Integração do Suporte à Visualização num Modelo de Sistemas de Partículas

Manuel Próspero dos Santos
Departamento de Informática
FCT/UNL, Quinta da Torre
2825-114 Caparica
ps@di.fct.unl.pt

Fernando Pedro Birra
Departamento de Informática
FCT/UNL, Quinta da Torre
2825-114 Caparica
fpb@di.fct.unl.pt

Renata Piotrowska
Computer Science Department
Faculty of CS and Management
Technical Univ. in Wroclaw, Poland

Sumário

É apresentado um sistema de modelação e síntese com base em partículas, destinado fundamentalmente à visualização de dados científicos e simulação. O modelo conceptual é criado a partir da existência de quatro componentes fundamentais, que são interligados de modo original. O suporte às técnicas de visualização é integrado no próprio modelo por uma extensão coerente do mesmo, que dá origem a uma nova arquitectura do sistema. Os componentes, implementados com JavaBeans, permitem ainda a programação visual. A comunicação termina com um exemplo de visualização da simulação de um fenómeno natural, utilizando-se para tal o modelo integrado.

Palavras-chave

Visualização de dados, sistemas de partículas, simulação, reutilização de componentes, JavaBeans, modelação de fenómenos naturais.

1. INTRODUÇÃO

Os sistemas de partículas têm vindo a ser usados no desenvolvimento de novas técnicas de visualização de dados científicos, particularmente quando estes representam um campo vectorial definido no espaço tridimensional [Andrade93]. Uma técnica experimental muito comum em Mecânica dos Fluidos consiste em largar tinta ou partículas de um material adequado e observar a sua evolução espacial ao longo do tempo. A simulação por computador, no entanto, permite visualizar de uma forma mais alargada as propriedades dum fluido muito difíceis (ou até impossíveis) de visualizar recorrendo exclusivamente à experiência física.

A utilização de um sistema de partículas como base de um ambiente de visualização de dados foi introduzida por Pang [Pang94] numa técnica que o autor denominou *spray rendering*. A metáfora é o uso de latas de tinta em *spray* para pintar, e assim visualizar, as propriedades dos conjuntos de dados antes considerados invisíveis. Dependendo do tipo de tinta, diferentes características dos dados poderão ser evidenciadas. Trata-se, portanto, duma combinação dos sistemas de partículas usuais com as técnicas de animação de comportamentos.

Nos trabalhos de investigação do grupo de Computação Gráfica da FCT/UNL, o modelo de Pang serviu de base para o estudo, concepção e desenvolvimento dum sistema de partículas de forma a ser adequado, não só à visualização de dados, como também à simulação de diversos fenómenos naturais onde a utilização de partículas se acomoda particularmente bem [Birra97].

Assim, foi o SiPaViS (Sistema de Partículas para Visualização e Simulação) concebido de modo a permitir a utilização consistente de sistemas de partículas numa vasta gama de aplicações com necessidades de visualização de dados, incluindo especialmente aplicações onde a simulação é baseada na modelação física de um determinado fenómeno [Birra98b]. O modelo conceptual do SiPaViS, sumariamente descrito em [Birra98a, Birra99], tem, entre outros, os seguintes objectivos:

- flexibilidade – permitindo a construção de diversas técnicas envolvendo interacções entre os objectos do sistema.
- extensibilidade – permitindo a reutilização de componentes em diversas situações, assim como a adição de novos componentes.
- suporte embebido para visualização de dados e simulação – permitindo a prototipagem rápida de algoritmos e reduzindo a quantidade de código necessário para a sua realização. No limite, poderá proceder-se apenas à reutilização de componentes anteriormente desenvolvidos modificando o modo como se interligam.

Apesar do sistema desenvolvido ter sido utilizado com sucesso na recriação de técnicas de visualização conhecidas, tais como o algoritmo de *ray-casting* (para campos escalares) ou o seguimento de partículas e técnicas de fitas (para campos vectoriais), constatou-se a necessidade de repensar o modelo no sentido de se

resolver um conjunto de problemas práticos importantes. De entre estes, ressalta-se o facto do modelo inicial pecar pela falta de suporte à especificação de câmaras, visores, iluminação, técnicas de sombreamento e outros parâmetros globais que controlam a aparência final das imagens de síntese, sendo para isso utilizados pelo programador métodos ad hoc. Por outro lado, também não era fácil tirar partido da capacidade de aceleração disponível nas actuais placas gráficas 3D da plataforma de hardware. Por último, também se pretendeu incorporar no novo modelo, ainda que parcialmente, o paradigma de programação visual. Nesta comunicação, além duma apresentação do modelo inicial, foca-se a alteração feita que, tendo em conta os objectivos acabados de referir, o melhorou e tornou muito mais abrangente.

2. MODELO DO SISTEMA DE PARTÍCULAS

O sistema de partículas SiPaViS foi concebido para possuir quatro tipos de componentes base e qualquer algoritmo de visualização ou simulação deveria idealizar-se criando e ligando entre si objectos desses tipos:

- Partícula
- Fonte de partículas
- Campo
- Interacção

As partículas são os elementos do sistema que permitem a visualização das propriedades do espaço circundante. Nalgumas técnicas de visualização poderão funcionar como sondas enviadas para o ambiente, procurando características específicas desse mesmo ambiente. Os dados por elas transmitidos são codificados numa representação visual recorrendo a cores, formas, transparências, posições e velocidades, entre outros atributos possíveis. O movimento destas sondas pode ser influenciado, tanto por propriedades do ambiente, como até de outras partículas.

As fontes de partículas, como o nome sugere, são os objectos que criam as partículas, controlando quando e como é que elas são libertadas no ambiente. Para além de existirem fontes com geometria pré-definida, tais como fontes circulares, esféricas e rectilíneas, é sempre possível definir outras formas que dependam das propriedades do meio ou de valores de atributos de partículas. Uma fonte de partículas é, na realidade, uma agregação de dois componentes: um componente temporal, que define o comportamento da fonte ao longo do tempo, e um componente espacial, que determina o local onde deverão ser criadas novas partículas.

Os problemas de visualização de dados científicos lidam principalmente com entidades físicas denominadas campos. Estes são, em geral, funções definidas no espaço tridimensional e cujo valor pode assumir qualquer dimensão. Pode dar-se, como exemplo, um campo escalar de temperaturas ou um campo vectorial de velocidades num fluido. Os campos poderão, temporalmente, ser caracterizados como estáticos ou dinâmicos, permitindo o sistema a coexistência de ambos os tipos.

Surge, finalmente, a interacção. A sua noção física é de extrema importância. Na verdade, os fenómenos físicos regem-se por interacções entre os intervenientes. Se se quiser um bom exemplo, bastará pensar nos fenómenos gravitacionais. Mas as interacções tanto podem representar forças aplicadas às partículas como agentes de alteração na sua aparência (cor, tamanho, transparência, etc.). De acordo com esta interpretação, as interacções são as entidades responsáveis por introduzir alterações no estado de cada partícula.

Tendo em conta que uma interacção não é mais do que uma relação, de aridade arbitrária, entre os elementos do seu domínio, uma característica básica e original do nosso modelo é precisamente a definição do domínio das interacções. Cada elemento desse domínio pode representar, quer um objecto particular do sistema, quer o conjunto de todos os objectos de um mesmo tipo (ou classe). Uma interacção é activada desde que estejam presentes no sistema todos os elementos do seu domínio. Uma vez mais se realça a semelhança com o mundo físico real recorrendo, de forma simplista, ao exemplo da interacção gravitacional: esta funciona pelo simples facto de estarem presentes, pelo menos, dois corpos com massa.

No modelo conceptual não se verifica a necessidade das partículas comunicarem directamente entre si ou com os campos. É da responsabilidade das interacções o estabelecimento de dependências entre os elementos do seu domínio. Assim, poderemos ter uma interacção que modifica a cor das partículas de um determinado tipo em função do valor de um campo para as posições das respectivas partículas, ou ter interacções mais simples, tais como atracções ou repulsões entre partículas.

Conceptualmente, todo o sistema foi idealizado com vista à utilização de componentes de software. A cada tipo de componente corresponde uma classe abstracta base oferecendo a funcionalidade apropriada.

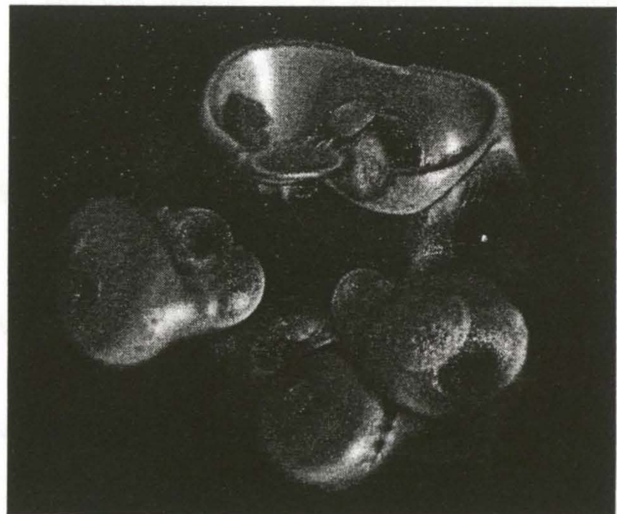

Figura 1: Superfície nodal numa função de onda para a órbita de um electrão.

O programador, recorrendo a mecanismos de herança, desenvolve classes derivadas onde se implementam os comportamentos específicos pretendidos. A especificação

de quais as classes a usar, que objectos se criarão e como eles se irão interligar é remetida para um guião, escrito de acordo com regras gramaticais que o programador terá que conhecer.

A Figura 1 mostra, como exemplo de aplicação do sistema SiPaViS, o resultado da técnica de visualização em volume por *ray-casting*, com dados científicos fornecidos pela University of North Carolina - Chapel Hill. Na mesma figura constata-se a utilização de fontes de iluminação que dão maior realismo tridimensional à imagem. No entanto, o modelo conceptual anteriormente descrito não apresenta qualquer suporte em questões de *rendering*. Por outro lado, o uso de guiões (não mostrados no exemplo), exigindo a aprendizagem de uma linguagem de especificação muito particular (descrita em [Birra97]), não é aliciante em termos da programação.

JavaBeans [Hamilton97] é uma tecnologia de componentes que permite aos programadores criar componentes de software reutilizáveis e que podem ser manipulados visualmente. Um *bean* pode ser encarado como um *building block* para a construção de aplicações. As principais vantagens da utilização desta tecnologia são resultado dos mecanismos oferecidos, tais como:

- manipulação visual no interior de *builders* – permitida pelos mecanismos de armazenamento persistente e de acesso a propriedades, capacidades de parametrização dos componentes e o suporte à introspecção.
- mecanismo de disseminação de eventos.

A utilização desta tecnologia adapta-se com naturalidade ao modelo conceptual do SiPaViS, permitindo reter, e até estender, a sua funcionalidade. O exemplo mais concreto de aumento dessa funcionalidade consiste na exploração do paradigma da programação visual, incomparavelmente superior à utilização convencional de bibliotecas e de guiões.

3. ARQUITECTURA DE SUPORTE À VISUALIZAÇÃO

Para, no próprio modelo, ter em linha de conta os aspectos de suporte à visualização e acesso à programação visual, referidos na secção anterior, redesenhou-se a arquitectura e implementou-se assim o actualmente denominado JavaSiPaViS. De acordo com a Figura 2, reconhece-se facilmente os componentes de visualização e simulação já antes mencionados (partículas, fontes de partículas, campos e interacções). Quanto aos novos componentes, ditos de visualização, são os seguintes:

- Luzes (Classe *Light*)
- Vistas (Classe *View*)
- Janelas (Classe *Frame*)
- Visores (Classe *Viewport*)
- Câmaras (Classe *Camera*)

Estes últimos apenas intervêm no processo de síntese da imagem, não tendo qualquer influência durante as fases de simulação. Por seu lado, os componentes do primeiro grupo, para além de serem os agentes de uma simulação, possuem ainda uma interface de síntese que lhes permite gerar a sua própria representação visual.

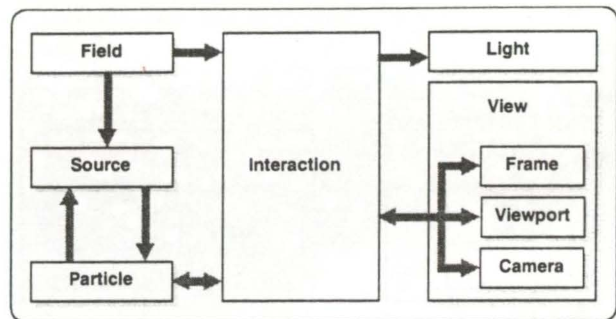

Figura 2: Arquitectura de Visualização do JavaSiPaViS.

A característica fundamental do modelo reside na coerente utilização das interacções, quer para manipular os componentes de simulação, quer para manipular os componentes de visualização. Do mesmo modo que se especificam interacções para mudar os atributos de partículas, podem-se especificar e/ou desenvolver interacções para efectuar alterações dinâmicas aos objectos de visualização, tais como as câmaras, as luzes ou os visores.

Com este mecanismo podemos especificar câmaras cujo centro de atenção incida, a título de ilustração, num grupo de partículas com a restrição de, em cada imagem, haver a necessidade de enquadrar todas as partículas. Outro exemplo reside na especificação de uma interacção que obrigue a câmara a seguir os movimentos de uma dada partícula ou de um grupo delas. Até mesmo as luzes poderão sofrer interacções que as obriguem a iluminar partes da cena com maior interesse.

Com o objectivo de simplificar a tarefa do programador, o JavaSiPaViS gere, de modo automático, todo o mecanismo de síntese das imagens. A arquitectura concebida segue a abordagem das linguagens orientadas por objectos: cada componente possui uma interface de síntese que poderá ser redefinida pelo programador de modo a sintetizar esse objecto. A implementação JavaSiPaViS [Piotrowska99] foi realizada em linguagem Java, como seria de esperar, e utiliza o OpenGL [Woo96] para efectuar a síntese das imagens, sendo sobejamente conhecidas todas as actuais vantagens de tal sistema gráfico.

Resume-se, seguidamente, o conjunto de aspectos mais significativos introduzidos pelos componentes de visualização.

3.1 Classe View

Cada componente desta classe permite estabelecer uma associação entre os triplos janela-visor-câmara. O utilizador pode criar várias vistas, com diferentes combinações de janelas, câmaras e visores. O grau de complexidade da especificação de uma simulação recorrendo a várias janelas é da mesma ordem de grandeza que no caso de se pretender apenas uma. É possível ter vários visores na mesma janela ou partilhar

câmaras entre janelas diferentes. Na Figura 3 apresenta-se um exemplo de uma simulação requerendo três vistas diferentes, com recurso a três visores, duas janelas e duas câmaras.

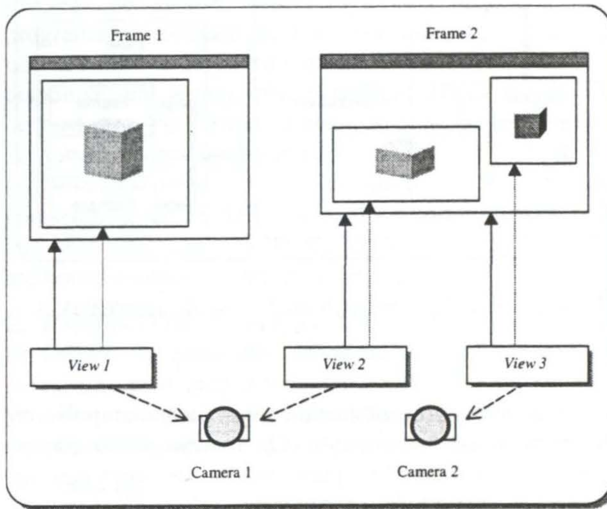

Figura 3: Diagrama para ilustrar possíveis combinações de componentes de visualização.

As vistas (*views*) desempenham ainda um papel de extrema importância no encadeamento temporal das chamadas aos métodos de síntese de cada objecto do sistema. Essas operações de síntese encontram-se divididas em três fases distintas, correspondendo a cada uma dois tipos de métodos: métodos estáticos que permitem factorizar operações comuns a todas as instâncias e métodos próprios dos objectos. Ei-las:

- **Inicialização** – No início da simulação, cada vista invoca os métodos de inicialização de síntese das classes de simulação. Esta será a altura ideal para se efectuarem operações tais como o carregamento de dados externos (texturas ou modelos geométricos). Nesta fase, os métodos estáticos da classe são invocados antes dos métodos das instâncias.
- **Síntese** – A vista invoca os métodos de síntese para cada um dos seus componentes. Numa primeira fase, é executado o código de síntese relativo às luzes, visores, câmaras e janelas. Este processo acabará por invocar os métodos de síntese dos componentes de simulação do sistema, tais como as partículas, os campos, as fontes e as interacções.
- **Conclusão** – No final da simulação, cada objecto terá a sua oportunidade de efectuar a libertação dos recursos por si usados para efeitos das operações de síntese. Posteriormente, cada classe poderá ainda libertar os recursos comuns a todas as suas instâncias.

3.2 Classes Frame e Viewport

Um objecto da classe *Frame* guarda e manipula o contexto de OpenGL para uma janela. Esta classe é derivada da classe *GL4Java.GLFrame* que, por sua vez, deriva de *java.awt.Frame*. A classe expõe algumas propriedades que poderão ser modificadas pelo programador ou editadas num *builder*, tais como: tipo de sombreamento, utilização de *Z-buffer* ou cor de fundo da janela. As dimensões de cada *frame* são apenas de leitura, mas poderão ser modificadas pelo utilizador, durante a simulação, fazendo o redimensionamento da janela. Os visores serão automaticamente redimensionados, por forma a manterem-se as mesmas proporções iniciais. Para tal efeito, cada *Frame* possui uma lista dos seus *Viewports*. O método de síntese da classe *Frame* refresca o conteúdo da respectiva janela, de acordo com o estado corrente da simulação, enviando a todos os componentes de simulação as respectivas mensagens de *rendering*.

Atributo	Descrição	Tipo	Valor por omissão
Left	Plano de recorte vertical esquerdo	double	-1.0
Right	Plano de recorte vertical direito	double	1.0
Top	Plano de recorte horizontal superior	double	1.0
Bottom	Plano de recorte horizontal inferior	double	-1.0
Znear	Distância ao plano de recorte anterior	double	1.5
Zfar	Distância ao plano de recorte posterior	double	20.0
PerspectiveType	Tipo de projecção	boolean	True (perspectiva)

Tabela 1: Atributos da classe *Camera* relacionados com a matriz de GL_PROJECTION.

Atributo	Descrição	Tipo	Valor por omissão
CameraPosition	Posição da câmara	Point3D	(0,0,5)
CenterPosition	Centro de interesse da cena	Point3D	(0,0,0)
UpVector	Vector de orientação	Point3D	(0,1,0)

Tabela 2: Atributos da classe *Camera* relacionados com a matriz de GL_MODELVIEW.

Por sua vez, os atributos expostos pela classe *Viewport* definem a área rectangular no interior da janela que define o visor. O método de síntese de cada *Viewport* apenas desencadeia uma chamada à função glViewport().

3.3 Classe Camera

A classe *Camera* foi desenhada tendo o objectivo de lidar com a matriz de projecção e a matriz de orientação da câmara. As propriedades relacionadas com a primeira matriz e expostas pela classe são as indicadas na Tabela 1. As outras propriedades, relativas à matriz de MODELVIEW do OpenGL e que definem a orientação da câmara, encontram-se enumeradas na Tabela 2.

3.4 Classe Light

A criação e a destruição dos objectos da classe *Light* está vedada aos programadores. O sistema cria inicialmente um conjunto destes que permanece único, pelo que as propriedades de cada luz são válidas para todas as vistas. Cada objecto da classe *Light* oferece uma interface semelhante à de OpenGL para manipulação das luzes, permitindo ajustar o seu tipo, cor, direcção e posição. Os comandos de iluminação relacionados com a cena e independentes de cada uma das luzes são implementados por intermédio de métodos estáticos da classe. Para conveniência do programador, os valores iniciais das propriedades dos objectos da classe *Light* coincidem com os valores por omissão em OpenGL.

O método de síntese desta classe é responsável por invocar todos os comandos necessários para especificar, ao sistema gráfico, as características de cada luz. Para, de forma dinâmica no decorrer de uma simulação, alterar os atributos de um objecto luz, é necessário definir uma interacção cujo domínio contenha um objecto deste tipo. No seu domínio, a interacção poderá conter outros objectos do sistema que poderão servir para determinar as alterações a efectuar à luz. Se, por exemplo, pretendermos que uma dada partícula se comporte como uma fonte de luz, a interacção partícula-luz perguntará à partícula, em cada instante, a sua posição e alterará a luz de forma correspondente.

4. EXEMPLO DE SIMULAÇÃO

Na simulação que se tomou para exemplo de aplicação, as partículas funcionam como flocos de neve. Estes são criados por uma fonte cujo comportamento temporal produz uma injecção de novas partículas a uma frequência constante. A criação dos flocos de neve occorre na parte superior da cena, fora do alcance da câmara (Figura 4). As posições iniciais desses flocos são distribuídas aleatoriamente numa porção de espaço limitada. Esta técnica causa a impressão de irregularidade natural que qualquer pessoa esperaria na geração dos flocos. Por outro lado, como se sabe, a queda dos flocos de neve é afectada pela força da gravidade, pelos ventos e pela resistência do ar ao movimento.

Cada floco de neve é um *bean* da classe *SnowParticleTex*. O método de síntese desta classe recorre ao uso do modo de *blending* do OpenGL: as partículas são quadriláteros com uma textura transparente em algumas zonas e são desenhadas no *frame buffer* usando, para tal, um modo de *blending* aditivo com o teste de profundidade desligado.

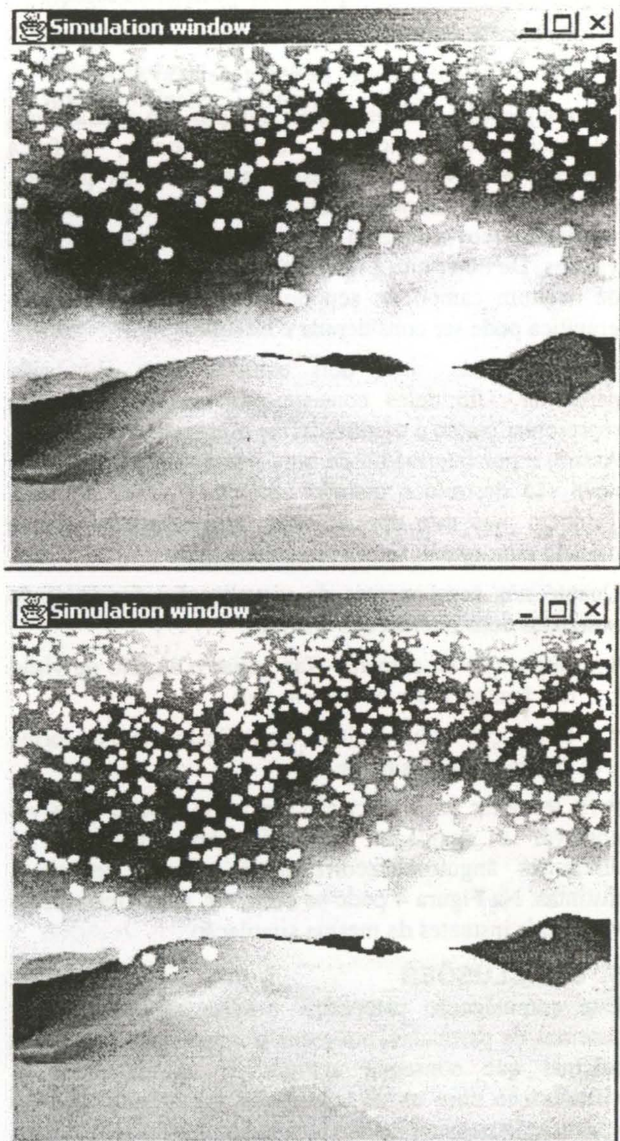

Figura 4: Simulação da queda de neve em dois instantes diferentes.

A síntese dos flocos de neve é efectuada em três fases distintas. No correspondente método estático da classe *SnowParticleTex* inicializa-se o sistema gráfico de forma a activar o modo de *blending* e a desactivar o teste de profundidade. De seguida, cada objecto (floco de neve) recebe uma mensagem para se desenhar no *frame buffer*. Finalmente, um último método estático é invocado para repor o estado do sistema gráfico.

No método estático de inicialização da síntese da classe *SnowParticleTex*, o conjunto de texturas disponíveis para os objectos da classe é carregado para memória. O método de inicialização de cada instância apenas selecciona qual a textura a usar para o respectivo floco de neve. O método estático de conclusão da síntese liberta todas as texturas alocadas pela classe. Deste modo, as texturas são partilhadas pelos diferentes objectos da classe.

Para recriar as trajectórias irregulares da queda dos flocos de neve, a simulação usa campos que estabelecem um conjunto de propriedades para cada posição do espaço:

- campo de ventos
- campo de coeficientes de resistência do ar

Em conjugação com estes dois campos são usadas duas interacções que calculam, respectivamente, a força exercida pelo vento e a resistência oferecida pelo ar. Cada um dos campos possui então uma média e uma variância que produzem efeitos oscilatórios nos seus valores. De notar que a interacção gravítica não necessita de nenhum campo no seu domínio, pois a aceleração gravítica pode ser considerada constante.

Existem, ainda, mais dois campos nesta simulação particular. Um deles consiste num *height field* para representar o solo e o outro define o aspecto a dar ao céu. Assim, e por intermédio de uma interacção, os flocos de neve são destruídos quando atingem o nível do solo. Também para este tipo de teste, coerentemente com o modelo conceptual, se usa uma interacção.

Quanto aos componentes de visualização, foram ainda utilizadas duas interacções distintas:

- Interacção de câmara – faz variar o ângulo de visão da câmara (*zoom in* e *zoom out*).
- Interacção de luz – altera, periodicamente e de forma ligeira, a cor das luzes.

Para terminar esta descrição, refira-se que é possível observar a simulação em diferentes janelas e de diferentes ângulos, recorrendo à criação de vistas distintas. Na Figura 4 pode-se observar, para uma mesma vista, dois instantes da mesma simulação.

5. CONCLUSÕES

Esta comunicação pretendeu mostrar, no âmbito dos sistemas de partículas, um modelo conceptual coerente e original que consegue agregar os componentes de visualização com os de simulação, permitindo ainda a reutilização de componentes e a programação visual. Para tal, na implementação usou-se JavaBeans. Existiriam, porém, soluções alternativas, delas se destacando, pela grande disseminação, a da tecnologia COM. Não pretendendo entrar em comparações exaustivas entre as duas abordagens, refira-se apenas que, após tentivas de construção de um protótipo eficaz, a tecnologia COM foi preterida pela falta de mecanismos de herança de implementação [Pinto99]. Como é sabido, COM apenas fornece herança de interfaces, o que não se adequa totalmente ao paradigma das linguagens orientadas por objectos.

Outro problema que se levantou foi o do acesso às funções de OpenGL a partir da linguagem Java. A escolha recaiu sobre o uso de GL4Java [Jausoft98], fundamentalmente pelas seguintes razões:

- Utilização da mesma convenção de nomes que a interface nativa.
- Suporte para as funções glu*.
- Diminuída penalização em termos de desempenho, pois é utilizado o mecanismo de acesso à interface nativa (Java Native Interface). Deste modo, GL4Java consiste apenas numa espécie de invólucro para as chamadas da biblioteca de OpenGL.

Resta ainda referir que, ao contrário do modelo adoptado por Pang [Pang94], a inteligência dos componentes de visualização ou simulação se encontra distribuída pelas diversas interacções e não centralizada em cada uma das partículas. Considerando como aplicação as simulações manifestando comportamentos emergentes (ou seja, resultantes dos comportamentos individuais das partículas) fica mais simples, deste modo, a partição de cada uma das fases dos comportamentos individuais por diferentes interacções.

6. AGRADECIMENTOS

O programa de intercâmbio SOCRATES/ERASMUS permitiu que o trabalho, referido neste artigo e suportado pelo projecto PRAXIS/P/EEI/10089/1998, resultasse numa tese de M.Sc. [Piotrowska99] sob orientação da FCT/UNL.

7. REFERÊNCIAS

[Andrade93] G. Andrade, M.J. Próspero. Avaliação de estratégias algorítmicas na visualização de campos de dados tridimensionais. *Actas do Quinto Encontro Português de Computação Gráfica*, pp. 1-16, GPCG — Eurographics Portuguese Chapter, Aveiro, Fevereiro 1993.

[Birra97] Fernando P. Birra. Sistema de partículas para visualização e simulação. Dissertação de Mestrado em Engenharia Informática, Universidade Nova de Lisboa, 1997.

[Birra98a] Fernando P. Birra, M. Próspero dos Santos. A Framework for Data Visualization based on Particle Systems. *The Sixth International Conference in Central Europe on Computer Graphics and Visualization'98 (WSCG'98)*, pp. 41-48, Plzen, Czech Republic, February 9-13, 1998.

[Birra98b] Fernando P. Birra, Manuel J. Próspero. SiPaViS – A Toolkit for Scientific Visualization and Simulation. *Proceedings of the 8th ICECGDG*, ISGG, pp. 480-484, Austin TX, USA, 1998.

[Birra99] Fernando P. Birra, Manuel J. Próspero. SiPaViS – A Toolkit for Scientific Visualization and Simulation. *Journal for Geometry and Graphics*, Volume 3 (1999), No. 1, pp. 47-55. Heldermann Verlag.

[Hamilton97] Graham Hamilton (Editor). JavaBeans™ API Specification, Version 1.01, Sun Microsystems, 1997.
http://java.sun.com/Beans/docs/specs.html

[Jausoft98] Jausoft. GL4Java documentation, Germany, 1998.
http://www.jausoft.com

[Pang94] A. Pang. Spray Rendering. *IEEE Computer Graphics and Applications*, 14, no. 5, pp. 57-63, 1994.

[Pinto99] Paulo Pinto. Sistemas de Partículas. Relatório de Projecto de Licenciatura, DI/FCT/UNL, 1999.

[Piotrowska99] Renata Piotrowska. A Component Object Model for Data Visualization Using a Particle System. MSc. Dissertation, Technical University in Wroclaw and FCT/Universidade Nova de Lisboa, 1999.

[Woo96] M.Woo, J.Neider, T.Davis. *The OpenGL Programming Guide, Second Edition*. Addison-Wesley, Reading, MA, USA, 1996.