**DÉBORA PEREIRA COURA**

# PRODUZINDO ANIMAÇÕES ATRAVÉS DA PROGRAMAÇÃO POR DEMONSTRAÇÃO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2006

**Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV**

T
C858p
2006

Coura, Débora Pereira, 1976-
Produzindo animações através da programação por demonstração / Débora Pereira Coura. – Viçosa : UFV, 2006.
xii, 105f. : il. ; 29cm.

Orientador: Vladimir Oliveira Di Iorio.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 102-105.

1. Linguagem de programação (Computador). 2. Jogos - Processamento de dados. 3. Programação visual (Computador). 4. Animação por computador. 5. Programação (Computadores). 6. Software educacionais - Jogos por computador. I. Universidade Federal de Viçosa. II.Título.

CDD 22.ed. 005.13

**DÉBORA PEREIRA COURA**

# PRODUZINDO ANIMAÇÕES ATRAVÉS DA PROGRAMAÇÃO POR DEMONSTRAÇÃO

**Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.**

APROVADA: 21 de dezembro de 2006

Prof. Alcione de Paiva Oliveira
(Co-orientador)

Prof. Marcos Vinícius Alvim Andrade
(Co-orientador)

Prof. Mauro Nacif Rocha

Prof. Roberto da Silva Bigonha

Prof. Vladimir Oliveira Di Iorio
(Orientador)

Antes de sair para a execução de suas tarefas, lembre-se de que é preciso abençoar a vida para que a vida nos abençoe.

*Chico Xavier*

# Agradecimento

Agora, que concluo mais uma etapa da minha vida, agradeço a Deus por ter me dado a força e a coragem necessárias para seguir em frente. Agradeço por colocar no meu caminho as pessoas certas, na hora certa. Através delas, Você, me dizia: “Vá em frente! Tudo se resolverá da melhor maneira possível!”. Agradecendo a Você, agradeço a todos que participaram e ainda participam da minha vida e dessa conquista.

Agradeço aos meus pais, José Raimundo e Vera. Sem vocês nada disso seria possível. Obrigada pelo exemplo de vida e de força. Obrigada pela paciência, pelo apoio, pelo incentivo e pela companhia durante tantas viagens. Obrigada por acreditarem nos meus sonhos e por me ajudarem a torná-los realidade. Obrigada por cuidarem, por mim, da pessoinha mais importante da minha vida. Sem vocês, com certeza, toda essa caminhada seria muito mais difícil. Amo muito vocês dois! Essa conquista é nossa, vocês são os mestres da minha vida!!! Obrigada pelo amor incondicional. Obrigada!!!!

Agradeço às minhas irmãs Rachel e Dalila! Agradeço pela paciência e por terem resolvido tantos problemas por mim. Vocês foram minhas mãos, minhas pernas, meus olhos quando o estudo não me deixava ter tempo. Durante esse período sobraram muitas tarefas e muitas vezes faltaram presenças, cuidados, carinhos... Obrigada!!!!

Agradeço ao meu amor, Romério! Nossa história começou nessa cidade, durante o curso de graduação. Hoje formamos uma família e terminamos mais uma etapa de nossas vidas. Obrigada pelo carinho, pela preocupação, pela paciência. Sei que foram muitas noites que passei em claro, muitas tarefas que você fez sozinho, muitas situações que não pudemos compartilhar e aproveitar juntos. Mas o seu apoio e a sua certeza de que eu conseguiria me deram forças para superar as dificuldades e chegar até aqui. Te amo muito!! Obrigada!!!!

Agradeço ao mestrado por ter realizado o meu sonho se ser mãe. Pedro, meu menininho, a alegria de ter você se confundia com a responsabilidade do estudo. E dessa forma, você foi participando de todos os momentos dessa conquista. Assitiu aulas, dividiu o colo com computador, papéis e livros. Fez inúmeras viagens para Viçosa, antes e depois de nascer. Foi o primeiro filho deste mestrado. E se comportou muito bem! Tudo poderia ter sido mais difícil, mas o seu sorriso, a sua tranquilidade, o seu bom humor, o seu carinho me davam a força necessária para te colocar no berço, ligar o computador e estudar. Obrigada por tantos momentos de descontração e de imensa felicidade. Obrigada pelo seu jeito doce de ficar na casa da vovó e do vovô. Assim, meu coração doía menos e eu podia acreditar que estava fazendo a coisa certa. Agora teremos muito tempo para brincar e para dormir juntos! Obrigada!!!!

Agradeço aos professores que tanto me ensinaram academicamente e pessoalmente. Obrigada pela paciência e pela colaboração de vocês. Obrigada Marquinhos!! Obrigada Alcione!! No meio de tantas situações atípicas vocês me deram tranquilidade para conseguir chegar até aqui. Obrigada a todos que contribuiram para a conclusão deste trabalho. Agradeço aos funcionários, especialmente, o Altino! Obrigada pela paciência e pela força. Obrigada!!!!

Agradeço, especialmente, ao meu orientador Vladimir! A você devo a conclusão deste trabalho. A sua calma, a sua ajuda, o seu apoio e a sua disponibilidade foram fundamentais. As dúvidas eram imensas, as incertezas, enormes, mas você esteve sempre presente. Obrigada pelas orientações!!! Tenha certeza que aprendi muito como pessoa, como estudante, como profissional. Sairei desse curso levando muitas lições valiosas. Obrigada!!!!

Agradeço aos amigos que estiveram ao meu lado! Agradeço, em especial, ao Vinícius e ao Mauro, que tanto me escutaram, me ajudaram e que tanto torceram para que tudo desse certo. Obrigada pelas caronas e pela preocupação de vocês.

Agradeço ao UnilesteMG pelo apoio financeiro. Agradeço ao Machado, coordenador do curso de computação, pelo apoio, pela compreensão e pela torcida.

Agradeço à Universidade Federal de Viçosa. Pela segunda vez uma parte importante da minha vida aconteceu aqui. Obrigada!!!!

# Biografia

Débora Pereira Coura, filha de José Raimundo de Castro Coura e Vera Lúcia Pereira Coura, brasileira nascida em 26 de junho de 1976 no município de Timóteo, no Estado de Minas Gerais.

No ano de 1994, após concluir o curso científico na cidade de Governador Valadares, ingressou no curso de graduação de "Bacharelado em Informática" na Universidade Federal de Viçosa, onde se graduou no ano 1997. Em 1998 trabalhou como Analista de Sistema em uma empresa na cidade de Timóteo. Em 1999 trabalhou em escola de curso técnico como professora na cidade de Ipatinga. Em 2000 começou a trabalhar em uma escola de curso técnico de Informática em Coronel Fabriciano. E iniciou um curso de pós-graduação oferecido pela Universidade Federal de Minas Gerais, terminando-o em 2001. Em 2001, após concluir a pós-graduação, começou a lecionar para o curso superior em uma faculdade da cidade de Coronel Fabriciano. Em 2004 foi aprovada na seleção do mestrado do Departamento de Informática - DPI, onde cursou o mestrado em Ciência da Computação na Universidade Federal de Viçosa - UFV, defendendo a sua dissertação em dezembro de 2006.

# Sumário

**Lista de Tabelas** viii

**Lista de Figuras** ix

**Resumo** xi

**Abstract** xii

**1 Introdução** **1**

1.1 Objetivos do Trabalho . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2 Organização deste Documento . . . . . . . . . . . . . . . . . . . . . . 4

**2 Ferramentas que utilizam PBD para produzir animação** **6**

2.1 Stagecast Creator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.1.1 Programar sem uma linguagem de programação textual . . . . . 7

2.1.2 Funcionamento do Stagecast Creator . . . . . . . . . . . . . . . 8

2.1.3 Recursos do Stagecast Creator . . . . . . . . . . . . . . . . . . . 13

2.1.4 Metas do Stagecast Creator . . . . . . . . . . . . . . . . . . . . 15

2.1.5 Evidência Empírica . . . . . . . . . . . . . . . . . . . . . . . . . 16

2.1.6 Conclusão . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

2.2 AgentSheets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

2.2.1 Programação Tangível . . . . . . . . . . . . . . . . . . . . . . . 20

2.2.2 Ambiente do AgentSheets . . . . . . . . . . . . . . . . . . . . . 21

2.2.3 Conclusão . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

2.3 Gamut . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

2.3.1 Conclusão . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

2.4 ToonTalk . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

2.4.1 Histórico do ToonTalk . . . . . . . . . . . . . . . . . . . . . . . 34

2.4.2 Código-fonte animado . . . . . . . . . . . . . . . . . . . . . . . . 38
2.4.3 Exemplo utilizando o ToonTalk . . . . . . . . . . . . . . . . . . . 38
2.4.4 Conclusão . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
2.5 Conclusão . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

**3 Sugestões para ambientes que utilizam PBD 45**
3.1 Perspectiva em Primeira Pessoa . . . . . . . . . . . . . . . . . . . . . 48
3.2 Herança . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
3.3 Regras Visuais com Condições Negativas . . . . . . . . . . . . . . . . 54
3.4 Conclusão . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

**4 Implementação do Tabajara Animator 57**
4.1 Máquinas de Estados Abstratas - ASM . . . . . . . . . . . . . . . . . 58
4.2 Modelo de Execução para Representação de Ambiente com Múltiplos Agentes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.2.1 Componentes da Representação XML do Modelo . . . . . . . . 62
4.2.2 Semântica dos Elementos . . . . . . . . . . . . . . . . . . . . . . 68
4.2.3 Interpretador para o Modelo . . . . . . . . . . . . . . . . . . . . 70
4.3 Ferramentas para Construção de Código . . . . . . . . . . . . . . . . 72
4.4 Recursos para Apresentação de Simulações Animadas . . . . . . . . . 77
4.5 Conclusão . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

**5 Exemplo - Avaliação e Validação 82**
5.1 O jogo Pacman . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
5.2 O jogo Pacman no AgentSheets . . . . . . . . . . . . . . . . . . . . . 85
5.3 O jogo Pacman no Stagecast Creator . . . . . . . . . . . . . . . . . . 89
5.4 O jogo Pacman no Tabajara Animator . . . . . . . . . . . . . . . . . 92
5.5 Conclusão . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95

**6 Conclusões e Trabalhos Futuros 98**
6.1 Conclusões . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
6.2 Trabalhos Futuros . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99

**Referências Bibliográficas 102**

# Lista de Tabelas

5.1 Diferenças na quantidade de Regras. . . . . . . . . . . . . . . . . . . . . . 96

5.2 Quantidade de Regras para o Ambiente Tabajara Animator. . . . . . . 97

# Lista de Figuras


2.1 Stagecast - primeiro passo para criar uma regra. . . . . . . . . . . . . . . 9
2.2 Stagecast - criando uma regra. . . . . . . . . . . . . . . . . . . . . . . . . 10
2.3 Stagecast - regra de movimentação criada. . . . . . . . . . . . . . . . . . 11
2.4 Stagecast - regra envolvendo dois personagens. . . . . . . . . . . . . . . . 11
2.5 Stagecast - janela "Character Window" de um personagem. . . . . . . . 12
2.6 Stagecast - janela Rule Maker com as várias ações de uma regra. . . . . 14
2.7 AgentSheets - ferramenta Gallery. . . . . . . . . . . . . . . . . . . . . . . 22
2.8 AgentSheets - ferramenta Worksheet. . . . . . . . . . . . . . . . . . . . . 23
2.9 AgentSheets - editor de comportamento para cada agente. . . . . . . . 24
2.10 AgentSheets - janelas de condições e ações para serem usadas na criação das regras. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.11 AgentSheets - janela Triggers. . . . . . . . . . . . . . . . . . . . . . . . . 27
2.12 AgentSheets - referência ao personagem Pacman, sem se preocupar com seu estado. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.13 Jogo do tipo Q*Bert criado totalmente no Gamut. . . . . . . . . . . . . 31
2.14 Termos da Ciência da Computação e termos do ToonTalk. . . . . . . . 35
2.15 Troca de números utilizando o ToonTalk. . . . . . . . . . . . . . . . . . 39

3.1 Stagecast Creator – Regras necessárias para movimentar um personagem. 47
3.2 Uso da herança (Repenning and Perrone, 2000). . . . . . . . . . . . . . . 50
3.3 Operadores utilizados no Creator. . . . . . . . . . . . . . . . . . . . . . . 55

4.1 Representa a definição de um ambiente. . . . . . . . . . . . . . . . . . . 63
4.2 Representa uma sequência de elementos. . . . . . . . . . . . . . . . . . 63
4.3 Representa uma escolhar entre elementos. . . . . . . . . . . . . . . . . . 63

4.4 Representa o conjunto de definições de atributos. . . . . . . . . . . . . 64
4.5 Representa as regras de transição. . . . . . . . . . . . . . . . . . . . . . . . 65
4.6 Representa os tipos de regras existentes. . . . . . . . . . . . . . . . . . 66
4.7 Representa expressões. . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.8 Representação de funções. . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.9 Representação para um endereço. . . . . . . . . . . . . . . . . . . . . . 68
4.10 Tabajara Animator - Editor da hierarquia de classes. . . . . . . . . . . 74
4.11 Tabajara Animator - Tela de criação de personagem. . . . . . . . . . . 75
4.12 Tabajara Animator - Editor de regras da classe Moveable. . . . . . . . 76
4.13 Tabajara Animator - Editor de regras da classe Pacman. . . . . . . . . 76
4.14 Tabajara Animator - Tela para criar regras. . . . . . . . . . . . . . . . 77
4.15 Janelas de perspectiva em primeira pessoa e perspectiva em terceira pessoa com escolha de personagem central. . . . . . . . . . . . . . . . . 78
4.16 Janela de perspectiva em terceira pessoa. . . . . . . . . . . . . . . . . . 79
4.17 Janela de perspectiva em primeira pessoa. . . . . . . . . . . . . . . . . 80
4.18 Problema que rotação que pode acontecer. . . . . . . . . . . . . . . . . 80

5.1 AgentSheets - regra para mudar a direção do Pacman. . . . . . . . . . . 85
5.2 AgentSheets - regras para o Pacman se mover por posições vazias. . . . 87
5.3 AgentSheets - regras para o Pacman comer pílulas. . . . . . . . . . . . 88
5.4 Stagecast - regras para o Pacman se mover por posições vazias. . . . . . 90
5.5 Stagecast - regras para o Pacman se mover e comer as pílulas. . . . . . 91
5.6 Tabajara – regra de movimento para a classe *Moveable*. . . . . . . . . . 92

# Resumo

COURA, Débora Pereira, M.Sc.,Universidade Federal de Viçosa, dezembro de 2006. **Produzindo animações através da programação por demonstração**. Orientador: Vladimir Oliveira Di Iorio. Co-Orientadores: Alcione de Paiva Oliveira e Marcus Vinícius Alvim Andrade.

Novas técnicas de programação, mais intuitivas e fáceis de serem utilizadas por não programadores, estão em desenvolvimento e são utilizadas, principalmente, na criação de jogos e simulações. Dentre elas a programação por demonstração se destaca e já é implementada em sistemas mundialmente conhecidos e utilizados. Essa dissertação apresenta um estudo sobre os principais sistemas que trabalham com essas técnicas, explora a sua utilização, suas vantagens e desvantagens. Levando em consideração os pontos fracos dos sistemas estudados são propostas três melhorias: a utilização de regras escritas em primeira pessoa, o uso da herança e o uso de regras com condições negativas. As propostas são implementadas em uma ferramenta especialmente desenvolvida para que os testes pudessem ser realizados. Para poder traçar uma comparação da nova ferramenta, que utiliza os recursos propostos, com sistemas similares, foi construído um exemplo envolvendo animação na nova ferramenta e em dois dos quatro sistemas estudados. Essa comparação permite verificar os benefícios proporcionados pelas propostas deste trabalho.

# Abstract


COURA, Débora Pereira, M.Sc.,Universidade Federal de Viçosa, December of 2006. **Producing animations with the programming by demonstration**. Advisor: Vladimir Oliveira Di Iorio. Co-Advisor: Alcione de Paiva Oliveira and Marcus Vinícius Alvim Andrade.

Producing Animation with Programming by Demonstration Modern tools for the creation of games and animated simulations frequently apply programming techniques which are intuitive and can be used by non-professional programmers. One of these techniques is known as *Programming by Demonstration (PBD)*, and consists of teaching the computer new behavior by demonstrating actions on concrete examples. This work studies the main systems for simulated animation using PBD and similar techniques, identifying the strong and weak points of each one. Using this study, three improvements are proposed: *first-person perspective* for visual rules, the use of *inheritance* and visual *negative conditions*. The proposals are implemented in a tool called *Tabajara Animator*. A relevant application is developed using the new tool and two similar systems. The results show that the proposed enhancements may bring a significant reduction on the number of required rules for a simulated animation.

# Capítulo 1

# Introdução

O computador é uma máquina poderosa que ainda esconde, da maioria dos seus usuários, uma grande quantidade de recursos. A melhor utilização desses recursos é feita por meio da programação. Mas programar, para a maior parte das pessoas, é uma tarefa difícil.

Elliot Soloway, diretor do Highly Interactive Computing Project da University of Michigan, estimou que, de um grupo de alunos que fizeram aulas de programação, menos de um por cento continua a programar quando as aulas terminam. Isto ocorre porque existe uma lacuna muito grande entre a representação que o cérebro usa quando pensa em um problema e a representação que o computador aceita. Segundo ele, existem duas maneiras de resolver esse problema, de preencher esta lacuna. A primeira maneira seria fazer com que o usuário ficasse mais perto do sistema, ou seja, o usuário deveria aprender a pensar como um computador. A segunda maneira seria fazer com que o sistema ficasse mais perto do usuário. A primeira opção é desconfortável para a maioria das pessoas. Elas demoram a aprender a fazer isso e quando aprendem não gostam de fazê-lo. Afinal, todos querem utilizar o computador sem ter que aprender uma nova linguagem, ainda mais uma linguagem não natural (Smith et al., 2000).

Hoje já existem estudos para que se possa utilizar a segunda opção, ou seja, fazer com que o computador fique mais perto do usuário. Isso pode ser feito por sistemas que se tornam cada vez mais fáceis de serem operados. Mas a dificuldade ainda existe na hora de criar técnicas de programação mais intuitivas e que permitam

que não programadores possam desenvolver os seus próprios programas.

A Programação por Demonstração é uma técnica utilizada para instruir o computador a aprender novos comportamentos por meio da apresentação de ações em exemplos concretos (Cypher et al., 1993). Essa técnica é conhecida pelo termo PBD, do inglês Programming by Demonstration, ou PBE, de Programming by Example. A sua aplicação permite ao usuário dizer, por meio de uma interface gráfica, o que deve ser feito, sem se preocupar em dizer como isso deve ser feito. Essas ações são salvas e podem ser usadas mais tarde com diferentes entradas de dados.

O mais importante da PBD é que ela pode ser utilizada por qualquer pessoa, mesmo as que não possuem conhecimentos de programação. Utilizá-la é tão fácil quanto utilizar o computador como simples usuário de softwares já desenvolvidos. Isso faz com que a PBD seja considerada uma abordagem alternativa para as linguagens sintáticas. Em linguagens sintáticas, é necessário que o programador guarde várias informações de como a linguagem funciona, já na PBD basta dar um exemplo para que o programa seja feito e possa ser utilizado em outros momentos.

O desenvolvimento e a implementação da PBD em ambientes para programação pode aumentar o número de usuários desses sistemas. Por ser uma técnica mais fácil de ser aprendida, um número maior de pessoas pode se interessar em desenvolver pequenas simulações, animações ou jogos. Os profissionais liberais ganham maior autonomia para desenvolver seus próprios programas. A área educacional ganha um forte aliado, pois os sistemas podem ser usados nas escolas para desenvolver o raciocínio das crianças de uma forma divertida e prazerosa. E também pode ser usado para iniciar os alunos nos estudos de programação.

Em estudos realizados e citados em (Smith et al., 2000), ficou provado que os estudantes que aprenderam a programar utilizando um sistema de PBD obtiveram, muitas vezes, resultados melhores do que os que aprenderam pelo método tradicional. E depois esses estudantes tiveram maior facilidade para aprender uma linguagem convencional e se tornaram melhores programadores.

A programação por demonstração tem sido usada com sucesso para simplificar tarefas repetitivas em áreas como a digitação de textos (Lau et al., 2001; Masui, 2001),

a navegação e criação de páginas para a Web (Sugiura, 2001), a criação de programas usando linguagens de programação convencionais (Ruvini and Dony, 2001), o reconhecimento de textos (Lieberman et al., 2001) etc. Outra área onde as técnicas de PBD têm forte aplicação é a construção de simulações animadas e jogos de computador, principalmente em sistemas voltados para a educação de crianças. Exemplos importantes desse tipo de sistema são o Stagecast Creator (Smith et al., 2000), o AgentSheets (Repenning, 2000), o Gamut (McDaniel and Myers, 1998) e o ToonTalk (Kahn, 1996).

## 1.1 Objetivos do Trabalho

A análise de sistemas que utilizam técnicas de PBD indica que eles trazem uma maior facilidade para o usuário comum, ou seja, o usuário que não é programador. Como os sistemas são mais intuitivos e gráficos, ele se sente mais estimulado a aprender e a utilizá-lo. Em algumas situações, o usuário para ensinar algum comportamento para o seu personagem, precisa criar uma grande quantidade de regras, onde a única diferença está na sua orientação. Isso acontece quando é necessário ensinar o personagem a andar, por exemplo. São necessárias quatro regras para que o personagem possa se locomover nas quatro direções. E em outras situações o número de regras pode ser ainda maior. Esse trabalho repetitivo pode acabar desanimando o usuário, já que esses ambientes possuem como principal atrativo serem fáceis de operar e proporcionarem maiores facilidades e pouco trabalho.

Nos sistemas estudados, quando dois ou mais personagens precisam realizar a mesma ação, a regra precisa ser demonstrada duas ou mais vezes, uma vez para cada personagem diferente. Não existe um mecanismo que possibilite o aproveitamento das regras já criadas. Elas terão as mesmas ações e as mesmas condições, apenas personagens diferentes. Isso aumenta o tempo necessário para criar a simulação sem proporcionar nenhuma novidade, é apenas uma repetição de ações. Sem mencionar que, no caso de uma pequena alteração nas regras criadas, tudo deverá ser feito duas ou mais vezes.

O objetivo geral do trabalho é propor e analisar formas de evitar o grande número de regras praticamente iguais, como as que seriam criadas nos exemplos citados acima. Especificamente pretente-se:

- Analisar os sistemas já existentes e divulgados, Stagecast Creator, AgentSheets, Gamut e ToonTalk, identificando como eles agem na hora em que o usuário precisa criar regras onde a única diferença está na orientação do personagem e quando vários personagens irão utilizar a mesma regra;
- Detalhar as três sugestões, Perspectiva em Primeira Pessoa, Herança, Regras Visuais com Condições Negativas, exemplificando as situações onde seriam aplicadas;
- Descrever como seria a implementação de um novo ambiente de programação que utilize as técnicas de programação por demonstração e que tenha os novos recursos implementados;
- Validar as propostas através da criação de um exemplo onde os três novos recursos pudessem ser usados.

Essas propostas são: ele terá que criar a mesma regra para vários personagens diferentes.

## 1.2 Organização deste Documento

- No Capítulo 2 é feita uma revisão bibliográfica apresentando os quatro sistemas estudados, Stagecast Creator, AgentSheets, Gamut e ToonTalk.
- No Capítulo 3 são apresentas as sugestões que podem melhorar o uso das técnicas de PBD em ambientes de programação. Nele são exploradas as motivações que levaram às sugestões e as situações onde elas seriam melhor aplicadas.
- No Capítulo 4 é descrito o arcabouço criado para testar e validar as sugestões apresentadas no capítulo anterior. Nele são discutidas as técnicas utilizadas na

criação do sistema que irá criar e interpretar os programas feitos por meio das técnicas de programação por demonstração.

- No Capítulo 5 é apresentado um exemplo de um jogo criado em dois dos sistemas estudados e no sistema desenvolvido neste trabalho. Por meio desse exemplo é feita a validação das sugestões apresentadas.

- No Capítulo 6 são apresentadas as contribuições e as conclusões desta dissertação, são também enumeradas possíveis melhorias e alguns possíveis trabalhos futuros que poderão ser realizados para o aprimoramento deste trabalho.

# Capítulo 2

# Ferramentas que utilizam PBD para produzir animação

Nesse capítulo são apresentados quatro sistemas que utilizam as técnicas de PBD (Programação por Demonstração). Eles foram escolhidos por serem ambientes de programação intuitivos, fáceis de serem aprendidos, especialmente desenvolvidos para que não programadores possam criar seus próprios jogos e simulações. E são ferramentas divulgadas e conhecidas internacionalmente. As ferramentas abordadas são o Stagecast Creator, o AgentSheets, o Gamut e o ToonTalk. O funcionamento, os recursos, as vantagens e desvantagens de cada ferramenta são apresentados nas seções que se seguem.

## 2.1 Stagecast Creator

O Stagecast Creator (Smith, 2000; Smith et al., 2000) foi um dos primeiros produtos comerciais a utilizarem a PBD. É voltado principalmente para as crianças, e permite que elas criem suas próprias histórias, jogos e simulações interativas. Ele foi lançado em 1999, com a finalidade de evitar a dificuldade de aprendizado existente nas então conhecidas abordagens de programação. Esse software reúne duas tecnologias: a **Programação por Demonstração (PBD)** e as **Regras Visuais Antes-Depois (before-after rules)**. Essas tecnologias permitem que a abordagem tradicional de

como aprender a programar seja eliminada.

De acordo com Smith et al. (2000), o Stagecast Creator é a culminação de um esforço de sete anos de pesquisa e de desenvolvimento, os primeiros cinco na Apple Computer. Ele é um novo sistema de programação para construir simulações no Stagecast Software, Inc., fundado em 1997. O projeto, chamado inicialmente de KidSim, foi rebatizado mais tarde de Cocoa, e finalmente tornou-se Creator. O objetivo era fazer com que os computadores fossem mais úteis na educação. Por várias razões, os co-inventores do Creator - os autores Smith e Cypher - focaram em simulações, uma ferramenta poderosa de ensino para transformar idéias abstratas em idéias concretas e mais compreensíveis. As simulações incentivam a experimentação, ajudando as crianças a desenvolverem o raciocínio seqüencial e causal, ou seja, o método científico. O objetivo do projeto Creator evoluiu para capacitar os usuários finais - professores e estudantes - para construírem e modificarem simulações através da programação.

### 2.1.1 Programar sem uma linguagem de programação textual

O grande desafio para os pesquisadores era desenvolver um programa onde os usuários pudessem programar um computador sem utilizar linhas de códigos, sem que precisassem aprender uma linguagem de programação. A solução encontrada foi combinar duas técnicas já existentes: **PBD** e **Regras Visuais Antes-Depois**. Na PBD, os usuários demonstram os algoritmos utilizando uma interface do computador, dizendo o que deve ser feito sem precisar informar como deve ser feito. O computador grava as ações do usuário e pode executá-las mais tarde com diferentes entradas. A característica mais importante da PBD é que qualquer pessoa, com conhecimento básico de usuário de computador, pode utilizá-la. Essa característica permite que a PBD seja considerada uma abordagem alternativa para as linguagens sintáticas, no caso da criação de jogos ou simulações. Enquanto na linguagem sintática é necessário guardar várias informações de como ela funciona, de como os comandos precisam ser escritos, na PBD basta dar um exemplo para o computador, e o seu programa estará pronto para ser usado em qualquer situação. Sem a necessidade de se digitar uma única linha

de código (Smith et al., 2000).

Um problema da PBD é a forma de representar um programa gravado para os usuários. Não é conveniente que os usuários possam facilmente criar um programa, mas que depois tenham que aprender uma linguagem sintática difícil para vê-lo ou modificá-lo, como acontece com a maioria dos sistemas de PBD. No Creator, primeiro os seus criadores mostraram o programa representando cada etapa em um formulário gráfico ou textual. Mas chegaram à conclusão que algumas representações ficariam muito grandes, apesar de serem representações elegantes. O fato de mostrar todas as etapas acabaria por desestimular os usuários, já que complicaria a visualização do programa.

Para evitar esse problema, eles utilizaram a técnica de **Regras Visuais Antes-Depois**. Essa técnica permite que o Creator mostre apenas o estado inicial e o final das regras. Caso a pessoa deseje, pode visualizar cada ação utilizada na regra criada. O Creator possui uma sintaxe própria para isso, que são as listas de regras de testes e de ações. Para não dificultar o entendimento do usuário, essas listas só são mostradas se houver interesse do programador. Assim pode-se programar por muito tempo sem ficar ciente dessa sintaxe. Essa característica é totalmente diferente das linguagens convencionais, em que os usuários devem saber detalhes da sintaxe, como por exemplo, a ordem dos parâmetros e onde colocar as variáveis, ponto e vírgula e parêntesis. A sintaxe utilizada no Creator é voltada para representar de forma visual, em gráficos ou textos, o que o usuário já criou através da programação por demonstração. Essa sintaxe só é utilizada quando o usuário sente a necessidade de modificar ou entender uma regra já criada.

### 2.1.2 Funcionamento do Stagecast Creator

O Stagecast Creator é um ambiente intuitivo, simples e de fácil aprendizado. Essa seção explica como criar regras nessa ferramenta e como utilizar outros recursos encontrados nela.

As regras atribuem funcionalidades ao jogo ou simulação. Depois de definidos os personagens, chega o momento de definir as ações executadas por eles. Para criar

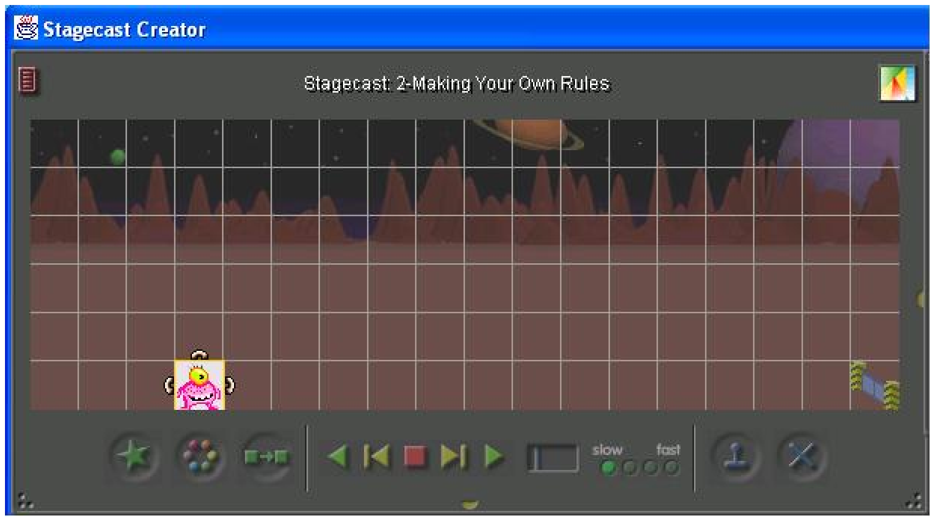


Figura 2.1: Stagecast - primeiro passo para criar uma regra.

uma regra, tem-se a necessidade de seguir, na ordem especificada, quatro passos. **Passo 1** - ativar a ferramenta de construção de regras e selecionar o personagem; **passo 2** - definir a área de atuação; **passo 3** - realizar a ação e **passo 4** - finalizar a regra.

Por exemplo, para fazer um personagem de uma animação se mover da esquerda para a direita basta executar os quatro passos. No passo 1 é preciso clicar no botão de criar regras ("Rule Tool") e depois no personagem que se deseja mover. O fundo irá se modificar, transformando-se em uma tela quadriculada, como na Figura 2.1. A tela permanecerá dessa forma durante toda a criação da regra.

O segundo passo é aumentar o contorno do personagem para que ele abranja o próximo quadrado a direita, como mostrado na Figura 2.2. O quadrado ou área em que se encontra o objeto destacado possui alças em várias direções. Para aumentar o seu contorno, basta clicar na alça da direita e arrastá-la até o término do próximo quadrado à direita. Na mesma Figura 2.2, pode-se ver uma janela chamada **Rule Maker**. Ela é utilizada para representar a regra que está em criação. Pode-se observar que existem duas figuras iguais do personagem selecionado, uma na esquerda e outra na direita. À medida que a regra for criada, a figura da direita irá se modificar para mostrar o resultado da execução daquela regra. Ou seja, na figura da esquerda temos o personagem no seu estado inicial e na direita temos o personagem em seu estado

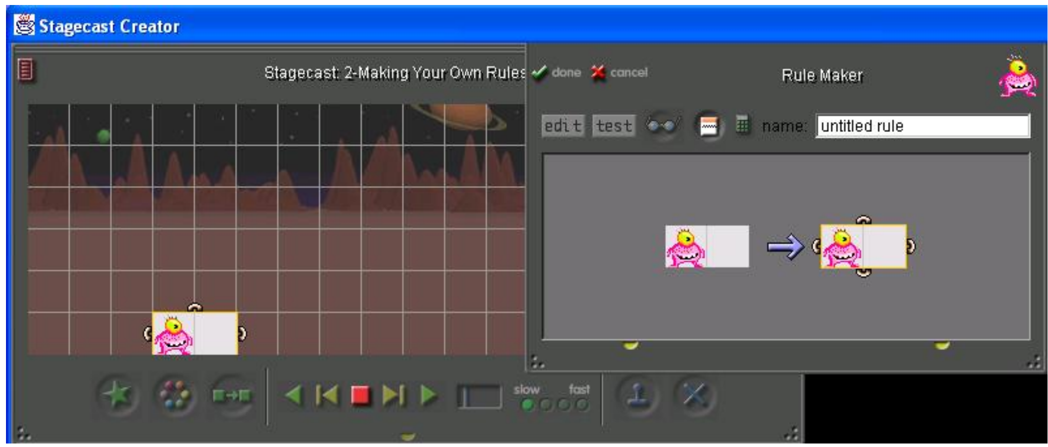


Figura 2.2: Stagecast - criando uma regra.

final. Isso pode ser feito graças ao uso da técnica de **Regras Visuais Antes-Depois**.

Para executar o terceiro passo é necessário clicar no personagem e arrastá-lo para o quadrado à direita, dessa forma a regra é atualizada na janela “Rule Maker”. Veja o resultado na Figura 2.3. Assim, o quarto passo, que finaliza a criação da regra, já pode ser executado. A única ação realizada nele e clicar sobre o botão “Done”, que se encontra na parte superior esquerda da janela “Rule Maker”. Sem a necessidade de digitar uma só linha de código, apenas seguindo esses quatro passos, teremos uma regra que fará com que o personagem se desloque para a direita. Ela pode ser lida da seguinte forma:

“Se o personagem estiver em um caminho reto e houver caminho reto e vazio à sua direita, então mova o personagem para a direita.”

Para ver o resultado da regra, basta clicar no botão “Play”, e o personagem se moverá para a direita a cada tique do relógio. E continuará se movendo até que encontre um obstáculo à sua direita, como por exemplo, uma pedra, outro personagem ou qualquer outro objeto.

As regras podem ser criadas para realizarem várias ações e podem envolver mais de um personagem da simulação ou do jogo. O processo para criar as regras independe da quantidade de ações ou personagens envolvidos, basta seguir os quatro passos apresentados acima. A Figura 2.4 mostra uma regra que envolve dois personagens: um boneco e uma rocha. Ela poderia ser interpretada da seguinte forma:

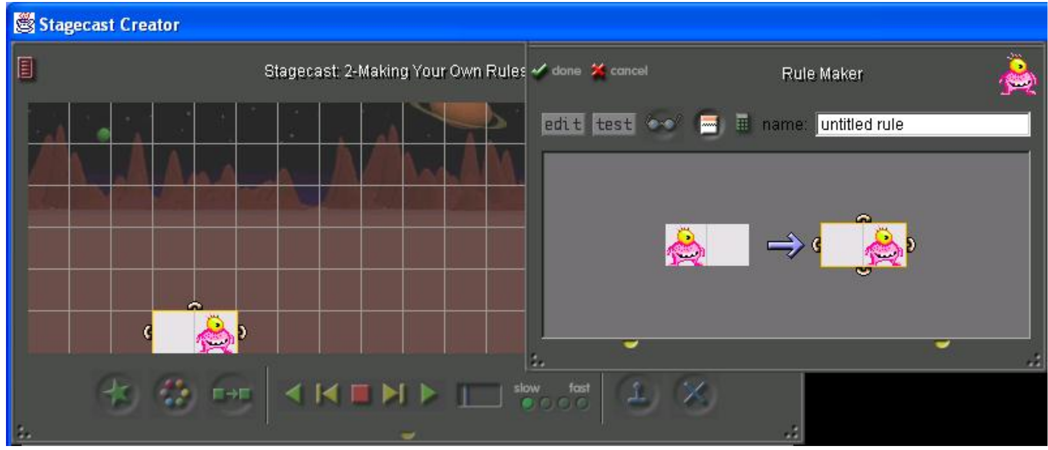


Figura 2.3: Stagecast - regra de movimentação criada.

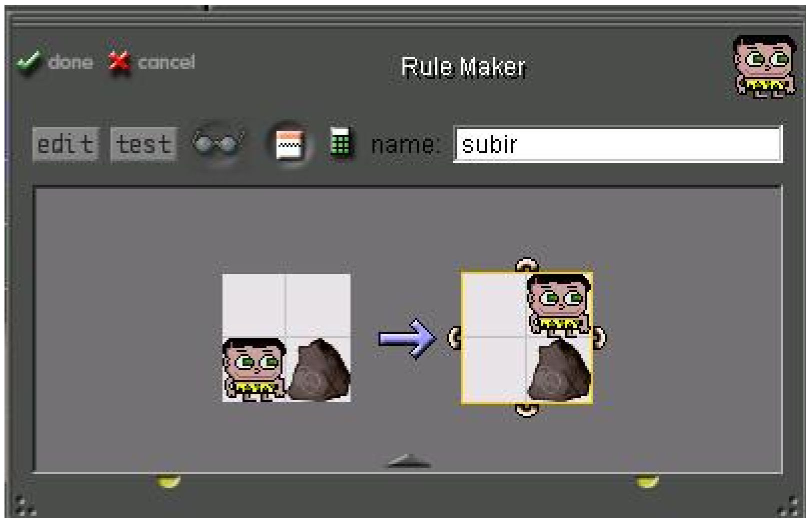


Figura 2.4: Stagecast - regra envolvendo dois personagens.

"Se no quadrado à direita do personagem houver uma rocha e não existir nada no quadrado acima dela, então mova o personagem para este quadrado."

Cada personagem pode ter, associado a ele, nenhuma, uma ou várias regras. Para ver todas as regras de um personagem basta acessar a sua "Character Window". A Figura 2.5 mostra um exemplo de um personagem com três regras.

Cada personagem possui a sua "Character Window", cada regra possui um nome e esse nome pode ser modificado para melhorar a organização das regras. As regras estão listadas por ordem de prioridade. Cada nova regra criada fica no topo da lista e possui prioridade sobre as que estão abaixo dela. Isso significa que, ao pressionar

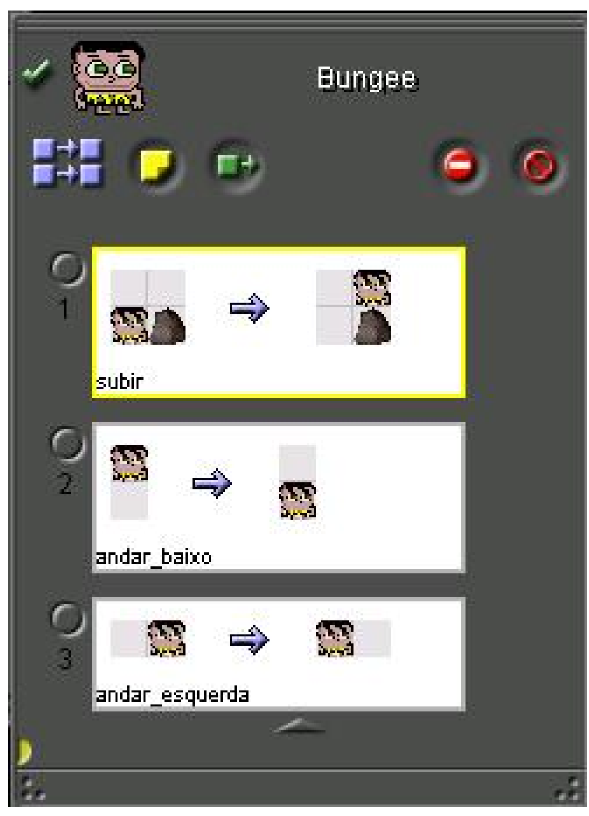


Figura 2.5: Stagecast - janela “Character Window” de um personagem.

o botão “Play”, a cada “tique” do relógio, na ordem em que aparecem, todas as regras do personagem em questão serão testadas. A primeira regra que combinar com as suas condições atuais será executada e as outras serão ignoradas. A prioridade das regras pode ser mudada, para isso basta alterar a posição das regras na “Character Window”. Isso é feito selecionando e arrastando as regras.

Usando a técnica de PBD, o programa é escrito na linguagem do domínio do problema, sem utilizar termos de linguagens de programação convencionais. O usuário ganha em comodidade e simplicidade. É uma forma bem mais fácil de programar, bem mais intuitiva.

O fato de mostrar apenas o estado inicial e final da regra faz com que fique mais fácil para o usuário entender os programas já criados e até os criados por outras pessoas, exceto quando são realizadas várias ações intermediárias. Nesse caso, o usuário, vendo apenas a regra, não terá idéia do que acontece entre o estado inicial e o final. É preciso executar a regra para saber todas as ações realizadas por ela. O sistema propicia ao usuário, caso ele queira, uma forma de ver todas as ações envolvidas na regra. Com apenas um clique, o usuário pode abrir a parte da janela que mostra todas as ações. A Figura 2.6, mostra todas as ações necessárias para que o cachorrinho suba em cima da lata de lixo, apareça o som do seu latido e depois desça da lata de lixo. Este recurso é muito útil, pois o usuário pode programar por muito tempo vendo apenas a regra, mas se precisar pode ver a representação que o sistema faz de cada uma das ações envolvidas naquela regra. E o interessante é que essas representações são feitas de forma bem intuitiva, para facilitar o seu entendimento.

### 2.1.3 Recursos do Stagecast Creator

O sistema Stagecast Creator possui ferramentas que auxiliam o usuário na construção das suas simulações. O usuário terá a sua disposição ferramentas para fazer cópias dos seus personagens. Suas regras podem ser apagadas. Ele possui também um editor gráfico que permite criar os personagens da simulação. Cada personagem pode ter mais de uma aparência armazenada e utilizada nas regras criadas. Todos os personagens criados ficam armazenados na “Character Drawer”, e podem ser inseridos

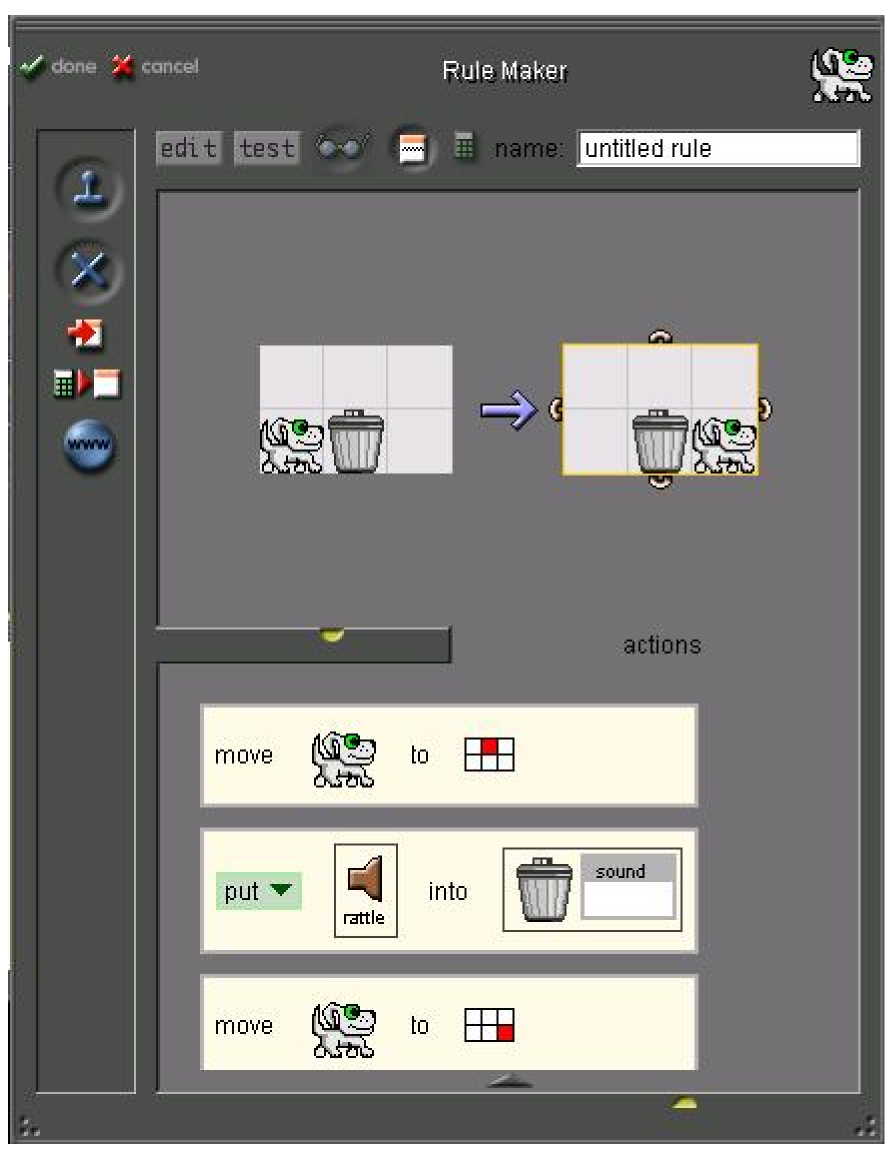


Figura 2.6: Stagecast - janela Rule Maker com as várias ações de uma regra.

na simulação a qualquer momento.

Para ajudar na programação e na detecção de erros, a ferramenta disponibiliza um sistema de depuração de regras. O usuário tem a possibilidade de mandar testar a regra criada. De uma forma bem gráfica será realçada na regra a parte com problema. Isso facilita a descoberta do erro existente.

As regras também podem envolver cliques do mouse e pressionamento de teclas do teclado. Por exemplo, pode-se criar regras para que um peixinho se mova dentro do aquário nas quatro direções quando as setas do teclado são pressionadas, ou pode-se fazer um fantasma desaparecer com um clique do botão direito do mouse sobre ele.

No Creator é possível armazenar arquivos de áudio para serem usados nas simulações. E tudo isso de uma forma gráfica e bem intuitiva.

### 2.1.4 Metas do Stagecast Creator

Segundo Rader et al. (1997), uma das metas do Creator é que os alunos possam utilizá-lo para explorar tópicos de ciências ou de outra matéria estudada. Neste caso é preciso que eles aprendam mais do que o mecanismo de criar regras para mover personagens na tela. Para criar um modelo que mostre um fenômeno, eles precisam decompor o problema em personagens com comportamentos particulares. Precisam aprender como programar os comportamentos desejados utilizando a capacidade de linguagem do Creator, além de aprender a distribuir os comportamentos nos vários personagens utilizados com a finalidade de alcançar o resultado global desejado para mostrar o conceito ou fenômeno de seu interesse.

De acordo com o que foi visto durante a demonstração do sistema Creator e de acordo com Rader et al. (1997), para que os estudantes possam projetar, implementar e depurar simulações complexas, eles precisam de um pouco de conhecimento das operações subjacentes do Creator. Essas operações são as seguintes:

- Ações individuais: Entender a ação de uma regra requer comparar o "antes" e "depois" nas figuras que determinam a ação. E em algumas situações, analisar a lista de ações envolvidas naquela regra.

- Ordenação das regras: Podem ser criadas várias regras para cada objeto do mundo em estudo. E a ordem em que essas regras são colocadas é muito importante. O sistema começa a avaliar as regras seguindo a sua ordem de apresentação. Assim que ele encontrar uma regra que possa ser usada, ela será executada. Isso pode fazer com que regras sejam executadas em uma ordem errada, ou que uma regra nunca seja executada por estar fora de ordem.

- Casamento de padrão: A regra só será executada se ocorrer uma situação exatamente igual no ambiente. Espaços vazios e outros objetos fazem toda a diferença na hora que o sistema escolhe qual regra será executada.

- Interação de objetos: Objetos interagem quando se movem pela tela em todas as direções, diretamente quando são incluídos vários objetos em uma regra, ou indiretamente quando um objeto "bloquear" ou "disparar" outro movimento.

No Creator também encontramos algumas características que são úteis para programar comportamentos mais complexos, como:

- Sub-rotinas, onde regras podem ser agrupadas para serem executadas em seqüência ou aleatoriamente.

- Propriedades (variáveis), em que os objetos podem possuir variáveis passíveis de serem modificadas ou avaliadas durante a execução do programa.

Alguns desses conceitos, como o de sub-rotinas e de propriedades, estão também presentes nas linguagens de programação convencionais. A diferença está no fato de que no Creator eles serão ensinados para os alunos de uma forma gráfica, divertida e bem intuitiva. E isso terá um impacto positivo quando ele tiver contato com esses mesmos conceitos em outros ambientes de programação.

### 2.1.5 Evidência Empírica

Segundo Smith et al. (2000), o Creator proporciona uma abordagem ideal para se trabalhar com alunos. Essas evidências foram conseguidas através da observação informal, do estudo formal dos usuários e dos relatórios informais.

Os dados foram conseguidos por meio de centenas de horas de testes feitos com crianças e adultos durante cinco anos. A maior parte foi feita com crianças entre seis e doze anos.

Foram implementados três protótipos do Creator, cada um menor e mais rápido e mais perto da qualidade prevista. O protótipo final - Cocoa - foi testado com centenas de alunos. Ele foi distribuído pela Internet, mas a melhor fonte de informação foi a conseguida por meio dos alunos da Califórnia. Os professores o utilizavam em suas aulas, propondo que os alunos resolvessem alguns problemas. E a surpresa foi que alguns alunos continuaram a trabalhar em suas simulações mesmo estando de férias. E eles não acharam a tarefa de programar complicada ou chata (Smith et al., 2000).

Os pesquisadores acabaram por concluir que as crianças podem programar utilizando esta abordagem e ainda podem gostar de programar. O Creator consegue transformar a tarefa de programar em uma grande e divertida brincadeira. Tanto que os estudantes que o utilizavam para desenvolver os seus projetos conseguiram resultados melhores do que os estudantes que utilizavam linguagens de programação convencionais. E o estudo também concluiu que os alunos que aprenderam primeiro a trabalhar com o Creator e depois aprenderam outra linguagem de programação se tornaram melhores programadores.

### 2.1.6 Conclusão

As técnicas de **PBD** eliminam a necessidade da linguagem sintática durante a construção do programa e as **Regras Visuais Antes-Depois** eliminam a necessidade de uma linguagem sintática para a sua representação. As representações usadas na programação ficaram mais próximas das representações usadas pelo cérebro humano. Isso torna o Creator um sistema gráfico muito intuitivo para a construção de programas. É mais fácil e agradável aprender a trabalhar com ele do que aprender e utilizar uma linguagem de programação convencional. Principalmente quando se trabalha com crianças ou adolescentes. E pode-se concluir que se crianças e adolescentes conseguem trabalhar com esse sistema, para os adultos a facilidade se torna ainda maior. E

os programadores ganham um ambiente mais prático para construir simulações e/ou jogos.

Um problema existente é que esse ambiente possui um domínio limitado para as simulações visuais ou para jogos. Ele não está preparado para implementar programas que realizem cadastros ou atividades comerciais convencionais.

Uma vantagem do Stagecast Creator é que as regras criadas não interferem umas nas outras, a não ser que uma tenha prioridade maior e por isso tenha sido executada primeiro.

Uma desvantagem é que as regras são criadas para objetos específicos. Por exemplo, se uma regra é criada para que um personagem possa subir em uma pedra, se outro personagem for colocado na simulação é necessário demonstrar, novamente, a regra para esse personagem, mesmo que a regra seja idêntica. Se uma simulação tem dez personagens que fazem a mesma coisa, será necessário criar a mesma regra dez vezes. Não existe o mecanismo de herança implementado. Isso provoca um aumento no tempo de programação e um aumento no número de regras. Outro problema encontrado é que ele possui um domínio limitado à simulações visuais ou jogos.

O número de regras também fica grande pelo fato de tudo ter que ser demonstrado nos seus mínimos detalhes. Por exemplo, não é possível ensinar o personagem a simplesmente andar. É preciso ensiná-lo a andar para a direita, para a esquerda, para cima e para baixo. Só para esse pequeno exemplo seriam necessárias quatro regras. E essas regras seriam criadas para cada personagem utilizado na aplicação.

## 2.2 AgentSheets

AgentSheets é uma ferramenta de simulação baseada em agentes, que permite que uma grande faixa de usuários finais (de crianças a profissionais e cientistas) criem suas próprias simulações e jogos de uma forma fácil e divertida (Repenning, 2000; Repenning and Sumner, 1995). É um ambiente de programação intuitivo e transparente, que permite combinar o uso de agentes, planilhas eletrônicas e a compilação do código para a linguagem Java. Dessa forma as simulações criadas podem ser exportadas

como applets interativos de Java, permitindo a sua disponibilização na Internet e sua grande divulgação.

Os agentes são os objetos programáveis do usuário final, reagem a operações com o mouse, entradas do teclado e movimentação ao seu redor, podem mudar de aparência, ler páginas da Web, enviar e-mail e resolver fórmulas.

Segundo Repenning (1991), um Agente é um objeto (ou pessoa) autorizado a agir para um cliente. O cliente pode ser outro agente ou um usuário do AgentSheets. Todo agente consiste em:

- Sensores: Sensores são métodos do agente que são ou ativamente disparados pelo usuário (por exemplo, clicando em um agente), ou que são usados para pesquisar o estado de outro agente. As classes embutidas de agentes providenciam um comportamento default definindo reações para todos os sensores. Para refinar esse comportamento, os métodos dos sensores podem ser modificados utilizando o paradigma de orientação objetos.

- Effectors: É um mecanismo para fazer a comunicação com outros agentes, através de mensagens enviadas para os agentes, usando as coordenadas relativas ou absolutas do grid. As mensagens ativam sensores dos agentes para que possam ser executadas. Adicionalmente, effectors também providenciam meios para que a representação do agente possa ser modificada.

- Estado: Descreve a condição na qual o agente está.

- Representação: A representação gráfica do estado do agente, por exemplo, a forma como ele está representado graficamente.

- Instância: Link para a classe do agente.

AgentSheets combina regras de **PBD** com **Regras de Reescrita Gráfica** (Cypher and Smith, 1995) em um paradigma de programação para usuário final. Regras de Reescrita Gráfica é uma técnica muito semelhante à que foi batizada de **Regras Visuais Antes-Depois (before-after rules)** no Stagecast Creator. As Regras de Reescrita Gráficas são linguagens poderosas para expressar o conceito de

mudança em uma representação visual. Elas descrevem transformações espaciais com uma seqüência de duas ou mais situações dimensionais contendo objetos. Situações podem ser interpretadas com respeito a objetos contidos e relações de espaço entre estes objetos. As diferenças entre as situações insinuam uma ou mais ações capazes de transformar uma situação em outra. No AgentSheets, qualquer número de Regras de Reescrita Gráfica pode ser agregado para criar comportamentos complexos para os agentes, e estes agentes e seus comportamentos combinam e interagem para simular qualquer comportamento desejado.

### 2.2.1 Programação Tangível

A linguagem de programação do AgentSheets, chamada Visual AgenTalk (VAT) (Repenning, 2000), é uma linguagem baseada em regras que se caracterizam pelo uso de condições e ações. As condições e ações são objetos completos que o usuário pode explorar.

A linguagem Visual AgenTalk emprega uma nova abordagem para a programação do usuário final, que é chamada de programação Tangível. A programação Tangível possui um conceito a mais do que a programação visual. Enquanto a maioria das linguagens de programação foi desenvolvida de uma perspectiva tecnológica, o design do ambiente do AgentSheets foi dirigido pela necessidade de pessoas comuns visualizarem, entenderem e comunicarem idéias. Dessas necessidades, talvez a mais importante seja a comunicação de idéias. E essa função o AgentSheets cumpre perfeitamente. Ela possui representações visuais que ajudam na legibilidade dos programas e possui também interfaces interativas para ajudar na escrita do programa (Repenning and Ambach, 1996).

A programação tangível permite experimentar a programação literalmente através de manipulação. A qualquer momento, o usuário seleciona uma condição ou uma ação e a testa sem ter que construir todo o programa primeiro. O usuário também poderá ter a explicação de qualquer condição ou ação através de ferramentas animadas existentes no AgentSheets. Estes recursos proporcionam uma avaliação imediata de cada regra criada e promovem a exploração de outras situações.

A Programação Tangível facilita a composição, a compreensão e o compartilhamento da programação (Repenning and Ambach, 1996). A composição, na Programação Tangível, faz com que os programas possam ser decompostos ou compostos ao longo de limites claramente definidos. A compreensão se torna mais fácil porque a percepção por meio da manipulação permite aos usuários finais examinarem a funcionalidade do programa eficientemente. Qualquer componente da linguagem, a qualquer hora, pode ser arrastado e colado sobre qualquer outro agente. O usuário poderá executar, com avaliação visual do componente, descobrindo condições que são verdadeiras ou falsas e mostrando as conseqüências da executação das ações. A Programação Tangível permite o compartilhamento de informações, pois suporta a migração da programação solitária para a programação social. Isso ocorre porque simulações inteiras, agentes individuais, regras e novas primitivas de linguagens podem ser compartilhadas pela Internet. E esse compartilhamento não exige que os usuários possuam nenhum arquivo em especial para utilizarem o material disponibilizado. O reaproveitamento é muito grande.

### 2.2.2 Ambiente do AgentSheets

No ambiente do AgentSheets, o usuário encontra ferramentas, com recursos visuais, que o ajudam a definir os agentes, o mundo onde o agente atuará e o comportamento de cada um deles. Esses recursos tornam a programação mais intuitiva.

A ferramenta Gallery, (Figura 2.7), permite que o usuário crie a representação dos agentes que serão utilizados na aplicação. Possui, também, as seguintes funções (Repenning, 1991):

- Editar as representações (Edit depictions): uma representação consiste em um bitmap e um nome que podem ser editados com um editor de representação. Dessa forma a representação do agente pode ser mudada em pequenos detalhes.

- Clonar representações: uma nova representação é criada por clonagem de uma já existente. No caso mais simples, a clonagem envolve apenas a cópia. Transformações extras podem ser adicionadas na clonagem. O conjunto de operações

Figura 2.7: AgentSheets - ferramenta Gallery.

da clonagem atualmente contém: operações unárias (cópia, giros múltiplos de 90 graus, inverter horizontalmente ou verticalmente e outras) e operações lógicas (e, ou, x-ou).

- Re-clonar representações: a modificação de uma representação pode ser propagada às representações dependentes através de re-clonagem deles.

- Palette: mostra as instâncias dos agentes. Nela são mostradas várias representações do agente, para que o usuário possa escolher a mais adequada.

- Guarda e mostra as representações (save and load depictions): permite que as representações sejam escondidas ou mostradas. Assim cada agente possui a sua lista de representações que pode ou não estar visível. Na Figura 2.7, o primeiro agente está com as representações visíveis, já o segundo não.

- Linkar representações para classes: toda representação é associada com uma classe de agente. Este link é usado quando instanciamos agentes.

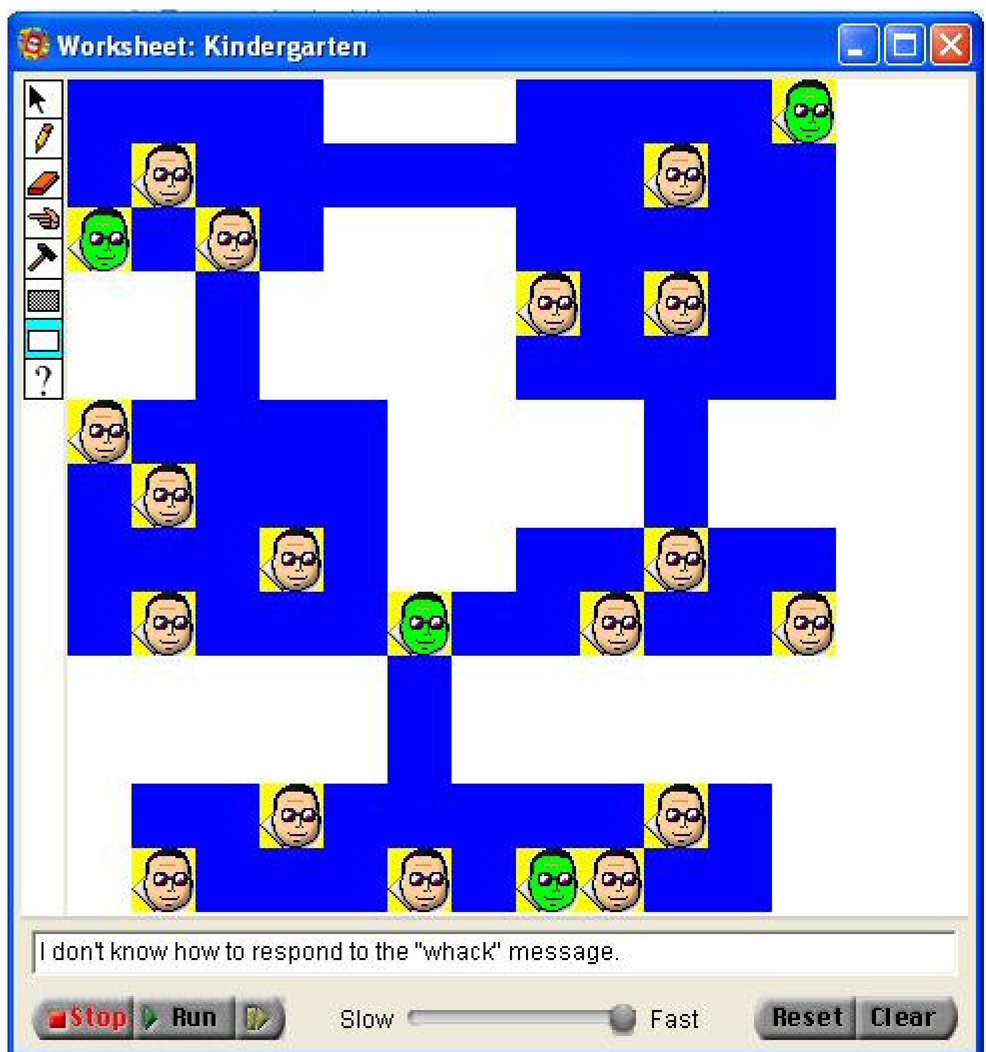


Figura 2.8: AgentSheets - ferramenta Worksheet.

- Editar o comportamento do agente (edit behavior): apresenta uma ferramenta onde o comportamento do agente pode ser editado.

A ferramenta Worksheet, mostrada na Figura 2.8, permite criar o mundo onde os agentes atuam. Ela possui uma barra de ferramentas que permite a realização de várias tarefas, como por exemplo: mover os agentes pelo worksheet, permitir que os usuários acrescentem novos agentes ou apaguem os já existentes e verificar quais são os atributos de cada agente.

Depois de definidos os agentes e o mundo onde eles atuarão, é preciso definir o comportamento de cada agente. Para isso, cada agente possui um editor de

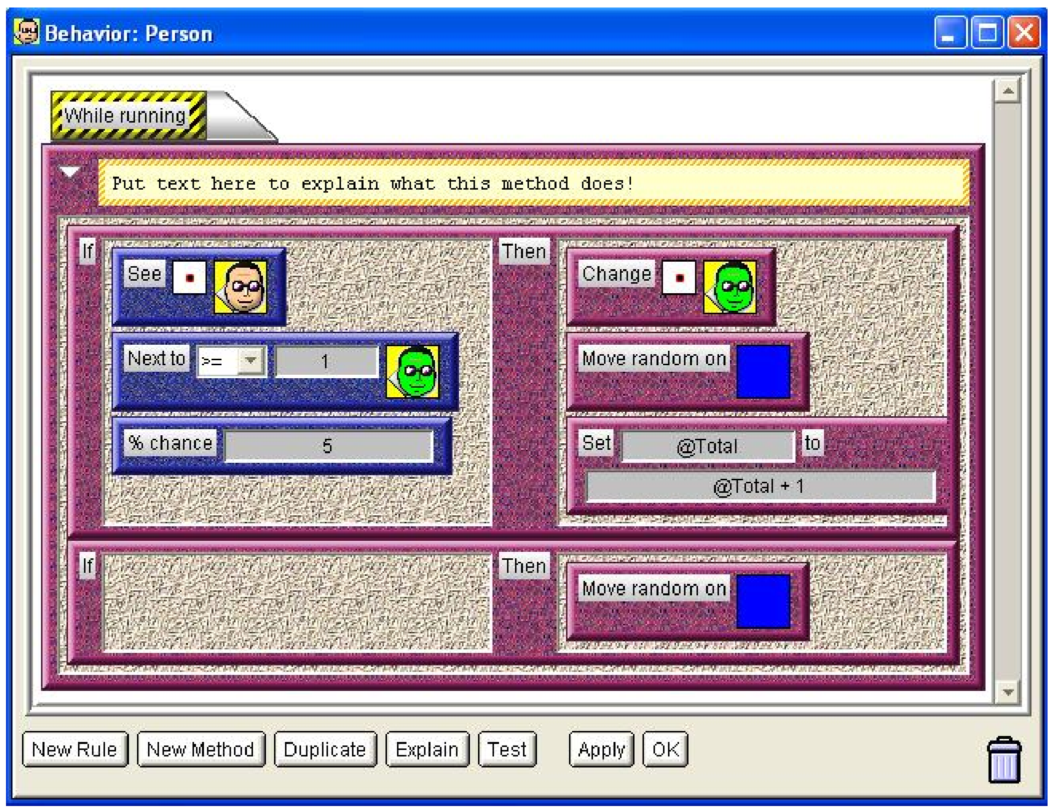


Figura 2.9: AgentSheets - editor de comportamento para cada agente.

comportamento (behavior editor). Esse editor é mostrado na Figura 2.9. Nele o comportamento do agente é criado por meio da construção de regras. As regras criadas para um agente descrevem o que ele faz e são formadas por condições e ações. Essas regras podem ser agrupadas em métodos. Os métodos, frequentemente, indicam uma tarefa que o agente poderia executar. O comportamento é expresso na linguagem Visual AgenTalk, como já foi visto na seção 2.2.1, por meio de regras IF-THEN contendo ações e condições. Para programar utilizando esta linguagem basta mover as condições e ações para as regras dos agentes localizadas em seu editor de comportamento (behavior editor).

Pelo editor é possível criar novas regras e novos métodos. Cada regra pode conter mais de uma condição e/ou mais de uma ação. As condições são ligadas através de um AND, para que todas as ações de uma regra possam ser executadas todas as condições devem ser verdadeiras.

As condições e ações utilizadas nas regras se encontram na janela Conditions

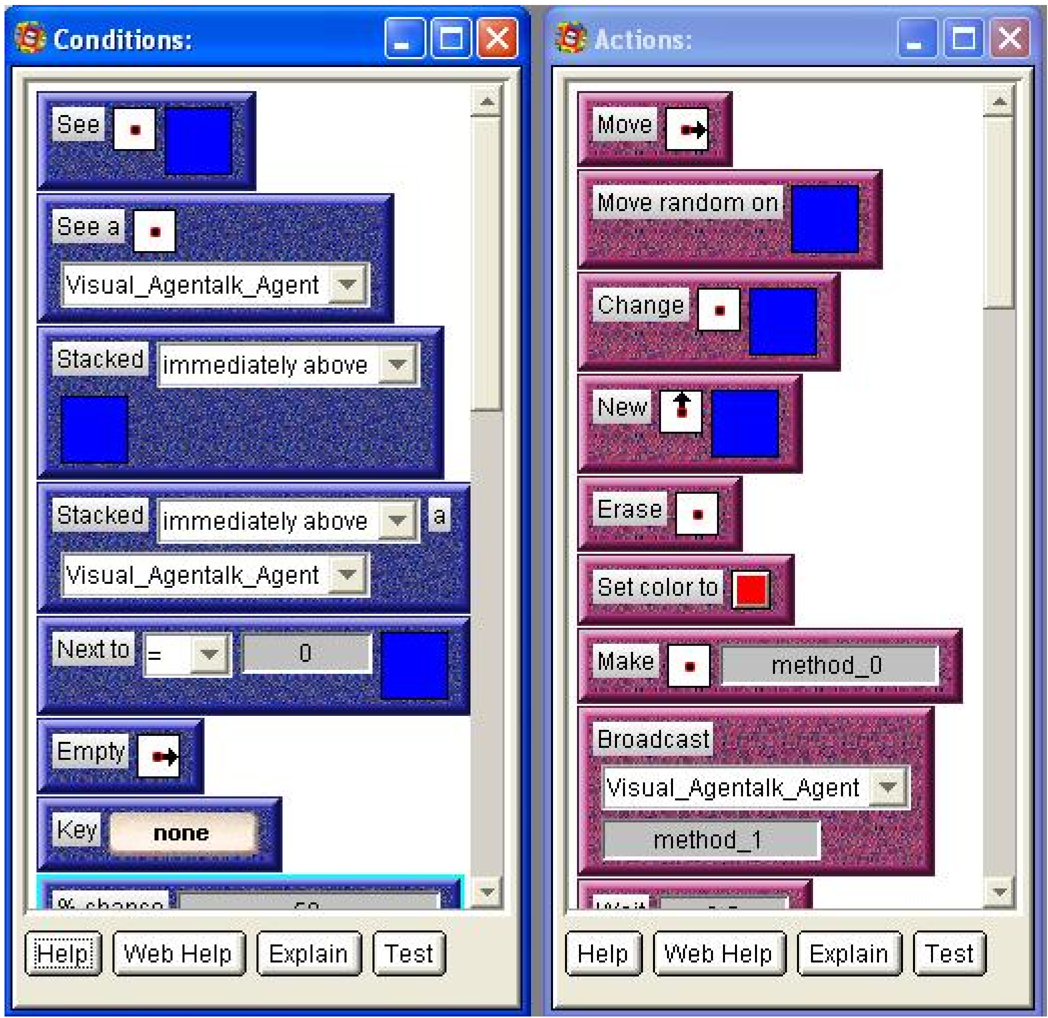


Figura 2.10: AgentSheets - janelas de condições e ações para serem usadas na criação das regras.

Palette e na janela Actions Palette, respectivamente. Essas janelas podem ser vistas na Figura 2.10. Para utilizar as condições ou ações basta selecionar a desejada e arrastá-la para a regra onde será utilizada. As condições e ações possuem parâmetros que podem ser modificados de acordo com a necessidade.

Cada uma das janelas possui os seguintes comandos: Help, Web Help, Explain e Test. Por meio do comando Help obtém-se a descrição de cada condição ou ação. O comando Web Help permite uma explicação mais detalhada, com informações relevantes sobre a ação ou condição selecionada que se encontram em páginas da Internet. Ele está diretamente indexado com o manual de referência do AgentSheets.

O comando Explain possibilita uma explicação mais interativa. A ação ou condição é explicada por uma ferramenta animada, onde as partes da condição ou ação e a parte correspondente da explicação são realçadas ao mesmo tempo. O comando Test permite que condições ou ações sejam testadas antes de terminado o trabalho de programação. Para fazer o teste basta selecionar a ação ou a condição, selecionar o agente no worksheet e então executar o comando Test. Esses recursos tornam a programação mais eficiente, pois, à medida que programa, o usuário pode saber se o que está fazendo dará ou não certo. E fica mais fácil para o usuário entender e utilizar os comandos disponíveis no ambiente de programação.

À medida que as regras são inseridas em um método, elas aparecerem no final do conjunto de regras. Na hora da execução, é obedecida a ordem em que as regras aparecem, ou seja, será executada a primeira regra que possuir condições verdadeiras. Por isso a ordem em que as regras são criadas é importante para o funcionamento da aplicação. Depois que as regras estiverem criadas, a sua ordem pode ser alterada sem prejuizo algum e de forma rápida e prática. Basta usar o mouse para arrastá-las e colocá-las na ordem desejada.

Cada método de um agente é composto de uma ou mais regras. Cada método é nomeado com uma trigger. A janela Triggers, Figura 2.11, mostra o conjunto completo de triggers. O editor de comportamento dos agentes, onde estão agrupados os métodos, também possui o comando Test e Explain. Pelo comando Test os métodos podem ser testados, bastando selecionar o agente e o método. O sistema irá passar por todas as regras dando o retorno do que será executado em cada passo. E pelo comando Explain, os métodos podem ser explicados interativamente.

Outro recurso do AgentSheets é que neste ambiente pode-se fazer referência a um personagem sem se importar com o seu estado. A Figura 2.12 mostra uma regra onde não importa a posição do Pacman, se ele está virado para a direita, para a esquerda, para baixo ou para cima, importa apenas que se existir um Pacman na posição especificada ele será eliminado. Esse recurso elimina a necessidade de regras que exemplifiquem cada possível estado do personagem.

Figura 2.11: AgentSheets - janela Triggers.

### 2.2.3 Conclusão

O AgentSheets é uma ferramenta mais profissional que o Stagecast Creator. O código em Visual AgentTalk é compilado para a linguagem Java, de forma otimizada. O resultado são animações que podem ser executadas em velocidade muito maior que a proporcionada pelo interpretador do Stagecast Creator. Por outro lado, pode-se considerar que a linguagem do AgentSheets se aproxima mais de uma linguagem de programação convencional do que os recursos oferecidos pelo Stagecast Creator, que são mais intuitivos.

A exemplo do Stagecast Creator, também no AgentSheets as regras são escritas levando-se em consideração a direção do agente, são escritas em terceira pessoa. Como foi explicado na Seção 2.1.6 não é possível ensinar o personagem a simplesmente andar, é preciso ensiná-lo a andar para a direita, para a esquerda, para cima e para baixo. Isso aumenta o número de regras necessárias para se demonstrar uma ação.

Outro problema que essa ferramenta divide com o Stagecast Creator é a impossibilidade de aproveitar uma mesma regra em vários agentes que possuam comportamento semelhante. E o seu domínio também está limitado a jogos e simulações.

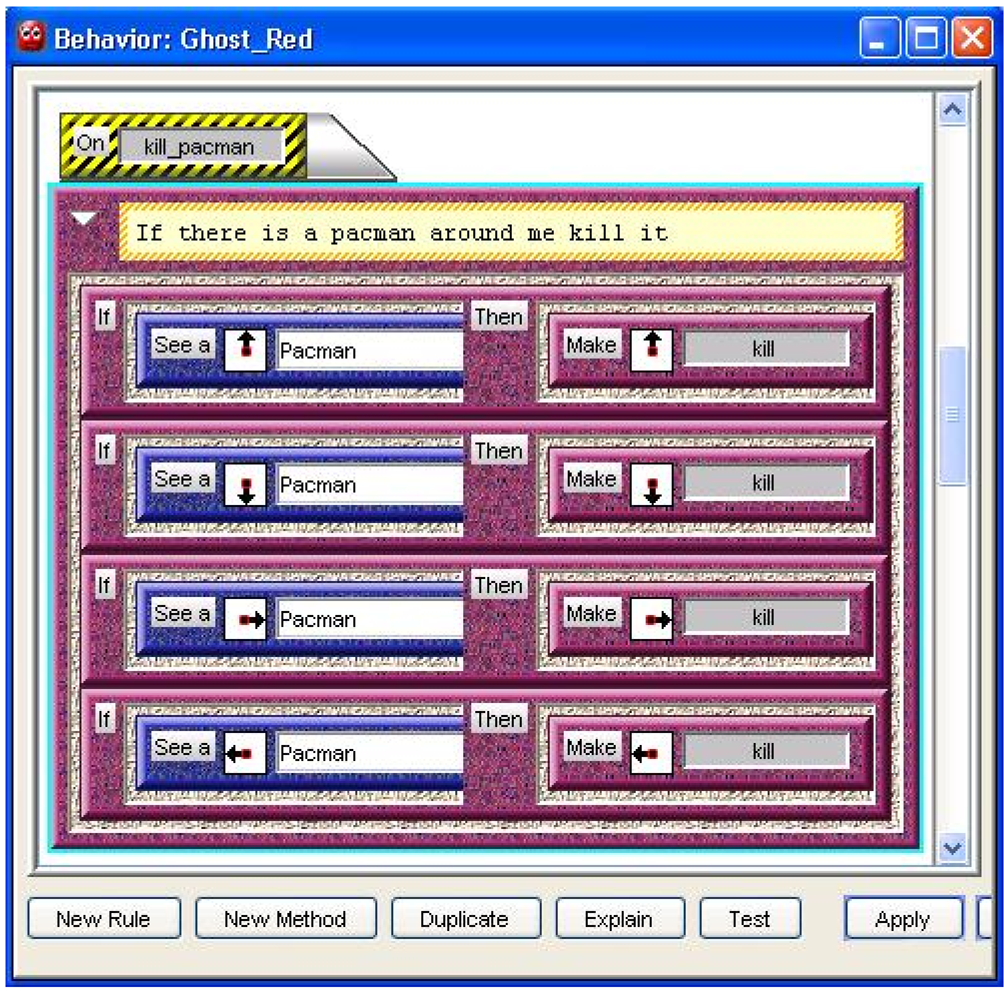


Figura 2.12: AgentSheets - referência ao personagem Pacman, sem se preocupar com seu estado.

## 2.3 Gamut

Gamut é uma ferramenta para não programadores, que utiliza as técnicas de programação por demonstração (PBD). O seu objetivo é que não programadores possam criar aplicações gráficas interativas, como softwares educacionais, simulações e, principalmente, jogos. O nome Gamut foi sugerido pelos seus autores como uma sigla para *Games Are Made Using This* (McDaniel and Myers, 1998).

A ferramenta Gamut é, essencialmente, um editor gráfico com habilidade de aprender por meio de exemplos. Possui técnicas de interação que permitem demonstrar facilmente os exemplos, tanto o que se deve fazer e o que não se deve. E tudo isso sem digitar uma única linha de código.

De acordo com McDaniel and Myers (1999), o sistema Gamut possui recursos que os que trabalham com um único-exemplo não possuem, ele é capaz de induzir comportamentos por meio de múltiplos exemplos e com isso criar comportamentos mais sofisticados.

Gamut começa construindo um novo comportamento do mesmo modo que os sistemas de único-exemplo. Porém, a cada vez que o desenvolvedor refina um comportamento, acrescentando um novo exemplo, o Gamut usa métricas para comparar o novo exemplo com o comportamento atual e escolher como o comportamento deveria ser modificado. Por exemplo, se o Gamut vê que parte de um código que era previamente executado não está sendo utilizado com o novo exemplo, ele colocará esse código em um comando if-then. Se o Gamut vê que um objeto começa com um valor diferente, usará o novo valor junto com heurísticas para escolher mudanças no código que produz aquele valor. O Gamut também usa um algoritmo de árvore de decisão para gerar código que não pode ser gerado apenas por meio de regras. Por exemplo, Gamut usa árvores de decisão para representar os predicados dos comandos if-then.

Gamut usa técnicas de interação para suportar seus algoritmos. Nele, o desenvolvedor pode criar objetos “guia” para representar o estado da aplicação que não faz parte de sua interface visível. Esses objetos são visíveis em tempo de design mas desaparecem em tempo de execução. O desenvolvedor também pode realçar obje-

tos em ordem para dar sugestões ao sistema. As sugestões guiam os algoritmos de heurística do Gamut quando eles selecionam relações para colocar em um comportamento. Gamut pode também requerer que o desenvolvedor destaque objetos quando os algoritmos chegarem a um impasse e não possam deduzir o código sem a sua ajuda.

De acordo com McDaniel and Myers (1998), os sistemas que induzem por meio de exemplos múltiplos, estão aptos a reconhecerem exemplos positivos e negativos. Um exemplo positivo demonstra uma condição quando um comportamento deve acontecer, e um exemplo negativo mostra uma condição quando o comportamento não deve acontecer. Sem exemplos negativos, um sistema não pode deduzir muitos comportamentos, como os que usam operadores booleanos. No Gamut, o desenvolvedor pode usar o botão **Do Something** ou o botão **Stop That** para criar um novo exemplo. Essas escolhas correspondem aproximadamente a criar um exemplo positivo ou um negativo.

Segundo McDaniel and Myers (1998), os aplicativos feitos em Gamut são como jogos de tabuleiro computadorizados. Esses aplicativos são bidimensionais e são compostos de um cenário fixo (tabuleiro) e objetos que se movem sobre ele. A Figura 2.13 mostra um exemplo de jogo criado com o Gamut. Esse jogo é baseado no vídeo-game Q*Bert. Nesta versão, o objetivo do jogador é, controlando o personagem, fazer com que ele pule sobre todos os cubos do cenário e consiga se desviar das bolas verdes que caem do topo da tela. Se uma bola acertar o personagem, ele perde uma vida. Esse exemplo pode ser considerado um jogo de tabuleiro pela definição do Gamut. Os cubos do cenário formam o tabuleiro ao passo que o personagem e as bolas são as peças que interagem com o cenário e entre si.

Os comportamentos, em jogos de tabuleiro, tendem a ser condicionais. Eles dependem de certas situações como, por exemplo, a vez do atual jogador. O Gamut é capaz de abranger objetos independentes em seus comportamentos para produzir dependências entre as peças e as várias situações no jogo.

As peças de um jogo tendem a seguir caminhos e possuir outras conexões com o tabuleiro e estes caminhos nem sempre são visíveis. O sistema deve providenciar uma forma de o desenvolvedor mostrar as informações que são normalmente escondidas

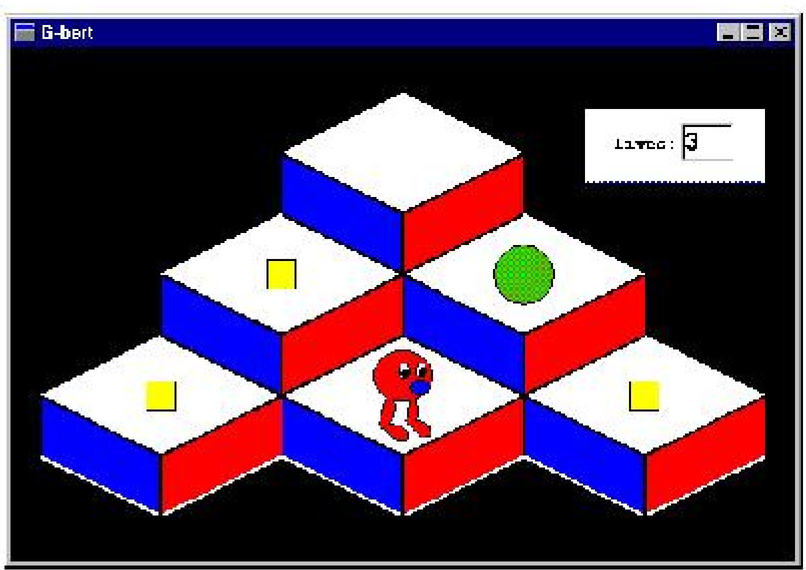


Figura 2.13: Jogo do tipo Q*Bert criado totalmente no Gamut.

quando o jogador vê o jogo. Por isso o sistema deve ser capaz de se referir a objetos indiretamente através de uma cadeia de descrições. Por exemplo, uma descrição para o local de destino de um peão no jogo Banco Imobiliário pode ser dada da seguinte forma: "irá para o quadrado que é o número do dado de quadrados a partir da posição do atual jogador". Essa descrição se refere aos muitos diferentes elementos do jogo, incluindo a configuração do tabuleiro, o número do dado, e a vez do jogador. Cada elemento desta descrição depende do resultado dos outros objetos. O Gamut é capaz de encadear múltiplas descrições para formar descrições complexas.

O Gamut pode produzir regras de jogo de tabuleiro, mas geralmente não pode gerar oponentes controlados pelo computador. Alguns jogos iriam requerer estratégias complexas que seriam muito difíceis de ensinar ao sistema. Estratégias simples são fáceis de serem ensinadas através do Gamut, mas estratégias complexas como as necessárias para um jogo de xadrez saem fora de seu domínio. Em um jogo de xadrez seria necessário ensinar ao computador ler a situação atual do tabuleiro e decidir qual seria a melhor jogada a ser realizada. Essa tarefa exigiria muitos conhecimentos e programação usando técnicas de IA (Inteligência Artificial), pois teria que ser preenchida uma base de dados contendo as possíveis jogadas de cada peça do tabuleiro e ensinar ao computador como ganhar o jogo. Isso foge do escopo desse ambiente de

programação.

O Gamut possui técnicas projetadas para permitir que o desenvolvedor mostre exemplos ao sistema de forma rápida e fácil e sem precisar aprender muitos novos conceitos (McDaniel and Myers, 1998). Essa rapidez e facilidade permitem ao desenvolvedor demonstrar um grande número de exemplos, reduzindo a importância de cada um deles. Isso permite que o sistema ignore elementos dos primeiros exemplos demonstrados se, naquela hora, eles não parecerem muito relevantes.

Os comportamentos demonstrados no Gamut podem ser incrementados. Comportamentos que dependiam de condições indisponíveis no momento da primeira demonstração podem ser completados, mais tarde, com outros exemplos que gerarão conflitos com o atual comportamento.

Ao desenvolvedor cabe o papel de demonstrar, através de exemplos, como o jogo funciona, e quando o sistema cometer um erro ao executar um comportamento que ainda não está completamente formado, o desenvolvedor deve reconhecer que um erro ocorreu e mostrar como deveria ser o comportamento a ser executado.

### 2.3.1 Conclusão

Projetar heurísticas para um sistema de PBD é uma tarefa muito difícil, pois seria necessário simular o mecanismo de inferência humano. O que se deseja é que os sistemas possam fazer inferências corretas. Para isso se torna necessário o uso de mecanismos de inferências sofisticados, que por sua vez trazem junto elaborados usos de interfaces para controlar a inferência, o que faz com que os sistemas se tornem mais difíceis de implementar. O importante é que em qualquer sistema PBD exista um mecanismo de avaliação bem projetado que permita que os usuários possam entender e controlar o que o sistema está fazendo e possam mudar o programa depois. Mas ainda necessita-se de muito mais pesquisa para se chegar a um nível apropriado de inteligência e de formas de avaliação em sistemas de PBD. O sistema Gamut tem habilidade para inferir complexos comportamentos que podem ser usados para construir aplicações interativas completas. Essa nova capacidade vem de uma coleção de técnicas de interação que induzem ao aprendizado.

Essa indução ao aprendizado permite a geração automática de regras, que nas outras ferramentas, deveriam ser construídas à mão, uma a uma. Apesar dessa vantagem, o Gamut não oferece ao usuário uma maneira de saber qual o resultado das inferências, ou seja, quais regras foram automaticamente geradas. A única forma que o usuário tem de obter alguma informação a respeito é executar o sistema em exemplos reais e observar o seu comportamento em situações específicas. Ele possui também a capacidade de rever um comportamento já deduzido depois que novos exemplos são demonstrados. Essa capacidade pode se tornar uma vantagem ou desvantagem, pois da mesma forma que ele pode consertar comportamentos deduzidos erradamente, ele pode estragar outros que estavam deduzidos corretamente.

## 2.4 ToonTalk

Um sistema que utiliza as técnicas de programação por demonstração proporciona aos seus usuários a facilidade de construir programas por meio de exemplos de entrada, sem a necessidade de digitar uma única linha de código. O problema do uso dessa técnica é a dificuldade de se ter mecanismos que permitam ao usuário generalizar as ações necessárias.

Segundo Kahn (1996), o ToonTalk é um sistema que utiliza as técnicas de programação por demonstração e que permite que o programador generalize as ações gravadas removendo, explicitamente, seus detalhes.

Em ToonTalk, o programa é executado como uma coleção de processos autônomos que se comunicam assincronicamente, e o comportamento do processo é especificado como um conjunto de cláusulas que são guardadas. Uma cláusula é construída pela execução de operações em uma única estrutura de dados de exemplo. Para fazer com que a cláusula seja capaz de operar em outras estruturas de dados, o programador precisa, apenas, remover detalhes do que está escondido ou a parte condicional da cláusula.

O ToonTalk é construído com a idéia de programação animada. Programas animados não são construídos pela digitação de texto ou pela construção de diagramas

ou seqüência de figuras. Em vez disso, o programador é colocado como um personagem dentro do mundo virtual animado onde as abstrações da programação são substituídas por comandos tangíveis. A Figura 2.14 mostra alguns exemplos das abstrações da programação e dos seus correspondentes objetos concretos criados para o ambiente do ToonTalk. Uma estrutura de dados, por exemplo, é uma caixa cujos buracos podem ser preenchidos com números, pedaços de texto, outras caixas, pássaros, ninhos e robôs. Pássaros e ninhos são analogias concretas da capacidade de enviar e receber dados em canais de comunicação. Um robô é uma estrutura treinada pelo programador para entrar em ação quando receber uma caixa. O pensamento bolha de um robô mostra as condições que precisam ser satisfeitas antes do robô executar uma ação. Para generalizar um robô, um programador precisa usar um aspirador de pó animado para remover os detalhes de dentro da bolha de pensamento do robô.

### 2.4.1 Histórico do ToonTalk

Segundo Kahn (2001), as crianças que conseguem programar podem transformar os computadores em jogos eletrônicos, simuladores, geradores de arte ou de música, bancos de dados, animações, controladores de robôs e muitas outras coisas que os programadores profissionais fazem.

Como programar é considerada uma tarefa difícil para se aprender e se realizar, principalmente para adolescentes e crianças, o ToonTalk começou com a idéia de que a tecnologia de animação e de jogos de computadores poderia tornar a programação uma tarefa mais fácil e divertida. Nesse caso, em vez de digitar programas textuais no computador ou utilizar o mouse para construir programas gráficos, a programação avançada passa a ser feita dentro de um mundo virtual animado e interativo.

Segundo Kahn (1996), os construtores do ToonTalk se esforçaram para seguir os seguintes princípios de concepção obtidos a partir dos bons jogos de vídeo game ou de construção, como o Lego:

1. Fazer com que a primeira experiência seja simples, aumentando a complexidade gradualmente;

| Computational Abstraction | ToonTalk Concretization |
|---|---|
| Computation | City |
| Actor<br>Process<br>Concurrent object | House |
| Method<br>Clause | Robot |
| Guard<br>Method preconditions | Contents of thought bubble |
| Method actions<br>Body | Actions taught to a robot |
| Message<br>Array<br>Vector | Box |
| Comparison test | Set of scales |
| Process spawning | Loaded truck |
| Process termination | Bomb |
| Constants | Numbers, text, pictures, etc. |
| Channel transmit capability<br>Message sending | Bird |
| Channel receive capability<br>Message receiving | Nest |
| Persistent storage<br>File | Notebook |

Figura 2.14: Termos da Ciência da Computação e termos do ToonTalk.

2. Incentivar a exploração e a curiosidade;
3. Apresentar e manter fantasias interessantes;
4. Ser um desafio constante, mas não frustrante;
5. Utilizar, frequentemente, princípios e técnicas de cinema e de animação (esta idéia provém exclusivamente dos jogos de vídeo).

Eles recorreram também à tecnologia dos jogos de vídeo-game. Por exemplo: imitaram a forma de colocar o jogador no mundo do jogo, disponibilizando um personagem, que o jogador controla e com o qual se identifica. Isso acaba por estimular as crianças. No ToonTalk, o programador é um personagem animado que constrói, testa e depura os programas.

Já ficou provado que a parte mais difícil da programação é a concepção dos programas. É onde as crianças e os adultos encontram os maiores problemas. Construí-los é, geralmente, uma seqüência de tarefas monótonas, como escrever ou corrigir erros de sintaxe. O ToonTalk é distinto dos outros ambientes de programação por fazer com que o processo de construção de um programa seja uma brincadeira divertida. Uma pessoa que utiliza o ToonTalk pela primeira vez para construir programas, pode não ser tão criativa quanto na construção de um modelo de avião ou quando segue as instruções para construir um brinquedo utilizando o Lego. Mas mesmo assim essa atividade é divertida. Uma forma muito eficaz de fazer com que as crianças aprendam a conceber algo é a de brincar com as boas concepções de outras pessoas.

O ToonTalk foi construído para ser um ferramenta de programação fácil de se aprender, mas também muito poderosa e flexível. Um dos motivos é que ele pega os conceitos da programação e os substitui por analogias concretas do dia-a-dia, como por exemplo, canais de comunicação são pássaros e ninhos. O seu grande desafio é pegar uma boa concepção de linguagem, de orientação teórica, que seja pequena e poderosa e encontrar um conjunto coerente de concretizações, que envolvam todos os elementos da linguagem.

Os programas em ToonTalk, embora construídos de forma interativa e animada, podem ser traduzidos para linguagens de lógica concorrente por restrições, em formas

textuais equivalentes.

A programação que utiliza lógica concorrente por restrições foi escolhida como base global para o ToonTalk porque depois de muitos anos de utilização ficou demonstrado que não há qualquer risco de que a linguagem seja inadequada à construção de uma grande variedade de programas complexos (Shapiro, 1989). As linguagens são pequenas, mas muito poderosas. Estes fatos fizeram com que a concepção do ToonTalk fosse muito mais simples do que se fosse baseada nas linguagens convencionais.

O mundo do ToonTalk se parece com uma cidade do século vinte. Existem helicópteros, caminhões, casas, ruas, bombas de bicicleta, caixas de ferramentas, aspiradores de pó, caixas, e robôs. Existem também os pássaros e seus ninhos.

Como já foi dito, o usuário do ToonTalk é um personagem em um mundo animado. Ele começa voando em um helicóptero por cima da cidade. Depois de pousar, ele controla uma pessoa em uma tela. A pessoa é seguida por um cachorro com uma caixa de ferramentas cheia de coisas úteis para a programação.

Uma computação completa, em ToonTalk, é representada na forma de uma cidade. A maioria das ações ocorrem nas casas. Os pássaros pombo correio providenciam a comunicação entre as casas. Os pássaros podem voar para o ninho, deixar coisas lá e voar de volta. Tipicamente, as casas contêm robôs que são treinados para realizar pequenas tarefas. Para treinar um robô, basta entrar em sua "bolha de pensamento" e lhe mostrar o que ele deve fazer.

Um robô se comporta exatamente como o programador o treinou. Esse treinamento corresponde, em termos da ciência da computação, a definir o corpo do método em uma linguagem de programação orientada a objeto, como Java ou Smalltalk. Um robô pode ser treinado para:

- enviar uma mensagem dando uma caixa ou bloco a um pássaro;
- gerar um novo processo derrubando uma caixa e um time de robôs em um caminhão;
- executar simples operações primitivas como adição ou multiplicação construindo uma pilha de números;

- copiar um item usando uma varinha de mágico;
- mudar uma estrutura de dados levando itens para fora da caixa e trazendo novos itens;
- terminar um processo disparando uma bomba.

### 2.4.2 Código-fonte animado

De acordo com Kahn (1996), a idéia fundamental por trás do ToonTalk é que o seu código fonte seja animado. Inclusive o seu nome, ToonTalk, significa "falar" por meio de bonecos, cartoons. Isto quer dizer que é por meio da animação que se comunica aos seres humanos e aos computadores todo o significado dos programas criados. O código fonte animado possui inúmeras vantagens, mas a construção das animações pode ser difícil e demorada. É mais difícil animar uma ação do que descrevê-la simbolicamente.

Por meio da animação dos jogos de vídeo game, as crianças podem produzir um conjunto de animações sofisticadas. Por exemplo, se um programa necessitar trocar valores presentes em dois locais, nada mais natural e simples do que pegar o conteúdo de um dos locais, guardá-lo, pegar o conteúdo do outro, colocá-lo no primeiro local e depois colocar o primeiro objeto no segundo local, como na Figura 2.15. Essa ação pode ser entendida e executada pelas crianças, enquanto que escrever o código abaixo, que é o equivalente a essas ações, só está ao alcance de pessoas que possuem conhecimento de programação:

```
temp := x;
x := y;
y := temp;
```

### 2.4.3 Exemplo utilizando o ToonTalk

Um problema simples da computação é o de dobrar números e somá-los sucessivamente. No ToonTalk, basta que a criança ensine o robô a fazer isso. Para isso, ela mostra o que o robô deve fazer com o número 1. Ou seja, ela mostra que ao encontrar o número 1, ele deve fazer uma cópia deste número, realizar a soma do número

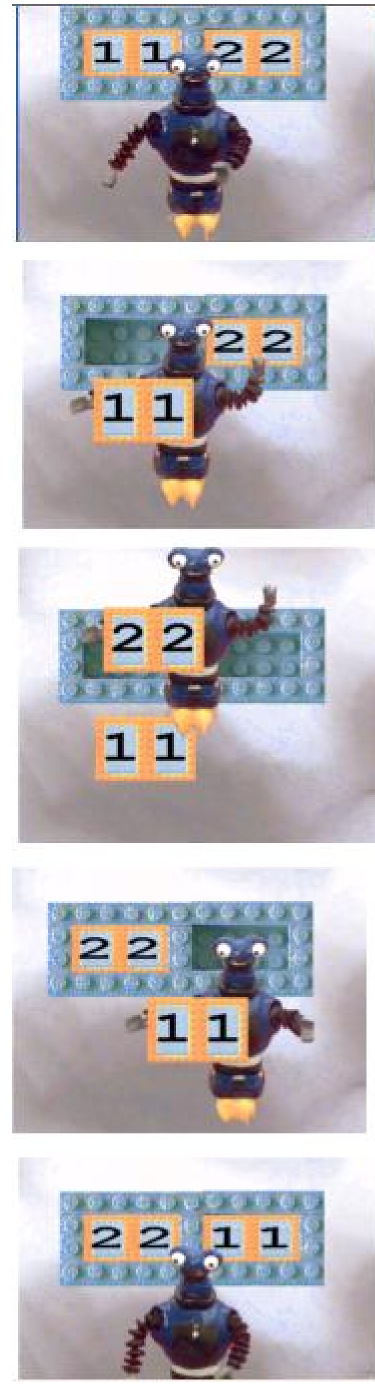


Figura 2.15: Troca de números utilizando o ToonTalk.

inicial com a a sua cópia (1 + 1), ao obter o resultado, 2, este deve ser guardado. Mas como a criança mostrou ao robô o que fazer com o número 1, ele só consegue agir quando encontra este número em sua caixa, pois é isso que existe na sua bolha de pensamento. Por isso ele não sabe, por exemplo, o que fazer quando encontra o número 2 em sua caixa. A criança então precisa apagar o número 1 da sua bolha de pensamento para generalizar as suas ações. Assim o robô conseguirá realizar esta operação com qualquer número, pois ficaram gravadas as ações que ele deve realizar (dobrar o número, somar e guardar o resultado) sem ficar especificado com qual número elas serão realizadas. O robô saberá apenas que irá receber um número, não se importará com qual. Esse problema da computação é resolvido sem a necessidade de se digitar uma só linha de código.

Ensinar o robô a ordenar uma seqüência de números é uma tarefa complicada. Todas as tentativas de ensiná-lo a ordenar os números acabam por conseguir treiná-lo, apenas, para fazer a permutação dos números. Ele irá trocar os números de lugar da mesma forma que foi treinado a fazer sem se importar com o valor de cada número. Isto acontece porque o ToonTalk não contém heurísticas de generalização de programas. Ele possui um número pequeno de regras muito simples de generalização. Com o treino do programador, um robô se move, pega e derruba coisas e usa ferramentas para copiar ou remover coisas. Eles se lembram das ações para as quais eles foram treinados com base na posição e na história dos objetos. Robôs ignoram o caminho e o tempo das ações, os rótulos (eles são como comentários em linguagens textuais) e outros detalhes.

### 2.4.4 Conclusão

De acordo com Cypher et al. (1993), as pesquisas realizadas provam que a programação por exemplo possui uma enorme vantagem sobre as linguagens convencionais.

O desenvolvimento de programas é muito mais fácil em ToonTalk por várias razões. Uma delas é que o cérebro humano está mais preparado para trabalhar com exemplos do que com abstrações. Cada cláusula do ToonTalk (i.e., robô) é uma unidade auto-suficiente cujo teste condicional é gerado automaticamente pelo sistema

e relaxado, se necessário, pelo programador.

Para se apoiar a programação por demonstração, é necessário o uso de um modelo de computação apropriado. O ToonTalk não tem nenhum teste condicional aninhado, nem teste condicional complexo e nem chamada de subrotina. Mas mesmo assim é uma linguagem de programação de alto-nível muito expressiva. Cada uma destas abstrações de programação que já estão difundidas na programação convencional interferem no processo de programar através de exemplos (Kahn, 2001). O poder e a simplicidade do ToonTalk estão no fato de o robô apenas executar uma seqüência de linhas de código sem condicionais ou chamadas de procedimento (entretanto este código pode incluir a geração de processos e a comunicação entre processos). A referência a variáveis é solucionada na caixa do robô ou com algumas variáveis temporárias locais que são concretizadas como algo que colocaram no chão.

Em ToonTalk, uma chamada de procedimento é apenas um padrão particular de geração de processos e é usada na comunicação de canais para devolver resultados. Um programador de ToonTalk freqüentemente cria pares de pássaros e ninhos e usa os pássaros para entregar valores computados para os ninhos. Se um programador de ToonTalk precisa computar algo antes de realizar alguma ação, então ele divide a tarefa entre dois robôs, um para gerar o processo para fazer o cálculo e o outro que usa o resultado depois que ele foi recebido.

No ToonTalk, o processo de programar é mais tangível, pois evita usar ícones e descrições e providencia analogias animadas e concretas de todas as abstrações computacionais. Tudo nele pode ser visto, apanhado e manipulado. Até mesmo operações como copiar e apagar estruturas de dados são expressas usando ferramentas animadas como a Magic Wand (animação de uma varinha mágica, capaz de copiar estruturas de dados) e Dusty the Vacuum (animação de um apirador de pó, utilizado para apagar dados ou estruturas de dados).

Segundo Kahn (2001), o Stagecast Creator é somente outro sistema de programação por demonstração projetado para crianças. O Creator confia em representações analógicas. "Programação é feita nos termos do domínio, como locomotivas e estradas de ferro, mais que em termos computacionais, como arrays e vetores" (Smith et al.,

2000). Em contraste, ToonTalk é programado em termos do computador, com o fato de que esses termos são "traduzidos" para objetos familiares e tangíveis tais como caixas, pássaros, e robôs. Por conseguinte, pode-se dizer que o ToonTalk é mais geral e poderoso, pois não se foca apenas na facilidade de uso, como o Creator faz, perdendo assim em generalidade e poder.

De acordo com Kahn (1996), a realização de testes, em pequena escala, com o ToonTalk mostrou que embora as crianças dominem facilmente os seus objetos e as suas ferramentas, ou seja, elas acabam dominando as "primitivas" da linguagem de programação, elas precisam de orientação para poder avançar no sistema, para fazer programas úteis. O sistema está atualmente sendo melhorado, incorporando exemplos, demonstrações, folhas de instruções e uma iniciação interativa, para permitir que elas possam avançar por si só.

Com o estudo da ferramenta ToonTalk foi possível perceber que ela realmente trabalha com analogias tangíveis dos termos usados na programação, mas é preciso tomar muito cuidado pois isso pode tornar a programação mais difícil para um não programador. O principal objetivo de um ambiente que utilize as técnicas de PBD é que ele seja fácil de operar, como visto nos exemplos da Seção 2.4.3, no ToonTalk nem sempre isso é intuitivo. O usuário teria que aprender vários conceitos de programação para conseguir trabalhar com a ferramenta. Talvez ele se torne interessante para ajudar alunos de programação a entenderem como os conceitos envolvidos em uma programação funcionam.

## 2.5 Conclusão

Os ambientes de programação que utilizam as técnicas de programação por demonstração trazem benefícios para os seus usuários. Tornam a programação mais divertida, intuitiva e livre dos conceitos de uma linguagem sintática. São ótimos ambientes para serem utilizados por crianças em fase de aprendizado e de desenvolvimento do raciocínio lógico ou método científico.

Os quatro sistemas abordados neste capítulo utilizam as técnicas de progra-

mação por demonstração, e por isso possuem todas as vantagens de seu uso. São sistemas mais intuitivos, não exigem que o programador tenha conhecimento de uma linguagem sintática.

O Stagecast Creator é um ambiente muito intuitivo e totalmente gráfico. Nele o usuário não escreve nem uma linha de código e pode programar sem possuir nenhum conhecimento sobre a programação convencional. Utiliza também as técnicas de **Regras Visuais Antes-Depois**. Possui recursos para a depuração do programa e tudo é feito de forma muito visual. As regras precisam ser criadas para cada direção de ação do personagem, mesmo que todo o restante da regra seja igual. Isso provoca um grande número de regras criadas para ensinar coisas básicas, como por exemplo, andar. Caso mais de um personagem utilize a mesma regra, ela precisa ser criada para cada um deles, não existe um mecanismo de aproveitar regras já criadas. O seu domínio é restrito à criação de jogos e simulações.

O AgentSheets é um sistema intuitivo e gráfico, mas as suas regras são criadas seguindo o formato das estruturas condicionais das linguagens convencionais. Combina regras de **PBD** com **Regras de Reescrita Gráfica**. Não exige conhecimento prévio do usuário porque apresenta tudo em um formato bem visual. As simulações criadas nele podem ser exportadas como applets interativos de Java. Utiliza uma linguagem de programação chamada de Visual AgenTalk (VAT). Existe o conceito de modularização, que permite o reaproveitamente de código/regras. Também possui ferramentas gráficas que auxiliam na depuração das simulações criadas. Da mesma forma que no Stagecast Creator, as regras precisam ser criadas para cada direção de ação do personagem, mesmo que todo o restante da regra seja igual. E não existe uma forma de utilizar uma regra que já foi criada para outro personagem. O seu domínio também é restrito à criação de jogos e simulações.

O Gamut é um ambiente gráfico de programação que possui características de Inteligência Artificial, pois pode realizar inferências. Esse recurso permite que a cada regra criada pelo usuário o sistema possa deduzir comportamente e gerar automáticamente outras regras. Um problema encontrado é que ele não possui mecanismos para que o usuário veja quais foram as regras geradas automaticamente. Isso facilita

a realização de inferências equivocadas.

O ToonTalk, diferentemente dos outros sistemas, utiliza a idéia de programação animada. O programador é colocado como um personagem do mundo virtual. As abstrações da programação são substituídas por comandos tangíveis. Exige um pouco mais de conhecimento ou de aprendizado do usuário, pois a sua forma de criar as animações não é tão natural quanto nos outros sistemas. Pode acabar confundindo o usuário leigo, pois exige um conhecimento das abstrações das linguagens de programação convencionais. Permite que sejam criados programas envolvendo os problemas clássicos da computação, como ordenar números, mais o seu desenvolvimento é mais complicado do que o código convencional.

Os quatro sistemas estudados permitem aumentar o número de pessoas capazes de desenvolver simulações, pois não exigem conhecimentos de programação. Mas como visto nas Seções 2.1 e 2.2, os sistemas Stagecast Creator e AgentSheets, em algumas situações, exigem que sejam criadas várias regras praticamente iguais. Essa exigência aumenta o tempo gasto pelo usuário, aumenta o seu trabalho e pode acabar por desestimulá-lo. Se os sistemas puderem oferecer recursos para eliminar o trabalho braçal, a área de abrangência desses ambientes aumenta. Eles podem ser usados por profissionais que procuram por formas mais rápidas de desenvolverem o seu trabalho e não perdem a sua simplicidade, nem o seu público leigo em programação.

Algumas sugestões para eliminar parte do trabalho braçal são mostradas no próximo capítulo.

# Capítulo 3

# Sugestões para ambientes que utilizam PBD

O capítulo anterior apresentou ambientes de programação onde não programadores podem criar suas simulações ou jogos. São ambientes utilizados, principalmente, em um ambiente educacional e possuem como objetivo ajudar os alunos nos estudos das disciplinas obrigatórias e estimular o raciocínio lógico. Eles não possuem como objetivo resolver problemas convencionais da computação, nem são usados para sistemas comerciais.

O que não impede que o uso desse tipo de ambiente de programação seja difundido para que possa ser utilizado por um número maior de pessoas e para outros fins. Ele poderia ser utilizado em escolas técnicas para introduzir os conceitos de lógica, por profissionais autônomos que quisessem ou precisassem criar simulações, por programadores que quisessem criar jogos simples ou auxiliar na programação de robôs.

Em todas essas situações pode-se utilizar a programação convencional com grande sucesso, mas a programação por demonstração permite uma programação mais simples, mais gráfica e por isso mesmo mais agradável para muitas pessoas. Ela não tem como objetivo substituir a linguagem convencional, nem resolver os mesmos problemas.

O estudo dos ambientes já existentes mostrou que o Stagecast Creator e o

AgentSheets são ambientes mais fáceis de serem utilizados por pessoas leigas e que possuem recursos para serem utilizados como um meio mais fácil de programadores criarem as suas simulações ou jogos.

Mas alguns problemas foram levantados durante esse estudo. Primeiro problema, no caso de um personagem precisar se locomover nas quatro direções (direita, esquerda, para cima e para baixo) é necessária a criação de quatro regras, sendo que a única diferença entre elas é a direção do personagem, todo o restante, condição e ação, permanecem iguais. São necessárias quatro regras, onde apenas uma poderia resolver o problema. Segundo problema, vários personagens podem realizar a ação de subir em uma pedra. A regra é a mesma, só muda o personagem que pode realizar a ação, mas a regra precisa ser criada para cada personagem. Isso aumenta o número de regras a serem criadas e consequentemente o tempo gasto na tarefa de criar apenas um tipo de regra. E o tempo gasto em possíveis modificações nessas regras também será grande, pois será necessário modificar as regras de todos os personagens. Terceiro problema, o personagem pode, em um ambiente com dez tipos de frutas, comer nove tipos. É necessário criar nove regras para explicar este comportamento para o personagem. Seria mais fácil se fosse possível especificar apenas a fruta que ele não pode comer.

Por esses motivos, este capítulo apresenta sugestões de melhorias para os ambientes que utilizam as técnicas de PBD. As sugestões visam diminuir o tempo e o número de regras gastas na programação de simulações e jogos e com isso tornar esses ambientes mais atrativos. Elas podem ser resumidas em três itens:

1. Utilização de perspectiva em primeira pessoa na definição de regras.

2. Utilização de mecanismos de herança.

3. Regras Visuais com Condições Negativas

Figura 3.1: Stagecast Creator – Regras necessárias para movimentar um personagem.

## 3.1 Perspectiva em Primeira Pessoa

A Figura 3.1 apresenta um conjunto de regras especificadas no sistema Stagecast Creator, cujo objetivo é instruir o personagem como se deslocar por espaços livres no ambiente.

As regras visuais estabelecem condições que, caso satisfeitas, fazem com que as alterações definidas sejam realizadas. Por exemplo, na regra 1, o sistema verifica se existe um espaço vazio abaixo do personagem. Caso positivo, promove um deslocamento para baixo. No Creator, as regras são executadas na ordem fornecida. Se a condição de uma regra for satisfeita, as regras seguintes não são executadas.

Pode-se observar que são necessárias quatro regras para ensinar o personagem a andar. Uma para que ele ande para baixo, outra para a esquerda, outra para cima e mais uma para que ele ande para a direita. As regras são muito semelhantes, a única diferença é a direção de deslocamento do personagem. As regras são criadas com perspectiva em terceira pessoa, por isso precisam levar em consideração essa direção de deslocamento.

Se as regras fossem criadas com uma perspectiva em primeira pessoa, ou seja, levando em consideração a visão relativa do próprio personagem, apenas uma regra seria necessária para especificar esse comportamento desejado. O único teste a ser feito seria o de verificar se, à frente do personagem, existe um espaço em branco.

Para implementar essa proposta, um sistema deve executar cálculos para identificar, em tempo de execução, o correto posicionamento das áreas do ambiente referenciadas nas regras. Esse posicionamento irá depender da orientação do personagem, que deve então ser um atributo obrigatório para objetos que forem representados graficamente pelo sistema. Do ponto de vista do usuário, nenhum tipo de conhecimento adicional é necessário para se usar esse novo recurso, além dos conhecimentos requeridos pelas ferramentas descritas no Capítulo 2.

O uso dessa técnica, no caso do exemplo citado, eliminaria três das quatro regras utilizadas. No exemplo, o personagem só se desloca em quatro direções. Caso ele pudesse se deslocar na diagonal, a economia na construção de regras seria ainda

maior, já que no esquema original seriam necessárias oito regras.

## 3.2 Herança

Analisando os ambientes citados no Capítulo 2, pode-se constatar que, se uma mesma ação pode ser realizada por personagens diferentes, ela precisa ser especificada para cada um deles. Esse fato provoca um aumento no número de regras e um aumento do tempo gasto na programação, sendo que essa programação torna-se uma tarefa mecânica, já que os mesmos comandos serão repetidos para vários personagens.

Herança, um importante conceito utilizado na teoria de sistemas orientados a objetos, é definida como a habilidade de uma classe herdar características de uma outra classe (Sebesta, 2000). Esse conceito sugere uma solução natural para o problema apresentado acima. Se dois tipos de personagens possuem parte de seu comportamento descrito por regras iguais, bastaria definir uma classe básica onde essas regras comuns são especificadas. Os dois tipos seriam definidos como subclasses dessa classe básica, podendo adicionar outros comportamentos, com regras próprias.

De maneira diversa à verificada com a proposta de perspectiva em primeira pessoa, a aplicação de herança requer que o usuário entenda conceitos relacionadas à programação convencional, no caso, a programação orientada a objetos, e que não são exigidos pelas ferramentas descritas no Capítulo 2. Isso pode ser considerado uma desvantagem da proposta, mas é importante notar que essa sugestão deve ser tratada como um recurso adicional do sistema. Isto é, um usuário pode construir suas simulações animadas sem usar esse recurso, entender a motivação do seu uso, e mais tarde aplicar os conceitos com sucesso.

Em estudos realizados em (Smith et al., 2000), ficou provado que alunos que aprenderam a programar utilizando as técnicas de PBD obtiveram maior êxito na programação convencional. O uso da herança em ambientes de PBD pode ser útil no ensino desse conceito para os alunos ou profissionais da área de desenvolvimento de software. O aluno aprenderia esses conceitos de uma forma gráfica e concreta. Encontrando maior facilidade na hora de aplicar esses conceitos na programação con-

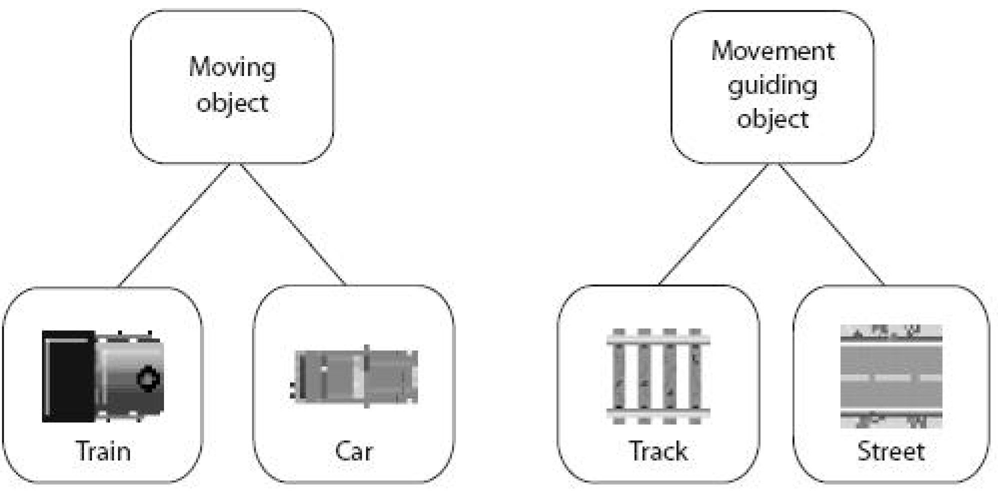


Figura 3.2: Uso da herança (Repenning and Perrone, 2000).

vecional.

No artigo (Repenning and Perrone, 2000), o autor é contra o uso da herança em sistemas de PBD. Ele dá um exemplo de uma simulação que reproduz o tráfego de uma cidade, com Trens andando em Trilhos e Carros andando em Ruas. Com o uso da herança, na hierarquia de classes, os Carros se tornam irmãos dos Trens e as Ruas se tornam irmãs dos Trilhos, para isso duas superclasses abstratas precisam ser criadas: **Moving Object** e **Movement Guiding Object** (Figura 3.2).

Ainda segundo Repenning and Perrone (2000), para permitir que Trens e Carros se movam, as regras gráficas precisam ser expressas em termos de **Moving Object** e **Movement Guiding Object**. Esta nova hierarquia de classes introduz dois problemas:

- A necessidade de abstrações: o usuário tem dificuldade para criar abstrações, principalmente para representá-la visualmente. A dificuldade pode aumentar quando os objetos são representados através de ícones definidos pelos usuários.

- Overgeneralization (termo usado pelo autor em (Repenning and Perrone, 2000)): se o tráfico de uma cidade roda com base nessa hierarquia de classes, torna-se possível que Trens se movam em Ruas e Carros se movam em Trilhos o que, embora teoricamente possível, não deve ser permitido em um domínio de

planejamento urbano. Essa proibição é obtida através de regras da vida real, o comportamento de Trens e Carros precisa ser especializado para prevenir estas combinações, não desejadas, dos objetos de **Moving Object** e **Movement Guiding Object**.

O autor do artigo (Repenning and Perrone, 2000) considera que a herança é um dos meios poderosos de generalização e que poderia aumentar a utilidade da PBD. Mas acredita que a tensão entre herança e PBD com respeito às concretas representações é difícil de solucionar. A herança cria a necessidade de introduzir e manipular representações abstratas, e a PBD tem como objetivo utilizar representações altamente concretas, que podem ser manipuladas e facilmente entendidas pelos usuários finais. Allen Cypher e David Canfield Smith também levantam esse problema no artigo (Cypher and Smith, 1995).

Essas considerações são importantes, mas não atrapalham a sugestão do uso da herança proposta neste capítulo. Como já foi dito, o uso da herança não será obrigatório para o usuário, ele poderá programar sem se preocupar com esse conceito. À medida que se familiarizar com o ambiente, com a programação, ele pode aprender e passar a utilizar as vantagens desse novo conceito.

Foram criados recursos para facilitar a visualização do uso da herança pelo usuário leigo em programação. Ao criar uma classe básica, o usuário define uma representação gráfica para ela. Como é uma classe genérica, ele usa a imagem que achar que melhor a representa, pode até usar desenhos de letras ou palavras que identifiquem bem a classe que está criando. Para as subclasses ou classes derivadas, o usuário escolhe outra figura, uma figura diferente da escolhida para a classe básica, mais específica, mais concreta. Todas as regras criadas para a classe básica usarão a imagem selecionada para ela, e todas as regras criadas para a classe derivada usarão a figura escolhida para a classe derivada, e as regras herdadas da classe básica são mostradas utilizando a nova representação.

Cada classe (personagem) possui uma janela onde as regras serão mostradas. Nelas todas as regras, inclusive as herdadas, aparecerão utilizando a imagem selecionada para a classe. As regras herdadas aparecererão marcadas para que o usuário

saiba que elas são apenas para leitura e saiba de que classe elas foram herdadas. Essa marca poderia ser o nome da classe básica ou a sua representação gráfica. Para editar uma regra herdada, o usuário precisa ir até a classe básica e fazer as modificações.

Por exemplo, a classe básica **Animais Terrestres** tem a regra que ensina os animais a andarem. A sua representação pode ser o desenho da letra **A** estilizado. Na classe **Elefantes**, derivada da classe **Animais Terrestres**, a regra que ensina a andar aparece marcada como uma regra herdada da classe básica, **Animais Terrestres**, mas a representação utilizada na regra é o desenho de um elefante (representação escolhida para a classe **Elefantes**). Na classe **Tigres**, também derivada da classe **Animais Terrestres**, a regra que ensina a andar é mostrada com a representação escolhida para a classe **Tigres**, provavelmente a figura de um tigre.

Esse recurso fará com que as representações voltem a ser concretas para o usuário, já que depois de criadas as regras da classe básica, ele terá acesso a elas através das regras mostradas em cada uma das classes derivadas. Só voltará a trabalhar com a representação abstrata quando tiver que modificar alguma regra na classe básica.

O problema da Overgeneralization não ocorre em todas as simulações. Em (Repenning and Perrone, 2000) foi citado um caso em particular, mas existem muitas outras simulações que podem ser construídas sem que esse problema seja detectado. Um exemplo de uma situação favorável ao uso da herança será visto na Capítulo 5.

O estudo da situação citada em (Repenning and Perrone, 2000) revela que o uso do polimorfismo paramétrico pode resolver o problema gerado pelo uso da herança.

Segundo Cardelli and Wegner (1985), em linguagens de programação, polimorfismo se refere à habilidade da linguagem de processar objetos de maneiras diferentes, dependendo do seu tipo ou classe.

A palavra polimorfismo significa "existindo em muitas formas". Na orientação a objetos é o conceito que possibilita o envio de uma mesma mensagem a objetos de diferentes classes sendo que cada objeto responderá de acordo com a implementação da sua classe, isto é, uma mesma mensagem pode resultar em ações diferentes quando recebidas por objetos diferentes. O polimorfismo visa a criação de classes com interfaces idênticas e implementações diferentes. Onde um nome simples pode

denotar objetos de diferentes classes que estão relacionadas por alguma superclasse em comum.

Dentre os tipos de polimorfismo existentes, se encontra o polimorfismo paramétrico. Através dele um mesmo objeto pode ser utilizado uniformemente como parâmetro em diferentes contextos sem necessidade de alterações. Uma função que exibe polimorfismo paramétrico (também chamada de função genérica) permite que o tipo do seu argumento seja determinado por um parâmetro de tipo implícito ou explícito, executando uma mesma operação, independente do tipo do argumento (Cardelli and Wegner, 1985).

Para resolver o problema descrito em (Repenning and Perrone, 2000), com a ajuda do polimorfismo paramétrico, bastaria criar uma regra com dois parâmetros, um representaria o objeto que anda e o outro a via onde ele poderia andar. Dessa forma, toda vez que a regra de movimentação do objeto fosse acionada, seriam informados dois parâmetros: o objeto que anda e a via por onde ele anda. Testes de restrição poderiam ser feitos para que o primeiro objeto informado tivesse relação com o segundo. Esse mecanismo impediria que o Carro andasse nos Trilhos ou o Trem andasse nas Ruas. Pois caso fosse informado um destes dois conjuntos de dados mensagens de erro seriam enviadas e as ações não seriam realizadas.

O polimorfismo paramétrico é um recurso interessante. A sua implementação permite aumentar as situações onde a herança pode ser utilizada sem gerar nenhum problema. É um recurso que precisa ser mais bem estudado para ser implementado. O sistema descrito no Capítulo 4 ainda não oferece esse recurso, mas ele está descrito como um trabalho futuro desta dissertação.

Ambientes como o AgentSheets trabalham com técnicas de analogias. Por exemplo, é possível copiar todas as regras da classe Carro para a do Trem, já que eles se movimentam da mesma forma, apenas em meios diferentes. Essa técnica resolve em parte o problema de criar todas as regras mais de uma vez. Isto porque, na verdade, é feita uma duplicação das mesmas regras, qualquer alteração necessária depois de realizada a analogia terá que ser feita em todas as classes. Isto provoca um trabalho desnecessário (Repenning and Perrone, 2000). Esse exemplo reforça a vantagem da

utilização da herança na programação por demonstração.

## 3.3 Regras Visuais com Condições Negativas

Na definição do comportamento de um personagem, pode ser interessante estabelecer que uma ação deverá ser executada se uma determinada condição não for satisfeita. Essa possibilidade, não disponível explicitamente em sistemas como Stagecast e AgentSheets, pode permitir que a definição seja especificada com um número menor de regras.

Por exemplo, suponha que um personagem possa encontrar, à sua frente, objetos de diversos tipos, entre eles a representação de um gramado, de um solo cimentado, de um solo com asfalto ou ainda de uma parede. Deseja-se definir que o personagem pode mover-se sobre qualquer desses objetos, com exceção da parede. Uma forma para isso seria especificar três regras, demonstrando movimento sobre gramado, cimento e asfalto. Se mecanismos para definição de regras com condições negativas estiverem disponíveis, bastaria especificar uma regra cuja condição é a presença de uma parede, e a ação é o movimento do personagem, caso essa condição não seja satisfeita. Assim, em vez de três regras, seria necessária apenas uma regra.

Regras visuais com condições negativas permitem trabalhar com a exceção, o que é útil em diversas situações e não exige nenhum conhecimento adicional do usuário, apenas raciocínio lógico.

No sistema Stagecast Creator existem vários tipos de operadores, um deles é o operador ***IS NOT***. Este operador permite trabalhar com algumas exceções, mas essas exceções estão relacionadas com os valores da variável selecionada. Ele equivale ao operador *diferente*, usado nas linguagens convencionais. A Figura 3.3 mostra todos os operadores que podem ser utilizados no Creator.

Figura 3.3: Operadores utilizados no Creator.

## 3.4 Conclusão

A herança e o uso de condições negativas são conceitos utilizados na programação convencional, mas ainda não explorados em ambientes que utilizam técnicas de programação por demonstração. A perspectiva em primeira pessoa foi pensada especialmente para esse tipo de ambiente. São sugestões simples, quase imperceptíveis para usuários não programadores, mas muito úteis para usuários com alguma experiência em programação.

A inclusão dessas novas técnicas permite simplificações significativas no processo de construção de simulações ou jogos. As sugestões possuem como objetivo diminuir a quantidade de regras necessárias para demonstrar um comportamento para um personagem. A perspectiva em primeira pessoa permite que o comportamente de um personagem, com várias direções, possa ser demonstrado com apenas uma regra. A herança elimina a necessidade de criar uma mesma regra para vários personagens. Além de evitar o trabalho de criar a mesma regra várias vezes, elimina a necessidade de fazer a mesma manutenção mais de uma vez. O ganho de tempo se torna grande. O uso de condições negativas permite trabalhar com exceções, elimina a necessidade de criar várias regras para tratar as situações permitidas. Em vez disso apenas uma regra é criada para demonstrar a exceção.

Uma grande importância na inclusão dessas sugestões é que o seu uso, pelo usuário, não é obrigatório. Em um ambiente que possui esses recursos, o usuário pode escolher usá-los ou não. Isso permite que o ambiente continue atrativo para os não-programadores e possibilita que usuários com um grau mais elevado de conhecimento possam usufruir de facilidades proporcionadas pelos ambientes convencionais, tornando o desenvolvimento mais atrativo e rápido. O que também amplia a possibilidade de uso desses ambientes. Eles podem deixar de serem simples sistemas para a criação de simulações e se tornarem importantes ferramentas capazes de facilitar a programação para robôs, por exemplo.

Exemplos apresentados no Capítulo 5 permitem um melhor entendimento dessas propostas, e ressaltam os benefícios proporcionados por elas.

# Capítulo 4

# Implementação do Tabajara Animator

Este capítulo apresenta um arcabouço próprio para produção de animação usando técnicas de programação por demonstração. As idéias sugeridas no Capítulo 3 foram implementadas no sistema, que foi batizado de Tabajara Animator. A implementação de simulações e jogos nesse novo sistema permite testar, na prática, se as sugestões dadas proporcionam um ganho de tempo e de eficiência para o usuário.

A construção de um novo sistema, para que os testes pudessem ser feitos, foi necessária porque os ambientes estudados no Capítulo 2 são sistemas proprietários, não permitem que o seu código seja modificado para englobar as sugestões propostas.

As funcionalidades do arcabouço podem ser divididas em três partes principais:

1. Modelo de execução para representar um ambiente com múltiplos agentes.

2. Ferramentas para produzir código para o modelo, a partir de metáforas visuais, usando técnicas de PBD (Programação Por Demonstração).

3. Recursos para apresentação de simulação animada.

O modelo de execução consiste na proposta de um modelo abstrato capaz de descrever um ambiente em que haja a interação concorrente de múltiplos agentes. A cada agente devem estar associados atributos que descrevem, entre outras coisas, uma posição determinada em um espaço bidimensional e o espaço ocupado por esse agente. Para que o sistema pudesse ser desenvolvido, foi proposta uma extensão do

modelo ASM *(Abstract State Machines)*, em português Máquinas de Estados Abstratas (Börger, 2000a; Börger and Stärk, 2003; Börger, 2000b). A Seção 4.1 descreve, resumidamente, o modelo ASM e a Seção 4.2 descreve a extensão produzida para esse sistema e apresenta a implementação de um interpretador para o modelo apresentado.

Para construir instâncias do modelo citado acima, ou seja, programar uma simulação animada, o sistema deve oferecer ferramentas que permitam que usuários com pouca ou nenhuma experiência de programação possam realizar essa programação. Para alcançar esse objetivo, as ferramentas devem usar técnicas de programação por demonstração (PBD), como aquelas empregadas em sistemas como Stagecast Creator (Smith et al., 2000) e Agentsheets (Repenning, 1991). O sistema desenvolvido ainda não oferece essas funcionalidades, que deverão ser incluídas em trabalhos futuros derivados dessa dissertação de mestrado. A Seção 4.3 apresenta as demais funcionalidades implementadas que foram suficientes para oferecer um ambiente de testes das melhorias, para sistemas de animação com PBD, propostas neste trabalho.

Uma vez que um "programa" foi produzido, sua execução pode ser apresentada de diversas formas. A forma mais direta, que é o objetivo inicial deste trabalho, consiste em uma simulação animada do programa. A Seção 4.4 apresenta alguns recursos oferecidos pelo sistema para apresentação dessas animações. Outras formas de execução podem ser geradas, por exemplo, envolvendo sistemas não virtuais, como robôs. Essas possibilidades são discutidas junto com os trabalhos futuros, no Capítulo 6.

## 4.1 Máquinas de Estados Abstratas - ASM

Introduzidas por Yuri Gurevich em (Gurevich, 1994), as Máquinas de Estados Abstratas (ASM), do inglês Abstract State Machines, são um conceito poderoso e elegante para a modelagem matemática de sistemas dinâmicos discretos. Utiliza conceitos simples e bem conhecidos, tornando a leitura e a escrita da especificação bastante direta. Foi originalmente proposta com o objetivo de prover semântica operacional para algoritmos de uma forma mais natural que a máquina de Turing.

O modelo ASM realiza a simulação de algoritmos através de transições de

estado. São definidos um estado inicial e uma regra de transição. Um novo estado é obtido a partir do estado atual e da execução de uma regra de transição. Estado é um conjunto com nomes de funções e relações e suas interpretações. O vocabulário do estado é definido pelo conjunto dos nomes de funções e relações. A interpretação de um nome de função ou relação é um mapeamento deste nome do vocabulário na respectiva função ou relação. Uma regra de transição modifica a interpretação de alguns nomes de função do vocalulário do estado. Ela se parece com um programa de uma linguagem imperativa comum, a principal diferança é a ausência de laços de iteração. A iteração é realizada por meio da execução, repetidas vezes, da regra de transição. As regras básicas são: atualização, condicional e bloco (Oliveira et al., 2004).

Existe um conjunto denominado de superuniverso do estado. Nele estão definidas todas as funções e relações. As funções podem ser estáticas ou dinâmicas. Elas são consideradas estáticas quando mantêm a mesma interpretação em todos os seus pontos durante uma execução. São consideradas dinâmicas quando podem sofrer mudanças na sua interpretação, como conseqüência de atualizações durante a execução da regra de transição ou mesmo por meio do ambiente.

A atualização de funções acontece por meio da mudança de interpretação da assinatura da função de um estado para outro. Uma função dinâmica sem parâmetros se parece com variáveis de um programa imperativo, pois seu significado é um nome que possui uma única interpretação de valor e que pode ser modificado. A atualização de uma função dinâmica de um ou mais argumentos $f(a1, \ldots, an) = v$ significa que a interpretação da função, $f$, nos argumentos, $(a1, \ldots, an)$, passa a ter novo valor, $v$.

A regra de atualização tem a forma: $f(\bar{x}) := y$, onde $\bar{x}$, é o comprimento de $x$, é igual à aridade de $f$. Esta regra cria, a partir de um estado $S$, um novo estado $S'$, tal que a interpretação da função $f$ no ponto $\bar{x}$ é $y$. Por exemplo, a regra $f(1) := 2$ determina um novo estado no qual o valor da função $f$, no ponto $1$, é $2$.

Uma regra condicional tem a forma: ***if g then R1 else R2*** onde $g$ é uma expressão booleana. Se $g$ for avaliado como verdadeiro então o estado resultante é o resultado da regra *R1*, caso contrário o estado resultante é o resultado da regra *R2*.

A regra de bloco tem a forma: *R1, ...,Rn* onde o estado resultante é formado pelo resultado da execução de todas as regras *Ri*, $1 \leq i \leq n$, em paralelo. Por exemplo, a execução da regra de bloco: *f(1) := 2*, *f(2) := 4* produz um novo estado, no qual o valor da função $f$ no ponto *1* é *2* e no ponto *2* é *4*.

A ASM possui, como uma de suas principais características, a execução paralela de regras. Elas são executadas em relação a um mesmo estado global. A execução de regras em um dado estado produz um conjunto de atualizações. Para o caso de um conjunto de atualizações consistente, que não apresenta contradições na atualização de alguma localização, aplica-se o conjunto de atualizações ao estado corrente no fim da transição para se obter o próximo estado.

## 4.2 Modelo de Execução para Representação de Ambiente com Múltiplos Agentes

Como foi mostrado no começo deste capítulo, o modelo de execução consiste na proposta de um modelo abstrato capaz de descrever um ambiente em que haja a interação concorrente de múltiplos agentes. A cada agente devem estar associados atributos que descrevem, entre outras coisas, uma posição determinada em um espaço bidimensional e o espaço ocupado por esse agente. Esses atributos podem ser usados para apresentação visual animada da execução do modelo, onde cada agente é associado a um personagem de uma animação. O modelo utilizado no sistema é inspirado nas Máquinas de Estado Abstratas (ASM). As características mais evidentes herdadas de ASM podem ser resumidas em:

- Inexistência de estruturas de repetição. A cada agente está associada uma regra de transição que é continuamente executada, produzindo alterações de estado.

- Regras simples para definir as transições de estado. São usadas regras de atualização de componentes do estado e estrutura condicional. Atualizações coletadas em um passo de execução são executadas em paralelo, no estilo ASM.

A diferença mais marcante do modelo adotado, com relação a ASM original, é a utilização do conceito de classes. O conceito de Herança também está presente no modelo e não é abordado em ASM original. Esse modelo estendido é chamado de ASM-OBJ, e é baseado nas idéias descritas em (Gottlob et al., 1991)

Em ASM-OBJ, uma classe possui atributos e uma regra de transição. Um programa completo é formado por um conjunto de classes e a definição de um estado inicial. No estado inicial, são criados os primeiros objetos das classes definidas. A cada passo da execução, o sistema seleciona objetos ativos e coleta as atualizações produzidas pelas regras desses objetos, disparando todas as atualizações em paralelo.

Nesta seção, o modelo é apresentado em detalhes. É proposta uma representação usando XML (van der Vlist, 2002) e os elementos principais dessa representação são discutidos. A semântica das construções que envolvem conceitos não presentes no modelo ASM, como classes e herança, é definida usando-se o próprio modelo ASM.

O modelo ASM não define uma sintaxe rigorosa, é apenas um modelo abstrato. As linguagens baseadas nesse modelo definem sua sintaxe própria, sendo que a mais difundida é a linguagem AsmL (Gurevich et al., 2004). Para representar programas em ASM-OBJ, optou-se por não utilizar uma sintaxe no estilo de linguagens de programação convencionais, mas sim um esquema XML. Essa escolha tem as seguintes justificativas principais:

1. Para um usuário que escreve um programa, é mais fácil usar construções simplificadas, utilizadas pela sintaxe da maioria das linguagens de programação, do que as regras rígidas de XML. Entretanto, neste sistema, o objetivo é que os programas sejam gerados automaticamente por ferramentas de PBD, o que elimina o inconveniente apresentado.

2. Uma representação em XML, formalmente definida por um esquema XSD, é adequada para comunicação de dados entre diferentes ferramentas. O sistema propõe que a execução do modelo possa ser realizada de diversas formas diferentes. Por exemplo, pode-se apresentar uma simulação animada em computador, ou produzir ações coordenadas em robôs. Nesses casos, a representação em XML

poderia ser utilizada por diferentes ferramentas, que gerariam uma apresentação visual da interpretação dos programas ou uma compilação para linguagem apropriada a uma família de robôs.

O sistema inclui um interpretador para o modelo adotado, escrito na linguagem Java, que é utilizado para simular as alterações de estado. Uma especificação é carregada na memória por meio de uma rotina gerada automaticamente por ferramentas de *XML Data Binding* (McLaughlin, 2002), a partir do esquema XML definido para o modelo. O interpretador permite controlar a execução passo a passo, e oferece facilidades para consultar o estado do modelo, a qualquer momento. Essas funcionalidades são utilizadas pelas ferramentas apresentadas na Seção 4.2.3.

### 4.2.1 Componentes da Representação XML do Modelo

Nesta seção, os principais elementos da representação proposta para o modelo são discutidos. São identificados os elementos que possuem semântica equivalente à utilizada no modelo ASM padrão. A semântica dos demais elementos é discutida na Seção 4.2.2.

Um *ambiente (environment)* é composto por um conjunto de definições de classes e um estado inicial. O estado inicial é definido por uma regra de transição. Uma definição de classe é representada pelos seguintes elementos: seu nome (elemento textual), o nome da sua classe ancestral imediata (pai) na hierarquia (também elemento textual), um conjunto de definições de atributos, uma regra de transição e um conjunto de ações. Este último elemento, conjunto de ações, ainda não é utilizado no modelo atual.

O esquema gráfico para a definição de um ambiente pode ser visto na Figura 4.1. Nela estão representados todos os elementos que compõem esse ambiente. Em todas as representações gráficas são utilizados dois símbolos, mostrados nas Figuras 4.2 e 4.3. Eles representam, respectivamente, uma "seqüência de elementos" e uma "escolha entre elementos".

O esquema para um conjunto de definições de atributos é exibido na Figura 4.4.

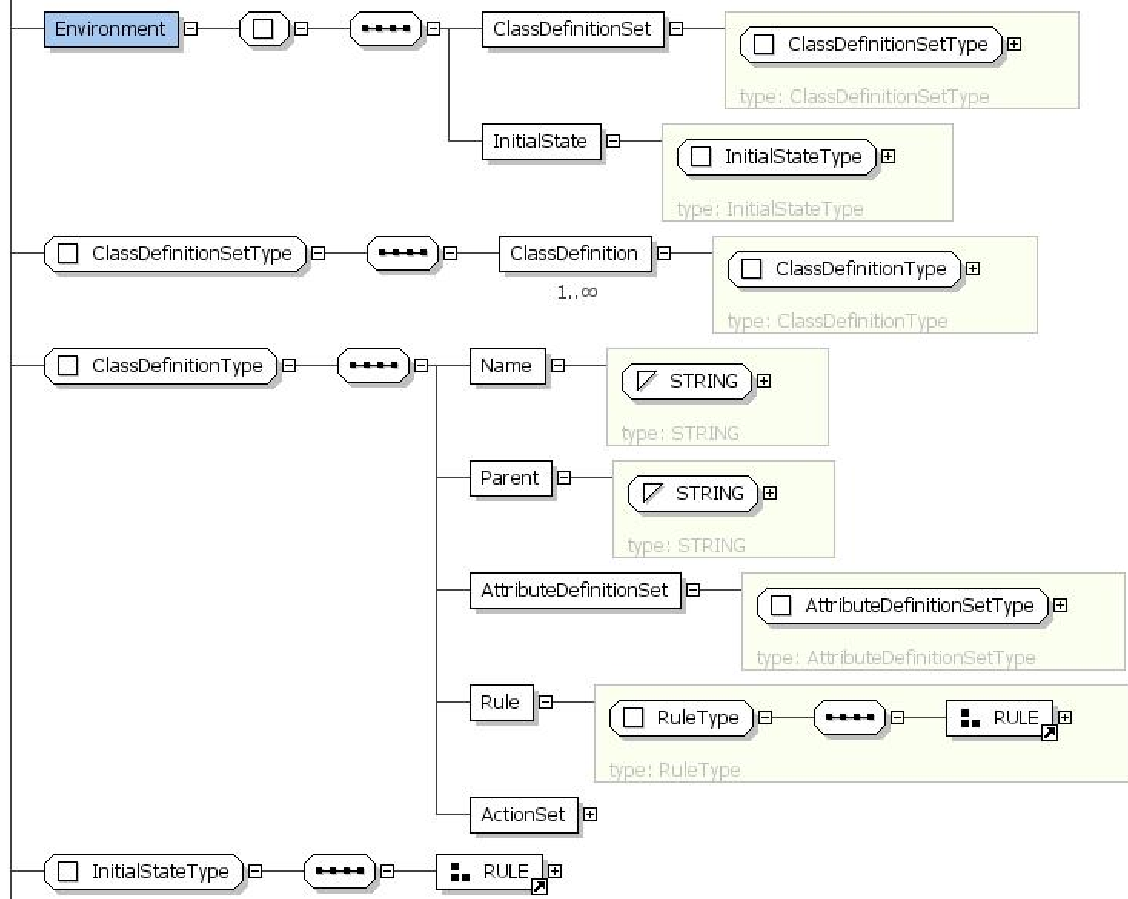


Figura 4.1: Representa a definição de um ambiente.

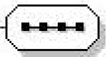

Figura 4.2: Representa uma sequência de elementos.

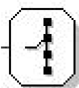

Figura 4.3: Representa uma escolhar entre elementos.

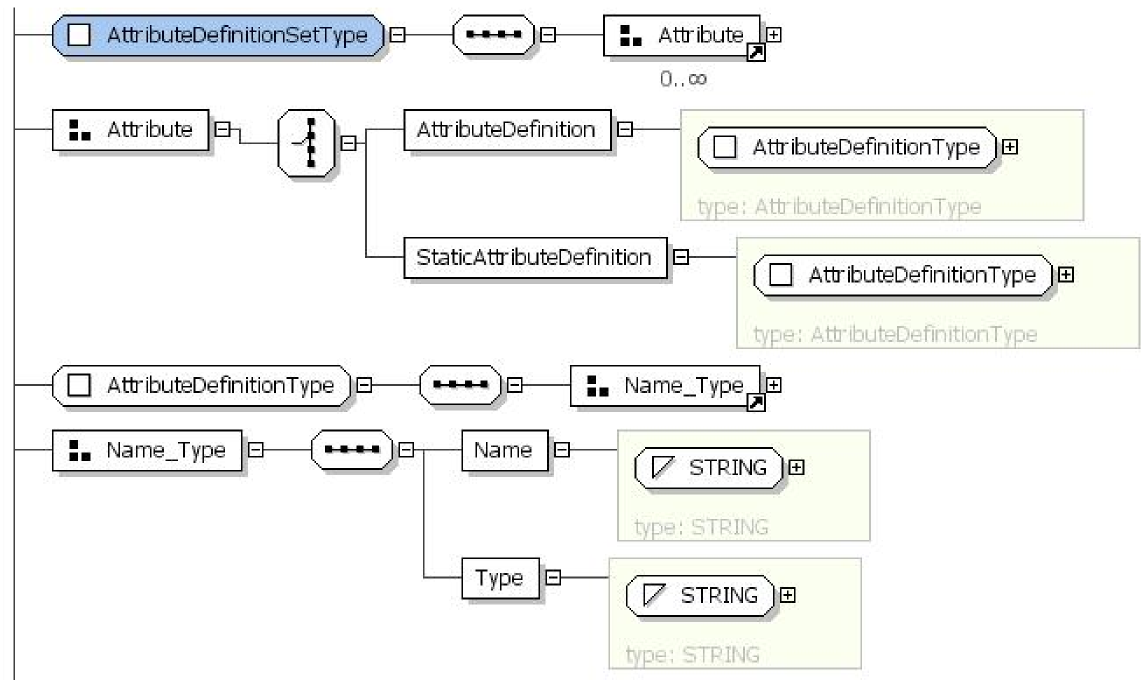


Figura 4.4: Representa o conjunto de definições de atributos.

Um atributo pode ser estático ou não estático, sendo composto por um nome e um tipo, ambos são elementos textuais. O tipo pode ser um dos tipos básicos oferecidos pelo sistema, que incluem representação de textos, números inteiros e reais, e também pode ser o nome de uma classe já definida.

O esquema para as regras de transição é apresentado na Figuras 4.5. De acordo com as idéias de ASM, existem cinco tipos de regras: bloco, condicional, atualização, escolha e criação de elementos em um universo.

A Figura 4.6 mostra o esquema gráfico para esses cinco tipos de regras. Um bloco é um conjunto de regras. A semântica, como em ASM, consiste no disparo em paralelo das regras do bloco. Uma regra condicional é formada por uma expressão e duas outras regras. Uma regra de atualização é formada por um lado esquerdo, que é um endereço (location), e um lado direito, que pode ser qualquer expressão válida. Regras condicionais e de atualização também seguem a mesma semântica definida no modelo ASM tradicional, diferindo apenas no conceito de endereço, que é explicado mais à frente, nesta seção.

As regras de escolha e criação de elementos de um universo envolvem a utili-

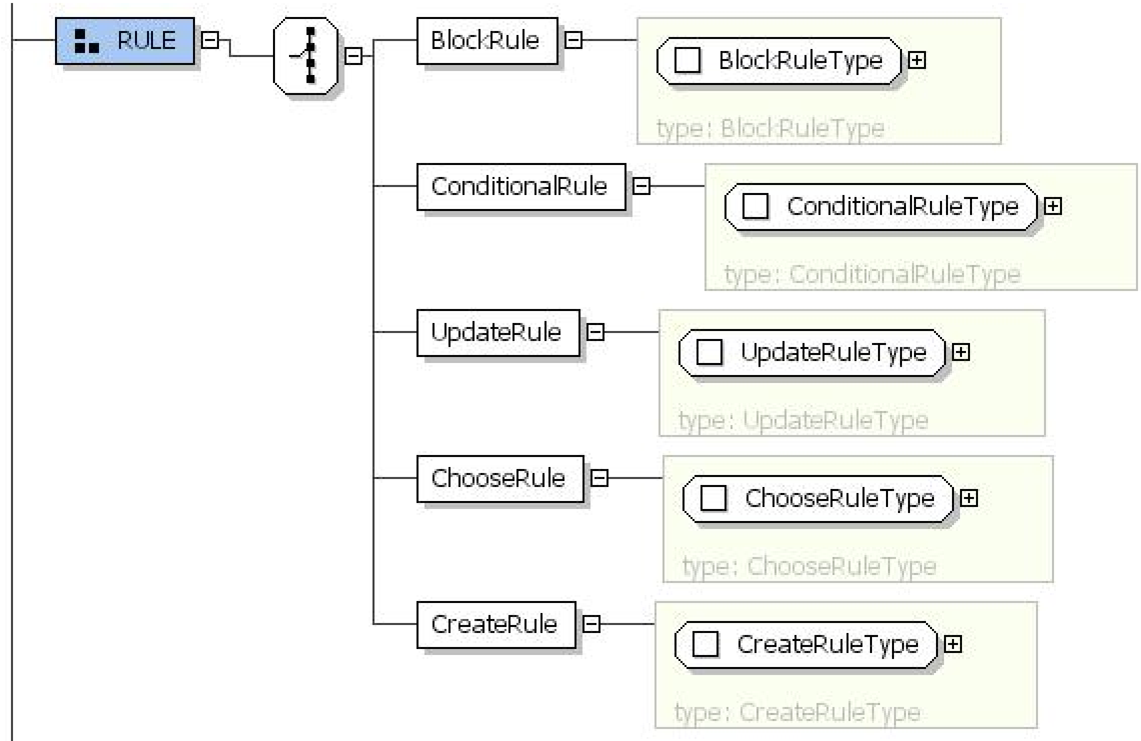


Figura 4.5: Representa as regras de transição.

zação de variáveis. Elas constituem mais um caso em que a semântica segue o padrão ASM, similares às regras *choose* e *extend* de ASM, sendo que o universo associado é um tipo definido dentro do ambiente.

A Figura 4.7 apresenta um esquema para as expressões. Uma expressão pode ser uma constante de tipo padrão, um endereço (*location*), uma referência a uma variável definida por regras como *choose* e *create*, uma chamada de função padrão ou uma chamada de função externa. Os tipos padrão incluem representação para números inteiros, números reais e outros valores.

A Figura 4.8 apresenta um esquema para chamadas de função padrão e de função externa. As funções padrão incluem operações básicas sobre os tipos padrão. As chamadas de funções externas seguem a semântica tradicional de ASM. Para deixar clara a distinção, o sistema usa sintaxe diferente para esses dois tipos de chamadas de função. Nas chamadas de função padrão, os argumentos são fornecidos na ordem esperada. Nas funções externas, cada parâmetro deve ter um identificador único, e os valores nas chamadas são associados a esses identificadores, em qualquer ordem.

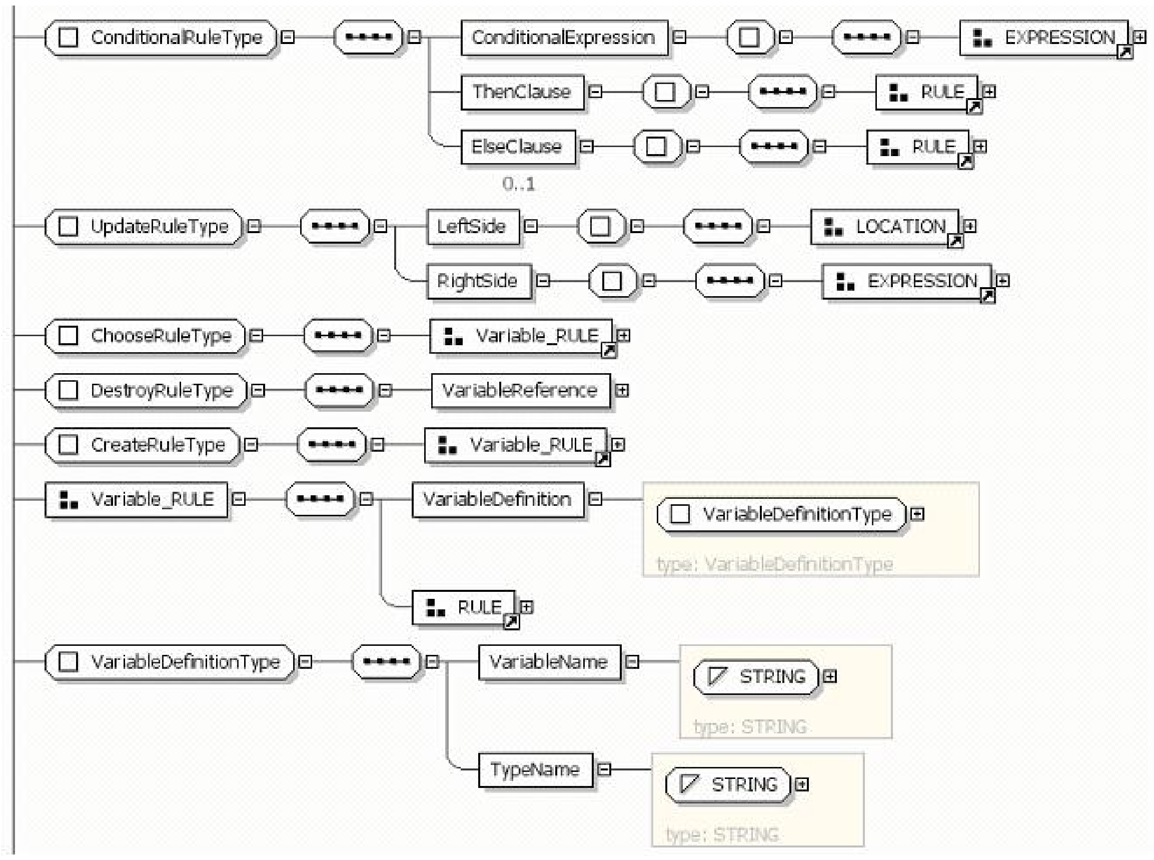


Figura 4.6: Representa os tipos de regras existentes.

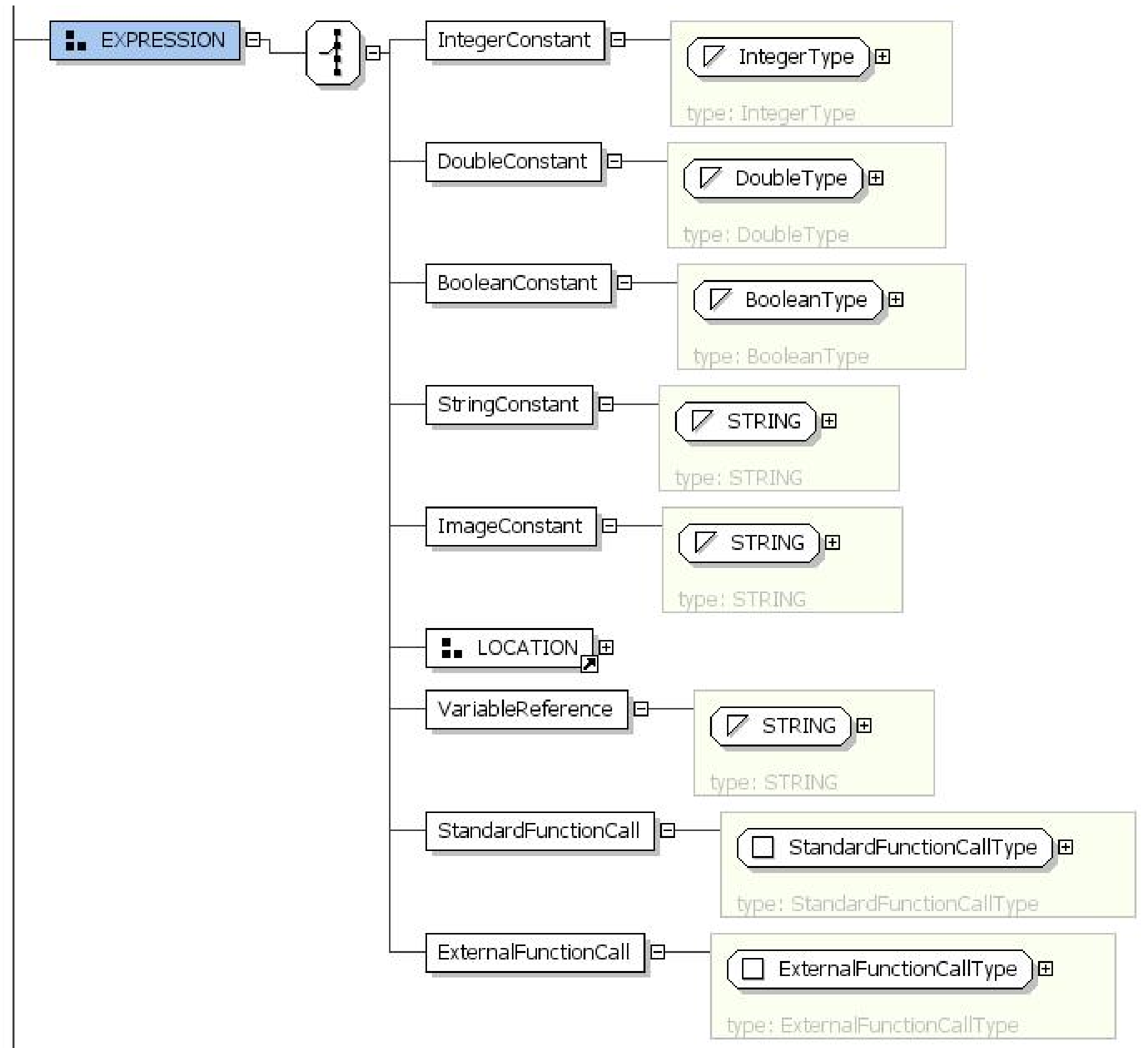


Figura 4.7: Representa expressões.

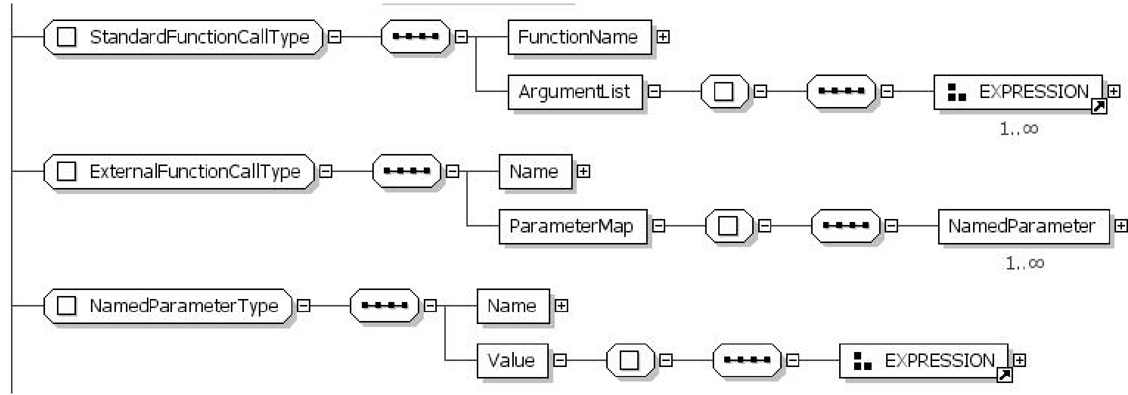


Figura 4.8: Representação de funções.

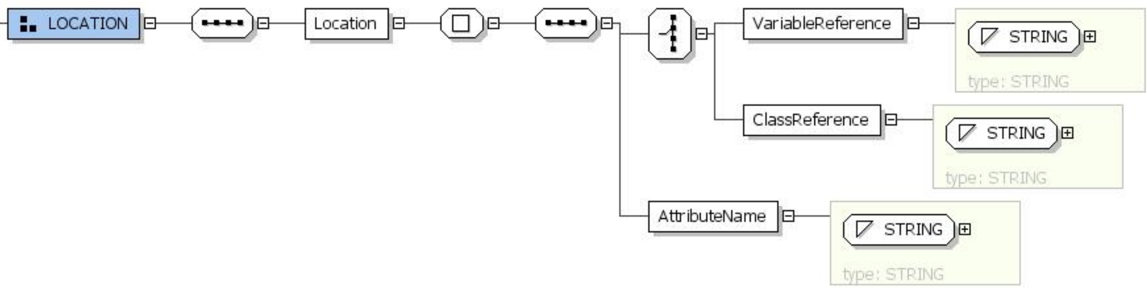


Figura 4.9: Representação para um endereço.

Finalmente, a Figura 4.9 apresenta um esquema para um endereço (*location*). Esse conceito difere do adotado em ASM tradicional, já que, no modelo proposto, os endereços estão associados a atributos das classes, em vez de funções de ASM. Um endereço associado a um atributo estático de uma classe é representado com o nome dessa classe. Um endereço associado a um atributo não estático é representado como uma referência a uma variável.

### 4.2.2 Semântica dos Elementos

A maior parte das construções apresentadas na Seção 4.2.1 possui semântica equivalente ao modelo ASM padrão. Nesta seção, é apresentada uma semântica para as construções que envolvem o conceito de classes, por meio de uma tradução para o modelo ASM padrão.

#### Atributos e Endereços

Cada atributo estático de uma definição de classe pode ser interpretado como uma função ASM 0-ária com identificador único, no conjunto de valores constantes do modelo. A exigência de identificador único é para garantir unicidade, no caso de classes diferentes possuírem atributos com um mesmo nome. O nome da função ASM pode ser resultado de uma combinação do nome da classe (identificador único) com o nome do atributo estático em questão.

Cada classe pode ser interpretada como um universo ASM. Objetos da classe são elementos do universo associado. Assim, cada atributo não estático de uma defi-

nição de classe pode ser interpretado como uma função ASM unária com identificador único, mapeando um elemento do respectivo universo a um valor do conjunto de constantes do modelo.

### Modelo de Execução

Uma execução do modelo começa pelo disparo da regra que define seu estado inicial. Espera-se que essa regra inclua regras *create* que irão produzir um povoamento inicial de um ou mais universos. A partir desse momento, inicia-se uma execução concorrente das regras associadas às classes não vazias, ou seja, cujos universos contenham pelo menos um elemento.

Os elementos dos universos associados às definições de classes podem ser vistos como *agentes* que executam as regras de transição das suas respectivas classes. A cada passo da execução, qualquer agente está apto a executar a sua regra de transição. Um novo estado global é produzido pela união das atualizações geradas pelos disparos das regras de cada agente. Nenhuma suposição é feita sobre a velocidade relativa dos agentes, assim, em um passo da execução do modelo, qualquer subconjunto dos agentes pode ter suas regras disparadas, até mesmo o conjunto de todos os agentes de todas as classes.

Quando a regra de transição associada a um agente é disparada, o nome *this* é interpretado como o agente em questão. Isso permite que um agente possa acessar seus "próprios" atributos.

Uma função especial indica que a execução terminou. Qualquer agente pode acessar essa função, definindo o término de uma execução.

### Herança

A hierarquia de classes afeta a semântica do modelo proposto, inserindo elementos que o diferenciam do modelo original de ASM. A herança múltipla não é tratada neste modelo.

Se uma classe C1 é filha de uma outra classe C2, isso significa que a classe C1 herda todos os atributos da classe C2. Esse fato afeta o conjunto de atributos que

podem ser acessados em regras de transição.

Se uma classe C1 é filha de outra classe C2, então C1 herda também a regra de transição de C2. A regra de transição de C1 é definida como a união de sua própria regra com a regra de C2. Essa definição é aplicada recursivamente, fazendo com que a regra de transição de uma classe seja a união de todas as regras das suas classes ancestrais na hierarquia.

Uma variável é criada por uma regra *choose* ou *create*. Nessas regras, é obrigatório definir o tipo da variável, ou seja, a que classe ela está relacionada. Assim, o tipo de uma variável é definido estaticamente. Os atributos acessíveis por meio de uma variável serão aqueles pertencentes à sua respectiva classe, ou às classes ancestrais a ela, na hierarquia.

### 4.2.3 Interpretador para o Modelo

Esta seção apresenta o interpretador desenvolvido para o modelo apresentado na Seção 4.2, aplicando, sempre que possível, padrões de projeto consagrados. A linguagem escolhida para a implementação foi a linguagem Java.

#### Análise Léxica e Sintática

Inicialmente, desenvolveu-se um analisador léxico e sintático para arquivos seguindo o esquema XML apresentado na Seção 4.2.1. Esse analisador recebe como entrada um arquivo XML e constrói uma árvore de sintaxe abstrata que representa a estrutura lida. Essa árvore é constituída por objetos Java conectados. Os nomes escolhidos para as classes foram os mesmos nomes adotados pelos tipos complexos do esquema XML. Cada elemento desses tipos complexos deu origem a um atributo nas classes construídas, que também receberam métodos para leitura e escrita (*get* e *set*) desses atributos, permitindo uma navegação na árvore e sua alteração.

Para alocação de objetos que compõem a árvore de sintaxe abstrata, utilizou-se o padrão de projeto **Fábrica Abstrata** (Gamma et al., 2005). Em nenhum ponto do código, componentes da árvore são alocados diretamente. Todas as alocações são realizadas por meio dessa **Fábrica Abstrata**.

O analisador léxico recebe, como parâmetro do seu construtor, uma *Fábrica Concreta* que implementa a interface descrita pela **Fábrica Abstrata** citada acima. Desse modo, é possível adicionar informações aos nós da árvore de sintaxe abstrata, adequadas a qualquer uso ao qual ela seja destinada. Essa funcionalidade é discutida mais à frente, no item ***Interpretador***.

### Análise Semântica

Uma vez alocada a árvore de sintaxe abstrata, uma análise semântica é executada. Nessa fase, a utilização de todos os identificadores é verificada. Estruturas são criadas para armazenar tabelas de símbolos e todas as referências aos símbolos encontrados são ligadas à entradas dessas tabelas.

A análise semântica está embutida no analisador léxico. Ao se requisitar a leitura de um arquivo XML representando um ambiente para o modelo, fases de análise léxica e semântica são automaticamente executadas, gerando uma árvore de sintaxe abstrata junto com as tabelas de símbolos.

### Interpretador

As classes criadas para representar a árvore de sintaxe abstrata podem ser utilizadas para diversos fins. Por exemplo, o ambiente lido pode ser compilado, gerando código em qualquer linguagem de programação desejada. Ou esse ambiente pode ser interpretado, procedimento que foi implementado neste trabalho.

Para executar uma interpretação do ambiente, pode ser interessante a adição de informações a nós da árvore de sintaxe abstrata. Por exemplo, pode ser interessante adicionar uma área de memória a identificadores das tabelas de símbolos associados aos atributos das classes do modelo. Essas áreas de memória podem armazenar o valor dos atributos, durante a interpretação.

A utilização do padrão **Fábrica Abstrata** no analisador léxico, citado nesta seção, permitiu que as classes da árvore de sintaxe abstrata não fossem "poluídas" com código associado especificamente ao interpretador. Para adicionar as informações desejadas a cada nó da árvore, foi criada uma nova classe, *Fábrica Concreta*, derivada

da **Fábrica Abstrata**, criada e implementada no analisador léxico. Nessa nova *Fábrica Concreta*, os métodos de criação de alguns nós foram redefinidos, alocando classes derivadas das classes originais, para os casos onde se fez necessário a adição de informações importantes para o interpretador.

O interpretador desenvolvido oferece operações que permitem:

- disparar a regra do estado inicial do ambiente;
- disparar um passo da execução do ambiente, que pode significar o disparo de regras de transição de diversos agentes;
- verificar se a execução terminou;
- identificar todos os componentes do ambiente, a cada estado da execução.

A última operação descrita acima permite que ferramentas possam apresentar a evolução dos estados em uma execução do modelo. Um exemplo disso são as ferramentas para apresentação de simulações animadas, desenvolvidas neste trabalho, que são discutidas na Seção 4.4.

## 4.3 Ferramentas para Construção de Código

O sistema desenvolvido irá oferecer ferramentas para gerar código de programas ASM-OBJ automaticamente, usando técnicas de PBD. Os recursos oferecidos serão semelhantes aos empregados em sistemas como o Stagecast Creator e o AgentSheets, discutidos no Capítulo 2.

Esses recursos, que geram os programas automaticamente, por meio de técnicas de PBD, ainda não estão implementados e não fazem parte desse trabalho. Serão executados em trabalhos futuros. Nessa seção será descrito como esse sistema deve funcionar quando estiver pronto. Por enquanto, todos os testes foram feitos por meio de programas escritos diretamente em XML, interpretados por meio do modelo descrito na Seção 4.2 e os resultados são exibidos por meio dos recursos de apresentação de simulação descritos na Seção 4.4.

Neste sistema, por enquanto hipotético, para que o usuário crie novas regras, basta que ele acione o comando de criar regras e selecione o personagem para o qual deseja criar a regra de comportamento. As regras são criadas no estilo de "regra visual antes-depois". Cada regra é traduzida automaticamente para uma estrutura condicional em ASM-OBJ, onde a condição avaliada é definida pelo estado "antes" e o comando a ser executado, caso a condição seja verdadeira, é um bloco com as atualizações definidas no estado "depois" dessa regra visual. Cada personagem possuirá uma janela onde as suas regras serão armazenadas.

Nessa ferramenta para construção de regras, as melhorias propostas no Capítulo 3 ficam evidentes em dois pontos. Primeiro, na percepção em primeira pessoa dos personagens. Isso faz com que o usuário não precise se preocupar em definir diversas regras onde a única diferença é a orientação do personagem. A segunda melhoria, que envolve o uso de herança, fica mais explícita na interface com o usuário: um editor para a hierarquia de classes permite que o usuário navegue nessa hierarquia, adicione, remova e altere classes e seus atributos.

Herança múltipla não é permitida na linguagem ASM-OBJ, assim a representação gráfica da hierarquia utiliza uma estrutura de árvore. Na Figura 4.10, pode-se ver um exemplo de uma tela do editor de hierarquia, onde as classes *Pacman* e *Ghost* são derivadas de *Moveable*, que é derivada de *VisibleClass*, assim como a classe *Brick*. No lado direito da janela, pode-se ver os atributos associados à classe *VisibleClass*.

A classe *VisibleClass* é pré-definida pelo sistema, não é editável, e serve de superclasse para todos os personagens que podem ter uma representação gráfica. Entre os atributos dessa classe, estão valores associados à posição em um espaço bidimensional e à rotação atual do personagem. Imagens são definidas como atributos *estáticos* (ou atributos de classe, com a mesma semântica de Java, por exemplo), pois a mesma imagem é usada por todos os personagens de uma classe. Subclasses de *VisibleClass* podem redefinir esse atributo e escolher uma imagem diferente.

A tela do editor de hierarquia, mostrada na Figura 4.10, fornece informações mais técnicas e será melhor aproveitada por usuários que já entendam os conceitos de herança. Os usuários que ainda não estão familiarizados com os conceitos de herança,

Figura 4.10: Tabajara Animator - Editor da hierarquia de classes.

podem utilizar a tela de criação de personagens, Figura 4.11. Essa tela terá um funcionamento e uma aparência semelhantes ao funcionamento e à aparência da tela do windows explorer, fornecendo a todos os usuários uma forma de visualizar todos os personagens existentes em sua simulação, bem como a sua posição na hierarquia de classes. Os personagens serão mostrados como se fossem pastas e estarão ligados à classe básica *VisibleClass* (pré-definida pelo sistema). Caso o usuário queira criar uma subclasse, basta selecionar a classe básica desejada e então acionar o comando para criar o personagem (classe). Ele terá, também, a opção de alterar a hierarquia das classes, para isso terá apenas que fazer a movimentação desejada, como se estivesse movimentando arquivos ou pastas.

Nessa tela ele poderá alterar a imagem associada ao personagem e, através de um clique duplo (ou botão de comando) terá acesso à janela com as regras do personagem. Essa janela, chamada de editor de regras, mostra a posição da classe na hierarquia, mostra todas as regras criadas exclusivamente para o personagem e as

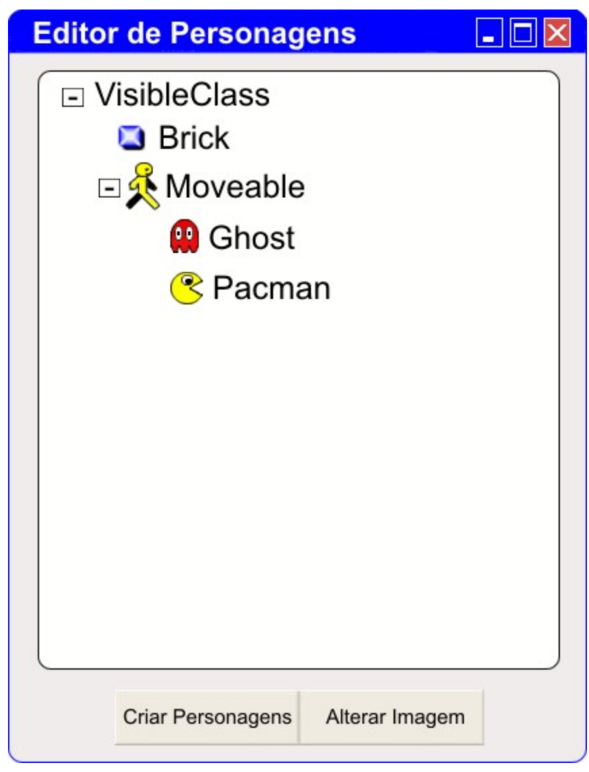


Figura 4.11: Tabajara Animator - Tela de criação de personagem.

regras herdadas de suas classes básicas. As regras herdadas serão mostradas com o personagem alterado para a representação utilizada na classe, mas não poderão ser modificadas, serão marcadas para que o usuário saiba que são regras herdadas e na frente estará especifica de qual classe foram herdadas. A Figura 4.12 mostra, como exemplo, a tela com as regras para a classe Moveable. Essa classe não possui regras herdadas, mas apenas uma regra que ensina o personagem a andar, desde que na sua frente não exista uma parede.

A Figura 4.13 mostra o editor de regras do personagem Pacman. Nele estão representadas a regra herdada da classe Moveable e a regra própria do Pacman. Nesta figura é possível identificar a diferença na representação de uma regra herdada e de uma regra criada para o personagem em questão.

No editor de regras de um personagem, o usuário poderá incluir novas regras ou modificar as já existentes, desde que não sejam regras herdadas. As regras herdadas só podem ser modificados no editor de regras da classe onde foram criadas.

Para criar ou editar uma regra será aberta uma tela como a mostrada na

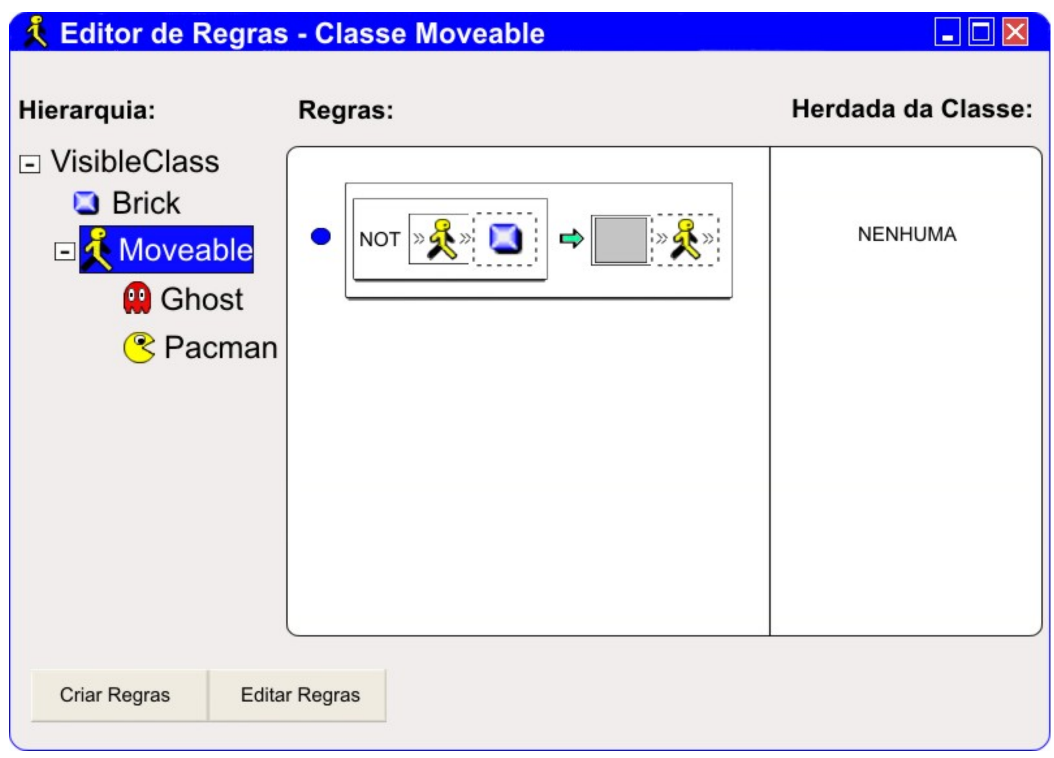


Figura 4.12: Tabajara Animator - Editor de regras da classe Moveable.

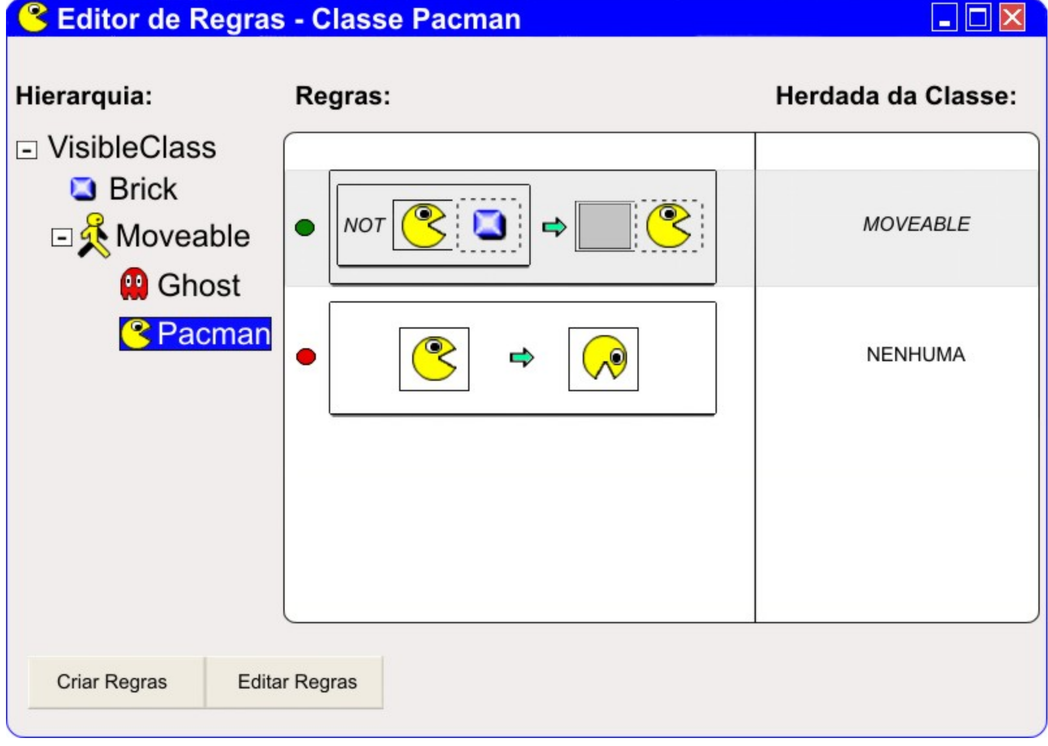


Figura 4.13: Tabajara Animator - Editor de regras da classe Pacman.

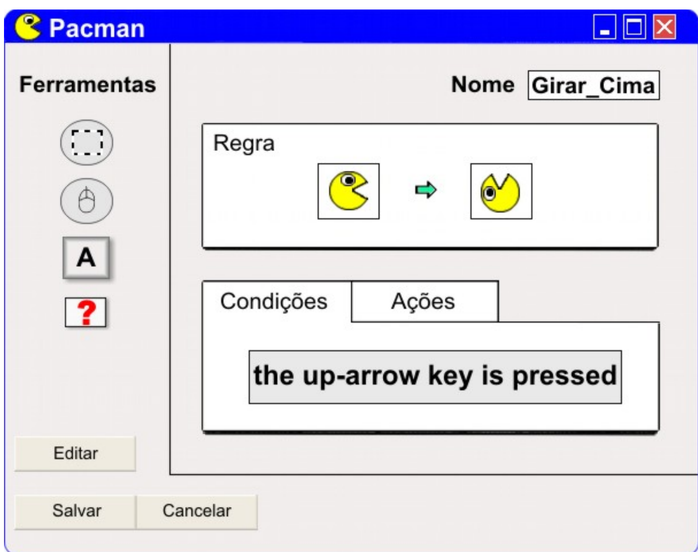


Figura 4.14: Tabajara Animator - Tela para criar regras.

Figura 4.14. Esta tela mostra o estado inicial e final do personagem, nela o usuário terá ferramentas para criar condições e ou ações para as suas regras. As condições podem utilizar teclas do teclado, cliques do mouse ou teste lógicos. Essas condições e ações são mostradas nas guias que aparecem abaixo da regra. A Figura 4.14 mostra uma das regras responsáveis pela mudança de direção do personagem Pacman. Nessa regra, o Pacman mudará a sua direção para cima se por pressionada a tecla UP-ARROW. Essa condição é mostrada na guia chamada de ***Condições***.

## 4.4 Recursos para Apresentação de Simulações Animadas

O sistema oferece recursos para apresentar uma representação animada dos estados produzidos pela execução de um programa ASM-OBJ. Esses recursos incluem janelas com diferentes perspectivas. No momento, são oferecidos três tipos de perspectiva:

1. perspectiva simples em terceira pessoa;
2. perspectiva em terceira pessoa com foco em um personagem central;
3. perspectiva em primeira pessoa.

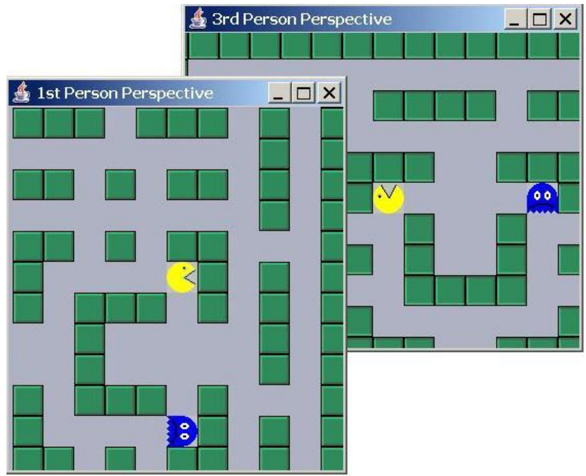


Figura 4.15: Janelas de perspectiva em primeira pessoa e perspectiva em terceira pessoa com escolha de personagem central.

As janelas com perspectiva do tipo 1 descrito acima apresentam uma porção fixa do ambiente. O usuário pode navegar pelo ambiente usando barras de rolagem, se ele for maior do que o tamanho da janela. No tipo 2, um personagem escolhido é posicionado, sempre que possível, no centro da janela. As janelas do tipo 3 se comportam de maneira semelhante às do tipo 2, com a caracerística adicional de não apresentar alteração visual na rotação do personagem escolhido. Quando o personagem escolhido altera sua rotação no modelo, a janela apresenta uma rotação em todo o resto do ambiente, na mesma proporção, na direção contrária.

Várias janelas podem ser abertas simultaneamente, para observar a evolução do modelo em diferentes perspectivas. A Figura 4.15 exibe uma janela de tipo 3 e uma janela de tipo 2, observando um mesmo ambiente. Ambas consideram como personagem central o Pacman.

Uma observação atenta das Figuras 4.16 e 4.17 mostra que elas representam a mesma situação do jogo Pacman. Essa conclusão é possível ao observar a posição do fantasma em relação ao Pacman. A Figura 4.16 mostra uma janela do tipo 1, ou seja, a imagem é mostrada em terceira pessoa, por isso é possível ver o Pacman se movimentando por todo o cenário. Já a Figura 4.17 mostra uma janela do tipo 3, onde a imagem é mostrada em primeira pessoa, por isso o Pacman permanece no

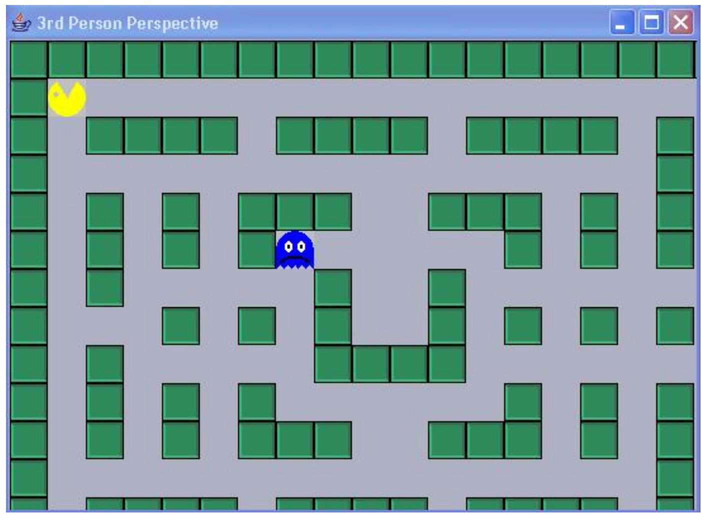


Figura 4.16: Janela de perspectiva em terceira pessoa.

centro da janela e todo o cenário se movimenta de acordo com a direção adotada pelo personagem.

A implementação desenvolvida até o momento provoca um problema de rotação dos personagens. À medida que eles mudam de direção a sua imagem pode ficar numa posição não muito natural. Esse problema será visto quando a janela em exibição for a de tipo 1. Por exemplo, se o Pacman estiver andando para a direita e tiver a sua direção alterada para a esquerda, a sua imagem sofrerá uma rotação de cento e oitenta graus. Isso fará com que a sua imagem apareça de cabeça para baixo. Esse problema ocorre porque a cada personagem só é associada uma imagem e quando a sua direção é alterada, a sua imagem é apenas rotacionada. Essa situação é mostrada na Figura 4.18.

Uma solução seria permitir que para cada personagem pudesse ser associada mais de uma imagem. E assim, de acordo com a rotação sofrida pelo personagem outra imagem seria selecionada. Essa solução faz parte dos trabalhos de continuação dessa dissertação.

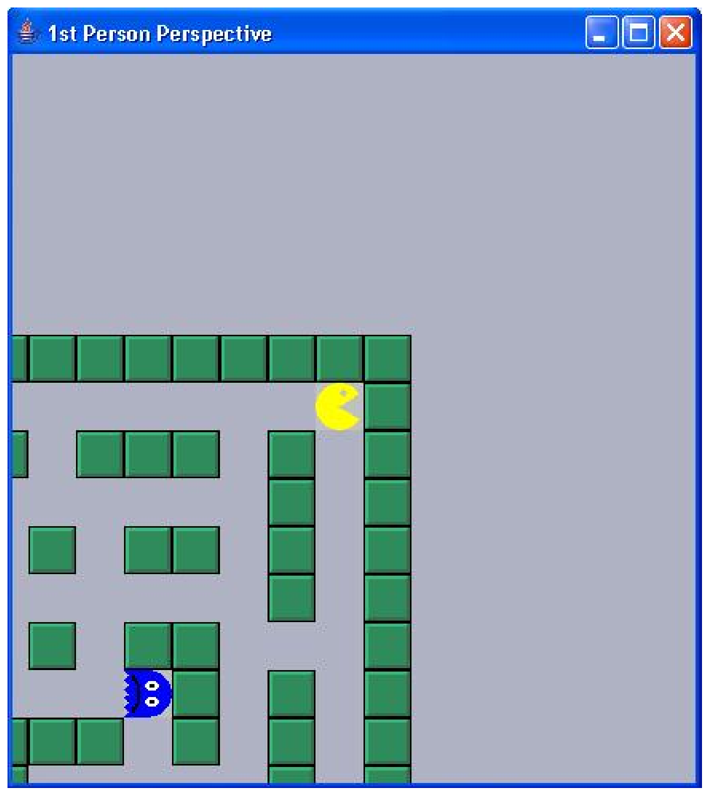


Figura 4.17: Janela de perspectiva em primeira pessoa.

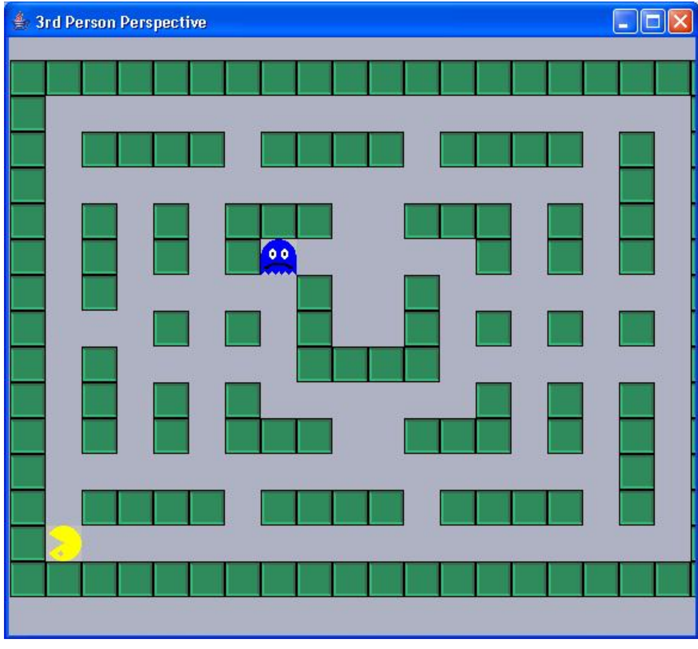


Figura 4.18: Problema que rotação que pode acontecer.

## 4.5 Conclusão

Este capítulo abordou o desenvolvimento do ambiente de programação Tabajara Animator. A criação desse sistema foi necessária porque os sistemas estudados não permitem que o seu código seja modificado para a implementação das sugestões propostas neste trabalho. O objetivo do Tabajara Animator é permitir a validação dessas sugestões. Devido à restrição de tempo e à complexidade envolvida na criação de um ambiente de programação, este trabalho contou com a colaboração de alunos de iniciação científica, do curso de Ciência da Computação da Universidade Federal de Viçosa, vinculados a esse projeto. Esses alunos implementaram duas das três partes desse ambiente de programação. Essas partes foram explicadas nas Seções 4.2 e 4.4. A parte de criação da ferramenta para construção de código através das técnicas de programação por demonstração (Seção 4.3) deverá ser feita em um trabalho futuro. Mesmo assim o sistema pôde ser testado através da implementação do jogo Pacman, mais detalhes do teste serão vistos no Capítulo 5.

# Capítulo 5

# Exemplo - Avaliação e Validação

Neste capítulo, é apresentada a implementação de partes de um jogo simples, nos ambientes AgentSheets, Stagecast Creator e Tabajara Animator. O AgentSheets e o Stagecast Creator não permitem que sejam feitas alterações no seu código, por isso as propostas deste trabalho não puderam ser implementadas nesses ambientes. O ambiente Tabajara Animator foi desenvolvido para que os recursos apresentados no Capítulo 3 pudessem ser utilizados. Essas implementações permitem fazer uma comparação mostrando como as técnicas propostas podem ser usadas para simplificar a especificação de jogos animados e simulações.

Para conduzir as comparações de implementações foi escolhido o clássico jogo *Pacman* (DeMaria and Wilson, 2003). A sua escolha foi motivada pela sua simplicidade de funcionamento, por ser um jogo bem conhecido e por proporcionar a identificação de situações claras onde as técnicas propostas no Capítulo 3 podem ser aplicadas. Assim, poderíamos verificar que a utilização dessas novas técnicas gera benefícios para o programador.

O jogo *Pacman* será implementado em apenas dois, AgentSheets e Stagecast Creator, dos quatro ambientes estudados no Capítulo 2 pelos seguintes motivos:

- a implementação em dois ambientes já proporciona material suficiente para comprovar a necessidade do uso dessas novas técnicas;

- são ambientes conhecidos e utilizados internacionalmente;

- possuem um ambiente intuitivo e de fácil utilização para não programadores;
- a forma de programar segue o mesmo estilo escolhido para o ambiente Tabajara Animator, o que permite comparar a quantidade de regras criadas em cada ambiente, principalmente em uma comparação entre o Tabajara e o Creator.

Nos dois ambientes escolhidos, os comandos são dados em forma de regras visuais. O AgentSheets usa a estrutura **IF ... THEN** e o Stagecast Creator usa a técnica de **Regras Visuais Antes-Depois**. Além disso, os dois ambientes apresentam as ineficiências identificadas e comentadas no Capítulo 2: excesso de regras para a mesma atividade, repetição de programação, etc. A implementação nas duas ferramentas, mostrando os problemas encontrados, e a implementação na ferramenta Tabajara Animator, nos dá uma idéia da diferença na quantidade de regras e no tempo gasto para a implementação utilizando ou não as técnicas propostas nesse trabalho.

A implementação no ambiente Gamut não foi abordada porque esse sistema utiliza regras de inferência que não estão presentes e nem foram estudadas para o sistema Tabajara Animator. O ToonTalk trabalha com uma abordagem parecida com jogos de vídeo-game, usando a idéia de uma programação animada onde o programador é colocado como um personagem dentro do mundo virtual animado, onde as abstrações da programação são substituídas por comandos tangíveis. Esses recursos não são contemplados no ambiente desenvolvido. A implementação do jogo Pacman nesses dois ambientes, Gamut e ToonTalk, não contribuiria para a comprovação da eficiência das propostas apresentadas no Capítulo 3.

## 5.1 O jogo Pacman

O jogo *Pacman* consiste em labirinto por onde se deslocam fantasmas e o personagem *Pacman*, controlado pelo usuário. O objetivo do jogo é que o personagem *Pacman* coma o máximo de pílulas que puder e que evite ser alcançado pelos fantasmas. Um jogador será o vencedor se conseguir fazer o *Pacman* comer todas as pílulas do labirinto. Cada pílula se converte em um ponto para o jogador. Esses pontos são acumulados em um contador.

O usuário controla o *Pacman* através do teclado. Ao pressionar uma das teclas de movimentação, a orientação do personagem muda e ele passa a se mover de acordo com a nova direção. Ao encontrar uma parede do labirinto, ele encontra um obstáculo, não consegue atravessá-la, pára e espera que o usuário mude a sua direção através das teclas do teclado. Se encontrar uma pílula na direção em que está se movendo, ele come a pílula e sua contagem de pontos sobe.

O objetivo do personagem fantasma é alcançar o Pacman e destruí-lo. Para isso precisa saber onde o *Pacman* está. Mas para executar uma perseguição eficiente ao *Pacman*, não basta que um fantasma saiba onde ele está, pois no jogo as paredes são obstáculos. Elas impedem a livre movimentação dos personagens, obrigando-os a contorná-las. Nas implementações desenvolvidas neste capítulo, a solução utilizada foi a de atribuir, para cada posição do cenário do jogo, um valor inteiro. À posição onde o *Pacman* estiver, será atribuído um valor maior que zero, por exemplo mil. O valor das outras posições será calculado de acordo com a maior ou menor proximidade com relação à posição do *Pacman*. O cálculo do valor das outras posições é feito por meio da média dos valores dos seus quatro vizinhos, dois na horizontal e dois na vertical. Essa pontuação é sempre atualizada, já que o personagem se encontra sempre em movimento. As paredes recebem valores nulos, o que permite que os valores se propagem ao redor delas e dessa forma possibilitem aos fantasmas contorná-las para continuarem a busca pelo seu alvo.

Os fantasmas, tendo à sua disposição esses valores, escolhem sempre as posições com maiores valores, e, assim, seguem de perto o *Pacman*.

O desenvolvimento desse jogo proporciona que o programador aprenda:

- a como criar personagens controlados pelo usuário (*Pacman*);
- a como criar sistemas de localização sofisticados (fantasmas que localizam o *Pacman*);
- a como construir um contador de pontos.

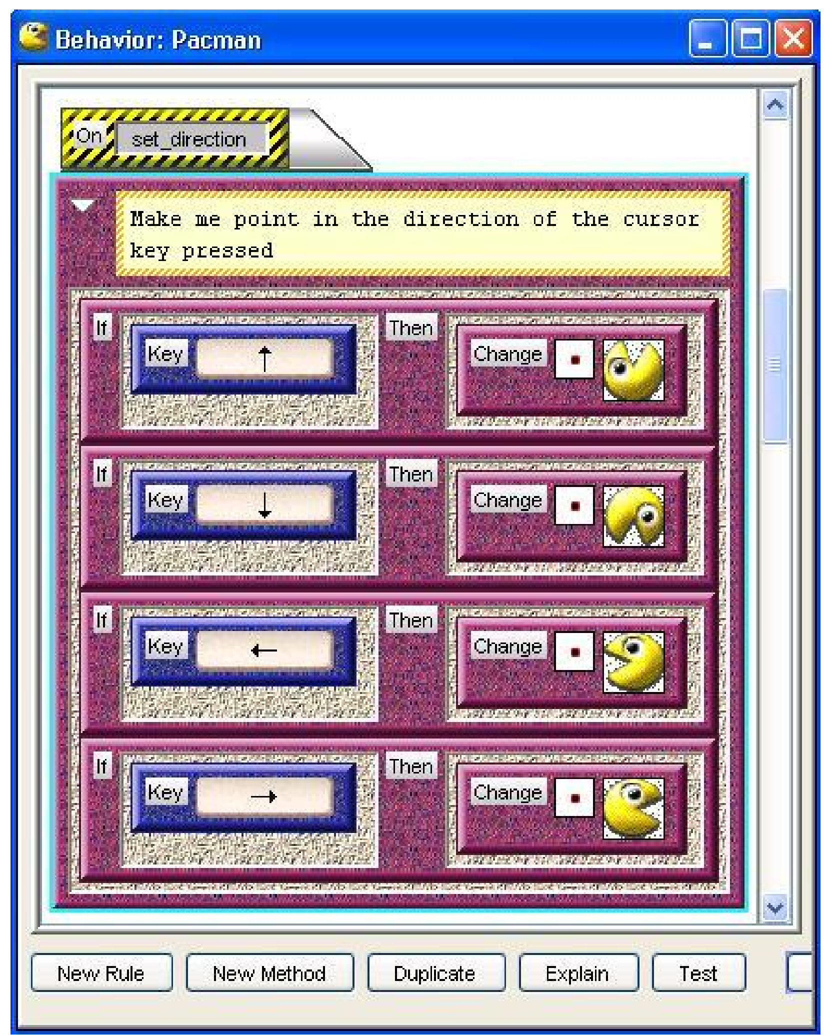


Figura 5.1: AgentSheets - regra para mudar a direção do Pacman.

## 5.2 O jogo Pacman no AgentSheets

O jogo *Pacman* foi completamente implementado no sistema AgentSheets, incluindo a contagem e apresentação visual do número de pontos conseguidos pelo usuário, que correspondem ao número de pílulas comidas pelo *Pacman*.

A implementação do jogo *Pacman* no AgentSheets envolve um número grande de Agentes e de regras. Algumas regras são agrupadas em módulos. O que é um ótimo recurso, pois evita a repetição de código.

A Figura 5.1 mostra a representação de um conjunto de regras para o personagem *Pacman*, dentro do editor de regras do AgentSheets. Nesta regra o personagem

muda a sua direção de acordo com a tecla pressionada. A representação usa a estrutura condicional com semântica óbvia: se a condição for verdadeira, a ação associada será executada. No caso do exemplo mostrado na Figura 5.1, a condição é saber qual tecla foi pressionada, para isso são necessárias quatro regras, uma para cada direção possível. Quando a tecla pressionada é identificada, a ação correspondente é executada. A ação é modificar o estado do Pacman, colocando-o virado para cima, para baixo, para a esquerda ou para a direita. O símbolo "`.`" (ponto), no AgentSheets, é associado à célula onde o personagem se encontra.

São necessárias mais quatro regras para fazer o Pacman andar pelas posições livres do labirinto. Uma representação gráfica dessas regras é mostrada na Figura 5.2. As setas usadas na representação servem para identificar células vizinhas à célula onde o personagem se encontra. Por exemplo, a condição da primeira regra apresentada é satisfeita se, na célula onde o personagem se encontra, houver uma representação do Pacman voltado para baixo, e a célula logo abaixo estiver vazia. A ação a ser executada manda que o Pacman se mova para a célula abaixo dele. Todas as regras apresentadas possuem a mesma ação, a única mudança se encontra na direção do Pacman apresentado na condição da regra.

A Figura 5.3 apresenta regras que ensinam o Pacman a andar e comer as pílulas do labirinto. A estrutura é semelhante à utilizada na Figura 5.2. Por exemplo, para que a primeira regra possa ser executada é necessário que na célula onde o Pacman se encontre, exista uma representação dele virado para baixo e na célula logo abaixo se encontre uma representação da pílula. A ação "`get_point`" é um módulo de regras definido separadamente no sistema, e executa procedimentos associados com a contagem e exibição do número de pontos obtidos.

Da mesma forma que acontece no conjunto de regras mostradas na Figura 5.2, o conjunto de regras da Figura 5.3 possui regras praticamente idênticas. A única diferença se encontra na direção de deslocamento do Pacman. Fora isso, as condições e ações executadas são as mesmas. Por esse motivo, cada conjunto de regras poderia ser resumido em uma única regra. Bastaria que a definição levasse em conta a perspectiva do próprio personagem, ou seja, observar o que está à sua frente, sem levar

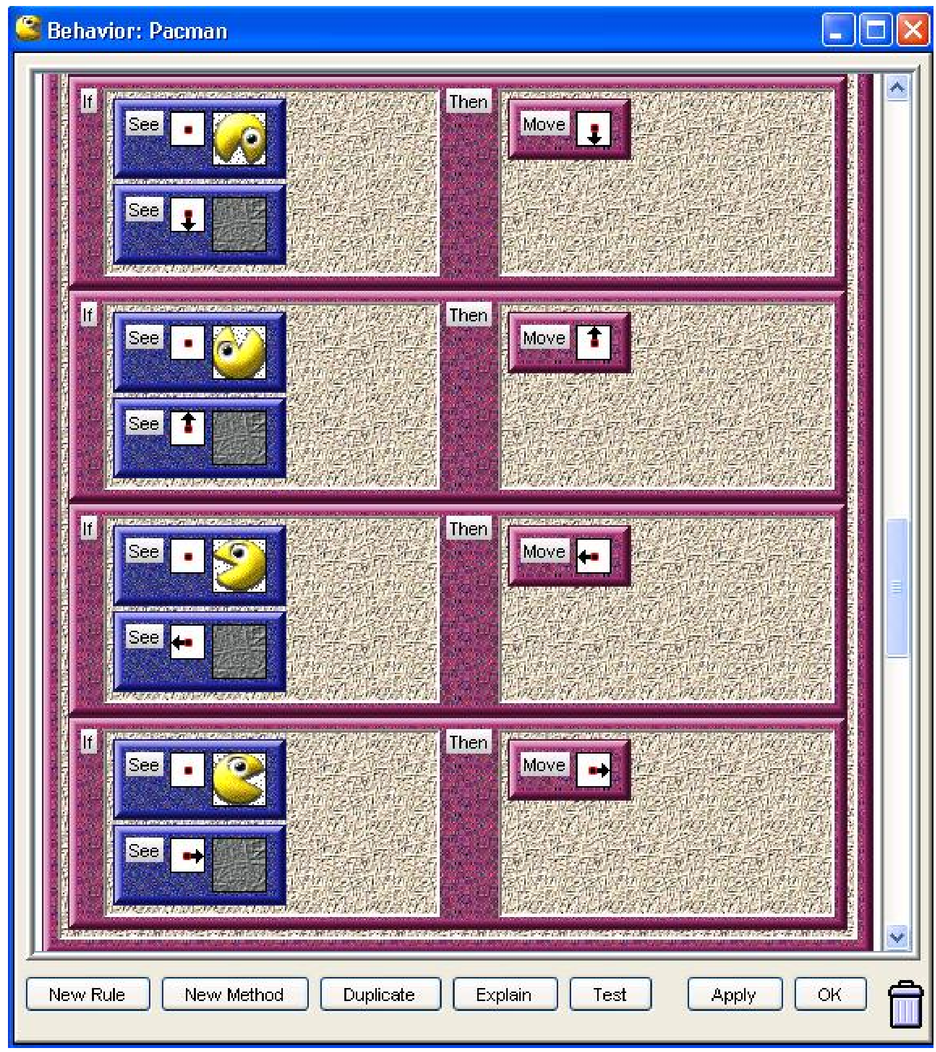


Figura 5.2: AgentSheets - regras para o Pacman se mover por posições vazias.

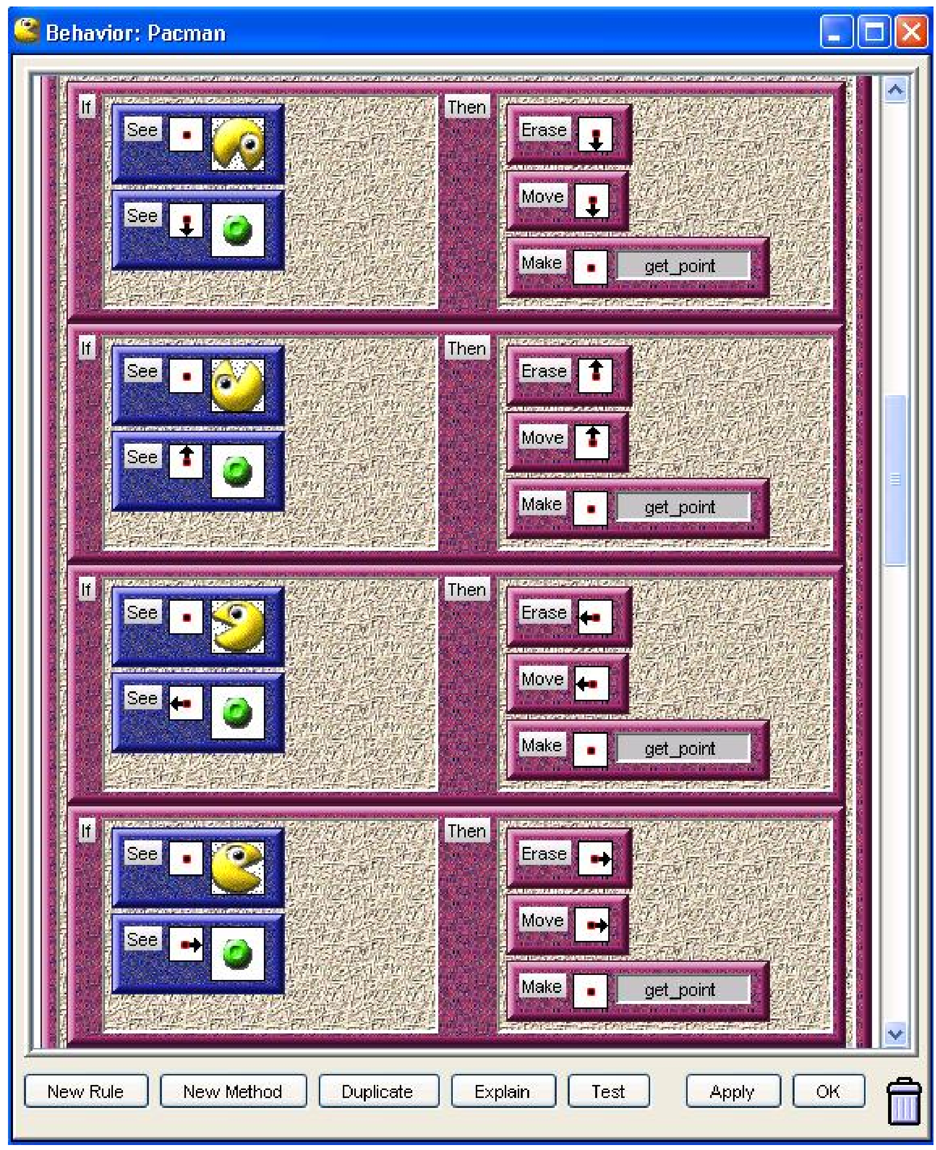


Figura 5.3: AgentSheets - regras para o Pacman comer pílulas.

em consideração a direção de deslocamento. O número de regras cairia de oito para duas regras.

Outro recurso que poderia ser considerado é o uso da herança. Tanto o Pacman quanto os fantasmas se movimentam, em posições livres, da mesma maneira. São necessárias quatro regras para que os fantasmas saibam como andar por espaços vazios. Então poderia ser criada uma classe básica que descrevesse o comportamento de andar por posições livres do labirinto. As classes Pacman e Ghost seriam derivadas dessa classe e trariam outras regras que ensinariam os personagens a mudarem de direção de acordo com as suas particulariedades. O personagem Pacman muda de direção através das teclas de movimentação do teclado, enquanto os fantasmas mudam de direção através do esquema sugerido na Seção 5.1, de atribuição de valores às posições do labirinto.

## 5.3 O jogo Pacman no Stagecast Creator

No Stagecast Creator, o jogo Pacman não foi totalmente implementado, pois a versão disponibilizada gratuitamente possui restrição quanto ao número de regras e de personagens que podem ser criados. Essas restrições impossibilitam a criação de todas as regras e personagens necessários. Mas mesmo sem a implementação completa, foi possível analisar os problemas a serem enfrentados no seu desenvolvimento.

No Stagecast Creator, as regras são escritas como apresentado na Figura 5.4. Nesse exemplo aparecem as regras responsáveis pela movimentação do Pacman nas posições vazias do labirinto. As regras utilizam os conceitos de **Regras Visuais Antes-Depois** (Smith et al., 2000). A primeira parte da regra mostra o personagem no seu estado inicial, exatamente do jeito que ele precisa estar para que a regra possa ser executada. Isso equivale à condição da regra escrita para o AgentSheets. Após a seta a regra apresenta o estado final do personagem, o que equivale às ações a serem executadas.

A implementação do jogo Pacman no Stagecast Creator pode seguir um padrão semelhante ao adotado para o AgentSheets, na Seção 5.2. A Figura 5.4 exibe as regras

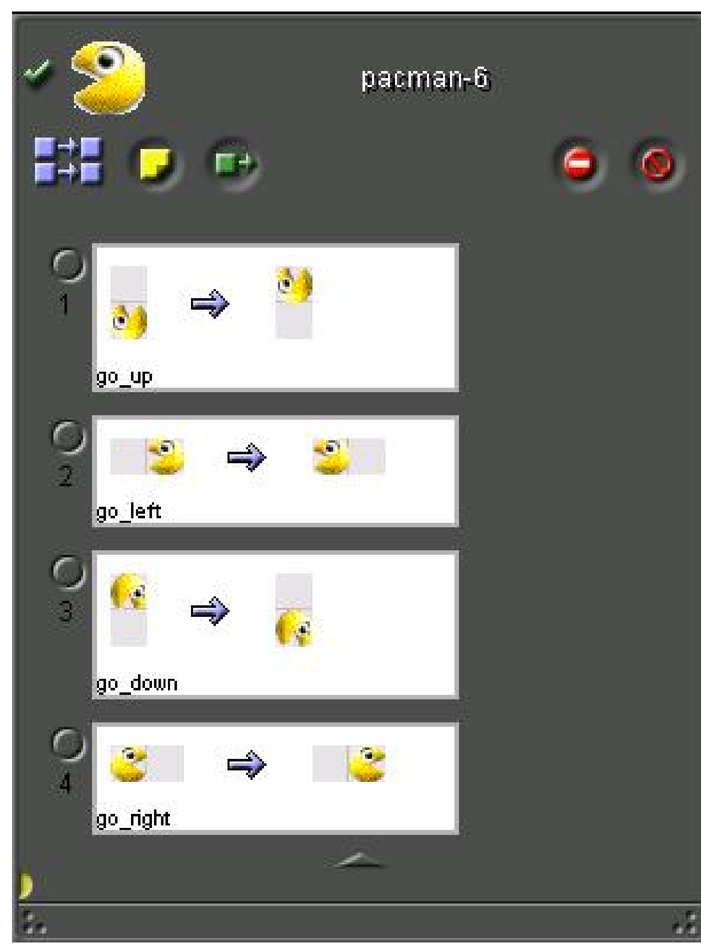


Figura 5.4: Stagecast - regras para o Pacman se mover por posições vazias.

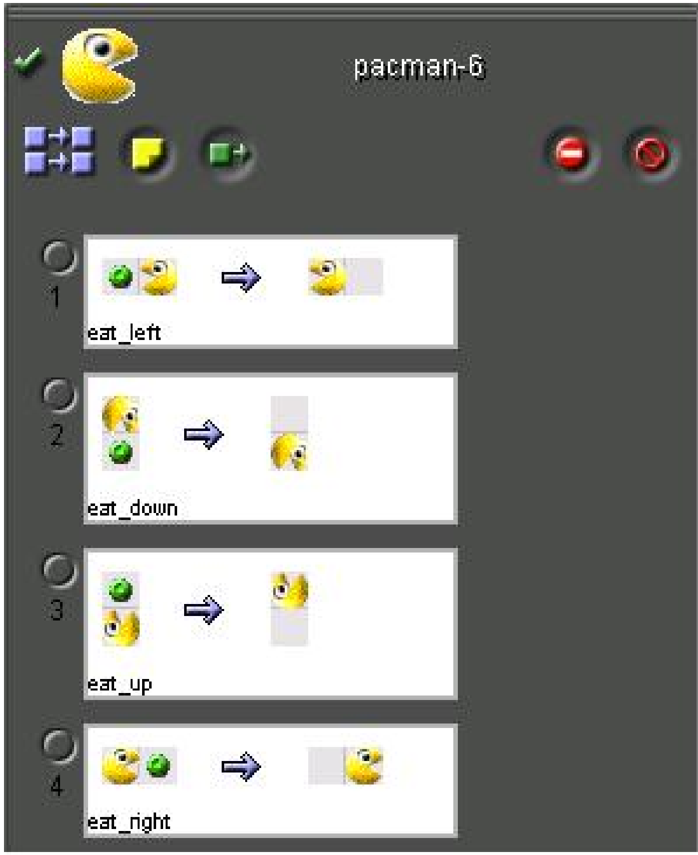


Figura 5.5: Stagecast - regras para o Pacman se mover e comer as pílulas.

responsáveis por controlar a movimentação do Pacman nas posições livres do labirinto. Assim como foi explicado na implementação do AgentSheets, as regras foram criadas levando-se em consideração o sentido de deslocamento do personagem. Por isso, foram necessárias quatro regras apenas para ensinar o Pacman a andar por posições livres. A Figura 5.5 mostra as quatro regras necessárias para ensinar o Pacman a comer as pílulas encontradas no labirinto. Esse número poderia ser reduzido se a implementação fosse feita sob a perspectiva do personagem e não do programador. De um total de oito regras utilizadas nessa implementação, seriam necessárias apenas duas.

A exemplo do AgentSheets, o Stagecast também poderia se beneficiar da utilização de herança na definição de suas regras. Tanto o Pacman quanto os fantasmas poderiam ser derivados da mesma classe básica. Nessa classe seriam criadas as regras que mostrariam como andar pelos espaços livres do labirinto e nas subclasses seriam colocadas as regras específicas de cada personagem. Neste exemplo, o uso da herança

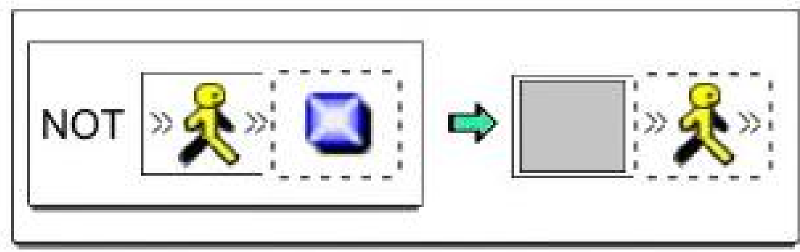


Figura 5.6: Tabajara – regra de movimento para a classe *Moveable*.

faria com que a quantidade de regras necessárias para ensinar o Pacman e os fantasmas a andarem por espaços vazios fosse reduzida para a metade. As regras seriam criadas apenas na classe básica, o que reduziria o trabalho e o tempo gasto com a implementação do jogo.

## 5.4 O jogo Pacman no Tabajara Animator

Para que as sugestões propostas no Capítulo 3 pudessem ser implementadas e testadas, foi desenvolvido o ambiente Tabajara Animator. Uma implementação completa do jogo Pacman, descrito na Seção 5.1, foi desenvolvida no sistema, utilizando as idéias apresentadas nesta dissertação.

A Figura 5.6 apresenta uma regra especificada para uma classe denominada *Moveable*. Essa regra deve ser interpretada da seguinte maneira: se à frente do personagem não houver uma parede (representada na figura por um bloco quadrado da cor azul), ele então vai se deslocar para frente. A classe *Moveable* é definida como a superclasse da classe *Ghost*, usada para representar os fantasmas do jogo, e também da classe *Pacman*, usada para representar o personagem principal, controlado pelo usuário. Essa associação pode ser observada na representação da hierarquia apresentada na Figura 4.10, do Capítulo 4. Outra classe envolvida na descrição do jogo é a classe *Brick*, usada para representar as paredes do labirinto.

Para construir a regra apresentada, devem ser executadas as tarefas enumeradas a seguir:

1. A classe para definição da regra é selecionada; neste caso, a classe *Moveable*. Na

Figura 5.6, a classe está representada por uma pessoa caminhando, figura que foi previamente escolhida pelo usuário.

2. Cria-se uma nova regra de reescrita antes-depois e define-se a pré-condição e a atualização a ser executada, caso essa condição seja satisfeita. A pré-condição, nesse caso, é representada por uma área selecionada ao lado do símbolo da classe, para onde foi arrastado um símbolo associado à classe *Brick*. Uma negação foi acrescentada à condição, representada pelo texto "`not`", indicando que ela será satisfeita se, na posição definida, não houver um objeto da classe *Brick*.

3. Definem-se as atualizações a serem executadas, caso a condição seja satisfeita. Isso pode ser realizado alterando-se propriedades dos elementos usados na condição. Neste caso, o símbolo associado à classe *Moveable* foi deslocado para a nova posição.

A regra da Figura 5.6 utiliza os três recursos sugeridos no Capítulo 3: herança, perspectiva em primeira pessoa e regras com condições negativas. A forma como esses recursos são aplicados na regra é discutida a seguir.

A utilização de herança na regra da Figura 5.6 é evidente. A regra é compartilhada por todas as subclasses de *Moveable*, assim as classes *Ghost* e *Pacman* vão herdar esse comportamento, podendo acrescentar regras específicas. Uma regra específica associada à movimentação do *Pacman* pode ser algo semelhante ao apresentado na Figura 5.1, no sistema AgentSheets. Essas regras são responsáveis por mudar a direção do Pacman. Uma regra específica para *Ghost* envolve a definição de uma nova orientação para os fantasmas, quando se encontrarem com uma parede do labirinto. Para definir essa nova orientação, na implementação desenvolvida, foi seguida a estratégia descrita na Seção 5.1 de atribuir valores para cada posição do cenário do jogo.

A utilização de perspectiva em primeira pessoa, na regra da Figura 5.6, é mais sutil. A pré-condição utilizada, que permite verificar a situação de uma área selecionada próxima a um personagem, só é válida para subclasses de *VisibleClass*. Um dos atributos de *VisibleClass* representa um valor associado à *rotação* do personagem.

Se esse valor é alterado, ou seja, o personagem altera sua orientação, o sistema se encarrega de calcular automaticamente uma nova localização para a área selecionada. É por essa razão que não é necessário criar quatro regras diferentes, uma para cada sentido de deslocamento do personagem, como foi apresentado na Figura 5.2. Basta criar uma regra, como a especificada na Figura 5.6, para que o personagem possa se movimentar, nas quatro direções, pelos espaços vazios do labirinto.

As ferramentas de simulação animada, apresentadas na Seção 4.4, oferecem facilidades para tirar proveito dos recursos de cálculo de posições baseados na orientação dos personagens. É definida apenas uma representação gráfica para cada personagem. Por exemplo, basta definir um desenho do Pacman se deslocando para a direita, para combinar com a regra da Figura 5.6. As janelas de perspectiva em terceira pessoa se encarregam de gerar automaticamente novas representações gráficas para os personagens, de acordo com sua rotação. As janelas de perspectiva em primeira pessoa geram imagens rotacionadas para todo o ambiente, mantendo a imagem original do personagem intacta.

As novas representações gráficas geradas para os personagens de acordo com a rotação realizada podem criar situações inusitadas. Por exemplo, se o Pacman estiver andando para a direita e sofrer uma rotação de cento e oitenta graus, ele aparecerá andando para a esquerda mas a sua imagem estará de cabeça para baixo. Isso irá acontecer porque o sistema apenas gira a imagem do personagem. Mecanismos para corrigir esse problema estão especificados no Capítulo 6, na Seção 6.2, como parte dos trabalhos futuros propostos por esta dissertação.

Usando a herança, a perspectiva em primeira pessoa e a condição negativa, a regra da Figura 5.6 elimina a necessidade de se criar quatro regras para a movimentação do Pacman por espaços livres e quatro regras para movimentação dos fantamas por espaços livres. E elimina também a necessidade de se criar as quatro regras que ensinam o Pacman a andar por espaços que contenham pílulas e as quatro regras que ensinam os fantasmas a andarem por espaços que contenham pílulas. Pode-se imaginar a extensão desse benefício, se o jogo for modificado. Ou seja, em vez de dezesseis regras será necessária a regra apresentada na Figura 5.6 e uma regra para ensinar o

Pacman a comer a pílula e com isso aumentar a contagem de pontos. Além disso, por exemplo, se forem acrescentados novos personagens que se movem pelo labirinto respeitando a limitação das paredes, a herança elimina a necessidade de se criar novas regras. Se for utilizado um labirinto em que exista deslocamento na diagonal, a perspectiva em primeira pessoa, elimina a necessidade de criar novas regras para a movimentação nas outras direções. Nos ambientes AgentSheets e StageCast Creator seria necessário criar novas regras nas duas situações descritas acima.

## 5.5 Conclusão

Este capítulo abordou a implementação do jogo Pacman em três ambientes: AgentSheets, Stagecast Creator e Tabajara Animator. Essas implementações permitiram confirmar os benefícios, diminuição do números de regras criadas e do tempo gasto com a programação, gerados pela utilização das sugestões propostas.

A Tabela 5.1 mostra a quantidade de regras necessárias para a implementação do jogo Pacman nos três ambientes citados acima. No ambiente Tabajara Animator a contagem das regras foi feita com a utilização da técnica de regras escritas com a perspectiva em primeira pessoa. Apenas essa inovação já trouxe uma diminuição significativa do número de regras necessárias. Como a implementação do jogo foi feita utilizando também a herança e o uso de regras visuais com condições negativas, o número total de regras necessárias para implementar essas funções cai para seis, isto é mostrado na Tabela 5.2. Essa diminuição ocorre porque as regras necessárias para andar por espaços vazios e por espaços com pílulas são agrupadas em uma só regra. Essa regra deveria ser criada para os personagens Pacman e Ghost, mas com o uso da herança, ela é criada apenas para a classe básica Moveable. Quando o Pacman se movimenta por espaços com pílulas, ele precisa comê-las e com isso aumentar o contagem de pontos do jogador, o que não acontece com o personagem do Fantasma. Essa situação faz com que seja necessário criar uma regra adicional para o Pacman. Essa regra ensina o que deve ser feito quando o seu movimento acontecer por um espaço que contenha uma pílula, a pílula deve desaparecer e a contagem de pontos

deve aumentar. Mesmo com a criação dessa regra específica, a diminuição do número de regras em relação aos outros sistemas é significativa.

| Tipo de Função Implementada | Quantidade de regras no AgentSheets | Quantidade de regras no Creator | Quantidade de regras no Tabajara |
|---|---|---|---|
| 1 - Movimentação do Pacman por espaços vazios | 4 | 4 | 1 |
| 2 - Movimentação do Fantasma por espaços vazios | 4 | 4 | 1 |
| 3 - Movimentação do Pacman por espaços que possuam pílulas | 4 | 4 | 1 |
| 4 - Movimentação do Fantasma por espaços que possuam pílulas | 4 | 4 | 1 |
| 5 - Mudança de direção do Pacman | 4 | 4 | 4 |
| 6 - Aumento da contagem de pontos | 0 | 0 | 1 |
| Total de Regras | 20 | 20 | 9 |

Tabela 5.1: Diferenças na quantidade de Regras.

| Tipo de Função Implementada | Quantidade de regras no Tabajara |
|---|---|
| 1 - Movimentação do Pacman por espaços vazios<br>2 - Movimentação do Fantasma por espaços vazios<br>3 - Movimentação do Pacman por espaços que possuam pílulas<br>4 - Movimentação do Fantasma por espaços que possuam pílulas | 1 |
| 5 - Aumento da contagem de pontos | 1 |
| 6 - Mudança de direção do Pacman | 4 |
| Total de Regras | 6 |

Tabela 5.2: Quantidade de Regras para o Ambiente Tabajara Animator.

# Capítulo 6

# Conclusões e Trabalhos Futuros

Essa dissertação apresentou sugestões que podem ser implementadas em sistemas que produzem animação usando técnicas de programação por demonstração (PBD). A sua contribuição é mostrar que essas sugestões, que possuem como objetivo reduzir a quantidade de regras necessárias para demonstrar um comportamento, podem ser implementadas e utilizadas de forma prática pelo usuário. Apresentou também uma ferramenta, o Tabajara Animator, parcialmente implementada, onde essas idéias puderam ser implementadas e testadas. Foi realizada uma comparação com os principais sistemas similares, usando um exemplo significativo para esse tipo de aplicação, o jogo *Pacman*. O fato de considerar apenas os sistemas AgentSheets e Stagecast Creator na comparação não deve ser visto como uma fraqueza do experimento, pois essas ferramentas são largamente utilizadas e reconhecidas internacionalmente.

Este capítulo apresenta as conclusões do trabalho e enumera alguns possíveis trabalhos futuros.

## 6.1 Conclusões

A principal conclusão do trabalho é que as sugestões propostas podem ser implementadas e realmente podem reduzir o número de regras gastas para demonstrar um comportamento. O número de regras que devem ser produzidas para se construir uma animação pode ser reduzido, quando se utiliza os conceitos de herança, perspectiva em

primeira pessoa e regras com condições negativas. Essa redução no número de regras não dificulta o trabalho do usuário na hora de criar a sua simulação ou jogo. Logo que a ferramenta Tabajara Animator estiver totalmente construída, podem ser feitos testes com usuários para verificar o grau de facilidade no uso dos recursos propostos.

O recurso de perspectiva em primeira pessoa não necessita de nenhum conhecimento adicional para os usuários, além daqueles requeridos pelos sistemas similares. Todo o trabalho adicional fica a cargo da própria ferramenta, que deve executar cálculos para identificar áreas do ambiente baseando-se na orientação dos personagens, e oferecer recursos de apresentação gráfica mais sofisticados, que também levem em conta a orientação dos personagens.

O conceito de herança, por outro lado, deve ser corretamente entendido para que um usuário possa obter os seus benefícios associados, e é algo que não é exigido nos sistemas similares. Entretanto, isso também pode ser visto como uma vantagem, pois esse tipo de sistema é conhecido por motivar o aprendizado dos usuários, inclusive crianças, e assim constituirá uma boa oportunidade para um contato inicial com o interessante conceito de herança. Além disso, o sistema pode ser usado, pelos professores de programação, como uma ferramenta de apoio na hora de ensinar os conceitos de herança para os alunos. Por meio da PBD e de um ambiente totalmente gráfico os conceitos de classe, classe derivada, métodos herdados e outros se tornam mais concretos e mais fáceis de serem assimilados.

O conceito de regras com condições negativas também não exige conhecimento adicional do usuário. É apenas mais um operador existente e que passa a ser utilizado na PBD. O usuário pode escolher se quer ou não utilizar esse recurso, que se bem utilizado permite que comandos sejam executados com um número menor de regras.

## 6.2 Trabalhos Futuros

Os planos para trabalhos futuros derivados dessa dissertação são bastante variados. Eles envolvem os três componentes em que o sistema foi dividido: o modelo de execução, as ferramentas para produção de código usando PBD e as ferramentas para

produção de simulação animada.

Nas ferramentas para construção de regras usando PBD, estudam-se mecanismos de inferência sofisticados, usando técnicas de inteligência artificial, similares aos aplicados pelo sistema Gamut. A vantagem sobre o Gamut seria, nesse caso, a possibilidade de apresentar ao usuário uma representação gráfica das regras geradas automaticamente pelos mecanismos de inferência. Dessa forma, o usuário poderia entender o que foi induzido e até alterar à mão alguma parte dessas regras geradas.

Os mecanismos para simulação animada oferecem uma infinidade de oportunidades para desenvolvimento de novos recursos. Estão sendo desenvolvidas duas novas categorias de janelas para apresentação gráfica do modelo. A primeira oferece a possibilidade de se definir diferentes imagens para cada personagem, que são selecionadas de acordo com atributos desses personagens. Pode-se, por exemplo, selecionar uma imagem diferente para cada intervalo de variação da rotação, criando uma animação muito mais sofisticada. Esse recurso acaba com o incoveniente do personagem andar de cabeça para baixo ou andar de costas, por exemplo, e é extremamente importante para dar veracidade à simulação ou jogo criado. A segunda categoria é uma extensão da perspectiva em terceira pessoa com foco em um personagem central. Será permitida a definição de vários personagens que a janela deve obrigatoriamente exibir. A janela vai se dividir automaticamente quando todos os personagens não puderem ser mostrados ao mesmo tempo, e se fundir novamente quando isso for possível.

Outras formas de utilização do código produzido, além da simulação animada, estão sendo estudadas. A adoção de uma representação em XML permitiu uma fácil comunicação com outras ferramentas e isso está sendo utilizado dentro de mais dois projetos associados. Em um dos projetos, procura-se definir o comportamento de agentes que vão ser executados dentro de um jogo de ação em terceira dimensão. O ambiente escolhido para isso foi o jogo comercial *Unreal Tournament* (Gerstmann, 1999). O trabalho consiste em traduzir código escrito em ASM-OBJ para programas em Java, utilizando as facilidades oferecidas pelo pacote *Gamebots* (Adobbati et al., 2001; Kaminka et al., 2002). Outro projeto envolve o estudo da aplicação de PBD para programação do comportamento de robôs. A plataforma escolhida foi o *Kit Lego*

*Mindstorm* (Erwin, 2001), por ser bastante difundida e de fácil utilização. Neste caso, o trabalho consiste também em traduzir código escrito em ASM-OBJ para programas Java, utilizando as facilidades oferecidas pelo pacote *LEJOS* (Laverde et al., 2002).

Extensões para o modelo de execução estão em planejamento. Uma dessas extensões é a inclusão da possibilidade de alteração do mecanismo de concorrência. No modelo atual, são coletadas as alterações produzidas por todos os agentes ativos a cada passo da execução, e essas atualizações são disparadas em paralelo. A extensão deve permitir ao usuário definir qualquer tipo de ordenação para a seleção dos agentes a serem disparados a cada passo, chegando ao ponto de especificar uma execução onde, a cada passo, apenas um agente é autorizado a executar um movimento. Outra extensão para o modelo de execução é a inclusão de mecanismos de modularização, semelhantes aos adotados pela linguagem VAT do AgentSheets. Os módulos devem ser conjuntos de regras reunidas, que podem ser disparadas usando o identificador do módulo. Esse recurso permite uma melhor organização das regras e possibilita o reaproveitamento delas.

Usando esse identificador, é proposta ainda a adoção de um mecanismo de polimorfismo, onde uma subclasse pode redefinir o comportamento de sua superclasse, associando um novo conjunto de regras a um identificador de módulo usado na superclasse. Esse recurso elimina problemas gerados pelo uso da herança, como o citado na seção 3.2. Essa é mais uma idéia que poderia ser estendida aos sistemas estudados, a exemplo do uso de herança, da perspectiva em primeira pessoa e das regras com condições negativas, discutidos nesta dissertação.

# Referências Bibliográficas


Adobbati, R., Marshall, A. N., Scholer, A., Tejada, S., Kaminka, G., Schaffer, S., and Sollitto, C. (2001). Gamebots: A 3d virtual world test-bed for multi-agent research. In *Proceedings of the International Conference on Autonomous Agents (Agents-2001) - Workshop on Infrastructure for Agents, MAS, and Scalable MAS.*

Börger, E. (2000a). Abstract state machines at the cusp of the millenium. In *ASM '00: Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, pages 1–8, London, UK. Springer-Verlag.

Börger, E. (2000b). The Origins and the Development of the ASM Method for High Level System Design and Analysis. *Journal of Universal Computer Science*, 8(1):2–74.

Börger, E. and Stärk, R. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag.

Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523.

Cypher, A., Halbert, D. C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B. A., and Turransky, A., editors (1993). *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, USA.

Cypher, A. and Smith, D. C. (1995). Kidsim: end user programming of simulations. In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 27–34, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

DeMaria, R. and Wilson, J. L. (2003). *High Score!: The Illustrated History of Electronic Games*. McGraw-Hill Osborne Media.

Erwin, B. (2001). *Creative Projects with LEGO Mindstorms*. Addison-Wesley.

Gamma, E., Johnson, R., Vlssides, J., and Helm, R. (2005). *Padrões de Projeto*. Bookman Companhia ED.

Gerstmann, J. (1999). Unreal Tournament : Action game of the year. Gamespot. (retrieved 28 May, 2006, from www.gamespot.com/features/1999/p3_01a.html).

Gottlob, G., Kappel, G., and Schrefl, M. (1991). Semantics of Object-Oriented Data Models – The Evolving Algebra Approach. In Schmidt, J. W. and Stogny, A. A., editors, *Next Generation Information Technology*, volume 504 of *LNCS*, pages 144–160. Springer.

Gurevich, Y. (1994). Evolving algebras 1993: Lipari guide. In Börger, E., editor, *Specification and Validation Methods*, pages 9–37. Oxford University Press.

Gurevich, Y., Rossman, B., and Schulte, W. (2004). Semantic Essence of Asml. Technical Report MSR-TR-2004-27, Microsoft Research.

Kahn, K. (2001). Generalizing by removing detail: how any program can be created by working with examples. pages 21–43.

Kahn, K. M. (1996). Toontalk - an animated programming environment for children. *J. Vis. Lang. Comput.*, 7(2):197–217.

Kaminka, G. A., Veloso, M., Schaffer, S., Sollitto, C., Adobbati, R., Marshal, A. N., Scholer, A. S., and Tejada, S. (2002). Gamebots: the ever-challenging multi-agent research test-bed. *Communications of the ACM*.

Lau, T., Wolfman, S., Domingos, P., and Weld, D. S. (2001). Learning Repetitive Text-editing Procedures with SMARTedit. In Lieberman, H., editor, *Your Wish is My Command: Programming by Example*, pages 209–225. Morgan Kaufmann.

Laverde, D., Ferrari, G., and Stuber, J. (2002). *Programming Lego Mindstorms with Java.* Syngress Publishing Inc.

Lieberman, H., Nardi, B. A., and Wright, D. J. (2001). Training Agents to Recognize Text by Example. In Lieberman, H., editor, *Your Wish is My Command: Programming by Example*, pages 227–244. Morgan Kaufmann.

Masui, T. (2001). Composition by Example. In Lieberman, H., editor, *Your Wish is My Command: Programming by Example*, pages 191–208. Morgan Kaufmann.

McDaniel, R. G. and Myers, B. A. (1998). Building applications using only demonstration. In *IUI '98: Proceedings of the 3rd international conference on Intelligent user interfaces*, pages 109–116, New York, NY, USA. ACM Press.

McDaniel, R. G. and Myers, B. A. (1999). Getting more out of programming-by-demonstration. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 442–449, New York, NY, USA. ACM Press.

McLaughlin, B. (2002). *Java and XML Data Binding.* O'Reilly.

Oliveira, F. F., Bigonha, R. S., and Bigonha, M. A. S. (2004). Otimização de código em ambiente de semântica formal executável baseado em asm. VIII Simpósio Brasileiro de Linguagens de Programação.

Rader, C., Brand, C., and Lewis, C. (1997). Degrees of comprehension: children's understanding of a visual programming environment. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 351–358, New York, NY, USA. ACM Press.

Repenning, A. (1991). Creating user interfaces with agentsheets. In *Symposium on Applied Computing*, pages 190–196, Kansas City, MO.

Repenning, A. (2000). Agentsheets: an interactive simulation environment with end-user programmable agents. Proceedings of the IFIP Conference on Human Computer Interaction (INTERACT '2000, Tokyo, Japan).

Repenning, A. and Ambach, J. (1996). Tactile programming: A unified manipulation paradigm supporting program comprehension, composition and sharing. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, pages 102–109, Washington, DC, USA. IEEE Computer Society.

Repenning, A. and Perrone, C. (2000). Programming by example: programming by analogous examples. *Commun. ACM*, 43(3):90–97.

Repenning, A. and Sumner, T. (1995). Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. *IEEE Computer*, 28(3):17–25.

Ruvini, J.-D. and Dony, C. (2001). Learning users' habits to automate repetitive tasks. In Lieberman, H., editor, *Your Wish is My Command: Programming by Example*, pages 271–296. Morgan Kaufmann.

Sebesta, R. W. (2000). *Conceitos de Linguagens de Programação*. Addison-Wesley Publishing Company, Bookman.

Shapiro, E. (1989). The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3):413–510.

Smith, D. C. (2000). Building personal tools by programming. *Communications of the ACM*, 43(8):92–95.

Smith, D. C., Cypher, A., and Tesler, L. (2000). Programming by example: novice programming comes of age. *Commun. ACM*, 43(3):75–81.

Sugiura, A. (2001). Web Browsing by Demonstration. In Lieberman, H., editor, *Your Wish is My Command: Programming by Example*, pages 61–86. Morgan Kaufmann.

van der Vlist, E. (2002). *XML Schema - The W3C's Object-Oriented Descriptions for XML*. O'Reilly.