Modelovanie umelého pohybu na báze VRML

Andrej Lúčny

Ústav informatiky, FMFI, UK Bratislava a MicroStep-MIS
E-mail: andy@microstep-mis.com

Abstrakt. VRML umožňuje modelovať 3D scény aj vrátane dynamiky. Nepodporuje však dodržiavanie fyzikálnych zákonov, takže model je nutné globálne režírovať v priestore i čase. Príspevok navrhuje taktiku ako zabezpečiť aby sa model vyvíjal sám na základe definovaných správaní jednotlivých častí. Ústrednou myšlienkou je nahradenie externých fyzikálnych zákonitostí tým, že sa im súčiastky modelu dobrovoľne podriadia v rámci svojho správania. Takéto modely je potom možné použiť na skúmanie pohybu živočíchov, napr. hmyzu.

1. Úvod[1]

Predpokladám, že každý kto skúšal dynamiku vo VRML[2] zažil veľké sklamanie, keď pustil proti sebe dve guličky a tie cez seba preleteli miesto toho, aby sa zrazili. Najmä ak zamýšľal použiť tento nástroj na modelovanie, ktoré by mu niečo prezradilo o povahe modelovaného. Napríklad nás zaujíma otázka ako je organizované riadenie pohybu šiestich nôh u mravca. Zoberieme teda mravca a skúmame ako navonok tento pohyb vyzerá za rôznych okolností: na rovine, z kopca, do kopca, pri prekonaní prekážky. Potom si vytvoríme virtuálny model mravca na základe jeho anatómie, vytvoríme virtuálnu scénu roviny, kopca, prekážky a skúšame vytvoriť riadiaci systém ktorý premostí modelované zmysly (senzory) na modelované svaly (aktuátory).

Štandardnými prostriedkami VRML možno síce na základe modelu mravca a scény urobiť dokonalý film, ako kráča po rovine, z kopca, do kopca, cez prekážku, ale len tak, že mu zadáme priamu postupnosť ako sa v čase menia jednotlivé stupne voľnosti (teda zohnutia v kĺboch) modelu (VRML nám pritom výrazne pomôže z ich rozfázovaním). Je smutné si pri pozeraní Jurského parku uvedomiť, že keď nejaký dinosaurus naháňa nejakého herca, nehýbe ním riadiaci systém, ktorý by videl hercovu polohu a podľa toho usmerňoval pohyb dinosaura, ale že smer tohto pohybu určuje animátor, ktorý pritom musí dávať pozor, aby mu dinosaurus neprebehol cez strom. Takéto modelovanie nám totiž nemôže prezradiť nič viac o jeho pohybe, než vieme vydedukovať z jeho anatómie, umožní nám akurát tieto poznatky zviditeľniť v 3D. Nás by ale zaujímali otázky typu:

1. Je na vyjadrenie pohybu mravca vhodnejšia centrálna alebo decentralizovaná štruktúra?[3] Pri ktorom z týchto spôsobov vieme pohyb nakódovať efektívnejšie?
2. Keď vyvinieme riadiaci mechanizmus známeho cik-cakovitého pohybu mravca po rovine, dokáže s ním prejsť aj cez prekážku? A bude to vyzerať rovnako ako u skutočného mravca? Resp. ak nie, čo treba do riadiaceho mechanizmu pridať?
3. Keď potom amputujeme určité končatiny nášmu modelu – bude sa pohyb produkovaný naším riadiacim systémom podobať pohybu mravca s amputovanými končatinami?

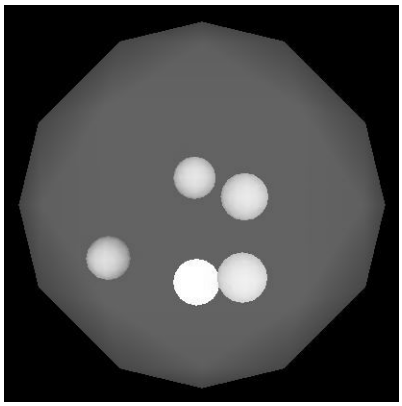
[1] Upozorňujeme čitateľa, že v celom článku u neho predpokladáme určité vedomosti o virtuálnej realite
[2] Virtual Reality Modeling Language, ver. 2.0, http://www.web3d.org
[3] Je známy výsledok, že pohyb možno vysvetliť na základe decentralizovaného riadiaceho systému [3].

4. Vieme riadiaci systém vybudovať na princípe adaptívneho systému? (Dokážeme naučiť neurónovú sieť? Dokážeme riadiaci systém evolvovať genetickým algoritmom?

2. Modelovanie fyzikálnych zákonitostí

Sústreďme sa najprv na čo najjednoduchší príklad, ktorý vyjadruje podstatu toho, čo musíme k VRML pridať, aby sme v ňom mohli vykonať modelovanie „bez podfuku“. Na tento účel si vezmeme pružné zrazy niekoľkých guličiek uzavretých vo veľkej guli, ktoré sa na začiatku pohybujú náhodným smerom a rýchlosťou (obrázok 1).

Štandardným riešením by tu bolo zaviesť do scény VRML časovaný script, ktorý buď posunie každú guličku v jej aktuálnom smere o zodpovedajúcu vzdialenosť, alebo zdetekuje jej zrážku, vypočíta jej nový smer a rýchlosť a posunie ju v tomto novom smere. Pri detekcii zrážky sa pritom spoliehame na aproximáciu v tom zmysle, že guličky sa môžu v mieste zrazu pružne deformovať o vzdialenosť zodpovedajúcu jednému tiku časovania. Nebudeme tu uvádzať rovnice, ktorými sa riadi pohyb guličky, uvedieme len, že je potrebné si všetky parametre guličky pamätať zvlášť pre predchádzajúci a nastávajúci časový okamih. Potom sa už dá kvadraticky od počtu guličiek aproximovaný pohyb spočítať.



Obrázok 1. Implementácia jednoduchého fyzikálneho modelu vo VRML.

Toto riešenie je však postavené na zvolenom zjednodušení a to síce:

1. objekty v scéne majú ideálny tvar (takže detekcia kolízie je hračkou)
2. v scéne máme objekty rovnakej povahy, takže nie je problém napísať škálovateľný script, ktorý už nebudeme musieť meniť, keď pridávame ďalšie objekty[4]

Dalo by sa riešenie upraviť tak, aby nevyužívalo tieto zjednodušenia? S bodom (1) nám môže pomôcť samotný VRML prehliadač, keďže niektoré jeho implementácie poskytujú prostriedky na detekciu kolízie[5]. Využívame tu pritom fakt, že paralelný pohyb viacerých objektov simulujeme postupným posúvaním vždy iba jedného z nich. Zjednodušenie (2) sa dá prekonať tým, že pre každý objekt v scéne zavedieme samostatný script, pričom všetky ich budeme časovať z jedného časovača. Tým vlastne tieto objekty personifikujeme: už nie sú pasívne objekty, s ktorými hýbu fyzikálne sily,

[4] Napísať takýto monolytický script pracujúci z množstvom objektov rôzneho typu je krkolomná robota
[5] Najďalej v tomto smere je prehliadač Cortona z www.parallelgraphics.com

ale aktívne objekty – tzv. **agenty**, ktoré **sa** hýbu. Pre každého agenta nakódujeme také správanie, že keď ich všetky súčasne vykonávame, deje sa to isté, ako keby sme globálne dohliadali na dodržiavanie fyzikálnych zákonov. Je to ako keby agenty tieto zákony dobrovoľne dodržiavali: gulička sa nemusí odraziť od inej, ale ona sa proste od iných guličiek naschvál odráža. Týmto duálnym prístupom k fyzike dosiahneme náš cieľ a to aby sme mali fyziku vyjadrenú spôsobom, ktorý nám umožní poľahky pridávať ďalšie objekty do scény. **Multiagentový systém** nám slúži ako vhodná reprezentácia fyzikálnych zákonov.

Guličke - agentovi samozrejme nestačí vedieť len to či zrážka nastala, musí sa dozvedieť smer a rýchlosť guličky do ktorej vrazila. Agenty musia komunikovať. Komunikácia tu môže byť – ako v každom multiagentovom systéme - priama a nepriama. Primárna je **nepriama komunikácia**, ktorou gulička prezrádza svoje parametre do tzv. **prostredia**[6] - objektu, ktorý má z hľadiska agentov pevnú adresu, dokáže do seba pojať pomenované dáta, a umožňuje agentom tieto dáta zapisovať a čítať [6]. Okrem jednoduchého read / write môže prostredie poskytovať služby na hromadnú manipuláciu s uloženými dátami. V našom prípade je užitočné aby dokázalo poskytnúť guličke parametre objektov, ktoré sa nachádzajú v jej blízkom okolí. V princípe takýto zoznam nie je viazaný na konkrétny typ objektov a agent musí podľa typu jednotlivého objektu zvážiť ako ho spracovať. Do pomenovaných dát môže agent uložiť aj referenciu na seba samého a tým umožniť iným agentom, ktorí si ju z prostredia prečítajú komunikovať s ním priamo. **Priama komunikácia** má zmysel vtedy, keď sa skôr hodí určitému objektu prikázať aby niečo so sebou urobil, než aby to usúdil sám zo stavu svojho okolia (čo preferujeme). Napríklad rozbitnej guličke môže iná gulička povedať „rozbíjam Ťa“ priamou komunikáciou, zatiaľ čo pri nepriamej gulička si povie „rozletím sa na kusy, lebo do mňa niekto tvrdo narazil“). Medzi ďalšie služby prostredia môžeme pridať schopnosť rozlišovať predchádzajúci a nasledujúci časový okamih, teda udržiavať v sebe dáta dupľovane.

Z implementačného hľadiska je script VRML uzol prepojený so špecifickým objektom v jazyku Java. Keďže pod prehliadačom sú tieto objekty púšťané v rámci jedinej JVM[7], stačí nám na implementáciu prostredia použiť triedu so statickými premennými. Priamej komunikácii bude zodpovedať volanie metódy jedného agenta iným agentom, na základe referencie získanej z prostredia.

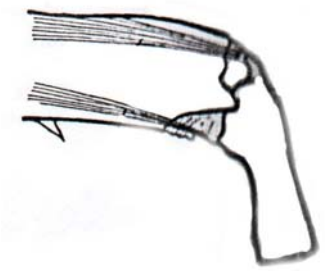
Týmto pádom máme stratégiu ako do scény implementovať fyzikálne zákonitosti. Treba však ďalej zohľadniť, že živočích nie je taký jednoduchý ako gulička. Gulička v predchádzajúcom príklade bola agentom, ktorý v sebe zahŕňal funkcie senzorov (kde sú iné guličky?), riadiaceho systému (ak nastala zrážka, zmeň smer) aj aktuátorov (pohnem sa týmto smerom). Bolo to však možné len vďaka tomu, že šlo o primitívny príklad. V skutočnom modeli musíme mať tieto funkcie oddelené.

3. Modelovanie aktuátorov

Pri modelovaní pohybu nejakého živočícha musíme zvládnuť veľký počet častí jeho tela pospájaných množstvom stupňov voľnosti, pričom ku každému prislúchajú minimálne dva svaly, prípadne jeden sval a jedna šľacha. Tradične sa takéto modely hýbu zmenou uhlov otočenia v kĺboch. My na to preferujeme používať silu, ktorou na

[6] anglický ekvivalent: space
[7] Java Virtual Machine, http://java.sun.com

kĺb pôsobia zmrštené svaly (obrázok 2.). Výhodou tohto prístupu je fakt, že môžeme na kĺby priamo rozpočítať vplyv tiažovej sily. Potom napríklad pri voľnom páde nielen telo padne na prvý bod dotyku so zemou, ale kĺb pri tomto bode povoľuje tak, ako mu to káže tiaž zvyšku tela, ktoré je ešte vo vzduchu.



Obrázok 2. Ukážka ovládania kĺbu dvojicou svalov u nohy hmyzu [1].

Z hľadiska fyziky sa vzťahy medzi časťami tela prejavujú nielen na atomickej úrovni ale v rámci určitej hierarchickej štruktúry. Napríklad, na zabezpečenie ohybu uvoľnenej hornej končatiny v zápästí sa mení v závislosti od ohybu v ramene (kvôli váhe ruky) stačia atomické agenty, ale na zabezpečenie voľného pádu, keď sa stojacej postave pokrčia naraz obe kolená, potrebujeme agenta, ktorý stráži celé telo. Našťastie VRML je priamo založené na hierarchickej štruktúre scény a je rovnako možné pýtať sa či narazil malíček ako aj či narazilo celé telo. Aktuátor bude teda implementovaný ako script majúci prístup k určitej časti scény, ktorý má určité parametre, ktoré je možné externe meniť. Pritom je dôležité domyslieť, že nie je vhodné ich meniť priamou komunikáciou, resp. nie je vhodné aby boli v scripte enkapsulované, ako je typické pre objektovú technológiu. Naopak, je dobré keď externá entita mení tento parameter tak, že zmení reprezentáciu tohto parametra v prostredí a script túto zmenu zaregistruje a „dobrovoľne“ vykoná. Povaha takejto zmeny parametra je typická pre multiagentové systémy a jej zmysel uvidíme pri modelovaní senzorov a riadiaceho systému.

Týmto spôsobom sa dá zabezpečiť, že keď riadiaci systém nášho modelu vydá povel vkročiť do steny, táto akcia sa podarí len po stenu a potom je detekovaný náraz. Podobne ak riadiaci systém pavúka zdvihne naraz všetky nohy, padne tento na brucho. Na simulovanie aktuátorov teda nepotrebujeme - mimo prostriedkov vyjadrenia fyziky pomocou multiagentového systému - nejaký ďalší špeciálny programátorský prostriedok, stačí vynaložiť obrovské množstvo mravenčej práce fyzika a anatóma.

4. Modelovanie senzorov

Ďalším problémom pri modelovaní vo VRML je získať vstupy, na základe ktorých by riadiaci systém mohol konať. Musíme zvládnuť simulovanie senzorov (zmyslov). Hmat má v princípe povahu detekcie kolízie, čiže sa dá previesť na test či je určitý hmatový orgán v náraze z iným objektom. Úd na ktorom sa hmatový orgán nachádza z tohto testu treba pochopiteľne vylúčiť, našťastie však určité prehliadače VRML túto možnosť priamo ponúkajú. Cortona dokonca podporuje aj vyslanie lúča ktorým sa zmeria vzdialenosť najbližšej prekážky, z čoho sa dá urobiť niečo ako echolokačný orgán (netopier). Nič viac však VRML priamo nepodporuje. Aby sme zrealizovali zrak

(a na úvod si dajme za cieľ vidieť len to, čo by videla v mieste očí daného živočícha kamera), musíme opäť niečo tvoriť. Nenapadlo ma nič rozumnejšie, než spriahnuť s modelovanou scénou jej duplikát, v ktorom sa avatar[8] hýbe tak, ako sa model hýbe v pôvodnej scéne. Obraz tohto duplikátu budeme potom fotografovať a posielať ako vstup do riadiaceho systému, ktorý hýbe pôvodnou scénou. V prípade niektorých prehliadačov (napr. CosmoPlayer) sa toto spriahnutie dá urobiť cez statickú pamäť, v Cortone však nie. Z hľadiska výkonu je každopádne lepšie, ak každú scénu modelujeme na vlastnom počítači (pokiaľ je to možné: pri zložených očiach hmyzu by to chcelo celkom slušné prístrojové vybavenie, takže by som sa skôr snažil aproximovať to, čo takéto oči vidia, z obrazu snímaného dvomi očami). Preto je potrebné zvoliť programátorské prostriedky, ktoré sú charakteristické pre distribuované programovanie. Existuje na to priamo analóg http protokolu zvaný vrtp, avšak my sa kloníme skôr k distribuovaným objektom na báze Java RMI[9]. Tu sa dostávame k významu nepriamej komunikácie medzi riadiacim systémom a aktuátorom: iba takto je totiž možné pokyn z riadiaceho systému odchytiť a zduplikovať ho v spriahnutej scéne. Je to krásny príklad toho, akou je nepriama komunikácia v multiagentových systémoch výhodou z hľadiska ich modifikovateľnosti. Ďalšou požiadavkou je, že aktuátory nesmú používať veci ako generátor náhody (riadiaci systém to môže), ich odozva musí byť deterministická. (Toto je zatiaľ len náš náhradný plán. Modelovanie zraku nie je nevyhnutné, lebo sa miesto neho dá použiť predstava, že výsledkom zrakového vnemu je určitá reprezentácia veľmi podobná reprezentácii samotnej scény vo VRML – riadiaci systém môže teda vnímať samotnú túto scénu a to VRML podporuje).

So simulovaním senzorov je to teda presne opačne ako so simulovaním aktuátorov: ide takmer o čisto programátorskú robotu.

5. Riadiaci systém

Samotný riadiaci systém môže byť realizovaný jediným agentom - kódom, ktorý pravidelne odčítava parametre zo senzorov a na základe toho nastavuje parametre aktuátorov. Pri zložitejšom modeli, je však potrebné vyjadriť jeho činnosť na základe jeho vnútornej štruktúry. Aj pritom môžeme ostať pri jednom agentovi, a vútornú štruktúru zabepečiť objektovým spôsobom:

- sériou správaní ako v behaviorálnej robotike,
- produkčným či expertným systémom na báze GOFAI[10],
- konekcionistickým modelom,
- interpretrom tzv. codelet-ov (napr. rutiny v LISPe, vhodné pre genetické programovanie)
- neurónovou sieťou (napr. Elmanova sieť)

Z nášho pohľadu je ale najjednoduchšie použiť pre implementáciu riadiaceho systému rovnaký spôsob ako v prípade senzorov a aktuátorov. Teda implementovať ich ako hromadu agentov, ktorí komunikujú navzájom i so senzormi a aktuátormi prostredníctvom pomenovaných odkazov v prostredí. Opäť je tu dôležitá nepriama komunikácia, aby sa jeden agent mohol dozvedieť aké pokyny vydáva aktuátorom iný

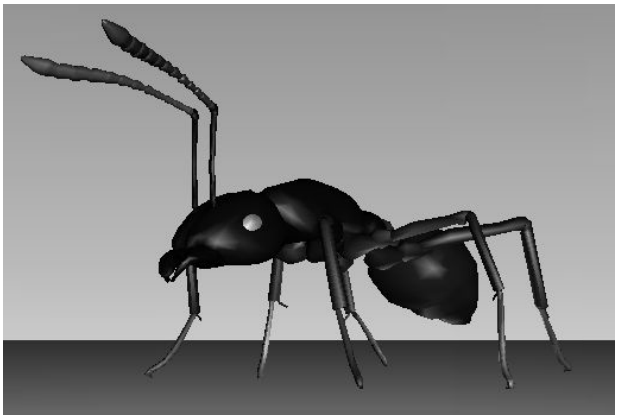
[8] fiktívny objekt reprezentujúci v scéne toho, kto sa na ňu pozerá
[9] Remote Invocation Method
[10] Good Old Fashioned Artificial Intelligence

agent a podľa toho upravil svoju vlastnú voľbu. Tým pádom je možné použiť pri implementácii riadiaceho systému princípy inkrementálneho vývoja podobné subsumpčnej architektúre [2].

6. Modelovanie tela

V neposlednom rade je pre modelovanie potrebné vytvoriť model tela modelovaného živočícha. VRML výborne podporuje vytvorenie ľubovolného tvaru zadefinovaním plôch z ktorých sa skladá a veľkosti uhla nad ktorou sa zadané zlomy interpretujú ako ohyby. Na získanie základných dát pre vytvorenie modelu sa dá použiť scanovanie modelovaného tela 2D laserom. Alternatívna metóda sa opiera o požitie dvoch všesmerových kamier so slušným objektívom (v prípade, že telo je malé). Tieto dáta je potom potrebné previesť na plochy a rozdeliť ich na anatomické časti. Toto „krájanie na hnáty“ je z celého procesu najzložitejšie a neobídeme sa pri ňom bez anatomických znalostí. Určité zakryté detaily sa musia dopracovať ručne a to žiaľ v prvom rade kĺby, rôzne výčnelky a pod.

Na internete je dnes možné sa dostať k už pomerne slušne predspracovaným dátam (obrázok 3), nie však priamo k „nakrájanému“ modelu. Anatomické údaje sa získavajú žiaľ oveľa ťažšie a dostatočne podrobný anatomický atlas je k dispozícii prakticky iba pre človeka, pre hmyz sú údaje nedostatočné aj v špecializovanej literatúre. Nám totiž nestačí vedieť, že mravec má 6 článkov na každej nohe, potrebujeme poznať presný tvar každého kĺbu alebo rozsah jeho ohybu v stupňoch (naopak vnútorné orgány nás nezaujímajú vôbec).



Obrázok 3. 3D model mravca [7].

7. Príklad

Na poskytnutie určitej predstavy ako to celé vyzerá pri implementácii uvedieme teraz schematický príklad modelovaného systému. V scéne budeme mať pohybujúcu sa guličku, ktorá sa odráža od prekážky. Napriek jednoduchosti, ponecháme v implementácii takú štruktúru akú by si vyžadoval zložitejší príklad. Preto táto guľka bude mať senzor, ktorý detekuje zrážku, aktuátor, ktorý ňou hýbe a riadiaci systém, ktorý bude meniť smer na opačný keď príde k zrážke.

```
#VRML V2.0 utf8

DEF MOVINGBODY Transform {
 translation -5 0 0
 children DEF BODY Shape {
  geometry Sphere {}
 }
}

DEF OBSTACLE Shape {
 geometry Box {}
}

DEF TIMER TimeSensor {
 loop TRUE
 cycleInterval 0.1
}

DEF SENSOR Script {
 eventIn SFTime tick
 field SFNode holder USE MOVINGBODY
 field SFNode body USE BODY
 eventOut SFInt32 bump
 url "vrmlscript:
  function tick() {
   var c = new Collidee();
   c.body = body;
   c.position =
    new SFVec3f(holder.translation);
   c.orientation =
    new SFRotation(holder.rotation);
   if(c.moveTo(holder.translation,
      holder.rotation)) bump = 0;
   else bump = 1;
  }"
}

DEF SENSORADAPTOR Script {
 url "Sensor.class"
 eventIn SFInt32 bump
}

DEF CONTROLLER Script {
 url "Controller.class"
 eventIn SFTime tick
}

DEF ACTUATOR Script {
 url "Actuator.class"
 eventIn SFTime tick
 field SFNode actuator USE MOVINGBODY
}

ROUTE TIMER.cycleTime TO SENSOR.tick
ROUTE SENSOR.bump TO SENSORADAPTOR.bump
ROUTE TIMER.cycleTime TO CONTROLLER.tick
ROUTE TIMER.cycleTime TO ACTUATOR.tick

# class-y v jazyku Java
import vrml.*;
import vrml.field.*;
import vrml.node.*;

public class Sensor extends Script {
 private Space space;

 public void initialize() {
  space = new Space();
 }

 public void processEvent(Event e) {
  if (e.getName().equals("bump")) {
   ConstSFInt32 bump =
    (ConstSFInt32) e.getValue();
   space.write("bump",bump);
  }
 }
}
```

```
public class Controller extends Script {
 private Space space;
 private SFVec3f dir;

 public void initialize() {
  space = new Space();
  dir = new SFVec3f(0.1f,0f,0f);
 }

 public void processEvent(Event e) {
  if (e.getName().equals("tick")) {
   ConstSFInt32 bump =
    (ConstSFInt32) space.read("bump");
   if (bump == null)
    bump = new ConstSFInt32(0);
   if (bump.getValue() > 0)
    dir.setValue(
    -dir.getX(),
    -dir.getY(),
    -dir.getZ()
   );
   space.write("motion",dir);
  }
 }
}

public class Actuator extends Script {
 private Space space;
 private SFVec3f pos;

 public void initialize() {
  space = new Space();
  SFNode node; Node base;
  node = (SFNode) getField("actuator");
  base = (Node) node.getValue();
  pos = (SFVec3f)
   base.getExposedField("translation");
 }

 public void processEvent(Event e) {
  if (e.getName().equals("tick")) {
   SFVec3f motion =
    (SFVec3f) space.read("motion");
   if (motion == null)
    motion = new SFVec3f(0f,0f,0f);
   pos.setValue(
    pos.getX()+motion.getX(),
    pos.getY()+motion.getY(),
    pos.getZ()+motion.getZ()
   );
  }
 }
}

public class Space {
 static Block[] block = new Block[100];
 static int count = 0;

 public int contains (String n) {
  for (int i=0; i<count; i++)
  if (block[i].getName().equals(n))
   return i;
  return -1;
 }

 public void write (String n,Object v){
  int i = contains(n);
  if (i != -1) block[i].setValue(v);
  else block[count++] = new Block(n,v);
 }

 public Object read (String n) {
  int i = contains(n);
  if (i == -1) return null;
  else return block[i].getValue();
 }
}
```

Tento príklad nebudeme podrobne komentovať, len uvedieme niekoľko implementačných skutočností, ktoré by bez komentára boli zarážajúce:

- Kód začína scénou v jazyku VRML a pokračuje implementáciou class-ov v jazyku Java.
- Zatiaľ čo riadiaci systém a aktuátor zodpovedajú jedinej class-e, senzor je rozdelený na kód v javascripte, ktorý je súčasťou popisu scény a jeho adaptér, ktorého jedinou funkciou je preniesť výsledok z javascriptu do prostredia (trieda Space) v Jave. Dôvodom je, že tu ide o konkrétnu implementáciu v prehliadači Cortona, ktorý v javascripte detekciu zrážky podporuje, ale v Jave nie. Na druhej strane v Javascripte sa nedá vymienať informácia medzi jednotlivými scriptami, zatiaľ čo v Jave to ide - cez statickú pamäť.
- Trieda Space je implementovaná pomerne primitívnym spôsobom, čoho dôvodom je fakt, že Cortona sa opiera o Microsoft JVM. Ide teda o prastarú verziu – Java 1.1.
- V implementácii Space používame objekt Block, ktorého kód neuvádzame. Je to preto, že je len púhou implementáciou dvojice mena a hodnoty.
- Grafickej stránke modelovania nie je v tomto príklade venované žiadna pozornosť, takže nepôsobí dobrým estetickým dojmom. Fázy pohybu guličky pri spustení scény môžeme vidieť na obrázku 4. Živý obraz v Internet Explorer-i možno nájsť na www.microstep-mis.sk/~andy/pub.htm.

Obrázok 4. Vývoj scény z uvedeného príkladu, štyri zábery.

8. Záver

V tomto príspevku sme sa pokúsili navrhnúť princípy modelovania pohybu živočíchov na báze VRML. Opreli sme sa pritom o multiagentové systémy s nepriamou komunikáciou medzi agentami. Upozoňujeme, že ide o prezentovanie zámeru, nie výsledkov. Avšak niektoré čiastkové výsledky nám naznačujú, že zámer nie je zlý.

Prínos tohto článku je ničím v porovnaní s prácou, ktorú ešte treba vynaložiť aby sa dosiahli ciele v ňom nastolené. Avšak nech by vytvorenie simulátora vyžadovalo akokoľvek enormnú prácu, vždy sa bude minimálne ľahšie udržiavať v chode než nejaký prototyp. Takisto znovupoužiteľnosť je výrazne vyššia.

V príspevku sme taktiež definovali určité typické otázky, ktoré by sa dali pomocou týchto modelov skúmať. Na záver by sa patrilo uviesť, prečo je vôbec dôležité sa týmito otázkami zaoberať. Už dvadsať rokov sa skúma technológia pohonov na báze materiálov s tvarovou pamäťou (SMA), kde sa pohyb dosahuje zmenou kryštalickej mriežky nikel-titánového vodiča na základe jeho ohrevu elektrickým prúdom. Tento pohon produkuje primárne nie rotačný, ale pozdĺžny pohyb. Ponáša sa teda na skutočný sval, nie je to motor [4]. Pomocou tohto pohonu bude raz možné zostrojiť roboty, v porovnaní s ktorými budú dnešné humanoidné roboty na báze motorov smiešne neohrabané, protézy, ktoré budú porovnateľné s chýbajúcou končatinou a pod. Nevýhodou tohto pohonu je, že vyžaduje spätnú väzbu (na rozdiel napríklad od servomotora). Na jeho ovládanie je nevyhnutný riadiaci systém vybavený senzormi.

Takéto riadiace systémy treba vyvinúť a na to je dobré uchýliť sa k nášmu najväčšiemu učiteľovi – matke prírode.

Literatúra

[1] Beláková A., Orságh I.: *Všeobecná entomológia*, PF UK, Bratislava, 1977.

[2] Brooks, R.: *Ambient Intelligence*, The MIT Press, Cambridge, Massachusetts, 1999.

[3] Cruse, H., Brunn, D., Bartling, Ch., Dean, J., Dreifert, M. Kindermann, Th. Schmitz, J.: *Walking – a complex behavior controlled by simple networks*. Adaptive Behavior 3, 385-418 (1995).

[4] Drahoš, P.: *Príspevok k syntéze aktuátora zo zliatiny s tvarovou pamäťou*. Dizertačná práca, KAR, FEI STU, Bratislava, 2003.

[5] Kelemen, J.: *Strojovia a agenty*, Archa, Bratislava, 2003.

[6] Lúčny, A.: *Reaktívny model inteligentného systému*, in. Kognitívne vedy II (Kvasnička V. Pospíchal J. eds.), Bratislava, 1999. http://www.microstep-mis.sk/~andy/pub.htm

[7] Sharov, A.: *Virtual Insects*. http://www.ento.vt.edu/~sharov/3d/virtual.html.

[8] Žára, J.: *VRML 97 – Laskavý prúvodce virtuálními světy*, Computer Press, Brno, 1999