kratší i když mají ve skutečnosti stejnou vzdálenost. Těleso se transformuje tak, že se složka každého jeho vektoru, která odpovídá ose pohledu zachová a ostatní se vynásobí perspektivním koeficientem K. Středové promítání můžeme vysvětlit na následujícím obrázku.



Středem promítání c vedeme promítací paprsky, které dopadají na průmětnu p. Z trojúhelníkové podobnosti a za předpokladu, že průmětna je v počátku souřadné soustavy dostaneme transformační matici ve tvaru:

$$T=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1/dc & 1 \end{bmatrix}$$

Uživatelské rozhraní

Uživatelské rozhraní je velice důležitou části programu. V grafických programech jeho důležitost ještě vzrůstá. Toto rozhraní zajišťuje komunikaci s uživatelem, to znamená, že zpracovává vstupně a výstupní proud dat. Při konstrukci programu byla snaha vytvořit za daných podmínek co nejvíce interaktivní prostředí, aby byla práce s programem efektivní a pohodlná.

Proč má program MDL vlastní uživatelské rozhraní a nevyužívá standardní prostředí (např. MS-WINDOWS)? Důvodů je více, ale zvláště ten, že vlastní prostředí je vytvořeno podle požadavků programu (tzn. je rychlé a účelné). Umožňuje například rychle přehrávat animované

sekvence anebo vykreslovat scénu. Je nutné přiznat, že autorské rozhraní nedosahuje uživatelského komfortu profesionálních rozhraní, ale to ani nebylo cílem práce.

Uživatel vede **dialog** s programem prostřednictvím rozhraní. Z tohoto důvodu se základním třídám prostředí říká **dialogy**. Uživatelské rozhraní (dále **UGI**) obsahuje několik základních dialogů. Tyto dialogy jsou odvozeny od společné třídy **Dialog**, která představuje abstraktní okno (proto můžeme říci, že UGI je systém založený na pojmu okna). Každé abstraktní okno obsahuje svůj název, polohu, rozměr a některé metody jako třeba změna polohy okna .

V objektové terminologii lze říci, že jde o abstraktní třídu se třemi čistě virtuálními metodami [17]. Tyto metody jsou: **metoda na vykreslení**, **metoda na činnost dialogu** a konečně **metoda pro systémové menu.** Každý potomek této třídy si definuje tyto metody tak, aby to vyhovovalo potřebám dané třídy. Co jednotlivé metody dělají? Metoda na vykreslení dialogu na požádání tento dialog vykreslí v dané pozici a s obsahem, který je právě validní. Metoda na činnost dialogu vytváří smyčku pro čtení událostí, které přicházejí od myši resp. z klávesnice. Je-li třeba tato metoda předá řízení metodě jiného dialogu. Poslední metoda definuje reakci na vyvolání systémového menu. Tato reakce u některých dialogů vyvolá menu, které mění vlastnosti tohoto dialogu.

Aby bylo možné spravovat všechny dialogy, které se právě účastní komunikace, byla vytvořena třída **správce**. Správce eviduje všechny dialogy a zajisťuje předávání aktivit mezi jednotlivými dialogy. V rozhraní je jeden dialog vždy aktivní, přijímající všechny zprávy od uživatele. Vyskytne-li se zpráva, která danému dialogu nepatří (např. kliknutí myši mimo okno dialogu), pak metoda pro činnost dialogu vrátí aktivitu správci a ten ji předá dialogu, který ji má dostat. Nyní si na příkladu podrobněji popíšeme činnost správce, a tím i činnost rozhraní.

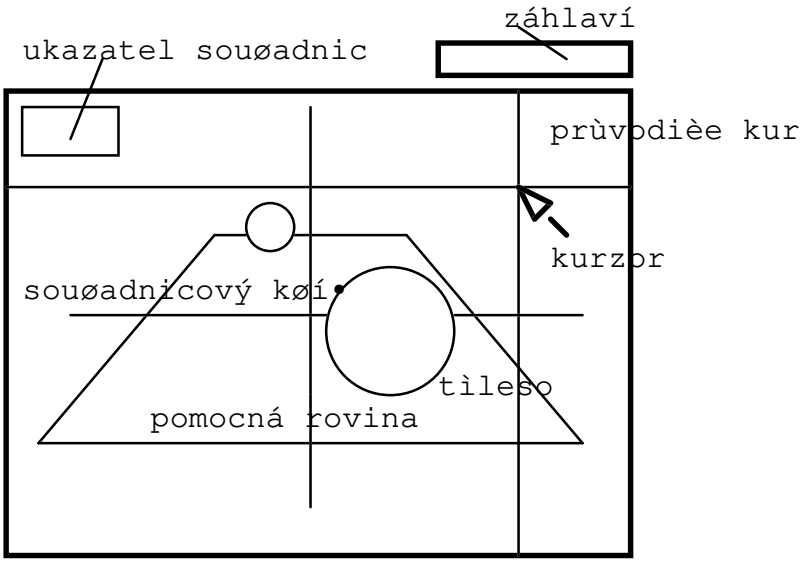
Předpokládejme, že správce eviduje již několik dialogů. Nyní je třeba, aby program získal vstupní data od uživatele, která jsou nezbytná pro další činnost. K tomuto úkolu je připraven nový dialog (resp. konkrétní instance třídy). Tento dialog je nainicializován na potřebné hodnoty a poté zaregistrován u správce. Správce zařadí tento dialog na začátek seznamu dialogů účastnících se komunikace, tím se stává dialog aktivním. Správce požádá všechny dialogy v seznamu, aby se vykreslily v obráceném pořadí, než jsou v tomto seznamu. Takto se zobrazí aktivní dialog na vrchu. Po vykreslení všech dialogů se předá činnost aktivnímu dialogu. V tuto chvíli začíná komunikace s uživatelem. Uživatel komunikuje s aktivním dialogem, chce-li pozornost přenést na jiný dialog, pak klikne myší mimo rámec aktivního dialogu. Správec přenese tento dialog navrch (překreslí celý desktop) a předá tomuto dialogu aktivitu. Doba, kdy činnost programu spravuje správce (resp. nějaký dialog) se nazývá **session**. Během každé session je seznamu dialogů definovaný správcem právě jeden dialog, který má možnost ukončit tuto session. Tím je uživatel nucen po čase vykonat určitou činnost. Skončí-li session, pak je činnost předána nekomunikační části programu, ta zpracuje data a spustí další session. Uživateli se však zdá, že práce s okny je kontinuální.

UGI obsahuje následující dialogy: **grafické okno, textové okno, nápovědu, menu, toolbar, seznam, celočíselný valuátor, okno zprávy, editační linka, tlačítko a ikona.**

Grafické okno

Nejdůležitější a zároveň nejkomplikovanější částí UGI je grafické okno. Tento dialog má vstupně-výstupní charakter, jednak slouží k zobrazování určité scény, ale také umožňuje například z této scény číst polohu. Okno se v podstatě skládá z obdélníkového výseku, který představuje část projektivní roviny. Na tuto rovinu se zadanou projekcí promítá již ztransformovaná scéna.

Okno si můžeme schématicky znazornit takto:



Většina prvků zobrazených v okně (resp. na schématu) je volitelná, tj. uživatel sám určí, jestli daný prvek chce zobrazovat nebo ne. Každé okno má svou vlastní **konfiguraci**, tj. specifické nastavení těchto prvků. Tato konfigurace se změní pouze tehdy, jestliže uživatel skončí práci s programem, nebo změní nastavení určitého prvku. Konfigurace okna se mění pomocí systémového menu.

Každé okno obsahuje již při svém vytvoření odkaz na určitou scénu. Program MDL umožňuje více oknům mít odkaz na tutéž scénu. Tento princip umožňuje sledovat tutéž scénu z různých pohledů, nebo umožňuje zadávat vstupní údaje v okně, které je pro daný účel nejvhodnější. V praxi tento způsob chápání oken přináší některé problémy, především pro správce. Například, chci-li smazat určité okno, tak musím nejprve projít seznam ostatních oken. Existuje-li okno, které má odkaz na stejnou scénu, jako rušené okno, tak zničím pouze rušené okno, jinak zruším i scénu, na kterou toto okno ukazovalo.

Nyní si ve zkratce vysvětlíme, jak okno **vykreslí zobrazovanou scénu**. Nastane-li důvod pro vykreslení celé obrazovky, pak správce postupně zavolá všechny dialogy, které se vyskytují na scéně resp. jejich překreslovací metody. Překreslovací metoda okna nejprve podle konfigurace zjistí, jestli se okno vykresluje standardně (tj. společně s ostatními dialogy), nebo přes celou obrazovku (užívá se pro animaci). Je-li okno přes celou obrazovku, pak metoda dále zjišťuje nastavený grafický mód. Je-li třeba, tak se inicializuje grafika potřebným způsobem. Každé okno má svou tabulku transformací, tato tabulka obsahuje transformace v následujícím pořadí: změna měřítka, rotace a translace. Je-li zadáno zobrazení pomocí projektivní transformace, tak se na konec výše zmíněné tabulky připojuje transformace odpovídající perspektivní projekci. Takto naplněná tabulka se aplikuje na zobrazovanou scénu. Po těchto přípravných operacích se vykreslí rám okna, nastaví se aktivní viewport a zavolá se požadovaná vykreslovací metoda. Tyto metody jsou podrobněji popsány v kapitole věnované zobrazování.

Jako další příklad metody této třídy lze uvést metodu, která identifikuje vektor, jehož obraz uživatel označil myší. Důležitá je i metoda, která načítá do pomocného tělesa ve scéně posloupnost uživatelem zadaných vektorů atp.

Nyní se vrátíme k pojmu konfigurace okna. Okno je velice variabilním dialogem, jehož stav je uložen v několika argumentech. Nejdůležitějším argumentem je styl okna, který obsahuje sadu bitových přepínačů. Tyto přepínače z velké části popisují stav okna (viz. tabulka). Které vlastnosti lze u okna měnit? Lze například nastavit přítomnost některých systémových těles na scéně (osový kříž nebo deska v rovině XY), nebo zvolit různé módy zobrazování kurzoru, nastavit perspektivní projekci, zvýraznit vrcholy, zvolit druh zobrazení.

Většina přepínačů se nastavuje v systémovém menu. V tomto menu je možno nastavit dále argumety jednotlivých transformací. Při manipulaci s tělesy je možné je nahradit jejich obalovými tělesy, pro snadnější a především rychlejší manipulaci.

Tabulka bitových přepínačů uložených ve stylu okna

přepínač	popis činnosti
W_ANIM	Speciální bitový přepínač určený pro animaci scény.
W_CROSS	Je-li tento bit nastaven, pak je ke scéně přídán kříž souřadné soustavy. Tento kříž nelze modifikovat.
W_GROUND	Je-li tento bit nastaven, pak je ve scéně přítomný obdelník zobrazující rovinu XY. Tuto rovinu neni taktéž možno modifikovat.
W_WHOLE	Tento přepínač se uplatňuje pouze při animaci. Nastavuje okno na celou obrazovku a nevykresluje jeho rám a ani záhlaví.
W_PIXEL	Bit určující jsou-li řídící vrcholy těles vidět.
W_EXECUTE	Je-li bit nastaven, pak okno před zobrazením uplatní svou tabulku transformací.
W_GET	Tento přepínač načítá do pomocnéh tělesa lomenou čáru. Pomocným tělesem je tzv. buffer. Toto těleso slouží ke konstrukci jiných těles.
W_IDENT	Bit užívaný pro identifikaci vektoru na scéně.
W_TOP	Je-li bit nastaven, pak je okno aktivní.
W_WINDOW	Tento bit je nastaven a určuje správci druh dialogu.
W_DIRECT	Tento přepínač umožňuje oknu předávat zprávy části programu, která je obvykle nezpracovává.
W_RENDER	Je-li bit nastaven, pak se užije aktuální render.
W_MOVEABLE	Je-li bit nastaven, pak lze hýbat s dialogem
W_LEADER	Je-li bit nastaven, pak má grafiký kurzor v okně průvodiče
W_COORD	Je-li bit nastaven, pak se zobrazuje poloha kurzoru.
W_ALONE	Tento bit neumožní předat zprávu jinému dialogu.

Textové okno

Dalším důležitým dialogem je **textové okno**. Textové okno umožňuje editovat zobrazovaný text. Tento dialog je potřebný především při psaní animačních algoritmů v jazyce MDL.

Editor poskytuje standardní operace jako jsou: mazání celé řádky, odskoky na začátek a konec zobrazené stránky, posun po stránkách, skoky po slovech, označování bloků, přesun bloku apod. Textové okno dále umí ukládat text na disk a nahrávat ho zpět, měnit font a jeho velikost. Okno při vykreslení zobrazí na svislé ose celkovou polohu stránky vzhledem k textu (scrollbar). Ve spodní části okna se zobrazí přehled o stavu okna.

Text je v okně uložen jako dynamický seznam řádek, kde řádka je chápána jako dynamický seznam znaků zakončených nulou.

Modifikace scény

Je-li scéna již vytvořena, pak je možno měnit vlastnosti jejích těles. Jde především o přesné stanovení polohy a tvaru, dále o barvu a jméno tělesa a jeho obalového tělesa. Uživatel má možnost tyto vlastnosti měnit interaktivně.

Princip modifikace závisí na pojmu **definice**. Definice je množinou vybraných řídících vektorů, které se na scéně nacházejí. Uživatel může různým způsobem tyto vektory specifikovat. Jsou-li vybrány všechny vektory tělesa, je pak vybráno i celé těleso a naopak, je-li vybráno celé těleso a vyloučíme-li z definice jediný jeho vrchol, pak už těleso není označeno jako vybrané.

Na vybranou definici aplikujeme požadovanou operaci. Operace se podle své povahy aplikuje na všechny vybrané vrcholy z definice, nebo jen na vybraná tělesa. Jak se v programu zadávají argumety transformací? Jako příklad vezmeme translaci. Translace definované množiny je zadána vektorem posunutí. Tento vektor se zadá tak, že v průmětu uživatel označi původní vektor a jeho výsledný obraz. Tím je zadán vektor posunutí. Abychom dostali odpovídající výsledek (tj. aby po transformaci přešel zadaný vektor do požadované polohy) a tento výsledek se v tělese uschoval, nemůžeme pouze ke všem vektorům z definice přičíst vektor posunutí, ale navíc je nutno po přičtení provést inverzní transformaci (k okením transformacím) **báze** tělesa. Výsledek inverzní transformace se převede do **matrice**. Takto se výsledek modifikace na trvalo uschová.

V programu lze nahradit celé těleso (již vytvořené) jeho obalovým tělesem. Toho lze využít pro urychlení chodu programu (zvláště vykreslování) a pro snadnější manipulaci s tělesy. Obalová tělesa nahrazují vlastní tělesa stále, kromě případu finálního vykreslení.

Jazyk MDL

V úvodní kapitole diplomové práce byly popsány různé způsoby popisu animační sekvence. Úkolem diplomové práce bylo implementovat jeden z těchto popisů, a to konkrétně popis animace pomocí animačního jazyka. Výsledkem této práce je jazyk **MDL (Motion Description Language),** který je součástí stejnojmenného programu.

Jazyk MDL popisuje chování vytvořené scény v čase pomocí algoritmu. Algoritmus se zapisuje do textového souboru. Podmínkou vytvoření animace není pouze algoritmus animace zapsaný v jazyce MDL, ale i přítomnost scény s definovaným tvarovým stromem.

V reálném světě se všechna tělesa pohybují (nebo jsou v klidu) ve stejném časovém okamžiku a navíc se pohybují více či méně nazávisle. Tato vlastnost okolního světa nutí zavést do animačního jazyka jisté rysy **paralelizmu**. Tomuto trendu se přizpůsobil i jazyk MDL, který v podstatě simuluje paralelní chování těles v zobrazované scéně. Všechna logicky samostatná tělesa na scéně se nazývají **entitami**. Jazyk MDL umožňuje popisovat polohu a podobu entit v čase. Jednotlivé entity mohou ovlivňovat chování ostatních entit. Entity mají přiřazený popis a mohou vykonávat různé činnosti. Popis entity je zadán pomocí tvarového stromu a činnosti jsou popsány algoritmicky.

Jazyk MDL je v podstatě procedurálním jazykem, se zabudovaným aparátem vektorové algebry. Kromě standardních algebraických operací je v jazyce možno využívat řadu standardních funkcí. Pojem procedury má některé odlišné rysy ve srovnání s klasickými jazyky (**C**,**Pascal**).

Algoritmus animace zapsaný v textovém souboru je nejprve převeden preprocesorem do mezikódu a tento mezikód je následně interpretován. V každém časovém kroku projde interpretr téměř celý zdrojový text a nastaví argumenty transformací všech entit. Podle těchto argumentů se scéna zobrazí a zvýší se čas. Animace končí není-li splněna podmínka na její trvání.

V následujících kapitolách si uvedeme pouze základní rysy jazyka MDL. Popis jazyka společně s popisem preprocesoru a interpretru je v uvedena v příloze této práce.

Animace v jazyku MDL

Nyní si popíšeme jakým způsobem vytváří program animační sekvenci. Animace se v programu vytváří v diskrétních časových krocích. V každém časovém kroku se spočte nová poloha a orientace všech entit na scéně. Uživatel mění polohu entit na scéně **nepřímo** tak, že mění hodnoty polohových transformací. Například, chce-li uživatel posunout entitu v ose X o 100 jednotek, tak nastaví metodě translace vektor posunutí na hodnotu **(100,0,0).**

Argumenty transformací odpovídající určitému časovému kroku, tedy plně popisují stav entity v daném čase. Aby mohl program (resp. jazyk) simulovat paralelizmus skutečného děje, je nutné aby entity, které se odkazují na argumenty jiných entit, četly tytéž hodnoty během celého výpočtu jednoho časového kroku. To je zajištěno tak, že se stav scény (resp. všechny použité argumenty) před novým výpočtem ukladá a je možné se na něj v následujícím kroku animace odkazovat.

Program bere v každém časovém kroku postupně všechny entity deklarované v animaci. Zavolá všechny varianty popisu entity a všechny činnosti příslušející entitě. Podle stavu scény se entitě přiřadí jeden popis a vykonají se ty činnosti, které splňují dané podmínky. Vliv činností (tj. změny argumentů) se v případě více činností sčítají. Vykonali-li se v daném časovém kroku všechny činnosti aktuálního herce, tudíž jsou aktualizovány tabulky argumentů, pak program vykoná zadanou posloupnost transformací s aktuálními argumenty, a tím případně změní jeho polohu nebo orientaci. Provede-li se výše zmíněný postup pro všechny entity, pak je scéna připravena k vykreslení snímku odpovídajího danému časovému kroku. Před vykreslením se provedou pohledové transformace podle argumentů entity-kamery. Po vykreslení se uloží staré hodnoty argumentů a zvýší se hodnota časového kroku. Animace skončí, není-li splněna podmínka na její trvání.

Výslednou animovanou sekvenci lze uložit do souboru. Tuto sekveci lze přehrávat i mimo vlastní program MDL pomocí programu **PLAY**. Tento program byl vytvořen proto, aby si uživatel mohl přehrát sekvenci pomocí jednoduchého přehrávače. Sekvence se skládá z posloupnosti bitových map, které jsou poměrně rozsáhlé. Přímé ukládání těchto map na disk by bylo velice neefektivní. Jedna mapa představující jeden snímek má velikost nejméně 300 KB. Je jasné, že při uložení celé sekvence by zabrala neúměrně velké místo. Tento problém byl v jazyce MDL řešen ve dvou krocích. Prvním krokem je komprimace jednotlivých snímků pomocí modifikace metody **RLE** [18] a druhým krokem je zaznamenávání diferencí mezi jednotlivými snímky. Přehrávaní sekvence probíhá tak, že se vykreslí celý první snímek a pak se vykreslují pouze odchylky mezi jednotlivými snímky. Metoda komprimace je vhodná k rychlému vykreslování. V současném stavu zvládá program vykreslovat 12 snímků za vteřinu.

Prostor

Na počátku činnosti animace (programu) uživatel vytvoří prostor, ve kterém bude tato animace probíhat, a to z toho důvodu že se vlastnosti prostoru mohou měnit pro jednotlivé animace. Konkrétně to znamená, že si uživatel může například pro danou animaci navolit požadovanou tabulku transformací, která bude popisovat polohu každé entity.

Doba trvání animace je spojena s dobou trvání tohoto prostoru, deklarujícího se pomocí klíčového slova **`space`** a trvá tak dlouho, jak dlouho platí podmínky uvedené za klíčovým slovem **`condition`**. Při vytvoření prostoru dostává uživatel k dispozici některé objektové typy. Mezi tyto typy patří například typ vektor, homogenní vektor, matice apod.

Velice důležitou částí deklarace prostoru je popis argumentů prostoru. Jedná se o seznam proměnných různého typu a identifikátorů transformací, které následují za klíčovým slovem **`arg`** . Proměnné slouží jako argumenty udaných polohových transformací. Uživatel tedy musí zadat společně s typem argumetu i transformaci, k níž tento argument patří. Tím se vytvoří uspořádaná dvojice. Tyto dvojice se postupně ukládají. Po ukončení deklarace prostoru jsou tedy v programu přitomny dvě tabulky, tabulka transformací a tabulka jejich argumentů. Tabulka argumentů se následně zkopíruje do všech mechanicky nezávislých částí scény. Uživatel může měnit polohu jednotlivých částí scény pouze tak, že mění argumenty transformací v tabulce odpovídající dané části. Transformace v určitou chvíli změní polohu odpovídající části právě podle zadaných argumentů. Transformace se vykonávají v takovém pořadí, v jakém jsou uvedeny za klíčovým slovem **`arg`**. Jazyk zatím umožňuje sestavovat kombinace čtyř základních transformací, které jsou popsány následujícími identifikátory : **`translation`** (translace), **`rotation`** (rotace), **`dofrotation`** (dof-rotace) **`a magnify`** (zvětšení). Od rotace a dof-rotace jsou odvozeny transformace, které vykonavaji tutéž činnost, ale vzhledem k zadanému bodu. Poloha tohoto bodu se vždy zadává jako druhý parametr. Parametry se oddělují čárkou. Identifikátory těchto transformací jsou **`relrotation`** a **`reldofrotation`**.

V seznamu argumentů musí **vždy** být deklarace časového kroku. Tato deklarace určuje typ hodnoty časového kroku a dále jeho identifikátor . Zápis deklarace je stejný jako deklarace transformací s tím rozdílem, že místo identifikátoru transformace se zapíše klíčové slovo **`timestep`**. Je důležité, aby byl čas pro uživatele přístupný v podobě proměnné, protože animace je často popsána parametrickými rovnicemi závislými na čase.

Nyní si uvedeme příklad deklarace prostoru pro animaci která bude trvat dvacet časových cyklů, a kde bude moci uživatel ovládat polohu mechanických částí prostřednictvím výše zmíněných transformací resp. jejich argumentů. Text za dvěma lomítky je komentář.

```
        // začátek deklarace prostoru
space Animace
        // začátek deklarace argumentů
        // prvni argument je časový krok určený identifikátorem
        // cas, který je typu integer
 arg cas(i) timestep
        // druhý argument definuje vektor zvětšení určený identifikátorem
        // zvětšení mající složky typu integer
  zvetseni(i:vector) magnify
        // třeti argument definuje vektor rotace určený
        // identifikátorem rotace mající složky typu double
        //  čtvrtý argument definuje nový střed
        // rotace určený vektorem center (téhož typu jako předcházející vektor)
  rotace(d:vector),center(d:vector) relrotation
        // pátý argument definuje vektor posunutí určený
        // identifikátorem posunutí mající
        // složky typu integer
  posunuti(i:vector) translation
        // podmínka určující dobu trváni animace
 condition cas###20
 ...
```

Všechna argumenty jsou na konci deklarace nastaveny na nulovou hodnotu. Tyto hodnoty argumetů nemají vliv na změnu polohy odpovídající části.

Jak je patrné z příkladu, je doba trvání animace závislá na množině podmínek (**`condition`**), které jsou nutné a postačující k tomu, aby animace skončila. V každém časovém kroku se tyto podmínky testují, jsou-li splněny, pak animace končí. Způsob zápisu podmínek je uveden v příloze práce.

Entity

Základním logickým prvkem jazyka MDL je **entita**. Entita představuje buď konkrétní množinu těles na scéně, nebo fiktivní těleso. V prvním případě se entita nazývá **herec** a v druhém případě **kamera**.

Uživatel může pomocí programu měnit polohu entit. Entita obsahuje alespoň jednu instanci struktury, která popisuje její polohu a orientaci. Má-li entita více mechanicky nezávislích částí, pak každá tato část (**blok**) obsahuje také kopii této struktury. Touto strukturou je samozřejmě tabulka argumentů transformací, o které jsme se zmíníli v předcházející kapitole. Transformace spojené s tabulkou argumentů mění podle zadaných hodnot polohu a orientaci entity. Identifikátory argumentů jsou po celou dobu vyhrazeny právě pro identifikaci těchto proměnných. Tyto proměnné tvoří speciální třídu (viz. příloha)

Deklarace entity v programu má tyto části: **popis entity a popis činnosti**.V popisu entity se přiřazuje entitě na základě udaných podmínek tvarový strom (podrobněji v popisu herce).V popisu činnosti se udává seznam aktivit entity, které mohou měnit její vlastnosti (probněji viz. herec a činnost).

Deklarace entit v jazyce MDL je z obecného hlediska totožná, a proto jednotlivé kroky deklarace entity popíšeme podrobněji na herci, u kamery popíšeme pouze odchylky od deklarace herce.

Herec

Běžná geometrická tělesa (např. auto) a jejich chování se dá popsat ve struktuře herec. Každý herec musí vědět **"v jakém kostýmu hraje a jaký je scénář hry, kterou hraje".** Tyto údaje jsou specifikovány při deklaraci herce. Ve zjednodušené formě je ekvivalentem k těmto údajům v jazyce MDL následující uspořádaná dvojice:

herec = (popis herce, seznam činností).

Popis herce udává z jakých grafických elementů se herec skládá a definuje jeho nezávislé části. Popis je jedinou částí v deklaraci herce, která spojuje animaci (popsanou jazykem) s vlastní grafickou reprezentací těles. Spojení je realizováno přes abstraktní stromovou strukturu, které se říká tvarový strom. V deklaraci herce může být uvedeno více podob herce. To umožňuje měnit podobu herce v čase v závislosti na daných podmínkách. Tyto podoby jsou ale závislé na vnějších podmínkách. Jsou-li splněny podmínky popisu, pak má herec podobu odpovídající tomuto popisu. Je-li splněno více podmínek (tj. je více možných popisů), pak má herec podobu posledního splněného popisu. Podrobněji v kapitole věnované popisu herce.

Seznam činností určuje činnosti, které se mají při splnění určitých podmínek vykonat, tj. jsou to jakési eventuality chování herce. Je-li v daném okamžiku splněno více spouštěcích podmínek, pak se samozřejmě vykonají postupně všechny. To v praxi znamená, že se jejich vlivy na polohu herce "sčítají". Činnosti mohou být ovlivňovány vstupními argumenty, s kterými se volají. Podrobněji v kapitole věnované činnostem.

Herec může komunikovat s ostatními herci pomocí zpráv. Stejně jako tabulky argumentů jsou zdvojeny, tak i seznamy došlých zpráv jsou zdvojeny pro současný a předcházející časový krok. Pomocí speciálních funkcí (**`getmsg, sendmsg`**) může herec posílat zprávy jinému herci, nebo může zjišťovat, jestli nepřišla konkrétní zpráva. Při přechodu na nový časový krok se starý seznam nahradí novým. Tento mechanizmus umožňuje herci reagovat na stav jiného herce. Obráceně platí totéž, má-li příjemce implementovanou odezvu na danou zprávu. Ptá-li se herec na zprávy, které mu přišli, pak se tyto zprávy vybírají ze seznamu zpráv odpovídajícího konci předcházejícího časového kroku.

Deklarace herce je uvozena klíčovým slovem **`actor`**, za nímž následuje identifikátor herce. Za touto deklarací může následovat deklarace proměnných a konstant (uvozených klíčovými slovy **`var`** a **`const`**). Pak následuje seznam popisů a seznam činností (uvozených klíčovými slovy **`description`** a **`activity`**) Nyní si uvedeme příklad deklarace herce:

```
...
// přiřazení prvního tvaru
description Auto
 condition cas###21
 shape Auto
...
// přiřazení druhého tvaru
desciption Auto_naraz
 condition cas ### 20
 shape Auto_naraz
...
// deklarace první činnosti
activity Jizda
 ...
```

```
...
// deklarace druhé činnosti
activity Jizda_naraz rychlost(i)
 ...
...
// začátek deklarace herce
// za klíčovým slovem následuje identifikátor herce
actor Auto
  // deklarace proměnných herce
 var rychlost(i)
  // seznam popisu
 description      Auto
                  Auto_naraz
  // seznam činností
 activity Jizda
          Naraz (rychlost)
 ...
```

~~Popis herce~~

V úvodu popisu jazyka MDL jsme se zmínili, že k vytvoření animace je třeba kromě algoritmu zapsaného v textovém souboru i scéna s definovanou stromovou strukturou. Přesněji řečeno, jedná se o seznam tvarových stromů, kdy každý strom představuje jeden popis. V animaci je možné každému herci (dokonce entitě) přiřadit v každém časovém kroku jeden tvarový strom - popis. Tvarový strom popisuje hierarchickou strukturu herce, která specifikuje jeho mechanické vlastnosti. Každý uzel stromu představuje mechanicky relativně nezávislou část tělesa, které budeme říkat **blok**.

Každé těleso, které patří danému herci je součástí nějakého bloku. Všechna tělesa patřící do daného bloku (buď přímo, nebo patřící do jeho podbloku) jsou ovlivňována vlastnostmi bloku. To znamená, že všechny uzly na větvích vycházejících z nějakého uzlu (tj. patřící do bloku) mění svou polohu při polohové změně či změně orientace nadřazeného uzlu. To přináší podstatné zjednodušení pro animaci. Změním-li polohu kořene tvarového stromu, pak se změní poloha celého herce. Naopak změním-li polohu jednoho jeho uzlu, pak se změní pouze poloha tohoto uzlu a uzlů na jeho podvětvích. Teoreticky to odpovídá do sebe vnořeným souřadným systémům, kdy souřadný systém nadřazeného uzlu obsahuje všechny systémy svých poduzlů.

Každý uzel stromu odpovídá mechanicky nezávislé části a lze jej popsat uspořádanou trojicí:

uzel stromu=
(identifikátor, seznam argumentů, seznam tělesa, seznam poduzlů (větví)),

kde **identifikátor** označuje název tohoto uzlu. Tímto identifikátorem se odkazuje na daný blok (uzel) v algoritmu animace. Například pro specifikaci argumentu **translace** v uzlu **`kolo`**, který je poduzlem uzlu **`auto`**, napíšeme v programu : **`auto.kolo.tran`**.

Seznam argumentů představuje tabulku argumentů transformací vytvořenou při deklaraci prostoru. Když proběhnou všechny změny argumentů daného herce, tj. změny ve všech tabulkách argumentů, každém bloku, interpretr přikáže tvarovému stromu, aby vykonal transformace podle zadaných argumentů. Tuto činnost provádí rekurzivní algoritmus, který vykoná transformace (podle tabulek) v nejnižších úrovních stromu, a pak postupuje směrem ke kořenu tvarového stromu.

Seznam těles v každém uzlu může obsahovat odkazy na tělesa, která patří danému uzlu (bloku). V popisu scény bylo řečeno, že tělesa na scéně jsou uložena v lineárním seznamu, proto odkazy v uzlech mají formu indexů do tohoto seznamu.

Seznam poduzlů (větví) obsahuje odkazy na uzly, které patří danému uzlu.

V animaci přiřadíme danému herci (entitě) vytvořený strom (ze seznamu stromů) v deklaraci popisu, pomocí klíčového slova **`shape`**. Deklarace je uvozena klíčovým slovem **`description`**. Popisy herce se mohou v čase měnit, to zajišťuje podmínka na přiřazení tvaru (**`condition`**). Příklad přiřazení tvaru je následující:

```
...
// začátek deklarace popisu
description Popis
 // volba proměnných a konstant potřebných k výpočtu
 var ...
 const ...
 // podmínka na přiřazeni tvarového stromu
 condition ...
 // přiřazení tvaru
 shape Novy_tvar
...
```

Kamera

Podobně jako herci jsou určeny i kamery na scéně . Rozdíl v deklaraci herce a kamery je pouze v tom, že deklarace kamery je uvozena klíčovým slovem **`camera`**. S kamerou je možno hýbat na scéně podobně jako s hercem, tj. měnit její orientaci a polohu. Kamera se liší od herce ještě v popisu tvaru. Je-li kameře přiřazen nějaký tvarový strom, pak má smysl měnit pouze tabulku argumentů v kořeni stromu, protože ostatní větve tohoto stromu jsou ignorovány. Na scéně muže být definována nejvýše jedna kamera. Není-li kamera definována, pak je scéna snímána z polohy, jaká je nastavena v aktuálním okně.

Všechna ostatní pravidla jsou totožná s pravidly užitými u herce. Příklad deklarace kamery je následující:

```
...
// deklarace popisu
descriptiion Kam
 // přiřazení tvarového stromu
 shape cam
...
// deklarace činností
activity Pohyb_kamery
 ...
...
// začátek deklarace kamery
camera Kamera
 // určeni kdy kamera hraje
 condition cas ### 20
```

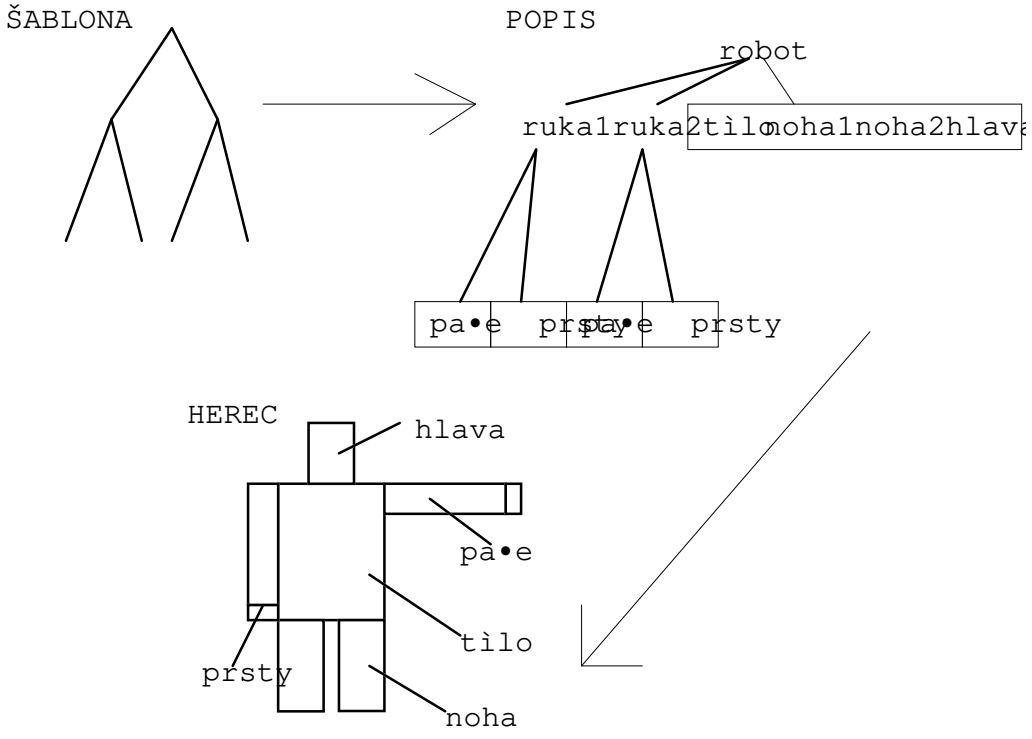
```
// přiřazení tvarového stromu
description Kam
// seznam činností
activity Pohyb_kamery
...
```

Editace tvarového stromu

Každé entitě je přiřazen tvarový strom, jehož strukturu jsme si popsali v předcházející kapitole. Tvarový strom je sice nezbytně nutný pro animaci, ale jeho tvorba neprobíhá přímo v jazyce. V prostředí programu MDL je definována třída, která tuto funkci vykonává. Tento dialog vytváří pro danou scénu seznam tvarových stromů. Ve vlastní animaci se pak odkazujeme pouze na již vytvořené stromy.

Strom se ve výše jmenovaném dialogu vytváří interaktivním způsobem. Uživatel může procházet všemi stromy uloženými v seznamu. Tyto stromy se před ním zobrazují . Myší je možné měnit vlastnosti jednotlivých uzlů. Například měnit jméno daného uzlu, přidat nebo smazat jeho podvětev, přidat nebo odebrat těleso z daného uzlu a zobrazit údaje o uzlu. V systémovém menu tohoto dialogu je možné měnit velikost celého stromu. Seznam stromů je součástí scény, proto se nahrává a ukládá na disk společně se scénou.

Postup práce s dialogem je jednoduchý (přesný popis je uveden v uživatelské příručce, která je součástí programu). Uživatel vytvoří nejprve kořen stromu a později přidává jednotlivé uzly stromu tak, aby vytvořil tzv. šablonu. Šablona je v podstatě strom, který obsahuje pouze údaje o uzlech, tj. popisuje pouze hierarchii entity, bez konkrétní specifikace těles, ze kterých se jednotlivé bloky skládají. Uživatel musí při konstrukci šablony dbát několika pravidel. Především by každá nezávislá část herce měla mít svou vlastní větev, aby bylo možno s ní nezávisle pohybovat. Dále by měl uživatel respektovat, že tvarový strom reprezentuje hierarchii herce, tudíž by logicky vyšší jednotky herce měly být blíž kořenu stromu.



Jako příklad popisu herce je možno uvést jednoduchého robota (obrázek na předcházející straně). Šablona je popsána geometrií stromu. Konkrétní popis je realizován tím, že strom obsahuje již konkrétní odkazy na tělesa ve scéně.

Popis herce se může měnit podle daných podmínek (tedy i v čase), což umožní animovat i tvar tělesa, tedy informaci závislou na konkrétní reprezentaci.

Výše popsaným způsobem je animační jazyk MDL odstíněn od konkrétní geometrické reprezentace. Animační metody zasahují pouze do struktury tvarového stromu a o konkretní zobrazení se postará grafický systém.

Činnost

Činnost je procedura v jazyku MDL, která mění hodnoty v tabulkách argumentů. Činnost je vždy přiřazena konkrétní entitě. Změna hodnot argumentů se provádí pomocí posloupnosti příkazů. Příkazy obsahují algebraické operace s vektory, skaláry a některé funkce. Syntax a sématika jazyka je podrobně vysvětlena v příloze. Příkazy následují za klíčovým slovem **`is`**. Příkazy mohou pracovat s argumenty, proměnnými a kostantami. Proměnné se zapisují za klíčové slovo **`var`** a konstanty za **`const`**. Procedura činnosti může obsahovat podmínku na ukončení činnosti uvozené slovem **`condition`**. Nalezne-li interpretr v proceduře nějakou podmínku, pak zjišťuje, zdali je tato podmínka splněna. Není-li splněna, pak činnost procedury končí. Podmínek může být v činnosti i více.

Deklarace vlastní činnosti může být uvozena více klíčovými slovy. Je to proto, že účel dané procedury může být různý. Uživatel má proto přehled co mají jednotlivé procedury za úkol. Program MDL užívá následující klíčová slova: **`activity`**, **`trace`** a **`time`**.

Procedura označená slovem **`time`** provádí časovou substituci, tj mění lokální hodnotu času. Rozdělení výpočtu času a trajektorie je vhodný pro **timewarpy** [21]. Procedura označená **`trace`** by měla měnit trajektorii entity. A konečně procedura označená slovem **`activity`** nerozlišuje, o který z výše zmíněných typů jde.

V každé činnosti je možno volat jinou činnost (pokud byla definovaná dříve). Volaná činnost změní hodnoty tak, jak má (jsou-li splněny podmínky) a vrátí změměný stav nadřazené činnosti.

Uvedeme si příklad činnosti:

```
...
// začátek deklarace činnosti
// za identifikátorem činnosti následuje její parametr a jeho typ
activity Test radius(i)
 // deklarace proměnných
 var prom1(i) prom2(i:vector)
 // deklarace konstant
 const con1 (d) 0.45
 // posloupnost příkazů
 is prom1=con1*45+18
 prom2= cas * [10,20,30]+[100,20,30]
 // podmínka na návrat
 condition ((cas###20)###(cas###25))
 // posloupnost příkazů
 is prom1=con1*3+1
 prom2= (cas * [1,2,3]+[100,20,30])/25
 ...
...
```

Časové procedury

Při vytvoření prostoru se v něm začíná počítat čas, který se zvětšuje v diskrétních krocích (definovaných při deklaraci). Čas se počítá vždy od 0, ale dá se lokálně měnit. Klíčové slovo **`time`** uvozuje časovou substituci, která v určitém místě prostoru "deformuje čas" (**timewarpy** [21]). Oddělení trajektorie a času je pro animaci velice výhodné, především pro editaci animace. Popisem trajektorie zadáme dráhu herce a časovou substitucí rychlost pohybu herce. Tyto dvě složky pohybu lze editovat nezávisle, což je nezanedbatelnou výhodou například, chceme-li změnit rychlost herce a přitom zachovat trajektorii.

Jako příklad výše zmíněného užití lze uvést tlumené kmitání. Trajektorie je obyčejnou přímkou, zato časová substituce má tvar :

$$t = A.e^{-t}.\sin(B.t + C) + D.$$

Příklad deklarace lineární časové substituce:

```
time Linear a(i) b(i)
 is cas=a*cas+b
```

Trajektorie

Trajektorie je většinou popsána parametrickými rovnicemi. Uživatel v příkazech činnosti mění argumenty transformací v závislosti na čase. Pro určení trajektorie může využívat hodnot proměnných a konstant deklarovaných v proceduře, hodnot argumentů v tabulkách uložených v uzlech patřících aktuální entitě. Je možné se dokonce odkazovat na hodnoty argumentů v předcházejícím časovém kroku.

Příklad deklarace parametrické trajektorie je následující:

```
trace Spiral a(i) b(i) c(i)
 is x=a*sin(t)
 y=b*cos(t)
 z=c*t
```

Závěr

V uvedené práci byl popsán program MDL, který umožňuje vytvářet a animovat prostorovou scénu. Z předcházejícího popisu je patrné, že program svou koncepcí zasahuje do několika oblastí počítačové grafiky. Jde především o oblasti interaktivní tvorby prostorových modelů, zobrazování scény, uživatelského grafického rozhraní a hlavně o oblast animace.

Metody pro tvorbu a modifikaci scény jsou známy a úkolem diplomové práce v této oblasti byla jejich vhodná implementace . Metody popisu animace jsou naproti tomu ve stádiu vývoje. Úkolem animační části diplomové práce bylo navrhnout a implementovat nový popis animace v podobě jazyka MDL. Výsledek této práce je popsán v předcházejícím textu.

Na závěr bychom se měli zmínit o možných rozšířeních programu. Přestože je program plně funkční, není uzavřeným projektem. Vzhledem k rozsáhlosti práce existují možnosti jak zrychlit a zkvalitnit běh programu. Jde především o užití modifikací již použitých metod, o nové způsoby zobrazení scény (např. Phongův model) a o rozšíření nástrojů animačního jazyka (např. o knihovny základních pohybů).

Přílohy

Popis jazyka MDL

Jazyk je syntézou interaktivního a algoritmického přístupu k popisu animace. Dovoluje uživateli popsat animační sekvenci pomocí algoritmu zapsaného v jednom textovém souboru. Při tvorbě algoritmu je však možno přebírat některá data (o poloze a orientaci) interaktivně . Celý pohyb v prostoru resp. na scéně je generován v jednotlivých **snímcích** (**keyframe**). Tyto snímky vytvářejí **sekvenci**. Sekvence je popsaná pohybovým algoritmem, kterému budeme říkat **animace**.

Požadavky při implementaci jazyka

Hlavní část jazyka (animační) je samozřejmě nezávislá na implementaci, naproti tomu část, která zobrazuje objekty na scéně, závisí na konkrétní reprezentaci objektu, a to se musí projevit i v koncepci jazyka (viz. Popis herce). Konkrétnímu programu, v němž je naimplementovaná určitá reprezentace těles, a který umí tato tělesa zobrazit, budeme říkat **grafický systém**. V programu MDL je užito autorské grafické prostředí na bázi hraniční reprezentace.

Grafický systém by měl umět zobrazovat prostorové objekty, je lhostejno, zda-li je reprezentace na bázi **CSG**, důležité je však, že systém je schopen komunikovat s jazykem prostřednictvím předepsaných struktur a ty pak ukládat do seznamu, na jehož prvky se může animace odkazovat. Odkazy probíhají pomocí identifikátorů, takže je odstíněna vlastní geometrická reprezentace.

Zápis animace

Animace v jazyce MDL je uložena v jednom textovém souboru, který má standardně příponu **anm**. Tento text může obsahovat **identifikátory, klíčová slova, operátory, literály a komentá ř**. Animace musí dále splňovat podmínky pro zápis v tomto jazyce:

a) Animace se významově zapisuje od shora dolů.
b) Animace se skládá z deklaračních bloků. Deklarační blok začíná klíčovým slovem, které musí být deklarátorem prvního druhu (viz. deklarátory) a končí v místě, kde se při postupu od shora dolů objeví další deklarátor stejného druhu nebo kde je konec souboru.
c) Animace obsahuje alespoň jeden deklarační blok . Je to blok, v kterém se deklaruje prostor.
d) V deklaračním bloku je možno se v některých případech odkazovat na jiné deklarační bloky (pomocí identifikátoru). Tyto bloky však musí být deklarovány nad tímto blokem.
e) Všechny deklarátory se zapisují na nový řádek.
f) Jsou-li součástí deklaračního bloku příkazy (za klíčovým slovem **`is`**), pak se zapisují buď na samostatný řádek, nebo jsou odděleny oddělovačem příkazů (středníkem) od jiného příkazu na témže řádku, tj. za posledním příkazem nemusí být středník. Totéž platí i o deklaraci proměnných, impulsů a o argumentech volané funkce s tím rozdílem, že oddělovačem je zde mezera.
g) Při zápisu nesmějí být klíčová slova a identifikátory rozděleny vložením bílého znaku.

Základní prvky jazyka

~~Znaky~~

V animacích lze užívat následující znaky pro malá písmena:

```
abcdefghijklmnoprstuvwxyz
```

obdobně znaky pro velká písmena:

```
ABCDEFGHIJKLMNOPRSTUVWXYZ
```

číslice:

```
0123456789
```

a konečně speciální znaky:

```
/*+-()###############~;,&¦{}!:
```

Ostatní znaky lze používat pouze v komentářích.

~~Identifikátory~~

Identifikátor chápeme jako posloupnost písmen a čísel, začínající písmenem. V jazyce se rozlišují velká a malá písmena. Identifikátor se nesmí shodovat se žádným klíčovým slovem. Příklad správných identifikátorů:

```
Activity, aCtivity, goal....
```

~~Komentáře~~

Komentáře se řídí zčásti konvencí jazyka C resp. C++. Pro komentář je vyhrazen znak /. Komentář je uvozen dvojicí těchto znaků. Je-li před nějakým textem tato dvojice znaků, pak je tento text vynechán (ignorován preprocesorem) a nazývá se komentář. Vynecháni textu platí pouze do konce řádky.

~~Klíčová slova~~

Klíčová slova jsou vyhrazenými identifikátory pro některé standardní typy dat, některé příkazy, deklarátory atp. Píšou se vždy s malými písmeny. Klíčová slova lze užít jako klasické identifikátory, pouze v deklaraci proměnných a konstant a příkazech. Mezi klíčová slova patří:

```
activity
actor
arg
condition
const
d
description
dofrotation
for
homogen
i
if
is
magnify
matrix
reldofrotation
relrotation
rotation
```

```
space
time
timestep
trace
translation
var
vector
while
```

Výrazy

Výraz je konstrukce sloužící k výpočtu hodnoty. Sestavujeme jej z operandů a operátorů. Operátory jsou popsány v kapitole uvedené níže. Ve výrazech se neuplatňují logické a relační operátory. Operandy jsou proměnné, konstanty a standardní funkce. Standardní funkce se volají následujícím způsobem:

```
identifikator_standardni_funkce (parametry ).
```

Jazyk MDL poskytuje standardní funkce pro výpočet skalárního součinu (**dprod**), vektorového součinu (**vprod**), goniometrické funkce (**sin, cos**), exponenciální funkce (**exp**), čtení zprávy (**getmsg**), čtení složky vektoru a matice (**vmem,mmem**) a jiné funkce uvedené v kapitole Příkazy. Výše zmíněné funkce však vrací hodnotu, a proto je možno je užít ve výrazech.

Podmínky

Podmínky jsou logické výrazy, u kterých je možné stanovit pravdivostní hodnotu. Podmínky mohou být jednoduché a složené. Jednoduchou podmínkou je buď samotný výraz, nebo jeho logická **negace (!)**, nebo relace mezi dvěmi výrazy(**<,=,>**). Složené podmínky jsou v kulatých závorkách, skládají se z jednoduchých nebo opět složených podmínek a jsou odděleny logickými operátory. Tyto operátory jsou logický **and (&)** a logický **or (|)**. Složené podmínky se vyhodnocují tak, že se vyhodnotí nejprve nejhlouběji vnořené podmínky. Vyhodnocování výrazů v podmínkách je identické s vyhodnocováním klasických výrazů. Operátory použité v podmínkách jsou popsané v následujících kapitolách.

Příklad zápisu podmínek je následující

```
condition ((a=b)&(c<=12)&((d)|(getmsg(...))))
...
```

Příkazy

Příkazy mohou být součástí jednotlivých deklaračních bloků. Jsou uvozeny klíčovým slovem **is,** nebo deklarátory druhého druhu. Jednotlivé příkazy jsou od sebe odděleny buď koncem řádku, nebo oddělovačem (středníkem). Příkazy se vykonávají od shora dolů resp. zleva doprava. Příklad zápisu příkazů je následující :

```
// začátek příkazového bloku a prvni přikaz
is prikaz
 // více příkazů na řádce
```

```
 prikaz1; prikaz2
...
```

Mezi příkazy se řadí **jednoduché příkazy** (přiřazovací příkazy a volání funkcí), **složené příkazy** a **konstrukce if, while** a **for.**

~~Jednoduché příkazy~~

a) **Přiřazování**: přiřazování přiřazuje proměnné (reprezentované identifikátorem) novou hodnotu. Vykoná se pomocí operátoru **=**. Syntax je následující:

```
identifikátor=výraz .
```

Deklarační bloky lze volat dvojím způsobem, a to buď jako přímé volání deklaračního bloku, nebo volání tohoto bloku jako funkce. Dvojí způsob volání byl zaveden proto, aby bylo možno zapisovat volání bloků jak pomocí seznamu činnosti, tak se na něj odkazovat v posloupnosti příkazů.

b) **Volání deklaračních bloků**: funkce deklarované v deklaračním bloku se volají pomocí deklarátorů druhého řádu, za kterým následuje identifikátor bloku a seznam parametrů v kulatých závorkách a navzájem oddělených čárkou. Na dalších řádcích se může opakovat nový identifikátor bloku a jemu odpovídající parametry. Mezi identifikátorem bloku a seznamem parametrů je alespoň jedna mezera. Syntax:

```
deklarátor      identifikátor1 (parametry).
                identifikátor2 (parametry)
```

c) **Volání funkcí**: funkce mohou být buď deklarační bloky, nebo standardní funkce. Standardní funkce, které mají sami o sobě význam příkazů, jsou tři: nastavení elementu vektoru a matice (**vset,mset**) a odeslání zprávy(**sendmsg**). Volání funkce tímto způsobem může být pouze v animaci za deklarátorem **is**. Funkce se volají následujícím způsobem :

```
identifikátor_funkce (parametry ).
```

Za identifikátorem následují parametry v závorce a oddělené čárkou. Na tomto místě si můžeme uvést příklad jednotlivých příkazů:

```
...
// deklarační blok
activity Proc1 radius(i)
 ...
...
activity Proc2
 var translace(i:vector)
// začátek příkazového bloku
 is
 // příkaz typu a)
 translace=cas*###10,2,5###+###12,5,0###
 // příkaz typu c)
 Proc1 (100)
```

```
...
// definice herce
actor Herec
  // příkazy typu b)
 activity  Proc1 (100)
       Proc2
...
```

~~Složený příkaz~~

Několik příkazů je možno sdružit do bloku a na tento blok pak pohlížet jako na příkaz. Takovému příkazu se říká složený příkaz. Složený příkaz začíná a končí složenou závorkou. V souvislostí se psaním příkazů je nutno oddělit příkazy na stejné řádce středníkem. Za posledním příkazem nemusí být středník. Syntax složeného příkazu je následující:

```
{prikaz1;prikaz2
 prikaz3;...;prikaz_n}
...
```

~~Příkaz if~~

Tento příkaz umožňuje podmínit běh animace. Vykonání části příkazového bloku závisí na splnění zadané podmínky. Podmínka se zapisuje do kulaté závorky za klíčovým slovem **if**. Podmínky se musí řídit pravidly ze stejnojmenné kapitoly. Syntax konstrukce **if** je následující :

```
if (podminka) {prikaz}
...
```

Příkaz **if** vyhodnotí podmínku **podminka**, a je-li pravdivá, pak vykoná příkaz **prikaz**, není-li pravdivá, pak se nic neprovede.

~~Příkaz while~~

Tento příkaz vytváří cyklus v běhu animace. Cyklus se vytváří do té doby, dokud je pravdivý výraz v závorce. Zápis cyklu je uvozen klíčovým slovem **while**, za kterým následuje podmínka v kulaté závorce a příkaz ve složených závorkách (jako složený příkaz). Syntax konstrukce **while** je následující:

```
while (podminka) {prikaz}
```

Příkaz **while** vyhodnotí podmínku **podminka** a je-li splněna, pak vykoná příkaz **prikaz**. Potom opět vyhodnotí podmínku **podminka** a vše se opakuje, dokud platí podmínka **podminka.**

~~Příkaz for~~

Je také příkaz pro tvorbu cyklů. Lze říci, že jde o speciální případ cyklu **while**. Tento příkaz se zapisuje za klíčové slovo **for**, za nímž následují tří příkazy a jeden výraz v pořadí v jakém jsou zapsány v následujícím schématu:

```
for {prikaz1}(podminka){prikaz2} {prikaz3}
```

Význam je cyklu **for** ,ve smyslu definice příkazu **while**, je následující:

```
prikaz1
```

```
while(podminka)
 {prikaz2;
 prikaz3}
```

~~Operátory~~

Operátory určují co s daným operandem (-y) provést. Jazyk MDL zná **operátory unární a binární.**

Unární operátory jsou složeny z operátoru logické negace (**!**) a operátoru časového návratu (**~**). Operátor logické negace se používá v podmínkách. Je-li před podmínkou tento operátor, pak se logická hodnota podmínky neguje. Operátor časového návratu se používá pro odkaz na hodnoty argumentů (v dané tabulce) v předcházejícím časovém kroku. Operátor se užije tak, že se zapíše před identifikátor argumentu.

Binární operátory tvoří podstatně širší množinu. Mezi binární operátory patří operátor sčítání (**+**), odčítání (**–**), násobení (*****), dělení (**/**), přístupu (**.**), logické and (**&**), logické or (**¦**), rovnosti (**=**), menší (**<**) a větší (**>**). Tyto operátory lze dělit do třech skupin, operátory aritmetické, logické a operátor přístupu. Je-li více operátorů v jednom výrazu (resp. příkazu), pak jsou priority výpočtu stanoveny následujícím způsobem: Nejvyšší prioritu má operátor přístupu, pak jsou operátory aritmetické a operátory logické.

Operátory aritmetické se užívají velice často. Představují základní aritmetické operace. Tyto operace se vykonávají v případě, jsou-li operandy přípustného typu. Nejsou-li operandy přípustného typu, pak se interpreter pokusí provést typovou konverzi. Není-li možné provést konverzi, pak interpreter ohlásí běhovou chybu. Priority operandů a vyhodnocování výrazů je podobné jazyku Pascal (podrobněji v kapitole o interpretaci). Nejvyšší prioritu má násobení a dělení před sčítáním a násobením.

Typy

Typy užívané v jazyku MDL a jim odpovídající proměnné lze rozdělit do dvou tříd, na **elementární** a **deklarativní typy.**

ELEMENTÁRNÍ TYPY:

Elementární typy lze rozdělit na **argumenty, proměnné a konstanty**. Elementární typy odpovídají představě proměnných v jiných jazycích (C,Pascal apod.). Lze jich užívat ve výrazech, jako parametrů deklaračních bloků a standardních funkcí.

a) **Argumenty** uchovávají polohu a orientaci entity, protože jsou vstupními hodnotami prostorových transformací. Tyto argumenty jsou deklarovány v hlavním deklaračním bloku (viz. deklarace prostoru a popis herce). Touto deklarací se jejich identifikátory stanou vyhraženými a nemohou se používat jako identifikátory proměnných nebo konstant. Jak bylo řečeno ve vlastním textu diplomové práce, vytvoří se v deklaraci prostoru tabulka těchto argumentů společně s tabulkou transformací. Po této operaci se tabulka argumentů zkopíruje do všech uzlů všech tvarových stromů. Vždy, když užíváme argumenty, tak je nastaven jeden aktuální tvarový strom. Odkazujeme-li se na argumenty kořene tohoto stromu, pak stačí zapsat pouze identifikátory těchto argumentů, jinak musíme zapsat celou cestu k požadovaným argumentům pomocí operátoru přístupu. Obě možnosti si budeme ilustrovat na následujícím příkladě:

```
// deklarace prostoru s tabulkou argumentů
space Prostor
 arg  rot(i:vector) rotation
 tran(i:vector) translation
 mag(i:vector) magnify
 t(i) timestep
 condition t<100
...
...
 ...
// užití argumentů
 is
 // příklad odkazů na kořenové argumenty
 // v aktualním stromě
 tran=t*###10,2,5###+###12,5,0###
 // příklad odkazů na argumenty v podvětvích
 // aktualniho stromu nebo v jiném stromu
 auto.kolo.rot=letadlo.kridlo.tran+###10,0,0###
...
```

Zápis deklarace argumentů je uvozen klíčovým slovem **arg,** za kterým následuje posloupnost uspořádaných trojic : **identifikátor argumentu, typ argumentu a typ transformace**. Typ argumentu se zapisuje pomocí identifikátoru typu do kulatých závorek. Všechny typy transformací požadují jako argument některý z vektorových typů. Časový krok musí být naopak skalár.

b) V jazyce MDL je možné definovat **proměnné** různých typů. Tyto proměnné jsou vždy v deklaračním bloku uvozeny klíčovým slovem **var**. Jejich platnost se vztahuje pouze na úsek deklaračního bloku. Každá deklarace proměnné začíná jejím identifikátorem, za kterým následuje v kulatých závorkách identifikátor typu proměnné. Deklarace jednotlivých identifikátorů jsou vždy od sebe odděleny mezerou nebo novým řádkem. Jazyk poskytuje dva integrální typy **integer** a **double**, vektorové typy **vektor** a **homogenní vektor** a maticový typ **matice**.

Identifikátory typů se vytvářejí pomocí kombinace klíčových slov. Typ **integer** má identifikátor **i**, typ **double d**, typ **vektor** (se složkami typu **int**) **i:vector**, typ **homogenní vektor** (se složkami typu **double**) d:homogen, **matice** (se složkami **integer** a rozměrech 3x3) **i:vector:matrix** atp. V následujícím příkladě si uvedeme deklarace některých proměnných.

```
// začátek deklarace proměnných
// deklarujeme proměnnou a typu integer
// a proměnnou b typu double
var    a(i) b(d)
       // proměnnou rot typu integer vektor
       rot(i:vector)
       // matice 3x3 se složkami typu integer
       mat1(i:vector:matrix)
       // matice 4x4 se složkami typu double
       mat2(d:homogen:matrix)
```

```
...
```

c) Stejná pravidla jako pro deklaraci proměnných platí i pro deklaraci konstant. Rozdíl mezi těmito typy je pouze v tom, že hodnota konstanty je specifikována již při deklaraci, a že není možno přiřazovat hodnotu konstantě. Deklarace konstanty je uvozena klíčovým slovem **const**, za ním následuje identifikator konstanty, za ním deklarace typu konstanty a hodnota konstanty oddělená mezerou. Hodnoty konstant se zadávají pro skaláry zapsáním číselné hodnoty (např. zapsáním 0.45 nebo 10), pro vektorové konstanty se používá zápis vektoru (homogenu) pomocí hranatých závorek (např. ###12,5,0### nebo ###12,5,0,1###). a pro matici se užívá vyjádření podobné vektorům (zapisují se jako vektor složený z vektorů, tj. ### ###1,0,0######0,1,0######0,0,1######).

Příklad deklarace konstatnt :

```
...
// začátek deklarace konstant
// deklarujeme konstantu a
// typu integer inicializovanou hodnotou 10
const a(i) 10
        // konstanta rot typu integer vektor
        // inicializovaná hodnotou ###12,5,0###
        rot(i:vector)  ###12,5,0###
        // matice 3x3 se složkami typu integer
        // inicializovaná hodnotou ######1,0,0######0,1,0######0,0,1######
        mat1(i:vector:matrix)     ######1,0,0######0,1,0######0,0,1
######
...
```

~~Přehled elementárních typů~~

~~Číselné typy~~

Číselné typy jsou zcela identické typům definovaným v jazyce C resp. C++. Typ integer představuje přirozená čísla (2B) a typ double představuje číslo s plovoucí desetinou čárkou (8B). Celočíselný typ má následující rozsah:

i: −32768...32767

Typ **double** má rozsah:

d: 1,7E −308...1,7E+308

který platí jak v kladném, tak záporném oboru.

~~Vektorové typy~~

Od číselných typů jsou odvozeny vektorové typy. Jde buď o vektory z **trojrozměrného prostoru,** nebo z **prostoru homogenních souřadnic**. Vektor **identifikator** ve 3D prostoru se složkami typu **typ** deklarujeme pomocí klíčových slov **var** a **vector** následujícím způsobem:

```
var identifikator(typ:vector)
```

Obdobně se deklaruje homogenní vektor **identifikator** v prostoru homogenních souřadnic pomocí klíčového slova **homogen**:

```
var identifikator(typ:homogen)
```

S výše zmíněnými vektorovými typy je možno provádět následující operace : sčítání vektorů, odčítání vektorů, násobení skalárem. Dále jsou dostupné některé standardní funkce, jako třeba vektorový a skalární součin, nastavení a čtení jedné složky vektoru apod.

~~Matice~~

Dalším elementárním typem, který je dostupný uživateli po vytvoření prostoru, je typ matice. Chceme-li vytvořit matici **identifikator**, která má složky typu **typ** a působí na vektory z trojrozměrného prostoru nebo prostoru homogenních souřadnic, pak obdobně jako u vektorů deklarujeme instance tohoto typu pomocí klíčového slova **matrix** :

```
var identifikator1(typ:vector:matrix)
 // matice z trojrozměrných vektorů
 identifikator2(typ:homogen:matrix)
 // matice z homogennich vektorů
```

S výše zmíněným maticovým typem je možno provádět následující operace : sčítání matic, odčítání matic, násobení skalárem, složení matice a vektoru. Dále jsou dostupné některé standardní funkce, jako třeba nastavení a čtení jedné složky matice apod.

DEKLARATIVNÍ TYPY:

Deklarativnímu typu odpovídá v objektovém programování pojem třída, nebo struktura. Tyto typy mají většinou za úkol naplnit atributy určité struktury, a tudíž je nelze užít přimo v příkazech (výjimkou je volání činností). Naproti tomu deklarativní typy mohou obsahovat posloupnost příkazů. Deklarativnímu typu odpovídá vždy jeden deklarační blok. Jsou to typy **time**, **trace**, **activity**, **description**, **actor** a **camera.**

Tyto typy představují složitější struktury, které tvoří páteř celého jazyka. Všechna klíčová slova odpovídající těmto typům se nazývají **deklarátory**. Deklarátory se dále dělí na deklarátory **prvního a druhého druhu**. Deklarátory prvního druhu jsou samotná klíčová slova a druhého druhu jsou tato klíčová slova s prefixem alespoň jedné mezery.

Každá instance deklarativního typu se definuje právě jedním deklaračním blokem. A naopak každý deklarační blok definuje právě jednu instanci. Deklarační blok je úsek animace, který začíná deklarátorem prvního druhu. Tento deklarátor specifikuje o jaký deklarační blok jde. Deklarační blok končí buď s koncem souboru, nebo před dalším deklarátorem prvního druhu. Deklarační blok může obsahovat deklarátory druhého druhu a bloky jiného charakteru (např. příkazový blok).

Jak již bylo řečeno, mohou být deklarační bloky odpovídající činnosti (activity,trace a time) volány z jiných bloků. Z tohoto důvodu mohou mít tyto bloky parametry. Parametry se zapisují při deklaraci za identifikátor bloku. Jednotlivé parametry se oddělují mezerami.

Schématický syntaktický zápis deklaračního bloku je následující:

```
// začátek deklaračního bloku
deklarátor identifikátor parametry
 // deklarace proměnných a konstant
 deklarátory a bloky
 ...
// konec deklaračního bloku
```

```
// deklarátor začínající jiný deklarační blok
deklarátor
...
```

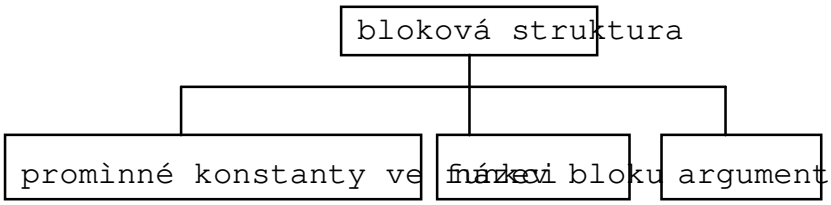
Význam jednotlivých deklarativních typů je popsát podrobně ve vlastním textu diplomové práce.

Preprocesor a interpretr

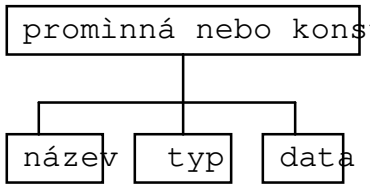
Jazyk je svou povahou **interpretr**. Jako zdroj animace používá textový soubor s koncovkou **.anm**. Aby byla interpretace co nejrychlejší, je před vlastním interpretrem spuštěn **preprocesor**, který textový soubor převede do mezikódu. Interpretr tedy neinterpretuje text ze souboru, ale mezikód připravený preprocesorem.

Předzpracování animace probíhá po řádcích zdrojového textu. Každý řádek je zpracováván nezávisle. Výjimkou je pouze udržování odkazů na nadřazené bloky. Předzpracovávaný řádek se přepíše na konci této činnosti mezikódem. Pravidla pro tvorbu mezikódu jsou následující:

Detekuje se deklarátor prvního druhu. Začínajícímu deklaračnímu bloku je přiřazena bloková struktura. Postupným načítáním řádek bloku se získají potřebné údaje o struktuře bloku (např. deklarace proměných a konstant, parametry bloku atp.) . Tyto údaje se zapíší do této struktury, která má následující podobu:



Struktura obsahuje identifikátor bloku, seznam proměných a konstant, číslo prvního řadku bloku atp. Seznami proměných a konstant jsou realizovány jako dynamické seznami tříd majících následující strukturu.



Instance této třídy představuje jednu proměnnou, nebo konstantu. Identifikátor proměné je udržován jen po tu dobu, kdy je třeba vytvářet odkazy na proměnné, pak je zrušen. Teprve před vlastní interpretací se alokuje prostor pro vlastní data proměnné, a to podle jejího typu.

Je-li naplněna bloková struktura odpovídající zjištěnému deklarátoru, pak je uložena do seznamu bloků. Jde-li o deklaraci herce, nebo kamery, pak se společně s blokovou strukturou vytvoří struktura popisující entitu . Tato struktura obsahuje dvě smyčky zpráv pro aktuální a předcházející časový krok. Dále obsahuje odkaz na odpovídající blokovou strukturu a argument specifikující aktuální tvarový strom. Strukturu entity zařadí preprocesor do seznamu entit.

Místo řádky s deklarátorem prvního druhu se zapíše pouze znak začátku bloku.

V kontextu deklarátorů prvního druhu se detekují deklarátory druhého druh (tj. mající před sebou alespoň jednu mezeru). Tyto deklarátory uvozují bloky, které končí novým deklarátorem (libovolného druhu), nebo koncem souboru. Deklarátory a jim odpovídající přepis bloků do mezikódu si popíšeme postupně v následující tabulce:

var	Odpovídající blok deklaruje posloupnost proměnných v jazyce MDL. Interpretr bere postupně všechny proměnné v bloku, detekuje jejich název a typ. Podle těchto údajů naplní instanci třídy popisující proměnné. Tuto instanci uloží do seznamu proměnných v aktuální blokové struktuře. Místo zdrojového textu nageneruje prázdný příkaz.
const	Odpovídající blok deklaruje posloupnost konstant v jazyce. Interpretr bere postupně všechny konstanty v bloku, detekuje jejich název, typ inicializační hodnotu. Podle těchto údajů naplní instanci třídy popisující proměnné. Tuto instanci uloží do seznamu proměnných v aktuální blokové struktuře. Místo zdrojového textu vloží do mezikódu příkazy na inicializaci konstant.
is	Blok následující za tímto deklarátorem obsahuje posloupnost příkazů. Preprocesor nahradí všechny indetifikátory proměnných, konstant a argumentů (pouze kořenových) odkazy do odpovídajících tabulek. Najde-li argumenty specifikované pomocí operátoru přístupu, pak celý řetězec specifikace označí interpretru. Na začátek a konec příkazového bloku se nagenerují v mezikódu znaky pro příkazový blok.
condition	Preprocesor přepíše blok podmínek podobně jako blok příkazů. Rozdíl je pouze v použitém znaku určujícím začátek a konec bloku podmínek.
shape	Tento blok přiřadí tvarový strom aktuální entitě. Preprocesor nalezne tvarový strom, specifikovaný identifikátorem, v seznamu stromů a jeho index zapíše spolu se speciálním symbolem do mezikódu.
arg	Tento blok popisuje vzor argumentů transformací. Preprocesor z načtených hodnot sestaví tabulku argumentů. Do textu se zapíše prázdný příkaz.
time, trace, activity	Tyto deklarátory druhého řádu určují blok činností. Preprocesor detekuje jednotlivé činnosti. Místo činností nageneruje do mezikódu odskok na danou činnost a nastavení parametrů.

Interpretace animace zapsané již v textu mezikódu se dá popsat následujícím algoritmem:

```
cas=0;
nakopírování tabulky argumentů do všech uzlů všech stromů;
// časová smyčka
while ( je splněna podmínka na trvání animace )
    {
      přenos aktuálního času do všech uzlů;
      // smyčka entit
      for (všechny entity na scéně)
            {
             vykonej příkazy mezikódu v bloku aktuální entity;
             ztransformuj entitu podle tabulek v jednotlivých uzlech
             aktuálního tvarového stromu;
            }
```

```
    transformuj scénu podle argumentů kamery;
    vykresli daný snímek;
    cas++;
}
```

tento zjednodušený algoritmus popisuje základní rámec interpretace animace. Nyní se budeme podrobněji věnovat způsobu interpretace mezikódu. Máme úkol interpretovat všechny příkazy resp. blok dané entity. Víme, že blok entity začíná na určité řádce, na této řádce začneme také s interpretací. V mezikódu se mohou vyskytnout následující znaky : **okraje bloků, odskoky na daný řádek, bloky příkazů, bloky podmínek a nastavení tvarového stromu**. Interpretace bloků v entitě zpravidla probíhá tak, že první blok je odskok na příkaz pro nastavení tvarového stromu. Odskoky se provádějí tak, že se aktuální pozice v mezikódu uloží na zásobník a ukazatel polohy v kódu se přesune na nové místo. Po návratu se ze zásobníku obnoví původní hodnoty. V našem případě šlo o odskok na místo, kde se přiřazuje nový tvarový strom. Interpretr nalezne na tomto místě příkaz na provedení přiřazení a ten také vykoná. Pak se běh interpretace vrátí zpět. Takto interpreter odskakuje podle jednolivých odkazů na vnější bloky. Nalezne-li interpreter blok příkazů, pak si tento blok překopíruje do pomocného seznamu. Tento seznam slouží jako vstup pro interpretr příkazů. Ten analyzuje tento seznam a rozděluje příkazy na následující množiny**: přiřazovací příkazy, volání funkcí (resp. bloků), složené příkazy, příkazy if, příkazy while, příkazy for a prázdné příkazy**. Tyto možnosti jsou dále analyzovány a jsou postupně volány interpretry výrazů, podmínek a rekurzívně opět interpretr výrazů. Syntaktická analýza a interpretace výrazů probíhá na třech úrovních. **Výraz** je rozložen na **termy,** mezi nimiž je oprerace sčítání a odčítání. Termy jsou rozděleny na **faktory** s operacemi násobení a dělení . Faktor je rozdělen **na proměnné, argumenty, konstanty a výrazy**. Podobně se analyzují i podmínky.

Postupným průchodem činností entity se změní hodnoty argumentů v aktuálním tvarovém stromu. Proběhnou-li všechny činnosti entity, pak se podle argumentů daného tvarového stromu ztransformují tělesa přiřazená tomuto stromu. Transformace ve stromu probíhá od listů stromu směrem ke kořenu.

Jsou-li všechny entity vyčerpány (v daném časovém kroku), pak se ještě podle hodnot argumentů aktuální kamery ztransformuje celá scéna, která se posléze zobrazí. Tím je dokončen jeden krok animace.

Literatura

[1] Počítačová grafika - principy a algoritmy: Žara a kol.:Grada 1992

[2] State-of-the-art in Computer Animation (Proceedings of Computer Animation 89): Nadia Magnenat-Thalman,Daniel Thalmann

[3] Computer Graphics - principles and practice (second edition): Foley, van Dam, Feiner, Hughes : Addison-Wesley 1990

[4] Pixel - dtp-design-tisk: časopis pro počítačovou grafiku2/93

[5] Four Dimensional Splines for Motion Control: Spencer-Smith, Wyvill: [2]

[6] Animace: Lálová - diplomová práce MFF UK 1994

[7] Motion Control:From keyframe to task-level animation: Thalman:[2]

[8] The Brown Animation Generation System, Ph.D. thesis: Technical Reports CS-88-22, Brow University, Providence květen 1988

[9] Computer Animation with Scripts and Actors: Reynolds:SIGGRAPH82

[10] S-Dynamics:Symbolics, Inc., Cambridge 1985

[11] Computer Animation: Theory and Practice: Magnenat-Thalmann, Thalmann: Springer-Verlag, Tokyo, 1985

[12] Picture Driven Animation: Baecker:SJCC,AFIPS Press 1969

[13] DIAL: A Diagrammatic Animation Language: Feiner, Salesin, Banchoff: CG & A, 2(7), September 1982

[14] Bbop-A System for 3D Keyframe Figure Animation: Stern: SIGGRAPH83

[15] Sketchpad: A Man-Machine Graphical Communication System:Sutherland:Spartan Book, Baltimore, 1963

[16] Modelování a animace v počítačové grafice: Rücker:výzkumný úkol, FJFI ČVUT 1994

[17] Programovací jazyky C/C++ : M.Virius

[18] Programování obrazových adapterů PC: Šnorek, Slavík: Grada 1993

[19] Anatomie IBM PC:Zápalka:Grada 1993

[20] Světlo, barvy a barevné systémy v počítačové grafice: Skala:Academia, Praha 1993

[21] Timewarps: A temporal Reparametrization Paradigm for Parametric Animation:Smith, Drewery: Eurographics Association,1991