Representação de Agentes Autónomos em VRML 97

Artur Caetano 5 João Pereira

INESC e DEI/IST
Rua Alves Redol 9, 1000-029 Lisboa, Portugal
{artur.caetano, joao.pereira}@inesc.pt

Sumário

Apresentamos neste documento três modelos para a representação em VRML de populações heterogéneas de agentes interactivos. Estes modelos permitem estender o conceito de interacção aos objectos da cena através da adição de comportamento autónomo e reactivo. Dois dos modelos assentam directamente na plataforma de prototipagem e de guiões patente no VRML. O modelo de representação externa utiliza o VRML apenas como meio de visualização, ficando o comportamento dos agentes definido externamente num módulo Java. São também discutidos os resultados de um estudo comparativo entre os diversos modelos em conjunto com os cenários típicos da sua utilização.

Palavras-chave

Agentes autónomos, agentes reactivos, realidade virtual, VRML, Java.

1. INTRODUÇÃO

A *Virtual Reality Modeling Language* (VRML[1]) afirma-se como uma das formas mais generalizadas de visualização e navegação em mundos virtuais [Carey97, VRML 97]. Este sucesso deve-se, entre outros factores, à sua fácil integração e difusão na WWW, e ao seu elevado grau de portabilidade.

A funcionalidade dinâmica do VRML assenta num modelo de acontecimentos de acordo com o padrão produtor-consumidor [Coplien95], onde cada acontecimento é capturado por um ou mais sensores e dirigido para este através de ligações. Os acontecimentos podem ser desencadeados através de acções temporizadas explicitadas na descrição da cena ou por interacção do avatar com os objectos do grafo da cena.

Contudo, o VRML não suporta directamente o conceito de interacção explícita entre objectos. Este factor limita a sua utilização em mundos onde existam objectos com comportamento dinâmico e que interajam autonomamente, independentemente da possível interacção com o avatar, como é o caso de mundos povoados por seres artificiais ou outros cenários de simulação dinâmica.

No entanto, o VRML ultrapassa esta limitação através da utilização de guiões[2] que podem ser associados aos vários objectos presentes no grafo da cena e que estendem o comportamento dinâmico de um objecto através da geração ou tratamento de acontecimentos. Na versão actual do VRML os guiões podem ser programados em Java *Script* ou Java. Quando o grau de complexidade é elevado (situação comum na representação de agentes autónomos), a última alternativa apresenta vantagens a nível da expressividade, funcionalidade acrescida e modularidade. Adicionalmente, o desempenho do Java é superior ao do Java *Script*.

Neste artigo apresentamos um modelo de representação de agentes que permite a sua gestão simples e flexível, e que suporta mecanismos eficientes de interacção não só entre o avatar e os agentes mas também entre os diversos agentes.

Um agente, uma entidade independente com capacidade de decisão autónoma, pode ser representado através de um par (corpo, comportamento). No contexto do VRML esta funcionalidade pode ser transcrita para um nó protótipo que representa os mesmos conceitos através de um par (geometria, guião).

Na secção 2 apresentaremos uma primeira aproximação à modelação de mundos virtuais multi-agente utilizando para tal as primitivas básicas do VRML. Nas secções 3 e 4 apresentaremos duas alternativas ao modelo não optimizado, visando a representação eficiente de agentes heterogéneos em VRML. Na secção 5 é apresentado um estudo comparativo dos vários modelos apresentados. Em apêndice são apresentados os interfaces das classes relativas aos modelos propostos, juntamente com um exemplo de aplicação.

2. MODELO NÃO OPTIMIZADO

O VRML permite adicionar guiões aos objectos do grafo da cena, o que facilita a extensão ou a criação de novos tipos de objectos, denominados protótipos, na terminologia do VRML. Este protótipos estendidos com guiões possuem funcionalidade acrescida e encapsulam toda a complexidade subjacente ao processamento adicional.

[1] VRML 97 (ISO/IEC DIS 14772-1)

[2] Do inglês *script*

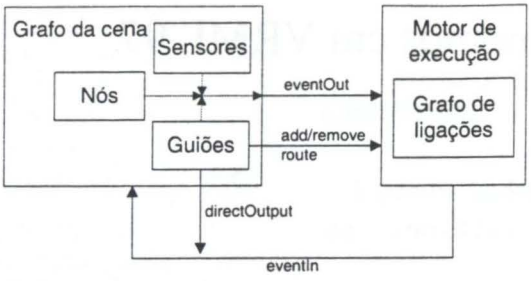

Figura 1: Modelo genérico de processamento de acontecimentos do VRML.

No modelo genérico de processamento de acontecimentos do VRML (v. Figura 1), cada nó define explicitamente o conjunto de acontecimentos de entrada e de saída que processa, sendo especificadas estaticamente as ligações entre estes através da primitiva `ROUTE` *`eventOut`* `TO` *`eventIn`*. Estas ligações são mantidas num mapa dentro do motor de execução do VRML que associa um acontecimento de saída aos respectivos acontecimentos de entrada. A semântica do processamento de acontecimentos garante que um acontecimento de saída gerado num dado instante temporal é encaminhado simultaneamente aos respectivos acontecimentos de entrada durante esse mesmo instante. Desta forma, é possível gerar acções coordenadas entre todos os nós que partilhem o mesmo acontecimento de saída. Para além das ligações estáticas, é também possível adicionar ou remover programaticamente as ligações existentes entre nós através de guiões.

Utilizando este modelo para representar um sistema com múltiplos agentes, obtemos a arquitectura ilustrada na Figura 2, onde cada agente é representado por um protótipo que define a sua geometria e comportamento.

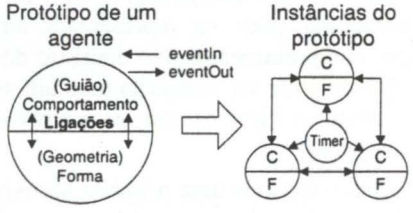

Figura 2: Representação de um agente e coordenação de agentes.

O guião que define o comportamento do agente tem como função o processamento das reacções a um ou mais acontecimentos de entrada. Estas reacções podem consistir na produção de um ou mais acontecimentos de saída (reacção externa) ou na alteração do estado interno do agente (reacção interna). O ritmo de actualização do estado dos diversos agentes é coordenado através de um temporizador partilhado por todos os agentes. Por exemplo, aplicando este modelo para representar um universo com três agentes semelhantes que interactuam através de um único acontecimento, obteríamos o seguinte grafo de cena em VRML:

```
PROTO AgentProto [
  eventIn  SFTime trigger
  eventIn  SFBool ein
  eventOut SFBool eout
]
{
  DEF BODY Transform { geometry }
  DEF RULES Script {
    eventIn  SFTime  trigger IS trigger
    eventIn  SFBool  ein     IS ein
    eventOut SFBool  eout    IS eout
    eventOut SFVec3f pos
    url "script"
  }
  ROUTE RULES.pos TO BODY.set_translation
}

DEF TIMER  TimeSensor { ... }
DEF Agent1 AgentProto { ... }
DEF Agent2 AgentProto { ... }
DEF Agent3 AgentProto { ... }

ROUTE TIMER.cycleTime TO Agent1.trigger
ROUTE TIMER.cycleTime TO Agent2.trigger
ROUTE TIMER.cycleTime TO Agent3.trigger

ROUTE Agent1.eout TO Agent2.ein
ROUTE Agent1.eout TO Agent3.ein
ROUTE Agent2.eout TO Agent1.ein
ROUTE Agent2.eout TO Agent3.ein
ROUTE Agent3.eout TO Agent1.ein
ROUTE Agent3.eout TO Agent2.ein
```

A simplicidade conceptual desta abordagem reside na utilização directa do modelo de coordenação e de guiões do VRML para se conseguir a ligação entre os diversos componentes do sistema. Neste caso, o guião apenas é responsável pela reacção aos acontecimentos de entrada. No entanto, esta abordagem possui diversos problemas:

- **Rigidez**. A introdução ou eliminação de um agente implica a actualização das ligações estabelecidas entre os restantes agentes.
- **Escalabilidade**. O número de entradas na tabela de encaminhamento para permitir a interacção entre n agentes é dada por $n \times (n-1) \times s/2$, onde s representa o número de acontecimentos de saída utilizados na interacção. Para um sistema de 10 agentes e com 2 acontecimentos de interacção (e.g. posição e direcção actuais do agente) é necessário explicitar 90 ligações na especificação da cena.
- **Não heterogeneidade**. O VRML não tem mecanismos para a determinação da origem de um acontecimento. Se a população de agentes for heterogénea (i.e. composta por diversos tipos de agentes) e se o comportamento depender do tipo do agente emissor, torna-se necessário criar um novo acontecimento que identifique o emissor. Tal facto faz crescer linearmente o número de ligações em função do número de tipos de agentes.

Apesar da simplicidade da especificação da cena e de programação do guião, conclui-se que este modelo apenas se revela adequado à representação de mundos compostos por agentes homogéneos e onde o número total de agentes e acontecimentos de interacção seja reduzido.

3. MODELO CENTRALIZADO

Os principais problemas do modelo não optimizado apresentado na secção anterior consistem no elevado número de ligações necessárias ao estabelecimento da rede de interacção entre os agentes e à dificuldade de suportar populações de agentes heterogéneos. Neste contexto, propomos um modelo centralizado que supera os proble-

mas referidos na secção anterior ao utilizar uma estrutura partilhada por todos os agentes o que permite eliminar as ligações explícitas entre os acontecimentos definidos nos guiões, facilitando assim a utilização de protocolos de interacção complexos (v. Figura 3).

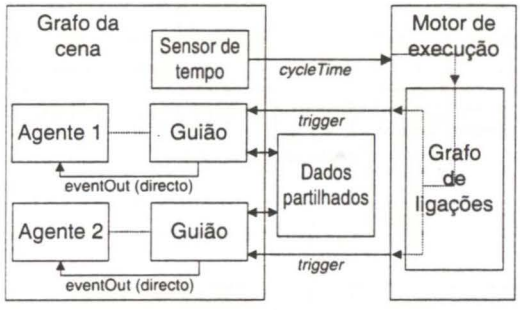

Figura 3: Encaminhamento de acontecimentos através do modelo centralizado.

O modelo centralizado permite obter a seguinte funcionalidade:

Simulação de comportamentos complexos e de populações de agentes heterogéneos.

Controlo do ritmo de resposta dos guiões através de estímulos temporais discretos.

Possibilidade de execução assíncrona, o que permite a utilização de guiões de elevada complexidade computacional sem comprometer a taxa de actualização do *browser* de VRML.

Integração com guiões já existentes.

Eliminação dos acontecimentos entre guiões. A comunicação passa a ser realizada pela estrutura partilhada sem recorrer a acontecimentos, libertando o motor de execução desta tarefa. Os únicos acontecimentos existentes serão entre o guião e a geometria a este associada.

Emissão de acontecimentos para a geometria de forma directa, sem recorrer à tabela de encaminhamento do motor de execução.

A arquitectura do modelo centralizado é constituída por quatro componentes básicos (v. Figura 4):

Registo centralizado (`AgentSharedData`). Contém a informação partilhada por todos os agentes, incluindo os dados necessários à interacção entre agentes.

Interface com o nó `Script` (`AgentScript`). Realiza o interface com o guião (nó Script) definido no grafo da cena em VRML. É responsável pela criação e eliminação do contexto do agente no registo centralizado e pelo encaminhamento dos acontecimentos de entrada para a respectivo módulo de processamento. Define também a informação básica de um agente (identificador e tipo).

Agente genérico (`GenericAgent`). Contém a informação comum (posição e orientação) a todos os agentes e define o mecanismo abstracto de processamento de acontecimentos temporais.

Agente (`AgentSpecialization`). Especialização de um agente genérico numa classe de agentes homogéneos. Múltiplas especializações permitem a definição de comunidades heterogéneas de agentes. Inclui os dados específicos ao estado interno do agente e a semântica associada ao processamento de acontecimentos por forma a traduzir o seu comportamento.

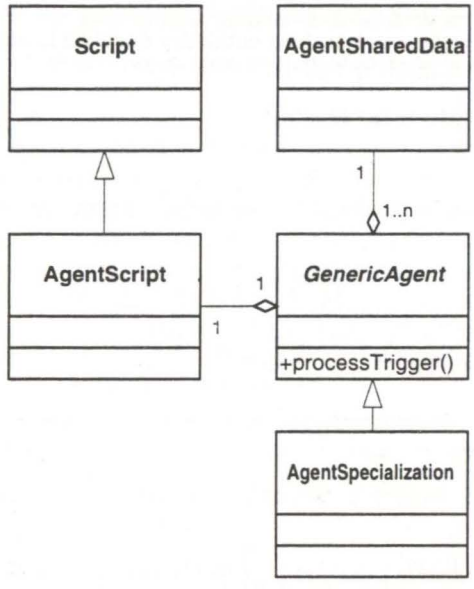

Figura 4: Modelo centralizado (diagrama de classes).

Segundo o modelo centralizado, o mesmo exemplo apresentado anteriormente (três agentes coordenados por um temporizador) seria estruturado da seguinte forma:

```
PROTO AgentProto [
  eventIn SFTime trigger
]
{
  DEF BODY Transform { geometry }
  DEF RULES Script {
    eventIn SFTime  trigger IS trigger
    url "script"
  }
}
DEF TIMER  TimeSensor { … }
DEF Agent1 AgentProto { … }
DEF Agent2 AgentProto { … }
DEF Agent3 AgentProto { … }

ROUTE TIMER.cycleTime TO Agent1.trigger
ROUTE TIMER.cycleTime TO Agent2.trigger
ROUTE TIMER.cycleTime TO Agent3.trigger
```

A principal vantagem deste modelo reside na separação completa da definição do comportamento dos agentes da definição da cena em VRML. Assim, todos os acontecimentos interactivos deixam de estar explícitos no VRML e passam a fazer parte do guião de cada agente. Este aproximação permite criar mundos de elevada dimensão e com padrões de interacção complexos sem o comprometimento da complexidade da cena. Este modelo apresenta um desempenho superior ao modelo não optimizado. Isto fica-se a dever a uma menor carga de processamento do motor de execução, pois o número de acontecimentos produzidos é sempre menor do que no modelo não optimizado (v. secção 5).

4. MODELO DE REPRESENTAÇÃO EXTERNA

Os dois modelos até agora apresentados assentam no VRML para a especificação da geometria e comportamento de um agente, ficando a visualização e a coordenação da interacção entre os agentes centrada no motor de execução do VRML. Contudo, existe a possibilidade de transpor toda a dinâmica do sistema para uma aplicação independente e utilizar o motor de execução do VRML apenas para a visualização do mundo virtual através. A interacção entre as duas entidades é realizada através *External Authoring Interface* [Marrin97].

4.1 Arquitectura do Modelo

O modelo de representação externa permite associar os elementos de um sistema multi-agente sem representação gráfica a uma representação geométrica dentro de um grafo de uma cena em VRML. Consegue-se assim a separação completa entre o comportamento de um agente e a sua representação gráfica. Este modelo, que denominamos de modelo de representação externa, assenta nos seguintes componentes (v. Figura 5):

- **Gestor de agentes** (`AgentManager`). Controla a simulação.
- **Agentes** (`Agent`). Representam o comportamento dos agentes.
- **Pares VRML** (`VRMLPeer`). Encapsulam a geometria VRML para cada tipo de agente e actualizam, através de acontecimentos directos, a geometria presente no grafo da cena.
- **Geometria VRML** (`Agent Geometry`). Representação gráfica de um agente no grafo da cena. É introduzida dinamicamente na cena pelo VRMLPeer.
- **Applet** (`EAIApplet`). Responsável pela comunicação inicial com o *browser* de VRML e apresentar o interface gráfico de controlo da simulação.

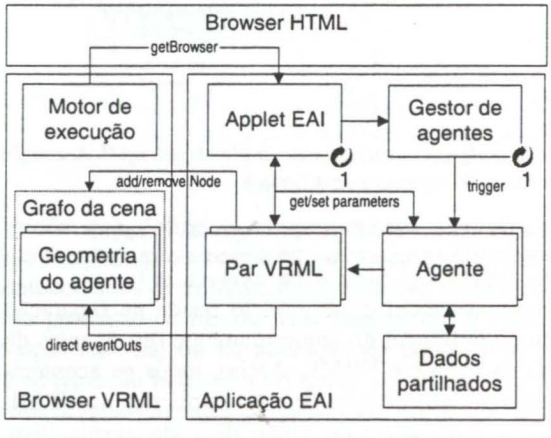

Figura 5: Arquitectura do modelo externo.

A Figura 6 descreve o diagrama de classes que serve de base ao modelo de representação externa.

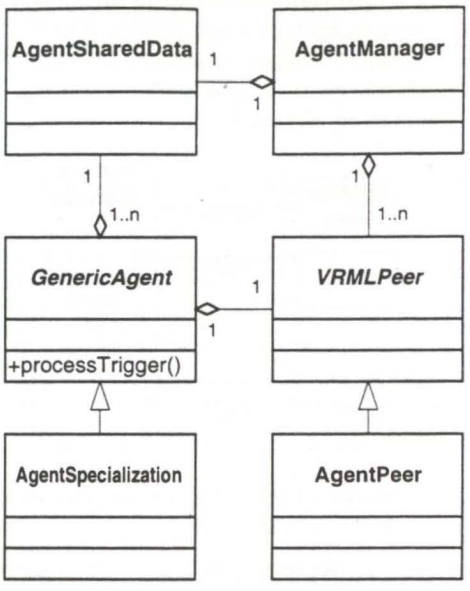

Figura 6: Modelo externo (diagrama de classes).

O gestor de agentes é responsável por sinalizar os agentes para actualizarem o seu estado interno (`processTrigger`) e por notificar o *browser* de VRML das alterações geométricas destes. Desta forma, o processamento do comportamento dos agentes e da geometria VRML associada encontra-se separado o que torna possível estabelecer ritmos de actualização distintos para cada acção.

4.2 Adição Dinâmica de Nós

Através da primitiva `createVrmlFromString` ou `createVrmlFromURL` é possível criar dinamicamente um conjunto de nós que podem ser inseridos no grafo da cena. Embora a semântica destas primitivas esteja bem definida, esta não é clara quanto à inclusão de guiões nos nós dinamicamente criados [Marrin97]. De facto, verifica-se que nenhum dos principais motores de execução de VRML suporta correctamente esta funcionalidade. O seguinte exemplo é suficiente para ilustrar o problema:

```
String vrml =
   "PROTO EmptyProto [ ] {
      Script { url "EmptyScript.class"  }
    } EmptyProto { }";
Node node[]=browser.createVrmlFromSring(vrml);
EventInMFNode set_addChildren =
    root.getEventIn("addChildren");
set_addChildren.setValue(node);
```

Este factor impede que sejam adicionados novos elementos ao sistema em tempo de execução, pois, quer o modelo típico não optimizado, quer o modelo centralizado, necessitam da utilização de guiões como meio de explicitar o comportamento dos agentes. Contudo, o modelo de representação externa contorna este problema, dado que permite associar comportamento a um nó geométrico sem recorrer ao mecanismo de guiões.

5. ESTUDO COMPARATIVO

Nesta secção avaliamos os três modelos propostos em função do número de acontecimentos envolvidos na comunicação entre os agentes e entre os agentes e a geometria VRML. Note-se que o desempenho global da cena depende de outras variáveis, como a complexidade da

geometria e a complexidade do guião. No entanto, estas não serão consideradas pois são comuns a qualquer modelo de representação. O sistema a avaliar é caracterizado pelas seguintes variáveis:

D número total de agentes

E_{sg} número de acontecimentos entre o guião e a geometria do agente

E_{sin} número de acontecimentos de entrada no guião

E_{sout} número de acontecimentos de saída do guião

C total de ligações ($C = C_{vrml} + C_{script}$)

C_{vrml} total de ligações explícitas

C_{scrip}: total de ligações directas a acontecimentos através do guião ou EAI

Modelo não optimizado

$$C = C_{vrml} = \frac{D(D-1)}{2} \times (E_{sin} + E_{sout}) + D \times E_{sg} + D$$

$$C_{script} = 0$$

Modelo centralizado

$$C_{vrml} = D \times E_{sg} + D$$

$$C_{script} = D \times E_{sout}$$

$$C = D \times (E_{sout} + E_{sg} + 1)$$

Modelo de representação externa

$$C_{vrml} = 0$$

$$C = C_{script} = D \times E_{sg}$$

A tabela abaixo apresenta os valores utilizados para a análise dos modelos em função da dimensão da população D. Os resultados da análise estão apresentados nas Figuras 7 e 8.

Propriedade	Valor	Significado
E_{sin}, E_{sout}	2	Posição e orientação (guião)
E_{sg}	2	Posição e orientação (geometria)

Tabela 1: Parâmetros de um agente.

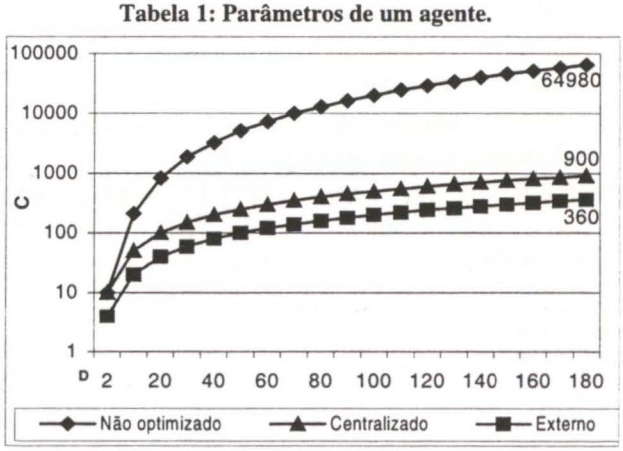

Figura 7: Ligações em função do número de agentes.

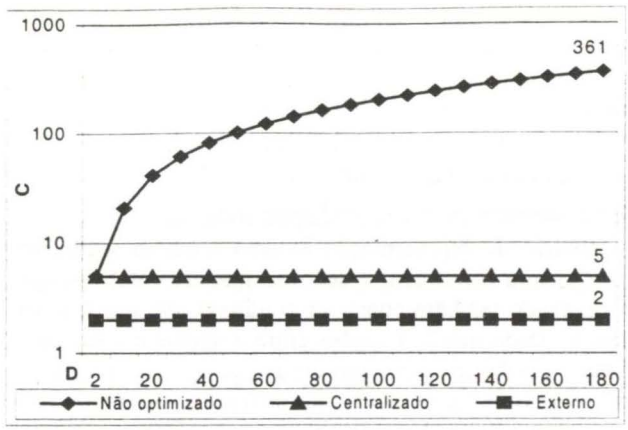

Figura 8: Ligações por agente em função do número total de agentes.

5.1 Discussão dos Resultados

Nesta secção apresentamos as conclusões retiradas dos resultados relativos aos três modelos propostos.

5.1.1 Modelo não optimizado

No modelo não optimizado, o número total de ligações aumenta quadraticamente com o número de agentes, i.e., $C_{total} \propto D^2$, enquanto o número de ligações de um agente individual cresce linearmente com o total de agentes, i.e., $C_{agente} \propto D$. A primeira observação limita o desempenho global do motor de execução e, consequentemente, a frequência de actualização, pois o número de acontecimentos necessários para a comunicação entre os agentes diminui o tempo útil para o processamento dos restantes acontecimentos. Este factor limita, por exemplo, a realização concorrente de animações e do processamento de um guião complexo, pois a taxa de amostragem dos acontecimentos do temporizador decresce com o número de acontecimentos que o motor de execução processa. A segunda observação refere-se ao aumento linear do número de ligações por agente. Este factor dificulta a descrição da cena dado que é necessário alterar esta sempre que um novo agente é introduzido. Adicionalmente, a especificação das ligações de um agente é tão mais complexa quanto o número total de agentes considerados.

5.1.2 Modelo centralizado

O número total de ligações no modelo centralizado apresenta um crescimento linear, pois toda a comunicação entre agentes é feita internamente, isto é, sem recorrer ao motor de execução. Este factor permite que a taxa de processamento de acontecimentos não diminua em função do número de acontecimentos. É assim possível dissimular o tempo de processamento do comportamento de um agente através da animação contínua da geometria deste. Neste modelo, o número de ligações por agente é constante e apenas inclui os acontecimentos para actualização da geometria. No entanto, através da utilização de protótipos é possível encapsular estas ligações (v. secção 8.4).

Geralmente são suficientes duas ligações para o controlo da geometria através de um guião, correspondendo uma à actualização da translação e outra à da rotação. Note-se que uma sequência de múltiplas rotações pode ser ex-

pressa numa única operação através da sua conversão intermédia para quaterniões [Heckbert94]. É assim possível representar a sequência das rotações na forma utilizada pelos nós de transformação VRML, isto é, $\{c_x, c_y, c_z, \alpha\}$, onde c_i representa o valor da componente no eixo i e α o ângulo da rotação combinada.

5.1.3 Modelo de representação externa

O modelo de representação externa também segue um crescimento linear em relação ao número total de ligações, sendo também constante o número de ligações por agente. Dado que as ligações entre o agente e a sua geometria são directas, o número total de ligações será sempre inferior à apresentada nos restantes modelos. Uma vantagem adicional reside na separação entre a temporização associada à visualização e ao processamento do comportamento dos agentes. Em cenários onde o número de agentes é elevado ou onde o comportamento é complexo, o tempo associado à gestão dos agentes pode comprometer a frequência de actualização do motor de visualização. Com o modelo de representação externa é possível privilegiar a simulação ou a actualização da geometria através da definição de prioridades.

- **Prioridade na simulação do comportamento.** Definição de uma elevada frequência de actualização dos agentes, em detrimento da frequência de actualização do motor de execução do VRML.
- **Prioridade na actualização geométrica.** Definição de uma baixa frequência de actualização do comportamento dos agentes por forma a reduzir o processamento global, o que permite aumentar a taxa de actualização do VRML. Neste caso, o baixo ritmo de alteração geométrica dos agentes pode ser compensado através da animação dos mesmos a partir do VRML.
- **Prioridade dinâmica.** A frequência de actualização dos agentes depende do ritmo instantâneo de actualização do motor de execução. Dado que uma aplicação pode determinar a qualquer instante a frequência de actualização do motor de execução, pode determinar-se em cada ciclo o tempo de espera óptimo entre a actualização do estado dos agentes.

5.1.4 Comentários finais

O modelo de representação externa permite maior liberdade na concepção de um sistema multi-agente heterogéneo. No entanto, a maior complexidade na definição dos agentes e a necessidade de utilizar uma aplicação adicional de controlo levam à sua escolha para a visualização de sistemas complexos ou onde a visualização e alteração dos parâmetros não dependa apenas do VRML. Como exemplo, pode referir-se um sistema multi-agente que utilize agentes remotos e permita configura-los dinamicamente.

O modelo centralizado é o mais equilibrado, pois permite a realização de simulações complexas, com elevado grau de interacção entre agentes, sem comprometer o desempenho do motor de execução. Através da utilização de protótipos torna-se simples encapsular populações heterogéneas de agentes. Por fim, e dado que este modelo não utiliza aplicações externas de controlo, revela-se o mais adequado sempre que a simulação seja auto-contida.

A utilização do modelo típico apenas se justifica em simulações de pequena dimensão e simplicidade, devido à explosão do número de acontecimentos a processar. Algumas ferramentas de edição de VRML permitem gerar as ligações entre os diversos agentes de forma automática ou semi-automática o que simplifica esta tarefa.

6. CONCLUSÕES

No estudo realizado, e cujos resultados aqui apresentámos, propusemos uma forma de representação de agentes baseada directamente no modelo de acontecimentos do VRML. Dadas as desvantagens desta aproximação, introduzimos duas alternativas a este modelo: o modelo centralizado e o modelo de representação externa. Verificámos que estas alternativas, com cenários de aplicação distintos, permitem uma representação simples e eficiente de sistemas compostos por múltiplos agentes, possivelmente heterogéneos, e com uma separação entre a definição da cena em VRML e a definição do comportamento dos agentes. Adicionalmente, estas aproximações tornam possível especificar comportamentos baseados na interacção entre os objectos da cena, estendendo assim a interacção destes objectos com o avatar, o que permite não só aumentar o realismo de uma cena mas também alargar a utilização do VRML à simulação e visualização de sistemas complexos.

7. BIBLIOGRAFIA

[Carey97] R. Carey, G. Bell. "The Annotated VRML 2.0 ReferenceManual". Addison Wesley Longman, 1997.

[Coplien95] J. Coplien, D. Schmidt (editors). "Patern Languages of Program Design", volume 1. Addison Wesley Publishing Company, 1995.

[Heckbert94] P. Heckbert (editor). "Graphics Gems IV". Academic Press, 1994.

[Marrin97] C. Marrin. "External Authoring Interface Reference". Silicon Graphics, 1997.

[VRML97] ISO/IEC DIS 14772-1. "VRML 97 Specification". http://www.vrml.org/Specifications/VRML97, 1997.

8. APÊNDICE

8.1 Interface do Modelo Centralizado

8.1.1 AgentScript.java

```
public class AgentScript extends Script {
    // returns the browser associated with this script node
    public static final Browser    getCurrentBrowser();

    // returns the shared registry data
    public static final SharedData getSharedData();

    // called before the scene graph is displayed and any event triggered. Registers the current
    // agent into the shared structure
    public final void              initialize();

    // called immediatly after the script node is removed from the scene graoph or the browser is
    // closed. Removes the current agent from the shared structure
    public final void              shutdown();

    // returns the agent instance associated with this script
    public GenericAgent            getAgent();

    // returns the registred agent identified by the pair (type, index)
    public static GenericAgent     getAgent(int type, int index);

    // returns the total number of agents of type type
    public static int              getAgentCount(int type);

    // called whenever one or more eventIns are routed to this script
    public final void              processEvents(int count, Event events[]);
}
```

8.1.2 SharedData.java

```
public class SharedData {
    // makes this agent visible to all other registered agents
    void registerAgent(GenericAgent agent);

    // returns a given agent identified by type and index
    GenericAgent                   getAgent(int type, int index);

    // returns the total number of agents of a given type
    int                            getAgentCount(int type);
}
```

8.1.3 GenericAgent.java

```
public abstract class GenericAgent {
    // Constructor. Automatically called by the AgentScript.
    GenericAgent(AgentScript script);

    // get the agent's type and id
    int                            getType();
    int                            getId();

    // called immediately after the generic agent is constructed and before any event is sent to
    // this node. id is the unique identifier of this agent.
    void                           initialize(int id);

    // called whenever a timer tick is sent to the script.
    void                           processTriggerEvt(Event event);

    // update the agent's VRML geometry
    protected void                 rotate();
    protected void                 move();

    // current position and orientation along each of the three major axis.
    protected SFVec3f              position;
    protected SFRotation           orientationX,
                                   orientationY,
                                   orientationZ;
}
```

8.2 Interface Externo de um Agente Genérico (AgentProto)

```
EXTERNPROTO AgentProto [
    eventIn       SFTime       trigger                    # timer tick
    exposedField  SFString     description                # viewpoint description
    field         SFInt32      type                       # the agent's type
    field         MFNode       geometry                   # the geometry nodes
    field         SFVec3f      iposition                  # initial position
]
"AgentProto.wrl"
```

8.3 Definição do Protótipo de um Agente Genérico (AgentProto.wrl)

```
PROTO GenericAgentProto [
    field         SFInt32      type                       # the agent's type
    field         SFVec3f      iposition      0 0 0       # initial position
    field         SFString     description    "Agent"     # viewpoint description
    eventIn       SFTime       trigger                    # timer tick
    eventIn       SFVec3f      translation                # new position
    eventIn       SFRotation   rotation                   # new rotation X, Y, Z
    exposedField  SFInt32      currentGeometry -1         # the selected geometry
    exposedfield  MFNode       geometry       []          # the set of geometry nodes
]
{
  # start prototype
  DEF TRANSLATION Transform {
    translation IS translation
    children DEF ROT Transform {
      rotation IS rotation
      children [
        Viewpoint {
          description IS description
        }
        DEF GEOMETRY_CHOICE Switch {
          whichChoice IS currentGeometry
          choice      IS geometry
        }
      ]
    }
  }

  DEF BASE_SCRIPT Script {
    field         SFInt32      type           IS type
    field         SFVec3f      iposition      IS iposition
    eventIn       SFTime       trigger        IS trigger
    eventOut      SFVec3f      position
    eventOut      SFRotation   rotation
    eventOut      SFInt32      currentGeometry
  }
  url "AgentScript.class"

  ROUTE BASE_SCRIPT.currentGeometry     TO GEOMETRY_CHOICE.set_whichChoice
  ROUTE BASE_SCRIPT.rotation            TO ROT_X.set_rotation
  ROUTE BASE_SCRIPT.position            TO TRANSLATION.set_translation
  # end prototype
}
```

8.4 Exemplo da Utilização da Plataforma do Modelo Centralizado

Mostra-se de seguida a definição de duas classes de agentes distintos (AgentType0_Proto e AgentType1_Proto) e a sua utilização num mundo virtual (World.wrl).

8.4.1 Definição do agente tipo 0 (AgentType0_Proto.wrl)

```
# include the generic agent prototype
EXTERNPROTO AgentProto [
     eventIn        SFTime          trigger                    # timer tick
     exposedField   SFString        description                # viewpoint description
     field          SFInt32         type                       # the agent's type
     field          MFNode          geometry                   # the geometry nodes
     field          SFVec3f         iposition                  # initial position
]
"AgentProto.wrl"

# instantiate the generic prototype for the agent type 0
PROTO AgentType0Proto [
   field        SFVec3f           iposition      0 0 0
   field        SFString          description    "agent type 0"
   eventIn      SFTime            trigger
]
{
   Transform {
      children AgentProto {
         geometry [
            Shape     { geometry Cone { height 2   } }          # Example of primary shape
            Shape     { geometry Box  { size 1 2 1 } }          # Example of secondary shape
            WorldInfo { }                                       # no shape
         ]
         type        0                                          # the type of this agent
        description IS description
        iposition   IS iposition
        trigger     IS trigger
     }
   }
}
```

8.4.2 Definição do agente tipo 1 (AgentType1_Proto.wrl)

```
# include the generic agent prototype
EXTERNPROTO AgentProto [
     eventIn        SFTime          trigger                    # timer tick
     exposedField   SFString        description                # viewpoint description
     field          SFInt32         type                       # the agent's type
     field          MFNode          geometry                   # the geometry nodes
     field          SFVec3f         iposition                  # initial position
]
"AgentProto.wrl"

# instantiate the generic prototype for the agent type 1
PROTO AgentType1Proto [
   field        SFVec3f           iposition      0 0 0
   field        SFString          description    "agent type 1"
   eventIn      SFTime            trigger
]
{
   Transform {
      children AgentProto {
         geometry [
            Shape     { geometry Sphere   { radius 1 } }        # Example of primary shape
            Shape     { geometry Cylinder { }          }        # Example of secondary shape
            WorldInfo { }                                       # no shape
         ]
        type        1                                           # the type of this agent
        description IS description
        iposition   IS iposition
        trigger     IS trigger
     }
   }
}
```

8.4.3 Instanciação dos agentes (World.wrl)

```
EXTERNPROTO Agent0 [
     eventIn      SFTime    trigger
     field        SFVec3f   iposition
     field        SFString  description
]
"AgentType0_Proto.wrl"

EXTERNPROTO Agent1 [
     eventIn      SFTime    trigger
     field        SFVec3f   iposition
     field        SFString  description
]
"AgentType1_Proto.wrl"

DEF AGENT0_1 Agent0 { iposition 10 10 10 description "foo 1" }
DEF AGENT0_2 Agent0 { iposition 20 20 20 description "foo 2" }
DEF AGENT0_3 Agent0 { iposition 30 30 30 description "foo 3" }
DEF AGENT0_4 Agent0 { iposition 40 40 40 description "foo 4" }

DEF AGENT1_1 Agent0 { iposition 15 15 15 description "bar 1" }
DEF AGENT1_2 Agent0 { iposition 25 25 25 description "bar 2" }
DEF AGENT1_3 Agent0 { iposition 35 35 35 description "bar 3" }
DEF AGENT1_4 Agent0 { iposition 45 45 45 description "bar 4" }

DEF TIMER TimeSensor {
   cycleInterval    0.05    # update frequency - 20 cycles per second
   loop             TRUE
}

ROUTE TIMER.cycleTime TO AGENT0_1.trigger
ROUTE TIMER.cycleTime TO AGENT0_2.trigger
ROUTE TIMER.cycleTime TO AGENT0_3.trigger
ROUTE TIMER.cycleTime TO AGENT0_4.trigger

ROUTE TIMER.cycleTime TO AGENT1_1.trigger
ROUTE TIMER.cycleTime TO AGENT1_2.trigger
ROUTE TIMER.cycleTime TO AGENT1_3.trigger
ROUTE TIMER.cycleTime TO AGENT1_4.trigger
```