UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
DEPARTAMENTO DE INFORMÁTICA
MESTRADO EM INFORMÁTICA

LUIZ MANOEL GEROSA

Um Framework para Criação Cooperativa de Jogos

VITÓRIA
2008

LUIZ MANOEL GEROSA

Um Framework para Criação Cooperativa de Jogos

Dissertação apresentada ao Programa de Mestrado em Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Mestre em Informática.

Orientador: Prof. Dr. Davidson Cury

VITÓRIA
2008

LUIZ MANOEL GEROSA

Um Framework para Criação Cooperativa de Jogos

COMISSÃO EXAMINADORA

Prof. Dr. Davidson Cury
UFES - Universidade Federal do Espírito Santo
Orientador

Prof. Dr. Orivaldo de Lira Tavares
UFES - Universidade Federal do Espírito Santo

Prof. Dr. Alexandre Ibrahim Direne
UFPR - Universidade Federal do Paraná

Vitória - ES, 22 de agosto de 2008.

RESUMO

A capacidade de entretenimento dos jogos eletrônicos e sua popularidade entre crianças e adolescentes despertaram o interesse de diversos setores da sociedade em usá-los na educação. Na abordagem tradicional, o jogador aprende enquanto joga. Na abordagem construcionista, seguida nesta dissertação, o jogador aprende construindo seus próprios jogos.

Nesta dissertação é proposta a ferramenta de criação cooperativa de jogos Affinity, que objetiva o uso por indivíduos que não sabem programação, a extensibilidade para que os próprios usuários incorporem novas funcionalidades e o trabalho cooperativo entre os membros da comunidade criativa. Nesse intuito, foi adotada a abordagem de Desenvolvimento Baseado em Componentes para decompor as funcionalidades dos jogos em componentes que possam ser criados, compartilhados e reusados pelos próprios usuários.

Palavras-chave

Jogos Educacionais; Construcionismo; Desenvolvimento Baseado em Componentes; Framework; Cooperação

ABSTRACT

The capacity of games to entertain and their popularity among children and teenagers raised the interest of various sectors of society to use them in education. In the traditional approach, the player learns while is playing. In constructionist approach, followed in this dissertation, the player learns building their own games.

In this dissertation is proposed the cooperative game construction tool Affinity, which aims the usability for who do not know programming, the extensibility to the users incorporate new features and the cooperative work between the members of the creative community. To that end, the Component Based Development approach was adopted to decompose the functionality of games into components that can be created, shared and reused by the users.

Keywords

Educational Games; Constructionism; Component Based Development; Framework; Cooperation

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

Introdução

Uma mídia que se tornou parte da cultura contemporânea são os jogos eletrônicos. De acordo com a ESA[1] (*Entertainment Software Association*), uma associação que levanta dados sobre a indústria de jogos americana, os jogos eletrônicos (de computadores e de consoles) foram responsáveis em 2007 por 9,5 bilhões de dólares em vendas. Outro fato interessante é que a média de idade dos jogadores é de 33 anos – mostrando que os jogos não estão restritos apenas às crianças e adolescentes. No Brasil, uma pesquisa realizada no Rio de Janeiro com garotos de classe média, na faixa etária de 10 a 17 anos, mostrou que 38% dos entrevistados jogam mais de 20 horas por semana [Clua et. al., 2002].

A capacidade de entretenimento dos jogos e sua popularidade entre crianças e adolescentes despertaram o interesse de acadêmicos, escritores, fundações, projetistas de jogos, empresas e governos em usar jogos eletrônicos na educação. Para Kafai [2001], duas possíveis abordagens para o uso de jogos na educação são a instrucionista e a construcionista.

A abordagem instrucionista está presente na maioria dos jogos voltados para ensinar. O jogador aprende algo enquanto faz uma determinada atividade, pois há uma integração do conteúdo que será ensinado com a idéia do jogo.

A outra abordagem é a construcionista, na qual a aprendizagem ocorre através da criação de jogos. O criador aprende através da reflexão, da formulação de estratégias e do compartilhamento de idéias subjacentes às suas criações [Harel, 1991]. Essa abordagem também desenvolve importantes aptidões de aprendizagem que dão flexibilidade e segurança numa sociedade caracterizada pelas constantes mudanças.

A criação de jogos pelos próprios jogadores está inserida no contexto mais amplo da Cultura Participativa, na qual os indivíduos criam e compartilham idéias e mídias em blogs ou em sites colaborativos como Flickr[2] (para fotos) e Youtube[3] (para vídeos) [Jenkins, 2006].

---

[1] www.theesa.com

[2] www.flickr.com

[3] www.youtube.com

Nessa cultura, os indivíduos não são apenas consumidores de conteúdos multimídia, mas são também produtores. A Nokia aponta essa tendência prevendo que, no ano de 2012, 25% do entretenimento mundial será criado e consumido dentro de comunidades [Nokia, 2007].

1.1. Objetivo

Este trabalho tem o objetivo de investigar os efeitos do processo de criação de jogos na educação e de desenvolver o protótipo do Affinity, uma ferramenta de criação de jogos que contempla os seguintes objetivos:

- Ser acessível para pessoas que não sabem programação.
- Ser extensível para que a própria comunidade de usuários possa incorporar novas funcionalidades.
- Possibilitar a construção cooperativa de jogos.
- Utilizar uma base tecnológica com suporte a recursos gráficos avançados.

1.2. Metodologia

Este trabalho teve início com uma revisão bibliográfica em artigos, livros e trabalhos acadêmicos sobre o uso de jogos eletrônicos na educação. Nesse estudo, foram investigados dois aspectos: o jogador como consumidor de jogos e o jogador como criador de jogos.

Após a constatação das capacidades pedagógicas do processo de criação de jogos, buscou-se na literatura por ferramentas apropriadas a esse processo. Entre os aspectos observados nessas ferramentas, destacam-se a acessibilidade para pessoas que não conhecem programação, os recursos de cooperação, a extensibilidade e a tecnologia utilizada pela ferramenta.

A partir desses estudos, foi proposta a ferramenta de criação de jogos Affinity. A primeira etapa do desenvolvimento foi pesquisar sobre os conceitos comuns em diversos jogos e propor uma arquitetura que pudesse contemplar os objetivos propostos para a ferramenta.

Com resultado desses estudos, a pesquisa foi direcionada para o Desenvolvimento Baseado em Componentes (DBC). Constatou-se que essa abordagem pode ser usada tanto no desenvolvimento da ferramenta quanto na criação dos jogos pelos usuários. Foram investigados os conceitos, os benefícios e sua aplicabilidade no desenvolvimento de jogos.

Definida a abordagem, o trabalho de desenvolvimento do protótipo teve início utilizando a tecnologia Microsoft XNA. A escolha dessa tecnologia foi fundamentada nos objetivos propostos para a ferramenta.

1.3. Organização do Trabalho

Além desta introdução, o restante do trabalho está organizado como segue:

- Capítulo 2 – *Jogos Eletrônicos na Educação*: fornece uma visão geral do papel dos jogos eletrônicos na educação. É apresentada uma comparação entre as abordagens instrucionista e construcionista, discutindo os principais benefícios de cada uma.
- Capítulo 3 – *Ferramentas de Desenvolvimento de Jogos*: fornece uma visão geral das tecnologias empregadas no desenvolvimento de jogos, enfatizando sua aplicabilidade na educação, principalmente quando os usuários são iniciantes e não conhecem programação.
- Capítulo 4 – *Análise e Projeto de Jogos*: apresenta uma análise e um projeto arquitetural aplicável no desenvolvimento de diversos jogos. Os resultados obtidos nesse capítulo servem de base para o desenvolvimento da ferramenta Affinity.
- Capítulo 5 – *Desenvolvimento Baseado em Componentes*: fornece uma visão geral da abordagem de desenvolvimento baseada em componentes. São apresentados os conceitos gerais de componente e *framework* de componente, além dos principais benefícios e dificuldades da abordagem.
- Capítulo 6 – *Affinity: Uma Ferramenta de Criação Cooperativa de Jogos*: apresenta a ferramenta de criação de jogos proposta neste trabalho. São apresentados os objetivos, o modelo conceitual, a arquitetura e a tecnologia empregada no desenvolvimento da ferramenta Affinity.
- Capítulo 7 – *Estudo de Caso: Breakout*: apresenta um estudo de caso da ferramenta Affinity na criação do jogo Breakout. Nesse estudo é observado se a ferramenta consegue contemplar alguns dos objetivos propostos.
- Capítulo 8 – *Conclusão*: apresenta as conclusões sobre o trabalho desenvolvido e as propostas para trabalhos futuros.

- Anexo A – *Descrição dos Componentes*: fornece uma descrição completa de todos os componentes disponíveis na versão atual do Affinity.

Capítulo 2

Jogos Eletrônicos na Educação

Este capítulo apresenta o estado da arte dos jogos eletrônicos na educação e está organizado da seguinte maneira: a Seção 2.1 faz uma comparação entre o instrucionismo e o construcionismo, a Seção 2.2 apresenta o papel que os jogos eletrônicos têm na educação dos jogadores e a Seção 2.3 explora como os jogos podem ser usados na abordagem construcionista.

2.1. Instrucionismo x Construcionismo

O pensamento das pessoas sobre educação está frequentemente associado à informação. Elas pensam sobre qual informação é mais importante para conhecerem, qual a melhor maneira de transmitir uma informação de uma pessoa (professor) para outra (aluno) e quais são as melhores maneiras de representar a informação para torná-la compreensível [Resnick, 2002].

A abordagem que usa o computador como meio para transmitir a informação ao aluno mantém a prática pedagógica vigente. O computador é usado para passar uma série de informações na forma de tutoriais e exercícios-e-prática. Do ponto de vista pedagógico, esse é o paradigma instrucionista. Isso tem facilitado a implantação do computador na escola, pois não quebra a dinâmica por ela adotada e não exige muito investimento na formação do professor, que basta ser treinado nas técnicas de uso de cada software [Valente, 1999].

Entretanto o foco na informação limita e distorce tanto o papel da educação quanto o papel do computador. A aprendizagem não é simplesmente uma questão de transmitir a informação e o computador pode ser muito mais que uma máquina para manipular a informação [Resnick, 2002, p.32].

Nos últimos cinquenta anos, psicólogos, fundamentados nos princípios do Construtivismo Cognitivo de Jean Piaget, defendem que a aprendizagem é um processo ativo no qual as pessoas constroem novos entendimentos do mundo à sua volta através da exploração, experimentação, discussão e reflexão.

Da mesma forma, o computador é um novo meio para as pessoas criarem e se expressarem apropriadamente. Resnick [2002, p.33] faz a seguinte analogia:

> Considere estas três coisas: computador, televisão, pintura a dedo. Qual dos três é diferente? Para a maioria das pessoas, a resposta parece óbvia: "pintura a dedo" não se encaixa. Afinal, o computador e a televisão foram inventados no século XX, ambos envolvem tecnologia eletrônica e ambos podem trazer informações para um grande número de pessoas. Nada disso é verdade para a pintura a dedo.
>
> Mas até nós começarmos a pensar no computador como pintura a dedo e menos como televisão, os computadores não vão mostrar todo o seu potencial. Da mesma forma que a pintura a dedo (e ao contrário da televisão), os computadores podem ser usados para desenhar e criar coisas. Além de acessar páginas na Internet, as pessoas podem criar suas próprias páginas. Além de baixar arquivos de música em MP3, as pessoas podem compor sua própria música. Além de jogar SimCity, as pessoas podem criar seus próprios mundos simulados.

Seymour Papert, criador da linguagem Logo, foi um dos precursores da idéia de usar os computadores como apoio para o processo de construção do conhecimento [Papert, 1980]. Ele expandiu o construtivismo de Piaget com a metodologia construcionista, que defende a idéia de que as melhores experiências de aprendizagem acontecem quando as pessoas estão engajadas em projetar e criar coisas que tenham significado para elas ou para os outros ao seu redor. Uma das diferenças para o construtivismo é a especial importância atribuída ao papel das construções "no mundo", tornando-se uma doutrina menos puramente mentalista [Papert, 1993].

Nessa abordagem, a aprendizagem ocorre através da reflexão, da formulação de estratégias e do compartilhamento de idéias subjacentes às criações [Harel, 1991]. Os estudantes também aprendem a ser um bons projetistas: como conceituar um projeto, como fazer uso dos recursos disponíveis, como persistir e achar alternativas quando as coisas dão erradas e como colaborar com os outros. Em resumo, eles aprendem a gerenciar um projeto complexo do início ao fim [Resnick, 2002, p.34].

O trabalho desta dissertação segue a linha construcionista através da criação de jogos eletrônicos, que será explorado com mais detalhes na Seção 2.3. A próxima seção apresenta uma visão geral do papel que os jogos têm na educação dos jogadores.

2.2. Jogadores como Consumidores

Jogos eletrônicos trazem algum benefício para seus jogadores ou são apenas "perda de tempo", como muitos adultos costumam afirmar? Pesquisas mostram que os jogos, mesmo sem um propósito educacional explícito, desenvolvem importantes habilidades cognitivas,

como pensamento estratégico e raciocínio rápido [Prensky, 2006] e que os princípios de aprendizagem encontrados nos jogos estão próximos aos melhores princípios de aprendizagem da ciência cognitiva [Gee, 2003, p.7].

Jogos usados com o propósito educacional, conhecidos com *serious games* (jogos sérios), também despertaram o interesse em diversos setores da sociedade. Governos, militares, empresas e escolas estão empregando jogos para conscientização, treinamento e educação.

Portanto, os jogos devem ser compreendidos como uma tecnologia educacional presente no contexto cibercultural do século XXI capaz de potencializar o processo de aprendizagem dos aprendizes contextualizados em uma sociedade do conhecimento [Clua et. al., 2002].

2.2.1. Benefícios Educacionais

Os jogos eletrônicos são reconhecidos por desenvolver nos jogadores importantes habilidades cognitivas [Prensky, 2006]:

- Os jogos eletrônicos afetam positivamente a atenção visual seletiva, ou seja, os jogadores aprendem a identificar e a se concentrar nas coisas mais importantes quando muitas outras estão acontecendo ao mesmo tempo;
- eles aprendem a deduzir as regras do jogo através de fatos observados do mundo;
- eles aprendem a manipular sistemas complexos através da tentativa e erro, e a formular estratégias para superar obstáculos encontrados;
- eles aprendem a tomar boas decisões rapidamente, recuperando informações de muitas fontes e organizando dados dispersos em uma visão coerente do mundo;
- eles aprendem a realizar várias tarefas ao mesmo tempo e fazê-las todas bem.

Dominar alguns jogos não é uma tarefa fácil e exigem muita dedicação dos jogadores. Apesar disso, os jogadores conseguem aprender as regras e formular estratégias necessárias para superar seus desafios. Gee [2003] argumenta que isso é possível pois esses jogos usam princípios para “ensinar” sobre o jogo que estão próximos aos melhores princípios de aprendizagem da ciência cognitiva. Dois desses princípios são descritos a seguir:

- As pessoas têm dificuldade para entender e lembrar de informações que recebem fora de contexto ou muito tempo antes de usá-las. Bons jogos apresentam

informações contextualizadas nas reais necessidades e objetivos dos jogadores que, por sua vez, podem aplicá-las imediatamente para visualizar o resultado.

- Os jogos se adaptam ao estilo e à capacidade do jogador para manter o desafio e a motivação constantes. Muitas vezes os jogadores nem percebem essa adaptação como, por exemplo, quando um jogo de corrida dá vantagens competitivas a quem está atrás para deixar a disputa mais emocionante. A motivação é o fator mais importante que impulsiona a aprendizagem, e os jogos são projetados com objetivo primário de manter o jogador motivado.

Em alguns casos, a aprendizagem não ocorre somente na interação do jogador com o jogo. Os chamados jogos *multiplayer* possibilitam que vários jogadores interajam pela rede. Prensky [2006, p.103] cita um caso de um menino de 10 anos de idade que aplicou, mesmo sem saber, diversos conceitos de economia e negócios jogando um jogo *multiplayer* cujos objetivos eram cumprir missões, batalhar, trocar e ganhar tesouros. Ele percebeu que jogando sozinho não conseguiria ficar rico (acumular muitos tesouros) e chamou alguns amigos para formar uma equipe. Cada membro tinha suas tarefas e obrigações. Nessa brincadeira, o garoto aplicou conceitos como cadeia produtiva, oferta e demanda, acúmulo de capital, gerência, comunicação e até corrupção (quando um membro da equipe não quis compartilhar o tesouro que havia conseguido).

2.2.2. Serious Game

A expressão *serious game* é a união do termo *game*, que lembra primeiramente diversão, com o termo *serious*, que indica que jogos eletrônicos também podem ter propósitos sérios.

Michael & Chen [2006, p.21] definem *serious games* como jogos que não tem o entretenimento ou a diversão como seu propósito primário. Os autores afirmam, entretanto, que isso não significa que esses jogos não são divertidos e que a diversão não é um ingrediente que você insere dentro do jogo e sim um resultado do ato praticado pelo jogador. O mesmo jogo pode ser divertido para algumas pessoas e para outras não. Por exemplo, exemplo, o jogo *America's Army*[4] foi desenvolvido pelo exército americano para ser uma ferramenta de treinamento e de propaganda para atrair novos recrutas. Quando ele é jogado

[4] http://www.americasarmy.com

por um civil com intenção de vivenciar a realidade de um soldado, a experiência pode ser divertida. Entretanto, quando é jogado por um militar em treinamento que está aprendendo táticas que podem salvar sua vida em uma batalha real, o entretenimento perde importância.

Alguns jogos que foram desenvolvidos com propósito primário de divertir também podem ser considerados *serious game* quando são aplicados com propósitos sérios, como na educação. Por exemplo, Ilha & Cruz [2005] descrevem a experiência de usar o jogo *Sim City*, no qual o jogador planeja, constrói e gerencia uma cidade, na escola Escola Técnica do Vale do Itajaí. A experiência envolveu professores de geografia, matemática e português. A pesquisa explora e argumenta sobre a necessidade dos professores adequarem o conteúdo e a estratégia didática de suas disciplinas para o ambiente multimídia, onde ocorrem mudanças no gerenciamento e regulação das situações de aprendizagem.

2.3. Jogadores como Produtores

A proliferação da tecnologia e o amplo acesso à Internet possibilitaram o surgimento do que Jenkins [2006] chama de Cultura Participativa, na qual os indivíduos criam e compartilham idéias e mídias em blogs ou em sites colaborativos. Essa mudança também está acontecendo com os jogos eletrônicos. Ferramentas possibilitam indivíduos, com pouco ou até mesmo nenhum conhecimento em programação, participarem da criação de jogos.

2.3.1. Processo Criativo

Resnick [2007] caracteriza um processo criativo pelas seguintes etapas: **imaginar** o que fazer, **criar** um projeto baseado nas idéias, **brincar** com as criações, **compartilhar** as idéias e criações com os outros e **refletir** sobre a própria experiência. O processo de imaginar, criar, brincar, compartilhar e refletir inspira novas idéias, retornando à primeira etapa e criando um novo ciclo, conforme ilustrado na Figura 1. É importante ressaltar que essas etapas não são necessariamente sequenciais e distintas.

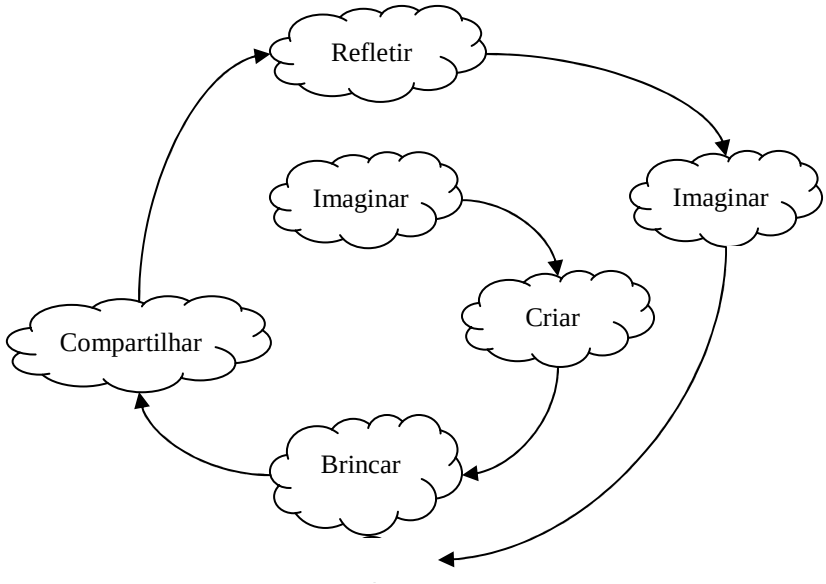

Figura 1. Etapas do processo criativo [Resnick, 2007].

Resnick acredita que passamos por esse processo no jardim de infância através de atividades de criação com, por exemplo, blocos de montar. Ele denominou essa abordagem de Aprendizagem Baseada no Jardim de Infância e afirma que, ao substituir blocos de montar por novas ferramentas, mídias e materiais, ela pode ser usada por pessoas de diferentes idades e com diferentes necessidades e estilos de aprendizagem. O processo criativo desenvolve o pensamento criativo, ou seja, a habilidade de achar soluções inovadoras para problemas inesperados, que é cada vez mais importante na vida profissional e pessoal [Resnick, 2007].

A criação de jogos eletrônicos é uma atividade que envolve um processo criativo. A primeira etapa para criar um jogo é imaginar a história, os personagens, a interface com o usuário e, o mais importante, como os jogadores interagirão no mundo do jogo (jogabilidade). Os próximos passos são criar o jogo utilizando ferramentas especializadas e brincar com o resultado da criação. A etapa de criação envolve utilizar diferentes mídias (textos, imagens, modelos 3D, animações, vídeos e sons) para programar o ambiente interativo que será explorado pelos jogadores.

O compartilhamento dos projetos é a próxima etapa no processo criativo. Algumas ferramentas disponibilizam uma comunidade *on-line* onde os usuários podem jogar, avaliar, reusar e compartilhar seus projetos e suas idéias. Os projetistas se tornam mais engajados com suas construções quando elas são compartilhadas com outros participantes da comunidade, que por sua vez, poderão se inspirar nos trabalhos disponibilizados. Monroy-Hernandez [2007] define quatro papéis ou estados de participação dos membros de uma comunidade criativa:

1. Consumidor Passivo: neste estado as pessoas acessam a comunidade para entender seus valores e idéias.
2. Consumidor Ativo: neste estado as pessoas participam na comunidade comentando, classificando e avaliando os projetos.
3. Produtor Passivo: neste estado os membros criam projetos, algumas vezes inspirados por outros projetos que eles viram na comunidade, mas não se sentem prontos para compartilhá-lo.
4. Produtor Ativo: no estado mais alto de participação, os membros da comunidade contribuem para o repositório de projetos.

A criação de jogos também favorece a colaboração, pois seu caráter interdisciplinar reflete na necessidade de diferentes perfis e habilidades: a habilidade artística para melhorar a aparência do jogo, a criatividade para criar a história, os personagens e definir a jogabilidade, e a lógica para programar o jogo.

A última etapa, refletir criticamente sobre a experiência de criação, é parte essencial do processo criativo. Quando essa abordagem de aprendizagem é aplicada em um ambiente escolar, o professor deve debater com os estudantes sobre as idéias que guiaram o projeto, sobre estratégias de como podem melhorá-lo e sobre conexões com conceitos científicos e fenômenos do mundo real [Resnick, 2007].

2.3.2. Benefícios Educacionais

Nas atividades de criação de jogos é possível trabalhar com uma grande variedade de assuntos e atender a estudantes com diferentes estilos e necessidades de aprendizagem. Quando os alunos criam seus próprios jogos educacionais sobre o conteúdo curricular, a aprendizagem ocorre através da reflexão sobre o assunto e do projeto de representações gráficas para melhor representá-lo. Professores mais experientes com a ferramenta também podem projetar jogos educacionais para aplicar em suas aulas, adequando-os as reais necessidades do conteúdo curricular que está sendo ensinado [Prensky, 2006, p.185].

Em Kafai [1995], foi elaborado um estudo que colocou uma classe de crianças de dez anos de idade para construir jogos educacionais sobre o conceito matemático de frações. As crianças tiveram que criar seus próprios personagens, histórias e interações durante um período de seis meses. Enquanto as crianças estavam imaginando qual jogo projetar e quais características incluir, elas também estavam envolvidas em entender o que estavam

aprendendo, no caso, as frações. Usando a linguagem Logo, as crianças também aprenderam conceitos de programação. Além disso, os estudantes lidaram com frações em seus jogos através de histórias e fantasias – contextos que são raramente promovidos em livros didáticos de matemática.

A interdisciplinaridade é uma característica muito importante para o ensino [Fazenda, 1994]. A atividade de criação de jogos é um processo criativo e lúdico que envolve conceitos em diferentes áreas: matemática, lógica, música, arte, gerenciamento de projetos e computação, além dos conceitos relacionados com a idéia do jogo que pode abranger qualquer disciplina (Física, História, Geografia, Ciências etc.).

Criar jogos eletrônicos ajuda os estudantes a desenvolverem um grau maior de fluência digital. Para explicar o que é fluência digital, Resnick [2002, p.32] faz uma analogia com a aprendizagem de uma língua estrangeira. Se uma pessoa só aprende algumas frases básicas, por exemplo, para pedir orientação na rua, ela não é considerada fluente naquela língua. Para ser fluente, é necessário saber se expressar com a língua. Analogamente, a maioria das pessoas que usam os computadores não possui fluência digital. Elas sabem usar um editor de texto e navegar na Internet, mas não sabem criar algo com significado. Segundo Resnick, a fluência digital é importante para facilitar a aprendizagem de novas tecnologias digitais, assim como saber ler e escrever fluentemente facilita a aprendizagem em vários outros assuntos.

Uma parceria público-privada americana, chamada de *Partnership for 21st Century Skills*, foi formada para criar um modelo de educação que prepare os indivíduos para o século XXI. Dentre os elementos chaves apresentados em seu primeiro relatório estão enfatizar aptidões de aprendizagem e usar ferramentas deste século para desenvolvê-las [Partnership for 21st Century Skills, 2003]. As aptidões de aprendizagem habilitam a aquisição de novos conhecimentos através da conexão de novas informações ao conhecimento pré-existente. A criação de jogos desenvolve aptidões de aprendizagem, conforme mostrado na Tabela 1.

Tabela 1. Como a criação de jogos desenvolve aptidões de aprendizagem.

Aptidão de Aprendizagem	**Criação de Jogos**
Aptidões de Informação e Comunicação	
Alfabetização em Informação e Mídia Analisar, acessar e criar informações em variadas formas e mídias. Entender o papel da mídia na sociedade.	Os projetistas trabalham com variados tipos de mídias, incluindo textos, imagens, animações e sons.

Comunicação Entender, gerenciar e criar efetivamente comunicações multimídias em variados contextos.	Os projetistas se expressam criativamente através da escolha e manipulação de variadas formas de mídia.
Aptidões de Pensamento e Resolução de Problemas	
Pensamento Crítico e Pensamento em Sistemas Realizar escolhas complexas entendendo a interconexão entre os sistemas.	Os projetistas devem programar o comportamento das entidades levando em consideração tempo, entrada, sensores, realimentações e outros conceitos fundamentais de sistemas.
Identificação, Formulação e Solução Habilidade de identificar, analisar e resolver problemas.	Os projetistas imaginam uma idéia de jogo e como dividir o problema em passos. Depois o jogo é implementado usando os recursos disponibilizados pela ferramenta.
Criatividade e Curiosidade Intelectual Desenvolver, implementar e comunicar novas idéias. Estar aberto a novas perspectivas.	Criar jogos é uma atividade, acima de tudo, criativa. Os projetistas aprendem a encontrar soluções inovadoras para problemas não esperados.
Aptidões Interpessoais e Autodidatas	
Interpessoal e Colaborativa Demonstrar trabalho em equipe e liderança. Adaptar-se a variados papéis e responsabilidades. Respeitar diversas perspectivas.	Criar jogos é uma atividade que favorece o trabalho cooperativo pelo seu caráter interdisciplinar. Algumas ferramentas também possibilitam o desenvolvimento cooperativo.
Autodidata Monitorar o próprio entendimento e necessidade de aprendizagem. Localizar os recursos apropriados e transferir o conhecimento de um domínio para outro.	Ter uma idéia e descobrir como programá-la na ferramenta requer persistência e prática. Quando os projetistas estão engajados com suas próprias idéias, eles estão motivados para enfrentar os desafios e as frustrações encontradas.
Responsabilidade e Adaptabilidade Exercitar a responsabilidade pessoal e a flexibilidade no contexto pessoal, do trabalho e da comunidade. Buscar altos padrões e metas para si próprio e para os outros.	Algumas ferramentas possibilitam e encorajam que os projetos criados sejam compartilhados e avaliados em uma comunidade. Dessa forma, os projetistas criam seus jogos imaginando como as outras pessoas reagirão e podem modificar o jogo baseado no retorno fornecido por elas.
Responsabilidade Social Agir responsavelmente de acordo com os interesses da ampla comunidade em mente. Demonstrar comportamento ético no contexto pessoal, do trabalho	Os jogos criados pelos projetistas podem estar relacionados com assuntos de interesse da comunidade como, por exemplo, preservação da natureza, reciclagem e prevenção de doenças.

e da comunidade.	

Na educação no século XXI não basta educar os estudantes com os conteúdos curriculares básicos. O desenvolvimento das aptidões de aprendizagem é fundamental para dar flexibilidade e segurança numa sociedade caracterizada pelas constantes mudanças. A criação de jogos é um processo lúdico que desenvolve essas aptidões.

Capítulo 3

Ferramentas de Desenvolvimento de Jogos

Há uma grande variedade de ferramentas de desenvolvimento de jogos, que foram projetas para atender a públicos diferentes. Algumas ferramentas são voltadas ao público iniciante, geralmente indivíduos que possuem pouco ou nenhum conhecimento de programação e nem conhecem as técnicas de desenvolvimento de jogos. Outras ferramentas são voltadas ao público especialista, geralmente profissionais da indústria de jogos ou programadores interessados em criar jogos por hobby.

Os iniciantes buscam por ferramentas que sejam simples de usar e que trabalhem em níveis de abstração mais altos, ou seja, mais próximas aos conceitos encontrados nos jogos. Por outro lado, os especialistas buscam ferramentas que provêem alto desempenho, recursos gráficos avançados, produtividade, flexibilidade e extensibilidade para criar jogos mais elaborados. Para isso, essas ferramentas trabalham em níveis de abstração mais baixos, ou seja, mais próximas ao hardware.

Este capítulo apresenta algumas tecnologias que são geralmente encontradas nessas ferramentas. A Figura 2 ilustra o relacionamento entre as tecnologias de desenvolvimento de jogos e o público-alvo.

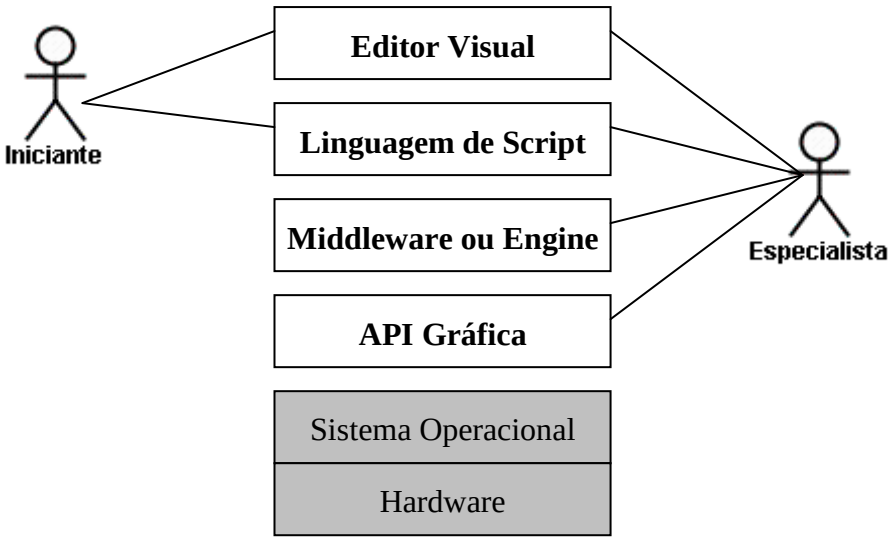

Figura 2. Relacionamento entre tecnologias de desenvolvimento de jogos e o público-alvo.

A seguir, cada tecnologia é apresentada em ordem crescente de abstração.

- **API Gráfica:** provê acesso ao dispositivo de vídeo da máquina de forma independente do fabricante e com alto desempenho. Essa tecnologia oferece baixa abstração, tornando o desenvolvimento do jogo complexo e demorado.
- ***Middleware*:** provê soluções reusáveis para diversos aspectos de um jogo como, por exemplo, gráfico, som, rede e física. Para acessar a API (*Application Programming Interface*) fornecida pelo *middleware*, é preciso utilizar uma linguagem de programação convencional, geralmente o C++.
- **Linguagem de Script:** é uma linguagem de programação projetada para ser mais fácil de usar e aprender. Alguns *middlewares* disponibilizam o acesso a sua API através de uma linguagem de script.
- **Editor Visual:** algumas ferramentas fornecem editores visuais para a criação completa de um jogo, facilitando a utilização por usuários que nunca programaram. Outras disponibilizam editores somente para algumas partes, como posicionar as entidades no cenário do jogo, enquanto a programação deve ser feita através da linguagem de script ou do *middleware*.

Os iniciantes trabalham em níveis de abstração mais altos com editores visuais e linguagens de script. Os especialistas desenvolvem a maior parte do jogo utilizando *middlewares* devido à maior flexibilidade provida pela linguagem de programação. Entretanto, eles também buscam tecnologias mais abstratas para aumentar a produtividade em determinadas tarefas, como a configuração do cenário do jogo. Algumas ferramentas disponibilizam as diferentes tecnologias de forma integrada.

Quanto menor o nível de abstração da tecnologia empregada pela ferramenta, maior é a curva de aprendizagem para dominá-la. Por exemplo, ferramentas que usam editores visuais são mais fáceis de aprender do que aquelas acessadas através de uma linguagem de programação, linguagens de script são mais fáceis de aprender do que linguagens convencionais e os *middlewares* abstraem diversos controles que seriam necessários nas APIs gráficas.

O restante do capítulo está organizado da seguinte maneira: a Seção 3.1 apresenta as API Gráficas e os *Middlewares*, a Seção 3.2 apresenta as Linguagens de Script, a Seção 3.3 apresenta as *Game Engines* e a Seção 3.4 apresenta as Ferramentas Visuais de Criação de Jogos.

3.1. API Gráfica e *Middleware*

Todos os jogos que usam recursos gráficos avançados necessitam acessar, direta ou indiretamente, uma API gráfica como o Microsoft DirectX[5] ou o OpenGL[6]. Essas bibliotecas provêem abstrações para acessar o dispositivo de vídeo da máquina de forma independente do fabricante e com alto desempenho.

Apesar das APIs gráficas abstraírem grande parte da complexidade relacionada com o hardware da máquina, o processo de desenvolvimento de jogos usando apenas essa tecnologia é complexo e demorado. Isso se deve ao fato delas não abordarem aspectos específicos dos jogos, já que foram projetadas para atender a outros tipos de aplicação como ferramentas CAD (*Computer-Aided Design*).

Essas dificuldades levaram ao surgimento de tecnologias reusáveis mais específicas ao desenvolvimento de jogos. Neste trabalho, todos os artefatos de software que fornecem uma API com esse propósito são denominados de *middlewares*. Outros nomes comumente encontrados são *engine*, *framework* e SDK (*Software Development Kit*). Os *middlewares* são classificados de acordo com o tipo de problema que se propõem resolver e os mais importantes estão listados abaixo:

- ***Middleware* de Renderização:** fornece diversos recursos gráficos usados pelos jogos e suportam gráficos em duas dimensões (2D) ou três dimensões (3D). Outros nomes comumente encontrados são: engine gráfica, engine de renderização ou 3D engine.
- ***Middleware* de Física:** muitos jogos usam simulações físicas para determinar o comportamento das entidades (por exemplo, num jogo de corrida, os carros devem se locomover de acordo com as forças aplicadas e colidir com outros carros ou obstáculos). Um middleware de física realiza simulações através de um sistema de forças e colisão.
- ***Middleware* de Som:** é responsável pelo gerenciamento dos arquivos de som usados no jogo. Além de abrir e tocar esses arquivos, algumas ferramentas também possibilitam um controle do som conforme o posicionamento das entidades no mundo do jogo.

[5] http://msdn.microsoft.com/en-us/directx/default.aspx
[6] http://www.opengl.org

- ***Game Middleware***: integra vários tipos de *middlewares* em um mesmo pacote, acrescentando funções mais específicas dos jogos como, por exemplo, sistema de menus, sistema de configuração e sistema de entidades.

Geralmente, os *middleware* são desenvolvidos e acessados com a linguagem C++, o que aumenta a complexidade, principalmente para programadores iniciantes. Outros são mais flexíveis e aceitam diversas linguagens de programação. Trabalhar nesse nível de abstração exige aprender a API fornecida, conhecer a arquitetura interna do *middleware* e conhecer outras áreas além da programação como, por exemplo, Computação Gráfica e Física. Portanto, essa tecnologia é mais voltada para os usuários especialistas.

Um possível uso dos *middlewares* como ferramenta educacional é empregá-los em cursos superiores de computação para motivar a aprendizagem de matérias como Linguagem de Programação e Computação Gráfica [Parberry, 2007].

3.2. Linguagem de Script

As linguagens de script são linguagens de programação usadas para manipular, customizar e automatizar sistemas existentes [Lutz & Ascher, 2003]. A plataforma de execução desses sistemas contém um interpretador, também conhecido como máquina virtual, que carrega, interpreta e executa scripts[7]. Um exemplo de linguagem de script muito usada na Internet para manipular objetos em páginas web é o JavaScript [ECMA, 1997].

As linguagens de script são projetadas para serem fáceis de usar e aprender, principalmente por programadores não profissionais. Para isso abstraem diversos conceitos como tipos de dados e gerência de memória.

Um exemplo desse tipo de linguagem amplamente aplicado com propósitos educacionais é o Logo, idealizada por Papert [1980]. Geralmente, as implementações do Logo contêm um interpretador dos comandos que manipulam um objeto gráfico (uma tartaruga) para desenhar figuras geométricas na tela.

No desenvolvimento de jogos, as linguagens de script são usadas para maximizar a produtividade durante o desenvolvimento e para dar flexibilidade para os próprios jogadores criarem variações do jogo [Buckland, 2005]. Ela reduz consideravelmente a complexidade, porém os usuários que nunca programaram poderão sentir dificuldades no uso dessa

[7] Script é nome dado aos programas escritos usando linguagens de script.

tecnologia, pois deverão conhecer conceitos básicos de programação, a sintaxe da linguagem e como acessar as funcionalidades fornecidas.

3.3. Game Engine

Game Engines integram várias tecnologias, como um *game middleware*, um conjunto de editores visuais e, geralmente, uma linguagem de script, para facilitar o desenvolvimento de jogos. Uma característica geralmente encontrada é a especificidade para um determinado gênero de jogo, por exemplo, jogos de tiro em primeira pessoa (*First Person Shooters* - FPS). Essa característica advém do fato de muitas produtoras de jogos comercializarem a *game engine* desenvolvida para a produção de um título específico.

Atualmente, essas ferramentas são vitais para a indústria de jogos. O custo e o tempo necessário para criar um jogo complexo sem usá-las se tornaram proibitivos [Rollings & Morris, 2004]. O custo de algumas *game engines* comerciais pode atingir milhares de dólares.

Apesar de focarem o mercado profissional de desenvolvimento de jogos, algumas iniciativas estão sendo feitas para que as *game engines* também possam ser usadas por não especialistas, tanto com propósito educacional quanto por hobby. Alguns jogos liberam e até incentivam que seus jogadores usem gratuitamente sua *game engine* para desenvolver variações não comerciais do jogo original, conhecidos como *Mods* [Prensky, 2006], alterando a aparência através dos editores visuais ou o comportamento através da linguagem de script.

A maior vantagem de usar *game engines* é sua alta qualidade gráfica. Gráficos mais elaborados propiciam aos usuários uma experiência de criação mais rica e motivadora. A principal desvantagem é a alta curva de aprendizagem necessária para usá-las, muitas vezes exigindo conhecimentos sobre Computação Gráfica, aprendizagem de uma nova linguagem de script e entendimento sobre sua arquitetura interna. Outra desvantagem peculiarmente grave para serem adotadas nas escolas brasileiras é a alta exigência do hardware da máquina.

Uma característica que também traz vantagens e desvantagens é a especificidade para determinado gênero de jogos. A vantagem é a maior facilidade para criar jogos do gênero suportado pela *engine*, pois há mais recursos específicos disponíveis como, por exemplo, um conjunto de scripts de comportamento para inimigos de um jogo FPS. A desvantagem é a maior limitação na hora de criar os jogos, pois eles terão a mesma mecânica que o jogo original. Apesar disso, essas ferramentas ainda são benéficas quando exploradas com temáticas sobre a matéria que os estudantes estão aprendendo nas aulas tradicionais. Por

exemplo, os estudantes podem criar um jogo que seja ambientado em um determinado período histórico, levando-os a uma aprendizagem através de pesquisas sobre os costumes, os eventos e os principais personagens da época.

Alguns exemplos de uso de *game engines* com propósito educacional são encontrados na literatura. Robertson e Good (2005) realizaram um workshop onde um grupo de adolescentes usou a ferramenta de edição do jogo *Neverwinter Nights*[8] para criar histórias interativas. O estudo mostrou que os estudantes puderam expressar melhor sua criatividade no ambiente de criação do jogo do que em linguagens escritas. Eles ficaram motivados e orgulhosos de suas criações. Entretanto, a ferramenta de criação gerou algumas dificuldades para realizar determinadas tarefas e se mostrou muito específica para a ambientação do jogo (medieval).

3.4. Ferramenta Visual de Criação de Jogos

As Ferramentas Visuais de Criação de Jogos possibilitam a criação de jogos usando somente interfaces visuais, ou seja, sem a necessidade de usar uma linguagem de programação textual. Elas fornecem editores para criar e posicionar as entidades dentro do cenário e para determinar a lógica do jogo, ou seja, como as entidades reagirão aos diversos eventos que podem ocorrer, tais como: duas entidades colidiram, a entidade foi atualizada, o jogador acionou o teclado ou o mouse, dentre outros.

A programação visual e a configuração de propriedades são duas abordagens comumente encontradas nessas ferramentas para determinar a lógica do jogo. A programação visual fornece editores visuais que reproduzem os conceitos encontrados nas linguagens de programação convencionais, tais como comandos, eventos, blocos de repetição e blocos condicionais. Outra abordagem é alterar o comportamento do jogo através da configuração de propriedades fornecidas pelo editor.

Comparado com a programação visual, a configuração de propriedades torna o processo de criação mais simples para os usuários que nunca programaram, pois diversos algoritmos que seriam necessários são abstraídos. Por exemplo, para fazer uma entidade se movimentar pelo teclado, a abordagem baseada em propriedades fornece uma interface para o usuário configurar as teclas, a velocidade de deslocamento, a animação que será executada, dentre

[8] http://nwn.bioware.com

outros. Na abordagem baseada na programação visual, é necessário programar todas as ações necessárias, possivelmente para as quatro direções (norte, sul, leste e oeste). Por outro lado, a programação visual fornece mais flexibilidade para definir os comportamentos e melhor desenvolve a habilidade de programação e o raciocínio lógico nos usuários.

Comparado com outras tecnologias, as Ferramentas Visuais de Criação de Jogos são mais acessíveis, pois possibilitam que usuários com pouco ou nenhum conhecimento de programação criem jogos. A proposta deste trabalho é criar uma Ferramenta Visual de Criação de Jogos e o restante da seção apresenta alguns exemplos de ferramentas nas duas abordagens.

3.4.1. Programação Visual

O *Klik & Play* [Clickteam, 1994] foi uma das primeiras ferramentas voltadas para iniciantes. Ele fez muito sucesso no Brasil e é gratuito para fins educacionais. O *Klik & Play* evoluiu e mudou de nome algumas vezes, sendo que a última versão se chama *The Games Factory 2*, específica para criar jogos ou *Multimedia Fusion 2*, para jogos e aplicativos [Clickteam, 2006]. Essa linha de ferramentas usa uma forma de programação tabular que pode ser usada para introduzir conceitos algorítmicos. A Figura 3 mostra o editor de eventos do *Klik & Play*, onde o projetista consegue visualizar todas as interações entre as entidades e entre o jogador e o jogo.

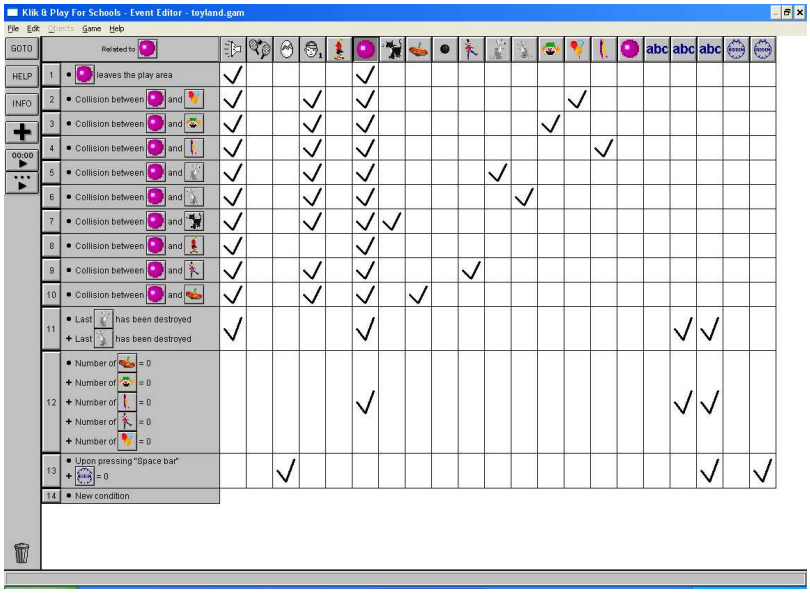

Figura 3. Editor de eventos do *Klik & Play* [Clickteam, 1994].

Outras ferramentas usam uma linguagem de programação visual que contém comandos representando alguma ação (por exemplo, movimentar, destruir e criar entidades) e estruturas

de controle: blocos condicionais e iteradores. Essas linguagens são orientadas a eventos, de forma que o projetista define um conjunto de comandos em resposta aos eventos que podem ocorrer durante o jogo.

O *Game Maker* [YoyoGames, 2007] é um exemplo desse tipo de ferramenta. Ele possui uma versão gratuita, com alguns recursos bloqueados, e possui uma linguagem de script para dar mais flexibilidade para os usuários avançados. A Figura 4 mostra parte da janela de edição de comandos do *Game Maker 7.0*. A coluna da esquerda mostra todos os eventos registrados na entidade, a coluna do meio contém os comandos configurados para o evento selecionado e a coluna da direita mostra o conjunto de comandos disponíveis.

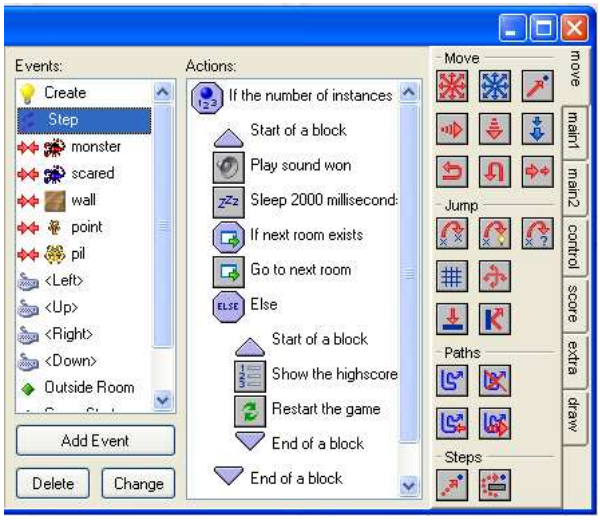

Figura 4. Janela de edição de comandos do *Game Maker* 7.0 [YoyoGames, 2007].

Outra exemplo é o *Scratch* [MIT, 2007] (Figura 5), desenvolvido pelo *Lifelong Kindergarten Group* do *MIT Media Lab* e possui o código livre. A sua linguagem visual é composta por estruturas de controles com o formato de blocos que se encaixam (Figura 6), o que facilita a aprendizagem e o uso da mesma. Um diferencial do *Scratch* são os recursos voltados ao compartilhamento e avaliação dos jogos criados pelos usuários através de uma comunidade online [Monroy-Hernandez, 2007].

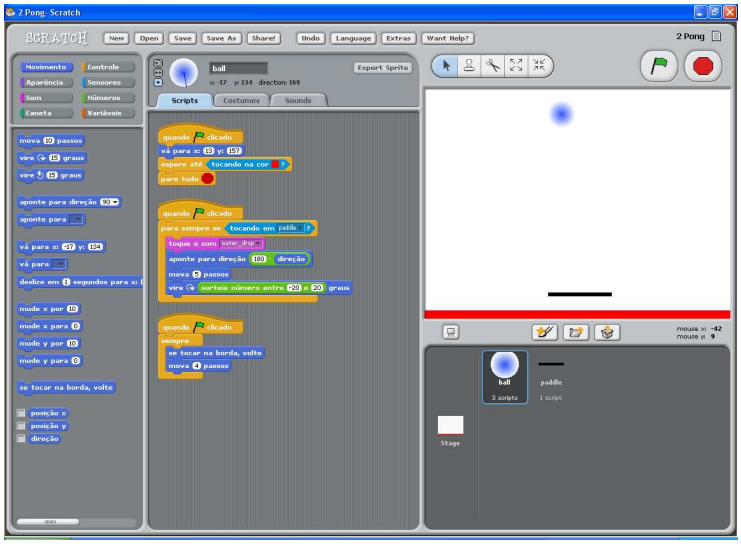

Figura 5. Imagem da ferramenta *Scratch* [MIT, 2007].

Figura 6. Linguagem de Programação visual usada no *Scratch* [MIT, 2007].

3.4.2. Configuração de Propriedades

Algumas ferramentas fornecem propriedades para determinar alguns comportamentos das entidades, enquanto outras são totalmente baseadas nessa abordagem. O *Klik & Play* fornece uma interface para definir qual o tipo de movimentação da entidade selecionada, conforme ilustrado na Figura 7.

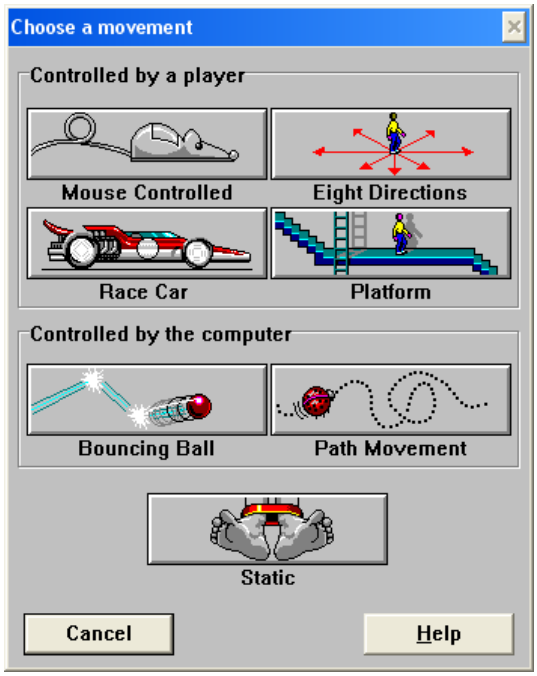

Figura 7. Interface de configuração da movimentação no Kilk & Play [Clickteam, 1994].

O *Torque Game Builder* (TGB) [GarageGames, 2007] é uma ferramenta comercial que fornece componentes para serem anexados e configurados através de um conjunto de propriedades. Cada componente representa uma funcionalidade que é comumente encontrada nos jogos, por exemplo, movimentar a entidade, seguir uma entidade, sofre dano, causar dano, dentre outros. O uso de componentes possibilita mais flexibilidade para compor o comportamento e maior extensibilidade para criar novos componentes. O TGB é mais voltado ao mercado profissional de desenvolvimento de jogos.

Capítulo 4

Análise e Projeto de Jogos

Algumas ferramentas de criação de jogos são específicas para um gênero de jogos, como jogos de plataforma ou jogos de tiro em primeira pessoa, e outras são mais genéricas, possibilitando a criação de jogos que, aparentemente, possuem pouca coisa em comum. A ferramenta Affinity, proposta neste trabalho e apresentada no Capítulo 6, segue a segunda abordagem. Este capítulo apresenta uma análise e um projeto arquitetural aplicável em diferentes jogos e que servirá de base para o desenvolvimento do Affinity.

O restante do capítulo está organizado da seguinte maneira: a Seção 4.1 apresenta uma análise para identificar os conceitos comuns a vários jogos e suas relações e a Seção 4.2 apresenta um projeto arquitetural capaz de prover flexibilidade, extensibilidade e reusabilidade.

4.1. Análise Conceitual

O *Game Ontology Project* (GOP) [Zagal et. al., 2005], desenvolvido pelo *Experimental Game Lab* no *Georgia Institute of Technology*, é uma ontologia de jogos que identifica e organiza hierarquicamente os elementos estruturais dos jogos. O objetivo do GOP é criar um *framework* para descrever, analisar e estudar jogos. O nível mais alto da ontologia consiste em cinco elementos: interface, regras, objetivos, entidades e manipulação de entidade. A interface provê o meio pelo qual o jogador experimenta o jogo e executa ações dentro dele. As regras definem e restringem o que pode e não pode ser feito no jogo. Os objetivos são as condições que definem o sucesso no jogo. As entidades são os objetos que existem dentro do mundo do jogo. Finalmente, a manipulação de entidade representa as ações ou verbos que podem ser executadas pelas entidades do jogo. O restante desta seção apresenta com mais detalhes os conceitos subjacentes a cada categoria definida pelo GOP.

4.1.1. Interface

A interface provê o meio pelo qual o jogador experimenta e executa ações dentro do jogo. Para se comunicar com o jogador é usada alguma forma de apresentação, visual ou sonora. A câmera é um conceito que representa a perspectiva de como o jogador percebe e interage com o mundo (ou cenário) do jogo. A câmera pode ter uma perspectiva em primeira ou terceira pessoa. Na perspectiva em primeira pessoa, o jogador percebe o mundo com a mesma visão do seu *avatar* (personagem que ele controla). Na perspectiva em terceira pessoa, o jogador percebe o mundo como um agente de fora.

Outros elementos de interface são os dispositivos de entrada como mouse, teclado ou *joystick*, usados para o jogador interagir com o jogo. O jogo interpreta os sinais enviados pelos dispositivos e executa alguma ação como, por exemplo, a manipulação de alguma entidade.

4.1.2. Regras

As regras definem e restringem o que pode e não pode ser feito no jogo. Zagal et. al. [2005] definem dois tipos de regras: regras da jogabilidade (*gameplay*) e regras do mundo (*gameworld*). As regras do mundo regulam o mundo virtual onde o jogo se passa, por exemplo, a gravidade imposta a todas as entidades. As regras da jogabilidade regulam e restringem em cima do mundo do jogo, por exemplo, ela define o número de vidas do jogador.

4.1.3. Objetivos

Os objetivos são as condições que o jogador deve alcançar para ter sucesso no jogo. Há objetivos com diferentes níveis de granularidade. Para todos os jogos, o nível mais alto é vencer o jogo ou jogar da melhor forma possível. Entretanto, para atingir esse objetivo, o jogador deve cumprir objetivos mais específicos, como achar uma chave ou derrotar o monstro [Zagal et. al., 2005].

4.1.4. Entidades

Entidades são os objetos que existem dentro do cenário do jogo. Para exemplificar o conceito de entidade, a Figura 8 identifica todas as entidades presentes no jogo Pong: a bola,

as paredes (superior e inferior), as raquetes dos jogadores (raquete 1 e raquete 2) e a área que representa os alvos onde cada jogador deve jogar a bola (alvo 1 e alvo 2).

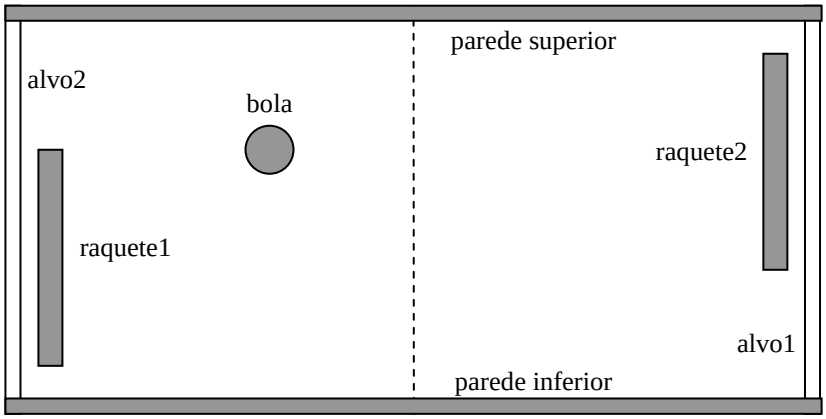

Figura 8. Cenário do jogo Pong.

Uma entidade não necessariamente precisa ser visível, como é o caso dos alvos dos jogadores, localizados atrás de cada raquete. Quando a bola colidir com essa entidade, o jogador adversário ganha um ponto e a bola recomeça em sua posição original.

Um sinônimo para entidade comumente encontrado é objeto do jogo (*game object*). Entretanto, adotar esse termo pode trazer confusão, pois o termo entidade está referenciando um conceito e não um objeto de software. Nem sempre uma entidade tem uma correspondência de um para um com o objeto de software [Rollings & Morris, 2004, p.482-483].

As entidades possuem alguns comportamentos comuns e outros distintos entre si. No exemplo anterior, as duas paredes possuem exatamente o mesmo comportamento. Elas permanecem estáticas durante todo o tempo e colidem com qualquer outra entidade. Elas diferenciam-se apenas pela posição dentro do cenário. Outro exemplo de comportamento comum é a bola e as raquetes colidirem com as paredes e ambas se deslocarem dentro do cenário de acordo com a velocidade aplicada.

Como exemplo de comportamento distinto, a bola é rebatida quando uma colisão com as paredes acontecer, enquanto a raquete para. Além disso, a bola se movimenta em qualquer direção e as raquetes só se movimentam na direção vertical. Também é possível encontrar diferenças no comportamento entre as duas raquetes (do jogador 1 e do jogador 2), pois cada uma responde a comandos de entrada diferentes no teclado.

Ao analisar as similaridades e as diferenças entres os comportamentos das entidades do jogo, é possível classificá-las de acordo com tipos de entidades que seguem um determinado

comportamento. Um tipo de entidade pode ter várias instâncias (entidades) dentro do cenário. Cada instância possui o mesmo comportamento, mas estão em estados diferentes. Por exemplo, duas entidades do mesmo tipo podem estar em posições diferentes dentro do cenário ou em faixas diferentes de uma animação. Esses conceitos são similares à classe e instância de classe da Análise Orientada a Objeto. A Figura 9 ilustra o relacionamento entre os conceitos apresentados.

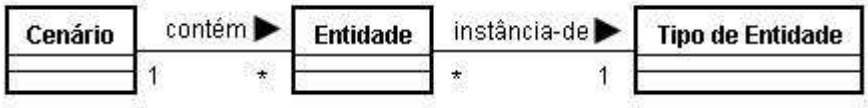

Figura 9. Relacionamento entre os conceitos de Cenário, Entidade e Tipo de Entidade.

No exemplo do jogo Pong, é possível classificar a parede superior e a parede inferior como instâncias do tipo de entidade Parede e a bola como instância do tipo de entidade Bola. No caso das raquetes e dos alvos, depende do ponto de vista do analista. Um analista pode considerar que as entidades raquete 1 e raquete 2 sejam instâncias de tipos diferentes por responderem a comandos de entrada diferentes. Outro analista pode considerar que a tecla responsável por movimentar a entidade seja uma propriedade do tipo de entidade e, portanto, as duas entidades são instâncias do mesmo tipo Raquete e diferenciam-se apenas no estado.

Ao definir o conjunto de tipos de entidades de um jogo, o domínio desse jogo está sendo modelado. Se for considerada a análise do domínio de jogos em geral, seria necessário definir todos os tipos de entidades que são comumente encontrados nos jogos. Entretanto, cada jogo é único e pode tratar de conceitos completamente distintos. Uma solução é definir uma entidade implicitamente por manipulações de entidade [Zagal et. al., 2005], que serão detalhadas na próxima seção. Essa abordagem torna a análise muito mais abrangente, pois não são introduzidos conceitos específicos de nenhum jogo.

4.1.5. Manipulação de Entidade

Manipulações de entidade são as ações ou verbos que são executados pelas entidades do jogo. As entidades possuem um conjunto de atributos (velocidade, dano, dono etc.) que são alterados pelas ações. Por exemplo, a ação de mover altera o atributo posição da entidade. Alguns exemplos de ações são:

- Se movimentar.
- Reagir como um corpo rígido.

- Emitir partículas.
- Executar um som.
- Seguir um caminho.
- Executar uma animação.
- Ser anexadas à outra entidade.
- Se destruir.

O comportamento de cada entidade é definido pelas ações que são executadas em resposta aos eventos do jogo. Por exemplo, quando duas entidades colidem, acontece um evento de colisão. Em resposta a esse evento, as entidades executam ações, como reagir como corpos rígidos, executar um efeito sonoro e emitir partículas. As entidades também podem responder a eventos causados por algum dispositivo de entrada: teclado, mouse ou *joystick*. Dessa forma o jogador consegue interagir com o jogo. Além dos eventos de entrada e de colisão, outros exemplos de eventos que acontecem dentro do jogo são:

- A entidade foi criada.
- A entidade foi destruída.
- A entidade está sendo atualizada.
- A entidade está sendo desenhada na tela.
- O jogo começou.
- O jogo terminou.
- Determinado tempo foi alcançado na linha do tempo do jogo.

Através de um conjunto de comportamentos comuns a vários jogos, é possível definir implicitamente uma entidade. Por exemplo, no jogo Pong, a entidade bola é definida por um comportamento de movimentar de acordo com a velocidade e outro de reagir como um corpo rígido na colisão com outras entidades. Esses dois comportamentos estão presentes na maioria dos jogos.

4.2. Projeto Arquitetural

A arquitetura de um jogo pode ser dividida em duas partes: a horizontal e a vertical. A arquitetura horizontal é composta por subsistemas reusáveis que fornecem uma infra-estrutura

de execução para diversos jogos. A arquitetura vertical é composta por subsistemas que implementam as funcionalidades específicas de um jogo [Rollings & Morris, 2004, p.612].

Geralmente os subsistemas da arquitetura horizontal provêem abstrações que facilitam o acesso aos vários dispositivos de hardware usados pelo jogo: vídeo, som, rede e dispositivos de entrada. Esses subsistemas são desenvolvidos pelos próprios desenvolvedores do jogo ou são incorporados à arquitetura na forma de *middlewares* provenientes de terceiros.

A arquitetura vertical é composta por subsistemas que implementam as funcionalidades específicas de um jogo e, portanto, não são reusáveis em projetos diferentes. Um exemplo é o subsistema de entidades que define o comportamento das entidades dentro do mundo do jogo, ou seja, define o jogo em si. No subsistema de entidades é onde estão definidos, por exemplo, as classes Personagem, Inimigo e Item.

Apesar dos subsistemas da arquitetura vertical tratarem de aspectos específicos, é possível identificar similaridades entre jogos e projetar esses subsistemas para serem reusados, ou seja, para pertencerem à arquitetura horizontal. Dessa forma, o desenvolvimento de um novo jogo passa a ser mais focado no conteúdo do que na tecnologia [Rollings & Morris, 2004, p.636]. Seguindo essa abordagem, esta seção apresenta o projeto arquitetural para o desenvolvimento de um sistema de entidades reusável.

Duas possíveis abordagens para o projeto de um sistema de entidades são: a hierarquia de entidades e a composição de comportamentos. Elas são baseadas nas duas técnicas mais comuns para a reutilização de funcionalidades em sistemas orientados a objetos: a herança de classe e a composição de objetos [Gamma et. al., 2005].

Ao analisar cada abordagem, será levado em consideração características de qualidade do projeto que visem um modelo capaz de ser reusado no desenvolvimento de diversos jogos, tais como: flexibilidade, extensibilidade e reusabilidade.

Entidades de diferentes tipos podem apresentar o mesmo comportamento. Por exemplo, no jogo Pong, as entidades dos tipos *Bola* e *Raquete* se movimentam pelo cenário. O modelo deve prover flexibilidade para configurar o mesmo comportamento em diferentes tipos. Outro fator importante é a capacidade de estender e especializar os comportamentos de acordo com as necessidades requeridas por um tipo de entidade específico.

A implementação de um comportamento deve ser reusada por todos os tipos de entidades que apresentam esse comportamento. No caso dos tipos *Bola* e *Raquete*, a

implementação deve ser feita apenas uma vez e reusada por todas as entidades que precisam se movimentar.

4.2.1. Hierarquia de Entidades

A hierarquia de entidades utiliza a técnica de reutilização por herança de classe para organizar todas as entidades do jogo em uma estrutura hierárquica.

A herança de classe possibilita definir a implementação de uma classe (subclasse) em termos de outra (superclasse). A subclasse contém todos os atributos e métodos da superclasse e pode acrescentar novos atributos, métodos ou redefinir métodos da superclasse.

A reutilização por meio de subclasses é frequentemente chamada de reutilização caixa branca, pois os interiores das classes ancestrais são visíveis para as subclasses [Gamma et. al., 2005].

A principal desvantagem no uso da herança é o forte acoplamento entre a superclasse e suas subclasses. Uma alteração na superclasse possivelmente implicará em alterar as subclasses. Outra desvantagem é a não possibilidade de mudar as implementações herdadas das classes ancestrais em tempo de execução, pois a herança é definida em tempo de compilação. Por causa desses problemas, a gangue dos quatro [Gamma et. al., 2005] apresenta um princípio de projeto orientado a objeto que diz para preferir a composição de objeto à herança de classe [Gamma et. al., 2005].

Por outro lado, a herança de classe apresenta a vantagem de ser mais simples de usar, pois é suportada pela própria linguagem de programação, e ser mais fácil de modificar a implementação que está sendo reutilizada, pois a subclasse pode redefinir alguns métodos da superclasse [Gamma et. al., 2005].

No domínio dos jogos, a herança de classes é usada na implementação de uma hierarquia de entidades. A classe *Entidade* é pai de todos os tipos de entidades do jogo e é responsável por todas as funcionalidades comuns. Cada tipo de entidade herda da classe *Entidade* ou de outra classe que agrupe suas funcionalidades comuns e especializa seus comportamentos específicos.

Uma possível hierarquia de classes para o jogo Pong é ilustrada na Figura 10. As classes *Parede*, *Bola* e *Alvo* herdam da classe *Entidade*. A classe *Raquete* herdou da classe *Parede*, pois ele tem o mesmo comportamento definido na *Parede* mais a habilidade de alterar sua posição [Rollings & Morris, 2004, p.492].

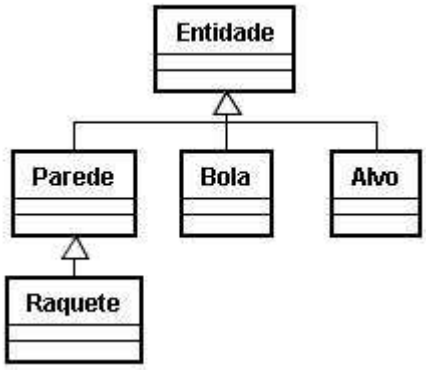

Figura 10. Hierarquia de tipos de entidades do jogo Pong.

Nessa abordagem, todos os comportamentos que são comuns aos diversos tipos de entidades acabam sendo implementados na classe *Entidade*. No final, a classe *Entidade* acaba implementando a maior parte das funcionalidades do jogo, o que a torna grande, difícil de manter e difícil de especializar. Além disso, algumas entidades definidas na hierarquia terão muito mais funcionalidades do que realmente precisam.

Mesmo ao expandir a hierarquia decompondo a classe *Entidade* em subclasses mais especializadas, outros problemas ainda permanecem. O primeiro é a dificuldade de compartilhar mais de uma funcionalidade na mesma entidade devido às limitações existentes no mecanismo de herança múltipla das linguagens de programação. Outro problema é a dificuldade de mudar o comportamento da entidade dinamicamente, pois sua hierarquia é definida em tempo de compilação.

4.2.2. Composição de Comportamentos

A composição de comportamentos utiliza a técnica de reutilização por composição de objetos para implementar cada comportamento apresentado pelas entidades do jogo em classes separadas.

A composição de objetos é uma alternativa à herança de classe [Gamma et. al., 2005]. Nessa técnica, uma nova funcionalidade é obtida compondo-se dinamicamente objetos que implementam as funcionalidades requeridas. A composição de objetos requer que os objetos que estão sendo compostos tenham interfaces bem definidas. Esse estilo de reutilização é chamado reutilização caixa preta, pois os detalhes internos dos objetos não são visíveis.

Os principais benefícios dessa abordagem são a maior flexibilidade e coesão. A flexibilidade aumenta, pois, como as interfaces são bem definidas, os objetos podem ser substituídos por outros em tempo de execução e haverá substancialmente menos dependências

de implementação. A coesão aumenta, pois cada classe fica responsável por uma única tarefa [Gamma et. al., 2005].

No domínio dos jogos, cada classe implementa um comportamento específico apresentado pelas entidades, possibilitando que qualquer entidade utilize essa funcionalidade. A classe *Entidade* ainda continua existindo, mas agora ela passa a ser um repositório de componentes, conforme ilustra a Figura 11.

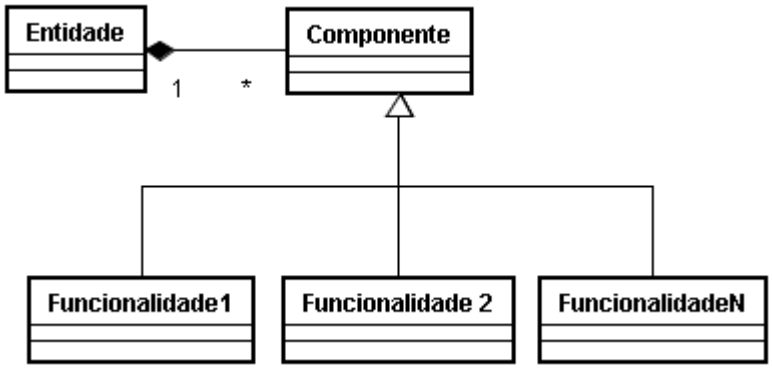

Figura 11. Composição de componentes no desenvolvimento de jogos.

Voltando ao exemplo do jogo Pong, a entidade bola agrega um componente de colisão configurado para rebater a entidade quando uma colisão ocorrer e um componente de física responsável em alterar a posição da entidade de acordo com sua velocidade atual. Já a entidade raquete agrega um componente de colisão configurado para parar quando uma colisão com a entidade parede ocorrer e um componente de controle de movimentação configurado para ler o teclado e alterar a velocidade das raquetes de acordo as teclas acionadas.

No Capítulo 5 será apresentada a abordagem de desenvolvimento baseado em componentes que possui o mesmo princípio de reutilização por decomposição de funcionalidades, mas emprega técnicas mais formalizadas.

Capítulo 5

Desenvolvimento Baseado em Componentes

Este capítulo apresenta os principais conceitos do Desenvolvimento Baseado em Componentes (DBC) e está organizado da seguinte maneira: a Seção 5.1 conceitua componente de software, a Seção 5.2 explica o que são *frameworks* de componentes, a Seção 5.3 descreve os benefícios e as dificuldades encontradas no uso dessa abordagem.

5.1. Componentes de Software

Para atingir a qualidade e cumprir o orçamento e os prazos previstos, não é mais possível começar a desenvolver software a partir do zero e estruturá-lo na forma de blocos monolíticos onde uma modificação propaga efeitos colaterais por todo o código, dificultando a manutenção e o reúso de suas partes [Werner & Braga, 2005, p.66]. O Desenvolvimento Baseado em Componentes (DBC) é uma abordagem que visa construir sistemas a partir de pequenas partes, chamadas de componentes, que já foram construídos anteriormente, aumentando a produtividade durante a fase de desenvolvimento e a qualidade final do produto [Almeida et. al., 2007].

A idéia de usar componentes para obter um maior nível de abstração e independência é empregada com destaque em outras engenharias. Por exemplo, na indústria automobilística, o projetista reutiliza a mesma peça em diferentes situações sem precisar entender o seu funcionamento interno, apenas sua interface [D'Souza & Wills, 1998, p.405].

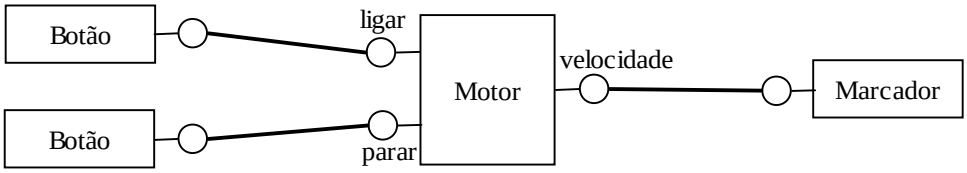

Figura 12. Exemplo de uso de componentes na indústria automobilística [D'Souza & Wills, 1998, p.405].

O exemplo da Figura 12, ilustra a componentização na ligação de um motor a dois botões (um para ligar e outro para desligar o motor) e a um marcador (que exibe a velocidade do motor). Os mesmos botões e o marcador são reusáveis em outras situações (por exemplo,

utilizando o marcador para mostrar a temperatura) e estes componentes são substituíveis por outros equivalentes, porém com implementações diferentes (por exemplo, os botões são substituíveis por uma chave liga-desliga e o marcador analógico por um digital). A interoperabilidade é propiciada pela compatibilidade entre os conectores. Além disso, o conjunto todo pode ser encarado como um componente de um sistema maior.

Na indústria de software, componentes são utilizados nas mais diversas áreas e com os mais diversos propósitos. Um exemplo bastante conhecido é o uso de plugins para estender as funcionalidades nativas das aplicações, como, por exemplo, nos navegadores Web, no Adobe Photoshop, no IDE Eclipse e no Apache Web Server.

Componentes visuais para construção de interface com o usuário é outro exemplo bastante difundido. Alguns ambientes de desenvolvimento disponibilizam um conjunto de componentes visuais que são utilizados para compor a interface com o usuário. O programador instancia e posiciona os componentes, configura suas propriedades e associa métodos a seus eventos, sem que para isso, precise ter acesso a seu código fonte.

5.1.1. Definição

Segundo [D'Souza & Wills, 1998, p.387], um componente de software é:

> Um pacote coerente de software que (a) pode ser desenvolvido e instalado independentemente como uma unidade, (b) tem interfaces explícitas e bem definidas para os serviços que provê, (c) tem interfaces explícitas e bem definidas para os serviços que espera de outros, e (d) pode ser utilizado para composição com outros componentes, sem alterações em sua implementação, podendo eventualmente ser customizado em algumas de suas propriedades.

Na revisão da literatura, foram encontradas diversas visões do que são componentes de software e, muitas vezes, o termo é usado para representar qualquer artefato do processo de desenvolvimento de software. Dentro do contexto do DBC, os componentes são artefatos de software que foram desenvolvidos seguindo-se padrões (modelo de componente) e que visam trazer benefícios propostos pela abordagem, tais como, manutenibilidade, reusabilidade, composição, extensibilidade, integração e escalabilidade.

Vale ressaltar que na literatura costumam-se encontrar alguns sinônimos para componentes, utilizados em contextos específicos, como plugin, add-on, módulo, serviço e widget.

Na UML, os componentes são representados de acordo com a Figura 13. Houve uma alteração na forma de representação da versão 1.x para a versão 2.0.

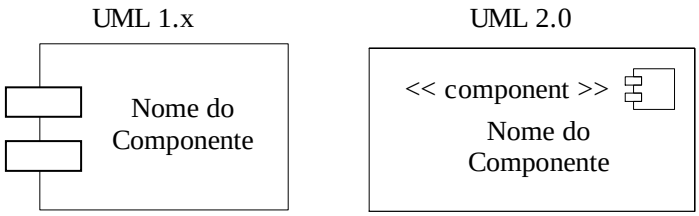

Figura 13. Representação de componente na UML 1.x e 2.0.

5.1.2. Interface

Componentes têm pontos de interconexão chamados de interfaces que representam seu contrato de utilização [Szyperski, 1997]. Respeitando os contratos, é possível alterar a implementação interna do componente ou substituí-lo por outro, com o mínimo de impacto nos seus clientes. O funcionamento da interface de um componente de software é semelhante à especificação dos pinos de um circuito integrado. Para usar o circuito, basta conhecer a especificação de sua interface externa e prever no circuito, o encaixe para ele. Não é necessário o conhecimento dos detalhes internos do seu funcionamento (abordagem caixa preta).

As interfaces são classificadas em dois tipos: interfaces fornecidas (*provided interfaces*) e interfaces requeridas (*required interfaces*). A primeira define os serviços oferecidos pelo componente. Para realizar uma determinada interface fornecida, um componente deve conter uma implementação de todas as operações definidas por aquela interface. Já a segunda define os serviços que o componente necessita de outros componentes. Um componente possui uma interface requerida se utiliza pelo menos uma operação definida naquela interface. Componentes se conectam por meio da interface requerida de um com a interface fornecida de outro [Barroca et al., 2005, p.4]. A Figura 14 ilustra o relacionamento entre dois componentes utilizando a linguagem UML. Nessa figura, o componente *Consumidor* provê a interface *IConsumidor* e o componente *Ordem* provê a interface *IOrdem* e requer uma interface *IConsumidor*.

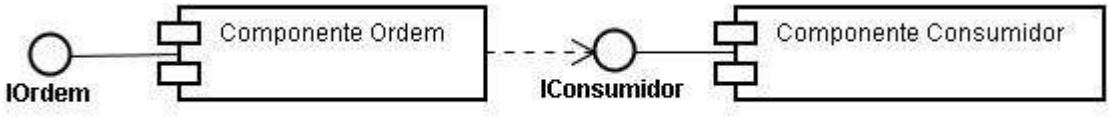

Figura 14. Relacionamento entre componentes em UML.

5.1.3. Modelo de Componente

Modelos de componente (*component model*) provêem padrões para a implementação e interoperabilidade de componentes. Eles podem definir a forma de implementar as interfaces, o padrão de nomeação, de composição, de versionamento e de empacotamento que devem ser seguidos. Um programador pode criar seu próprio modelo, eventualmente sendo uma especialização de um modelo disponível. Para compatibilizar componentes de um modelo para outro é necessário desenvolver adaptadores.

Alguns exemplos de modelos de componentes são: CORBA, OLE, COM+, Enterprise Java Beans, JavaBeans, Web Service e .NET. Alguns modelos, como CORBA e Web Services, possibilitam o reúso de componentes que não foram necessariamente desenvolvidos na mesma linguagem de programação.

5.2. *Framework* de Componentes

As definições de *framework* encontradas na literatura costumam variar. Duas definições que são comumente usadas são [Johnson, 1997]:

- Um projeto reutilizável de uma ou de todas as partes de um sistema, que é representado por um conjunto de classes e pela maneira que elas interagem.
- Um esqueleto de um sistema, que é instanciado e especializado para gerar uma família de aplicações.

Essas definições, na verdade, não são conflitantes: a primeira descreve a estrutura de um *framework*, enquanto a segunda descreve seu propósito.

Alguns *frameworks* são voltados à solução de problemas ligados à tecnologia, como interface com o usuário, persistência de objetos e suporte a *Model-View-Controller* (MVC). Outros *frameworks* são voltados para um determinado domínio, como por exemplo, aplicações bancárias, aviação, relacionamento com o cliente etc. Esses *frameworks* são embasados em teorias e modelos do domínio e definem uma arquitetura orientada para a área de aplicação específica [Fayad, 1999].

Além do reúso de código, *frameworks* também favorecem o reúso de projetos ao especificar a estrutura arquitetural a ser seguida pelas aplicações do domínio. Eles estão em um nível intermediário das técnicas de reúso, sendo mais abstratos e flexíveis que componentes de código e mais concretos e fáceis de serem reutilizados que projeto puro (mas

menos flexíveis e menos prováveis de serem aplicados) [Guizzardi, 2001]. Um definição que melhor expressa essa característica é apresentada Jia [2000]: um *framework* é um conjunto de classes cooperantes, que são partes de um sistema semicompleto e que representam estratégias reutilizáveis de projeto de software em um domínio particular de aplicação.

O conceito de *framework* está relacionado ao de componente – são conceitos complementares que contribuem para o reúso de software [Johnson, 1997]. Um *framework* de componentes segue e oferece suporte a um modelo de componentes, provendo uma infra-estrutura para apoiar sua execução. Ele oferece serviços de execução como criação de objetos, gerenciamento de ciclo de vida, persistência de objetos, licenciamento, acesso a dados, gerenciamento de transações, balanceamento de carga etc.

Frameworks de componentes também facilitam o desenvolvimento de novos componentes, oferecendo *templates* para implementá-los, especificações e a possibilidade de composição de componentes a partir de outros menores [Johnson, 1997].

5.3. Benefícios e Dificuldades da Componentização de Software

Esta seção apresenta os benefícios e as dificuldades encontradas no DBC.

5.3.1. Benefícios

Provavelmente um dos benefícios mais importantes propiciados pela componentização seja a maior modularização do software. A idéia de construir grandes sistemas a partir de pequenas partes (componentes) pode reduzir consideravelmente o tempo de desenvolvimento e melhorar a qualidade final do produto [Almeida et. al., 2007]. Grandes blocos monolíticos dificultam a manutenção, pois uma pequena alteração tem impacto em várias partes do código, e dificultam o reúso, pois há tanta interdependência entre as partes que compõem o sistema que dificilmente uma parte será reusada sem as outras.

O DBC aumenta a produtividade, pois a decomposição do sistema em componentes independentes possibilita que eles sejam subcontratados ou alocados entre os vários membros da equipe, o que favorece o desenvolvimento paralelo e em grupo.

A componentização favorece a manutenibilidade do sistema. Os componentes são substituíveis para atualização ou correção, muitas vezes sem precisar alterar ou recompilar a aplicação como um todo [D'Souza & Wills, 1998]. Além disso, eles são adicionados,

removidos ou substituídos por versões mais robustas ou mais apropriadas ao hardware, ao sistema operacional ou aos produtos legados com os quais o sistema tenha que operar. A modularidade obtida com a componentização facilita a localização do código a ser alterado e o encapsulamento da alteração. Algumas mudanças não acarretam em alterações no código, sendo resolvidas por re-composição da aplicação ou re-configuração dos componentes [D'Souza & Wills, 1998, p.397].

O reúso também é freqüentemente citado como um benefício da componentização. O reúso favorece a redução dos esforços de desenvolvimento e a qualidade do produto final, por colocar em uso código já utilizado e testado em outras situações [Krueger, 1992]. Além do reúso dos componentes em si, o *framework* de componentes provê diversos serviços básicos que são reusados como persistência, versionamento, comunicação etc. A componentização também promove o reúso nas diversas atividades do desenvolvimento: análise, projeto, implementação e testes.

Com a componentização, a aplicação é adaptável para diversas necessidades, selecionando e configurando os componentes mais adequados. Pode-se reimplementar um determinado componente para atender a uma necessidade específica ou adicionar novos componentes ou interfaces para estender os serviços providos, tornando o software desenvolvido mais adaptável e extensível [D'Souza & Wills, 1998, p.397]. Em alguns sistemas os próprios usuários finais recompõem e re-configuram os componentes.

Uma outra vantagem da componentização é o encapsulamento de conhecimento e uma programação de alto nível. Um desenvolvedor não precisa conhecer os detalhes de implementação dos componentes para utilizá-los para compor as aplicações. Quem integra componentes se especializa nesta atividade abstraindo os detalhes de implementação, tendo uma visão mais abrangente e mais próxima do domínio de aplicação.

5.3.2. Dificuldades

Com relação às dificuldades, o DBC demanda um grande esforço inicial para o desenvolvimento do *framework* e de uma biblioteca robusta de componentes reusáveis. Esse desenvolvimento requer especialistas tanto no domínio do problema quanto em Engenharia de Software [Chen, 2004, p.8]. Projetar e codificar um software para futuro reúso aumenta a necessidade de flexibilidade, documentação, estabilidade e abrangência [Moore & Bailin, 1991].

O custo do estudo e entendimento de como usar os componentes também é outra dificuldade. Os desenvolvedores devem aprender as APIs, os serviços e os parâmetros de configuração providos antes de reusá-los no desenvolvimento de novas aplicações [Chen, 2004, p.8].

Os custos e o esforço de desenvolvimento devem ser ponderados para avaliar a necessidade de desenvolver o *framework* e os componentes para um sistema novo. A menos que o investimento inicial seja amortizado por vários projetos ou que o ganho de produtividade e de qualidade seja expressivo, o DBC não é adequado.

Capítulo 6

Affinity: Uma Ferramenta para a Criação Cooperativa de Jogos

Criar jogos é uma tarefa que envolve diferentes aptidões. Nas empresas de desenvolvimento de jogos, essas aptidões são executadas, de um modo geral, por três profissionais: artista, desenvolvedor e projetista de jogos (*game designer*). O artista é responsável por criar os *assets*, que são arquivos relacionados com a arte do jogo, como arquivos de texturas, sons e modelos 3D. O desenvolvedor é responsável por programar o jogo e o projetista é responsável por definir as regras, os objetivos, os tipos de entidades e as entidades do jogo.

No processo de criação, o projetista geralmente define o jogo que deseja criar em um documento de especificação que serve de entrada para o trabalho dos desenvolvedores e artistas. A idéia central da ferramenta de criação de jogos Affinity é possibilitar que os próprios projetistas configurem seus jogos através de um editor visual, sem a necessidade de conhecer programação. O Affinity também possui uma comunidade onde os projetistas cooperam na criação dos jogos, compartilhando suas criações ou trabalhando em conjunto num mesmo projeto.

No Affinity, o projetista consegue trabalhar de forma independente dos desenvolvedores e artistas. Entretanto, os dois últimos também cooperam disponibilizando suas criações na comunidade. Para isso um jogo é decomposto por três elementos: componentes, *assets* e dados.

Os componentes implementam comportamentos do jogo e das entidades. Eles são disponibilizados pelo Affinity através do *framework* de componentes Affinity Game Framework (AGF) ou são implementados pelos desenvolvedores.

Os *assets* são criados pelos artistas utilizando ferramentas especializadas. O Affinity não provê nenhuma ferramenta de criação desses arquivos, apenas fornece mecanismos para armazenamento e compatibilidade com os principais formatos disponíveis.

Por fim, os dados especificam quais componentes são usados no jogo e suas configurações, os assets necessários, as propriedades do cenário, os tipos de entidades, o

posicionamento das entidades, dentre outros. Os projetistas configuram esses dados através da ferramenta visual Affinity Game Editor (AGE).

A Figura 15 ilustra o relacionamento entre os três elementos de um jogo e os atores do Affinity. É importante ressaltar que um mesmo usuário pode assumir mais de um papel no processo de criação.

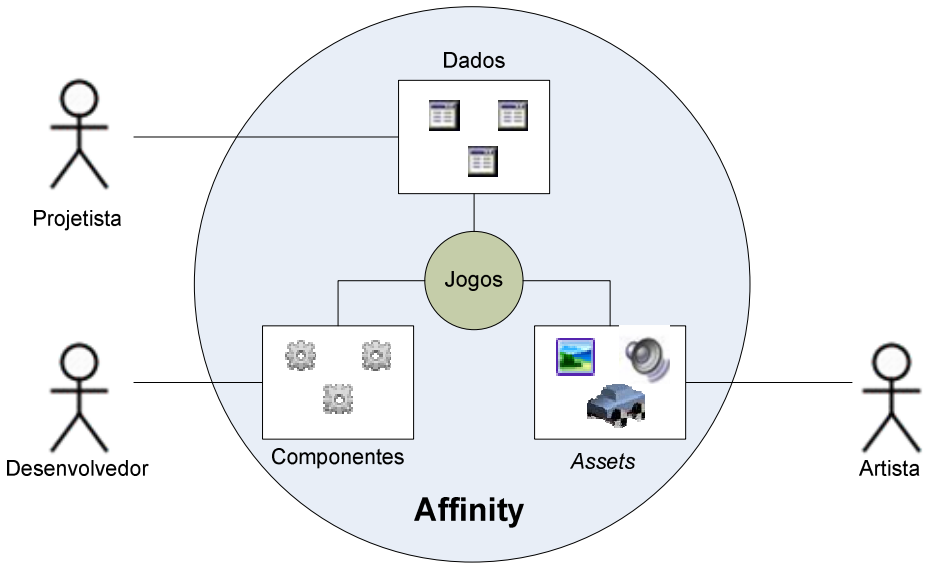

Figura 15. Relacionamento entre os elementos dos jogos no Affinity e os atores.

O restante do capítulo apresenta com mais detalhes o Affinity e está organizado da seguinte maneira: a Seção 6.1 apresenta os objetivos, a Seção 6.2 apresenta o modelo conceitual, a Seção 6.3 apresenta a arquitetura e a Seção 6.4 descreve a tecnologia usada no desenvolvimento.

6.1. Objetivos

A principal motivação para o desenvolvimento da ferramenta Affinity é fornecer uma ferramenta educacional que desenvolva nos usuários aptidões de aprendizagem importantes para suas vidas e carreiras. Ela foi pensada para atender aos seguintes objetivos específicos:

1. Ser acessível para pessoas que não sabem programação.
2. Ser extensível para que a própria comunidade de usuários possa incorporar novas funcionalidades.
3. Possibilitar a construção cooperativa de jogos.
4. Utilizar uma base tecnológica com suporte a recursos gráficos avançados.

Todas as decisões arquiteturais e tecnológicas tomadas no desenvolvimento do Affinity foram baseadas nesses quatro objetivos. Por exemplo, a abordagem de Desenvolvimento Baseado em Componentes (DBC), apresentada no Capítulo 5, contribuiu para tornar a ferramenta mais simples e extensível. A tecnologia Microsoft XNA foi escolhida pela facilidade de uso, recursos disponíveis, gratuidade e aceitação pela comunidade de desenvolvedores.

A seguir, cada objetivo do Affinity é detalhado, especificando quais foram os mecanismos utilizados para atingi-los.

6.1.1. Acessibilidade

Pelo caráter educacional da ferramenta Affinity, é fundamental fornecer mecanismos de acessibilidade para indivíduos que não sabem programação. O Affinity provê o AGE, uma Ferramenta Visual de Criação de Jogos (Seção 3.4), para os usuários criarem seus jogos, sozinhos ou em equipe.

Para facilitar a aprendizagem e o uso da ferramenta, foi adotado o conceito de componente para encapsular as funcionalidades comumente encontradas nos jogos. Por exemplo, há um componente pronto para movimentar entidades pelo teclado, bastando ao projetista configurar quais teclas deseja e qual a velocidade de deslocamento. Os componentes abstraem diversos controles que seriam necessários em softwares similares. Por exemplo, as ferramentas que trabalham com linguagens de programação necessitam de controles mais elaborados para obter o mesmo resultado.

6.1.2. Extensibilidade

Por mais que o Affinity contenha funcionalidades que são usadas na criação de diferentes jogos, o projetista sempre pode necessitar de uma variação nova que não está disponível. Nesse caso, a extensibilidade é um importante fator, pois possibilita que usuários incorporem novas funcionalidades implementadas por eles mesmos ou disponibilizadas na comunidade.

Para atingir esse objetivo, o Affinity provê o *framework* de componentes AGF, que é responsável por disponibilizar um conjunto de componentes básicos e fornecer funcionalidades que facilitem a criação de novos componentes, como mecanismos de

persistência, verificação de dependência, interfaces padronizadas, classes utilitária etc. O AGF será detalhado na Seção 6.3.

6.1.3. Cooperação

Cada ator apresentado na Figura 15 desempenha um papel fundamental no processo de criação de um jogo. Um dos objetivos do Affinity é possibilitar que esses atores possam trabalhar cooperativamente. Dois possíveis cenários de cooperação identificados no Affinity são:

- Artistas, desenvolvedores e projetistas compartilham suas criações com a comunidade, mas mantém o controle total para modificar as mesmas. Os demais participantes buscam na comunidade por essas criações com o intuito de se inspirarem ou mesmo de reusá-las em suas próprias criações. Nesse cenário de cooperação não há uma interação direta entre os atores.
- Artistas, desenvolvedores e projetistas trabalham em equipe na criação de um jogo específico. Os elementos criados por cada membro são compartilhados e sincronizados através de um servidor central. Quando um membro alterar um elemento e submetê-lo ao servidor, todos os outros membros recebem a alteração.

A versão atual do Affinity disponibiliza somente o segundo cenário de cooperação através de um servidor central, que será apresentado na Seção 6.3.2.

6.1.4. Base Tecnológica

O último objetivo é utilizar uma base tecnológica com suporte a recursos gráficos avançados. A base tecnológica empregada determina a qualidade gráfica dos jogos que ela é capaz de produzir. Se a tecnologia for baseada em bibliotecas do sistema operacional para criação de janelas, só será possível a criação de jogos 2D simples. Por outro lado, se a ferramenta utilizar, direta ou indiretamente, uma API gráfica como o DirectX ou o OpenGL, ela poderá trabalhar com recursos gráficos avançados em 3D.

Uma base tecnológica apropriada também possibilita o uso da ferramenta por profissionais interessados em produzir jogos comerciais, aumentando a base de usuários, as trocas de experiências e o número de extensões para a ferramenta.

6.2. Modelo Conceitual

O Affinity utiliza o modelo conceitual apresentado no Capítulo 4, que define um jogo através de entidades, tipos de entidades e componentes de comportamento. As entidades representam os objetos do mundo do jogo. Os tipos de entidade definem uma classe de entidades que seguem o mesmo comportamento, mas estão em estados diferentes. Esses conceitos são similares à classe e instância de classe da Análise Orientada a Objeto.

O comportamento de uma entidade é definido pelas manipulações de entidades que ela executa (Seção 4.1.5). A ferramenta Affinity trabalha com o conceito de componente para manipular as entidades. Os componentes são anexados às entidades para definir o comportamento ou a aparência delas. Por exemplo, há componentes para fazer a entidade seguir o ponteiro do mouse, movimentá-la através do teclado, desenhar uma textura na sua posição dentro do cenário, dentre outros. O Desenvolvimento Baseado em Componentes é uma abordagem que geralmente é usada na engenharia de software, mas no Affinity ele também está presente no modelo conceitual por ser a principal ferramenta de criação usada pelo projetista. As regras e os objetivos do jogo são definidos implicitamente pelos componentes.

A seguir é apresentado um caso de uso típico do projetista:

1. O projetista cria um novo tipo de entidade;
2. anexa componentes que definam a aparência da entidade como, por exemplo, um componente que desenha uma textura;
3. anexa componentes que definam o comportamento da entidade, indicando como ela interage com outras entidades, como ela se movimenta, como ela é destruída etc.;
4. cria as instâncias do tipo de entidade e as posiciona dentro do cenário do jogo;
5. e executa o jogo para visualizar o resultado.

A Figura 16, apresenta o modelo conceitual da ferramenta Affinity e, logo a seguir, cada conceito é descrito. Entre parênteses há o nome correspondente da classe no código para referência.

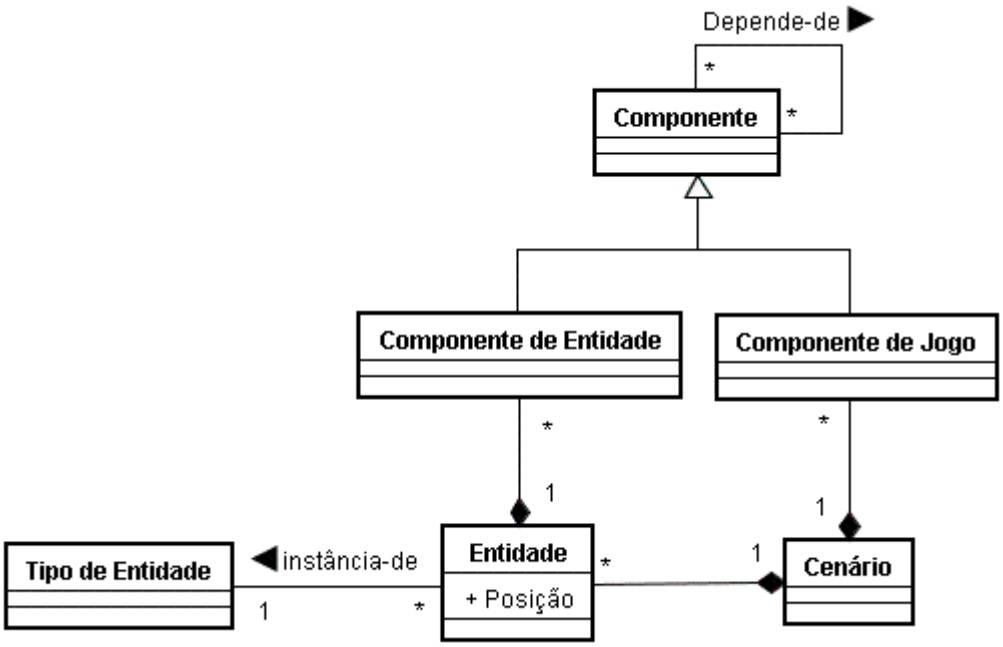

Figura 16. Modelo conceitual da ferramenta Affinity.

- **Cenário** (*Scene*): é o mundo do jogo. Todas as entidades e os componentes de jogo instanciados pertencem ao cenário. No Affinity, só há uma instância do cenário ativa.
- **Entidade** (*Entity*): é um objeto que existe dentro do cenário do jogo. O atributo Posição (*Position*) indica sua localização dentro do cenário. Se for observado nos jogos, as entidades podem ter muitos outros atributos como velocidade e pontos de vida, mas no Affinity, esses atributos pertencem aos componentes derivados por estarem relacionados com funcionalidades específicas.
- **Tipo de Entidade** (*EntityType*): é uma classe de entidades que seguem o mesmo comportamento, mas estão em estados diferentes. As entidades associadas a um tipo são chamadas de instância desse tipo.
- **Componente** (*Component*): são elementos que acrescentam alguma funcionalidade ao jogo ou as entidades. Um componente pode depender da existência de outros componentes para funcionar. Por exemplo, um componente de movimentação depende que haja um componente de física na mesma entidade que ele está anexado.
- **Componente de Entidade** (*EntityComponent*): são componentes que acrescentam funcionalidades às entidades do jogo.
- **Componente de Jogo** (*GameComponent*): são componentes que acrescentam funcionalidades que não estão associadas a uma entidade específica. Por exemplo, a gravidade é um comportamento que afeta a todas as entidades igualmente; um

contador de tempo não está associado a nenhuma entidade. Para esses casos, o Affinity trabalha com o conceito de componente de jogo.

No modelo conceitual apresentado, não há conceitos de um jogo específico, e sim um modelo genérico que deve ser estendido para incluir esses conceitos conforme a necessidade. Essa extensão deve ser feita derivando-se dos conceitos abstratos Componente de Jogo e Componente de Entidade. O Affinity provê um conjunto de componentes derivados pré-definidos para serem usados na criação de diferentes jogos.

A Tabela 2 e a Tabela 3 descrevem esses componentes resumidamente, enquanto o Anexo A apresenta-os detalhadamente. A primeira tabela apresenta os componentes de entidade derivados e a segunda apresenta os componentes de jogo derivados. A coluna "Desenhável" indica se o componente possui a habilidade de desenhar algo na tela e a coluna "Dependências" mostra os outros componentes que devem estar presentes na entidade ou no jogo para o funcionamento do componente.

Tabela 2. Lista de componentes de entidade disponíveis no Affinity.

Desenhá-vel	Nome	Dependências	Descrição
X	StaticSprite		Desenha uma imagem estática na mesma localização da entidade dona.
	PhysicComponent	PhysicEngine	Movimenta e colide a entidade dona com outras entidades.
	CollisionResponse	PhysicComponent	Configura uma resposta a uma colisão com outra entidade ou com o limitador de espaço da entidade dona.
	KeyboardMoviment	PhysicComponent	Movimenta a entidade dona de acordo com as teclas pressionadas no teclado
	MouseFollowing		Atualiza a posição da entidade dona com a posição do ponteiro do mouse.
	RandomStart	PhysicComponent	Inicializa a entidade dona com uma velocidade de magnitude constante e direção aleatória entre um ângulo mínimo e máximo.
	Shoot	PhysicComponent	Dispara outra entidade a partir da posição da entidade dona.

Tabela 3. Lista de componentes de jogo disponíveis no Affinity.

Desenhá-vel	Nome	Dependências	Descrição
	PhysicEngine		Responsável pela simulação (movimentação e colisão) de todos os corpos rígidos que são criados pelos componentes PhysicComponent.
	AudioEngine		Responsável pela execução dos efeitos sonoros e da

			trilha sonora do jogo.
X	Counter		Conta e mostra na tela valores numéricos.
X	TextWriter		Mostra um texto na tela.
X	Timer		Gerencia um contador de tempo e mostra na tela a contagem atual.
	DuplicateArea		Cria novas instâncias de um tipo de entidade, de acordo com a área e com o intervalo de tempo especificado.
	EntityMonitor		Monitora a criação e a remoção de entidades de um determinado tipo de entidade.
	LevelManager		Responsável pela manipulação das fases do jogo. Contém métodos para anunciar a vitória ou a derrota do jogador

6.3. Arquitetura

A arquitetura do Affinity segue uma topologia cliente / servidor, conforme apresentado na Figura 17.

Figura 17. Topologia da arquitetura.

O cliente é responsável por todas as operações relacionadas à construção do jogo. O servidor é responsável por manter os dados dos jogos centralizados e disponíveis pela Internet, para possibilitar o trabalho cooperativo entre os clientes.

A comunicação entre o cliente e o servidor é realizada através de um *web service*. *Web service* é qualquer serviço que está disponível pela Internet, usa um sistema de mensagem XML padronizado e não está amarrado a nenhum sistema operacional ou linguagem de programação [Cerami, 2002]. A principal razão para usar *web service* como meio de

comunicação no Affinity é possibilitar futuras versões do cliente em diferentes plataformas como *browsers*, dispositivos móveis e consoles.

6.3.1. Cliente

O Affinity provê dois subsistemas que estão em níveis de abstração diferentes: o Affinity Game Editor (AGE) e o Affinity Game Framework (AGF). O AGF é um *framework* de componentes e o AGE é uma ferramenta visual de criação de jogos que utiliza o AGF como base. Essa divisão na arquitetura do Affinity possibilita atender a dois atores: o projetista e o desenvolvedor de jogos.

O ator projetista interage somente com o AGE para definir o jogo através de um conjunto de funcionalidades disponíveis. Entretanto, as funcionalidades disponibilizadas pela ferramenta podem não ser suficientes para que o projetista crie o jogo que deseja. Para suprir essa limitação, a arquitetura do Affinity considera igualmente importante o ator desenvolvedor, que é o responsável por desenvolver novas funcionalidades que são disponibilizadas para o projetista através de *plugins*. O relacionamento entre os atores e os subsistemas que compõem o Affinity é apresentado na Figura 18.

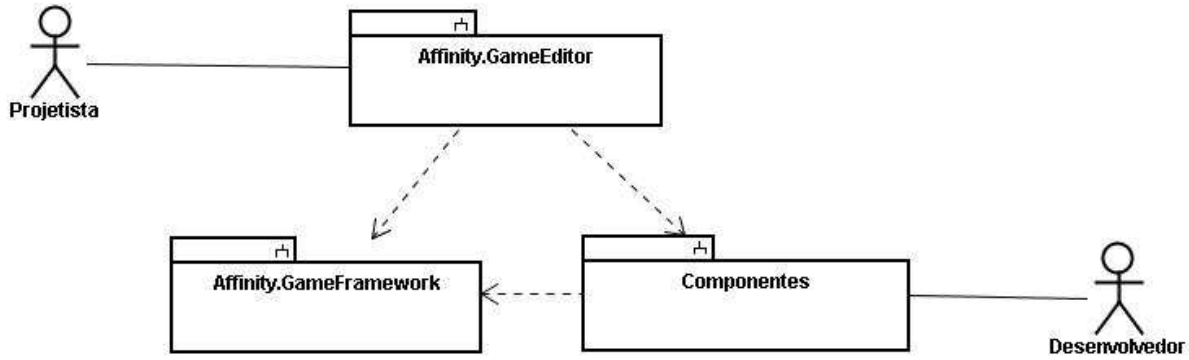

Figura 18. Relacionamento entre os atores e os subsistemas do Affinity.

O ator desenvolvedor conhece uma linguagem de programação orientada a objetos e é capaz de estender as funcionalidades presentes no AGF para criar jogos mais elaborados e únicos. Ele requer uma arquitetura que possibilite estender a ferramenta sem que seja necessário entender sua estrutura interna. Outro requisito importante é trabalhar com uma IDE que ele já está acostumado e que tenha os recursos necessários para realizar a tarefa.

Para atender a esse ator, o ambiente Affinity disponibiliza o *framework* de componentes Affinity Game Framework, que é responsável por:

- Implementar o modelo conceitual apresentado na Figura 16;
- implementar o conjunto de componentes básicos que estão listados na Tabela 2 e na Tabela 3;
- fornecer um conjunto de funcionalidades que facilitem a criação de novos componentes, como mecanismos de persistência, verificação de dependência, interfaces padronizadas, classes utilitária etc.;
- e armazenar as alterações realizadas no projeto do jogo e submetê-las ao servidor via web service.

A persistência dos projetos é feita em arquivo ou no servidor. A opção de salvar em arquivo geralmente é usada quando o projetista não deseja compartilhar seu projeto, não finalizou as alterações realizadas desde a última atualização ou a conexão com o servidor não está disponível.

6.3.2. Servidor

Na versão atual do Affinity, o servidor é responsável apenas por manter os dados dos jogos centralizados e disponíveis para que vários projetistas possam criá-los cooperativamente pela Internet. Em versões futuras, o servidor também será responsável por disponibilizar uma comunidade onde os membros possam compartilhar e classificar jogos, componentes e *assets*

O servidor armazena as informações sobre os projetos criados em um banco de dados. Quando dois projetistas trabalham no mesmo projeto, podem acontecer conflitos. Por exemplo, um projetista pode estar tentando atualizar as informações de um tipo de entidade que já foi removido. Nesses casos, o servidor retorna um erro para o projetista que está submetendo as novas informações.

6.4. Tecnologia

Esta seção apresenta a plataforma e o middleware usados no desenvolvimento do Affinity.

6.4.1. Plataforma

A plataforma de desenvolvimento e execução do cliente e do servidor do Affinity é o Microsoft .NET Framework[9]. Essa plataforma trouxe mudanças fundamentais nas ferramentas e técnicas usadas pelos desenvolvedores para construir sistemas para web, dispositivos móveis, aplicações Windows e jogos. A versão atual do .NET Framework é a 3.5, sendo que o Affinity usa a versão 2.0.

Uma das principais características da plataforma .NET é o seu suporte à multi-linguagem. A linguagem C# foi desenvolvida junto com o .NET e é amplamente aceita pela comunidade de desenvolvedores devido à sua facilidade de uso, flexibilidade e performance. O C# é a linguagem utilizada no desenvolvimento do Affinity.

O IDE (*Integrated Development Environment*) da Microsoft para o .NET é o Microsoft Visual Studio. A versão atual é a 2008, sendo que o Affinity usa a versão 2005. Esse IDE possui poderosos recursos de edição de código, refatoração e depuração. O Microsoft Visual C# Express Edition é uma versão gratuita e com menos recursos que pode ser usada pelos desenvolvedores para criar novas extensões para o Affinity.

No servidor, o SGBD (Sistema de Gerenciamento de Banco de Dados) utilizado foi o Microsoft SqlServer 2005 Express Edition[10], que também é disponibilizado gratuitamente.

6.4.2. Middleware

O *middleware* escolhido como base para o desenvolvimento do Affinity é o Microsoft XNA[11]. O XNA é uma biblioteca de classes baseada no .NET 2.0 e projetada especificamente para o desenvolvimento jogos e para promover o máximo de reúso entre as plataformas Windows e Xbox 360. O *XNA Framework* abstrai diversos aspectos envolvidos na programação de jogos como gráfico, áudio, entrada, armazenamento e controle de tempo. O XNA é disponibilizado gratuitamente.

A comunidade de usuários do XNA vem crescendo muito rapidamente. A principal razão disso é a possibilidade de criar jogos para o Xbox 360, uma iniciativa inédita da

[9] http://www.microsoft.com/net

[10] http://www.microsoft.com/sql/editions/express/default.mspx

[11] http://www.xna.com/

Microsoft em relação aos consoles de vídeo games que, até então, só grandes empresas tinham acesso.

Algumas dificuldades são encontradas ao usar o XNA. Por ser uma plataforma recente, as bibliotecas para reutilização não são tão difundidas. Entretanto, à medida que a comunidade de usuários aumenta, essa dificuldade tende a ser minimizada. Outra dificuldade é o requisito mínimo da placa de vídeo da máquina, que deve suportar o DirectX 9.0c e o Shader Model 1.1, impossibilitando o seu uso em máquinas antigas.

Capítulo 7

Estudo de Caso: Breakout

Este capítulo apresenta um estudo de caso da ferramenta Affinity para a criação do jogo *Breakout*.

O jogo *Breakout* foi desenvolvido pela Atari Inc. e introduzido em 1976. No jogo, há uma camada de blocos no topo e uma bola que se desloca e é rebatida por paredes nas laterais e na parte superior da janela. Quando um bloco é atingido, a bola é rebatida e o bloco é destruído. O jogador perde uma chance quando a bola encosta no limite inferior da janela. Para prevenir que isso ocorra, ele move uma raquete que rebate a bola quando há uma colisão. O objetivo do jogo é eliminar todos os blocos antes que as chances do jogador acabem. A Figura 19 mostra a versão original do jogo para o Atari 2600.

Figura 19. Versão do Breakout para o Atari 2600.

Para melhor ilustrar o processo de criação no Affinity, o *Breakout* será criado em três versões incrementais, com os seguintes objetivos para cada versão:

- 1ª Versão: configurar todas as funcionalidades básicas relacionadas à aparência, à colisão e à movimentação das entidades.
- 2ª Versão: configurar os eventos de vitória e derrota.
- 3ª Versão: estender o jogo com um novo componente específico para controlar a colisão da bola com a raquete, afim de melhorar o controle da direção da bola pelo jogador.

O restante do capítulo está organizado da seguinte maneira: a Seção 7.1 descreve os passos para criar a primeira versão do jogo, a Seção 7.2 descreve os passos para criar a segunda versão do jogo, a Seção 7.3 descreve os passos para criar a terceira versão do jogo e a Seção 7.4 finaliza o capítulo observando quais objetivos propostos para a ferramenta foram demonstrados.

7.1. Primeira Versão

Na primeira versão, todas as funcionalidades básicas relacionadas à aparência, à colisão e à movimentação das entidades são configuradas.

7.1.1. Tipos de Entidade

O primeiro passo para criar um jogo no Affinity é identificar os tipos de entidades presentes no mundo do jogo, bem como suas instâncias e seus comportamentos. No Breakout, os seguintes tipos de entidades são identificados:

- Raquete: entidade controlada pelo jogador que fica localizada na parte inferior da janela. Há apenas uma instância desse tipo. O jogador altera a posição horizontal da raquete utilizando o teclado. A raquete deve parar quando colidir com os limites laterais da janela.
- Bola: entidade que se movimenta livremente pela janela. Há apenas uma instância desse tipo[12]. A bola é rebatida quando colide com os limites da janela, com os blocos e com a raquete. As chances do jogador são reduzidas em um quando a bola colide com o limite inferior da janela.
- Bloco: entidade que deve ser destruída pela bola. Há diversas instâncias desse tipo localizadas na parte superior da janela. Quando a bola colide com um bloco ele é destruído. Quando não há mais instâncias do tipo de entidade Bloco, o jogador ganha.

Para criar um tipo de entidade no Affinity Game Editor, o projetista deve selecionar a pasta *Entity Types* da árvore do projeto e acionar o botão “mais” localizado na barra de ferramentas, conforme ilustrado na Figura 20.

[12] Existem variações do jogo original que contêm itens especiais para criar várias bolas ao mesmo tempo.

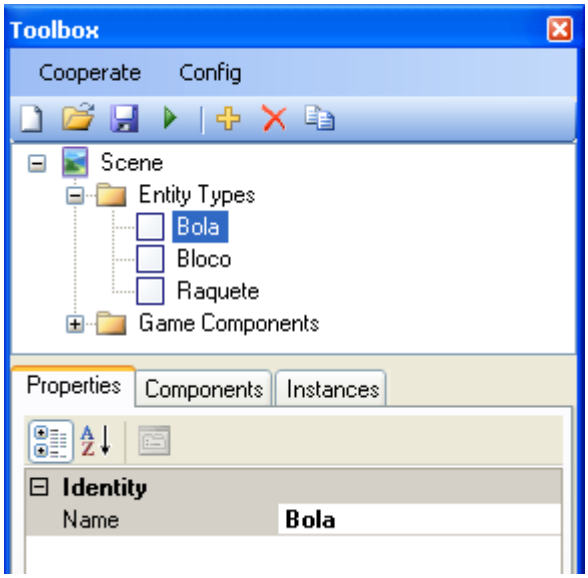

Figura 20. Painel do Affinity para criação dos tipos de entidades.

7.1.2. Componentes de Entidade

No Affinity, a aparência e o comportamento das entidades do jogo são definidos através de componentes de entidade. Os componentes de entidade são anexados aos tipos de entidades e todas as instâncias desse tipo compartilham o mesmo conjunto de componentes. Para anexar um componente de entidade no Affinity, deve-se selecionar o tipo de entidade na árvore do projeto e acionar o botão “mais” localizado no painel *Components*. Ao realizar essa operação, o Affinity abre um seletor com o conjunto de componentes disponíveis, conforme ilustrado na Figura 21.

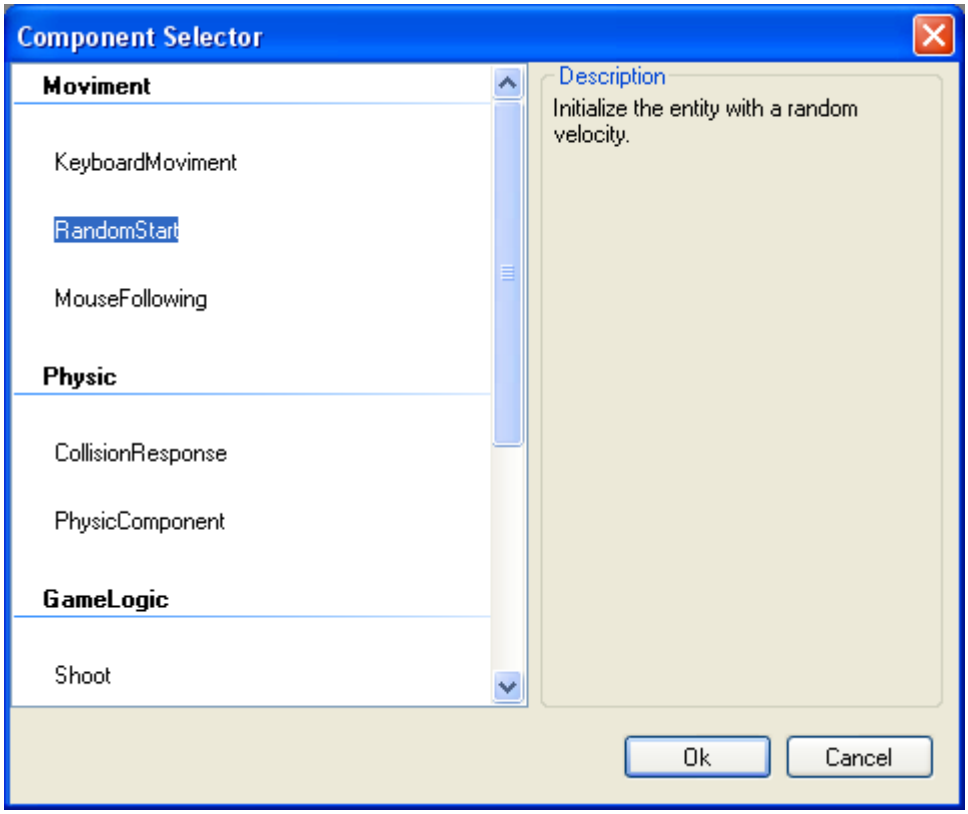

Figura 21. Seletor de componentes do Affinity.

Para configurar um componente, o projetista deve selecioná-lo na lista dos componentes anexados e editar suas propriedades nos campos fornecidos, conforme ilustrado na Figura 22.

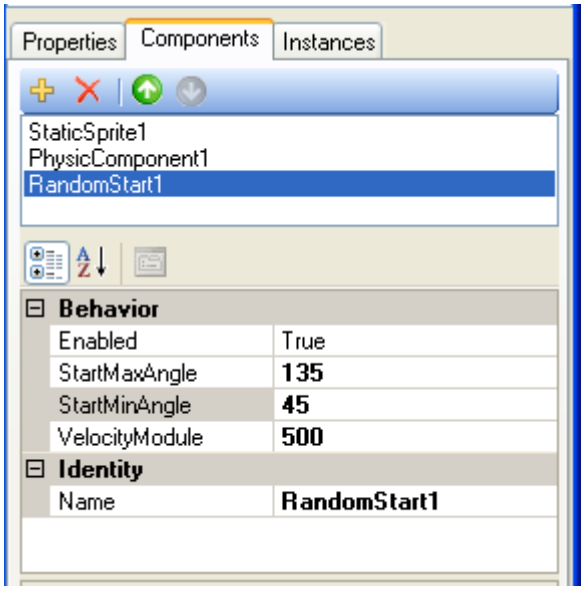

Figura 22. Painel do Affinity de edição dos componentes de entidade.

A Tabela 4 apresenta os componentes de entidade e suas principais propriedades para cada tipo de entidade da primeira versão do jogo Breakout. A descrição completa dos componentes e de suas propriedades está disponível no Anexo A.

Tabela 4. Componentes de entidade para o jogo Breakout.

Tipo de Entidade	Componente de Entidade	Principais propriedades
Bola	StaticSprite	TextureSource = “Content\soccer_ball.png” Width = 28 Height = 28
	PhysicComponent	Limit = {X:0 Y:0 Width:600 Height:400}
	RandomStart	StartMinAngle = 45 StartMaxAngle = 135 VelocityModule = 500
	CollisionResponse	OtherEntityType = All ResponseType = Bounce
Bloco	StaticSprite	ColorFilter = {192; 0; 0; 255} Width = 34 Height = 15
	PhysicComponent	
	CollisionResponse	OtherEntityType = Bola ResponseType = Destroy
Raquete	StaticSprite	ColorFilter = {64; 64; 64; 255} Width = 115 Height = 26
	PhysicComponent	Limit = {X:0 Y:0 Width:600 Height:400}
	CollisionResponse	OtherEntityType = SpaceLimit ResponseType = Stop
	KeyboardMoviment	MoveUpKey = None MoveDownKey = None MoveRightKey = Right MoveLeftKey = Left Velocity = 600

7.1.3. Instâncias dos Tipos de Entidade

Nas seções anteriores, os tipos de entidades foram criados e seus comportamentos foram definidos através de componentes. O próximo passo é “instanciar” os tipos de entidades, criando e posicionando entidades dentro do cenário. Para isso, o projetista deve selecionar o tipo de entidade na árvore do projeto e acionar o botão “mais” localizado na aba *Instances*. Ao realizar essa operação, uma nova instância do tipo é criada e fica visível no painel de visualização do cenário jogo. O projetista pode arrastar a entidade pelo painel para posicioná-la. A Figura 23 ilustra o cenário final do jogo após criar e posicionar todas as instâncias. A instância selecionada é indicada por uma caixa azul.

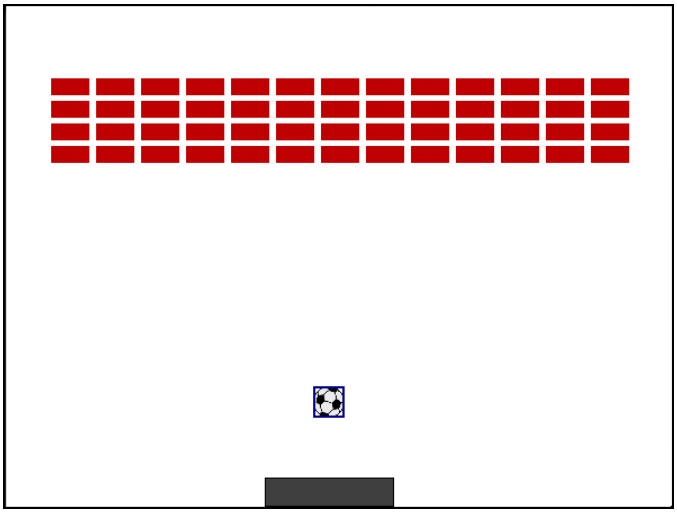

Figura 23. Painel de visualização do cenário do jogo Breakout.

7.2. Segunda Versão

A segunda versão do jogo Breakout contém os eventos de vitória e derrota configurados. Essa configuração é realizada através de componentes de jogo e através da configuração de ações para determinados componentes.

7.2.1. Componentes de Jogo

Componentes de jogo acrescentam funcionalidades que não são exclusivas de uma entidade específica. Para incluir um novo componente de jogo, o projetista deve selecionar a pasta *Game Components* da árvore do projeto e acionar o botão “mais” localizado na barra de ferramentas. Ao realizar essa operação, o Affinity abre um seletor com todos os componentes de jogo disponíveis. Para alterar as configurações de um componente, o projetista deve selecioná-lo na árvore do projeto e editar os campos presentes na aba *Properties*, conforme ilustrado na Figura 24.

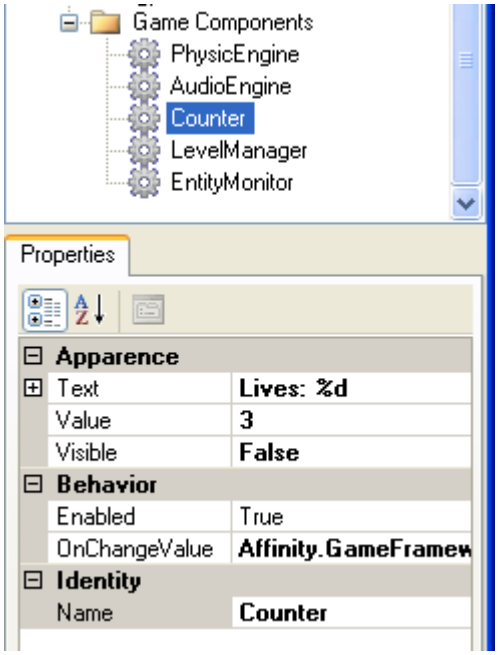

Figura 24. Painel do Affinity para configuração de componentes de jogo.

A **Erro! Fonte de referência não encontrada.** apresenta os componentes de jogo necessários para a implementação da 2ª versão do jogo Breakout.

Tabela 5. Componentes de entidade para o jogo Breakout.

Componente de Jogo	Principais propriedades	Função
PhysicEngine	Gravity = {0; 0}	Calcular movimentações e colisões entre as entidades.
Counter	Text.Text = “Chances: %d” Text.Position = {-283; 210} Text.Align = Left Value = 3	Contar a quantidade de chances que restam para o jogador.
LevelManager		Pausar o jogo e mostrar uma mensagem de vitória ou derrota para o jogador.
EntityMonitor		Contar a quantidade de blocos que restam no cenário para detectar a vitória do jogador.

Entretanto, não basta incluir esses componentes para atingir o objetivo da segunda versão. É necessário configurar ações em resposta aos eventos disparados pelos componentes.

7.2.2. Configuração de Ações

As ações são métodos que o projetista configura para serem invocados em resposta aos eventos disparados pelos componentes. As ações dão mais flexibilidade quando as propriedades dos componentes não são suficientes para configurar a lógica do jogo. Por

exemplo, toda vez que a bola do jogo Breakout atingir a parede inferior da janela, o contador de chances deve ser reduzido em um. Como não há nenhuma propriedade no componente de CollisionResponse para configurar esse tipos de comportamento, o projetista deve configurar uma ação para o evento disparado por esse componente.

Para configurar uma ação em um componente, o projetista deve acionar o botão elipses "..." que aparece ao clicar na propriedade correspondente ao evento do componente, conforme ilustrado na Figura 25.

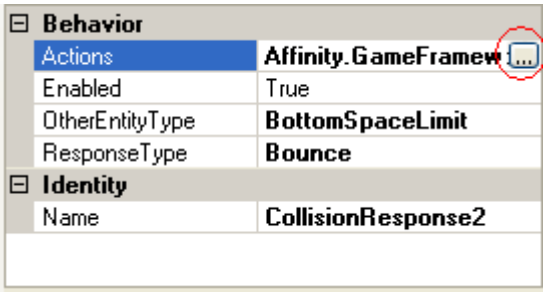

Figura 25. Propriedade correspondente ao evento de resposta à colisão.

Ao acionar essa opção, o Affinity abre a janela de configuração de evento ilustrada na Figura 26. O campo *Target* lista todos os componentes de jogo inseridos, o campo *Action* lista todos os métodos que podem ser chamados no componente selecionado e a tabela mostra todos os parâmetros que devem ser preenchidos para o método selecionado.

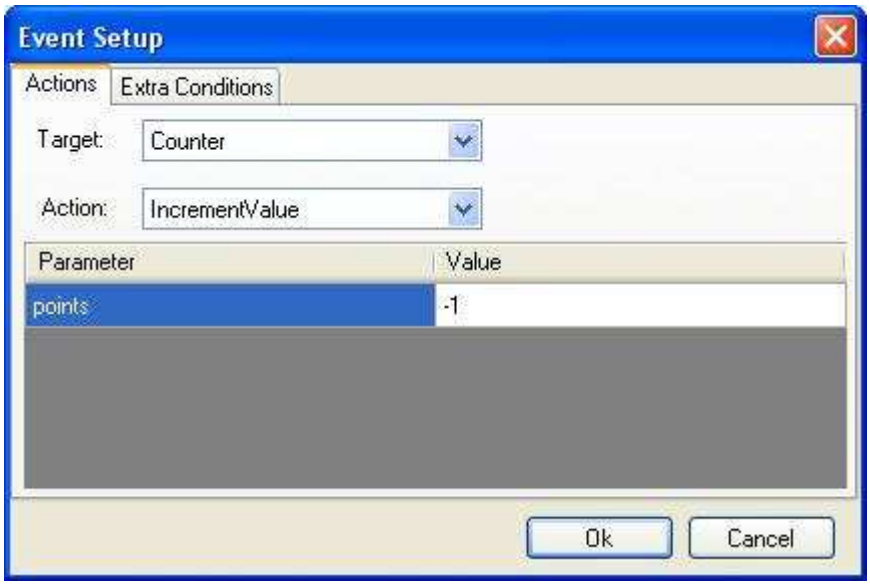

Figura 26. Painel do Affinity para configuração de ações do evento.

Um evento pode precisar que certas condições sejam satisfeitas para que ele seja disparado. Por exemplo, o evento que é disparado quando o contador de chances é alterado deve invocar o método que anuncia o fim de jogo somente quando ele chegar à zero. Essa

configuração é feita na aba *Extra Conditions* da janela de configuração de evento, conforme ilustrado na Figura 27.

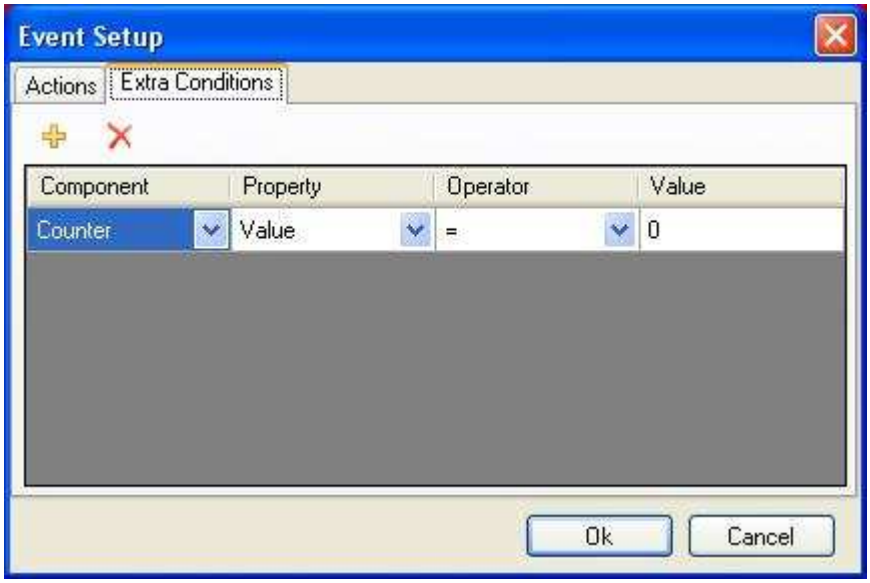

Figura 27. Painel do Affinity para configuração de condições do evento.

A Tabela 6 apresenta todos os eventos que devem ser configurados para a implementação da segunda versão do Breakout.

Componente Fonte	Componente Alvo	Método	Condições
Bola.CollisionResponse	Counter	IncrementValue(-1)	
Counter	LevelManager	GameOver("Fim de Jogo!")	Counter.Value = 0
EntityMonitor	LevelManager	Win("Você ganhou!")	EntityMonitor.Count = 0

Tabela 6. Eventos configurados para o jogo Breakout.

7.3. Terceira Versão

Na segunda versão, o jogo Breakout já está completo. Entretanto, o jogador não consegue controlar a direção que a bola é rebatida e isso atrapalha a jogabildiade, principalmente quando restam poucos blocos para serem destruídos. Isso acontece, pois, ao detectar que uma entidade colidiu com outra e que a resposta à colisão é rebater, o *framework* calcula a nova velocidade refletindo o vetor velocidade contra a normal da colisão, conforme ilustrado na Figura 28.

N: vetor normal
Va: vetor velocidade antes da colisão
Vd: vetor velocidade depois da colisão

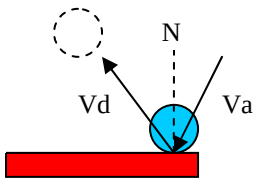

Figura 28. Cálculo padrão da velocidade ao rebater uma entidade.

Na terceira versão do Breakout, o jogador tem mais controle sobre a direção da bola. Uma forma de fazer isso é calculando o vetor velocidade resultante da colisão de acordo com a posição relativa que a bola colidiu com a raquete, ou seja, quanto mais à direita da raquete a bola colidir, menor será o ângulo, e quanto mais à esquerda, maior será o ângulo. Dessa forma, o ângulo do vetor velocidade resultante é limitado entre 45º e 135º e é inversamente proporcional à componente x do vetor diferença entre a bola e a raquete, conforme ilustrado na Figura 29.

N: vetor normal.

dx: componente x do vetor diferença entre a bola e a raquete.

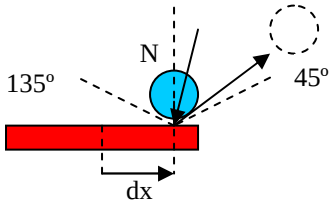

Figura 29. Cálculo modificado da velocidade para o Breakout.

Como não há nenhum componente de entidade pronto que implemente esse tipo de comportamento é necessário estender o Affinity com um novo componente. No Affinity, as extensões são realizadas através de *plugins*. Um *plugin* é uma dll que contém classes que implementam as interfaces dos componentes ou que herdam de classes que já façam isso.

Para criar um *plugin* no Affinity, é necessário utilizar as ferramentas de desenvolvimento disponíveis para a plataforma *Microsoft .NET*. Neste estudo de caso foi utilizado o *Microsoft Visual C# 2005 Express Edition*. O código presente na Figura 30 implementa o comportamento de colisão da bola com a raquete. A classe *BallCollision* herda do componente de resposta à colisão padrão do Affinity, o *CollisionResponse*, e sobrescreve o método *Collide*, que é chamado sempre que uma colisão com a entidade configurada acontece.

```
public class BallCollision : CollisionResponse
{
	private const double MAX_ANGLE = 3.0 * Math.PI / 4.0;
	private const double MIN_ANGLE = Math.PI / 4.0;

	public override void Collide(Entity otherEntity, Vector2 normal)
	{
		Vector2 dif = this.Entity.Position - otherEntity.Position;
		float xoffset = dif.X;

		// limita o offset máximo e mínimo
		// para melhorar o controle do jogador
		float xmax = otherEntity.Physic.Width / 2.0f;
		float xmin = -otherEntity.Physic.Width / 2.0f;
		xoffset = Math.Min(xmax, xoffset);
		xoffset = Math.Max(xmin, xoffset);

		// calcula o percentual do offset
		float perc = (xoffset - xmin) / (xmax - xmin);
		perc = 1 - perc;

		// aplica o percentual no ângulo
		double ang = perc * (MAX_ANGLE - MIN_ANGLE) + MIN_ANGLE;
		float mod = this.Entity.Physic.Velocity.Length();

		// Muda a velocidade
		this.Entity.Physic.Velocity = new Vector2(
			(float)Math.Cos(ang) * mod,
			(float)Math.Sin(ang) * mod);
	}
}
```

Figura 30. Código do componente de colisão da bola do jogo Breakout.

Após copiar a dll resultante da compilação para a pasta “plugin”, o componente *BallCollision* fica disponível para ser anexado na entidade bola.

7.4. Conclusão do Estudo de Caso

O estudo de caso apresentado neste capítulo demonstrou dois objetivos propostos para o Affinity:

- Ser acessível para pessoas que não sabem programação.
- Ser extensível para que a própria comunidade de usuários possa incorporar novas funcionalidades.

Nas seções 7.1 e 7.2, foi demonstrado que é possível criar uma versão simplificada do *Breakout* sem a necessidade de programar. Os tipos de entidades, suas instâncias e seus comportamentos foram configurados somente utilizando editores visuais baseados no modelo conceitual do Affinity, que foi apresentado na Seção 6.2.

Na Seção 7.3, foi demonstrado que o Affinity é extensível, quando um novo componente específico do *Breakout* foi desenvolvido e facilmente integrado ao framework de componentes.

Capítulo 8

Conclusão

Conforme discutido no Capítulo 2, criar jogos é uma excelente ferramenta educacional. Através do processo criativo e lúdico de criação de jogos, os usuários desenvolvem importantes aptidões de aprendizagem para suas vidas e carreiras. Eles desenvolvem a habilidade de pensar criativamente, ou seja, aprendem a achar soluções inovadoras para problemas inesperados; trabalham num ambiente cooperativo e interdisciplinar que envolve conceitos em diferentes áreas; e aprendem a ser bons projetistas: como conceituar um projeto, como fazer uso dos recursos disponíveis, como persistir e achar alternativas quando as coisas dão erradas e como colaborar com os outros.

O uso de ferramentas tecnológicas nesse processo também desenvolve a fluência digital nos projetistas, ou seja, eles aprendem a criar e se expressar efetivamente com a computação e não ser apenas consumidores passivos. Há cada vez mais consumidores assumindo o papel de produtor de conteúdos multimídia. Eles participam da criação de vídeos, textos jornalísticos, músicas e jogos.

A partir da análise das ferramentas utilizadas no desenvolvimento de jogos, apresentada no Capítulo 3, foi proposta uma nova ferramenta de construção de jogos, denominada Affinity, que contempla os seguintes objetivos:

1. Ser acessível para pessoas que não sabem programação.
2. Ser extensível para que a própria comunidade de usuários possa incorporar novas funcionalidades.
3. Possibilitar a construção cooperativa de jogos.
4. Utilizar uma base tecnológica com suporte a recursos gráficos avançados.

A abordagem de Desenvolvimento Baseado em Componentes (DBC), apresentada no Capítulo 5, contribuiu para atingir os objetivos propostos. O conceito de componente é simples que as pessoas encontram no dia a dia. No contexto deste trabalho, os componentes encapsulam as funcionalidades comumente encontradas nos jogos.

Para atender ao primeiro objetivo, o Affinity provê um editor visual, o Affinity Game Editor, para que os próprios projetistas componham e configurem os componentes sem a necessidade de usar uma linguagem de programação. No Capítulo 7, foi apresentado um estudo de caso do Affinity na criação do jogo *Breakout*, o qual mostrou ser possível obter uma versão "jogável" sem a necessidade de programar.

O DBC também contribuiu para aumentar a extensibilidade da ferramenta. Novos componentes podem ser incorporados pelos desenvolvedores sem a necessidade do conhecimento da arquitetura interna do Affinity e reusando diversas funcionalidades providas pelo Affinity Game Framework. No mesmo estudo de caso, foi facilmente desenvolvido um componente customizado para melhorar o jogo, o que demonstrou o segundo objetivo.

As funcionalidades referentes à cooperação foram implementadas usando web services para intermediar a comunicação cliente/servidor. Todas as alterações que um projetista submete ao servidor ficam disponíveis para outros projetistas atualizarem seus projetos. O terceiro objetivo foi demonstrado em testes com os usuários. Após um período de estudo da interface da ferramenta, os usuários recriaram o jogo *Breakout* em cooperação, cada um sendo responsável por um conjunto de funcionalidades do jogo.

Por fim, a tecnologia empregada no desenvolvimento do Affinity foi a *engine* Microsoft XNA. Essa *engine* está alinhada com o quarto objetivo do projeto, pois ela provê uma base tecnológica que possibilitará futuras expansões da ferramenta que incorporem gráficos mais elaborados.

8.1. Trabalhos Futuros

Como trabalho futuro, pretende-se evoluir o ambiente Affinity em duas frentes: no cliente e no servidor. No cliente, os trabalhos se referem ao desenvolvimento de novas funcionalidades para tornar a ferramenta mais fácil de usar e mais abrangente. No servidor, os trabalhos se referem a melhorar os mecanismos de cooperação entre os usuários.

No cliente, pretende-se suportar gráficos em três dimensões (3D). Para o desenvolvimento dessa funcionalidade alguns componentes deverão ser adaptados, mas não é necessário trocar de engine gráfica, já que o XNA tem suporte a 3D e ele foi escolhido justamente para possibilitar futuras expansões da ferramenta. Entretanto, a engine de física adotada só suporta simulações em duas dimensões e necessitaria ser trocada.

Pretende-se também implementar novas funcionalidades como o suporte para mais de uma fase por jogo, animações de textura e rotação de entidades. No editor, pretende-se incluir novos recursos que facilitarão o uso da ferramenta, como o suporte a *drag and drop* e teclas de atalho.

Por último, pretende-se portar o *framework* para executar os jogos no console XBox 360, através do XNA que já tem o suporte necessário. Com essa nova funcionalidade, os projetistas poderão criar jogos para a rede XBox Live, o que dará uma visibilidade muito maior tanto para os jogos criados quanto para o Affinity.

No lado do servidor, pretende-se incluir novos mecanismos para o compartilhamento de projetos, componentes e assets. Pretende-se também incluir um mecanismo de avaliação e classificação dos jogos disponibilizados e um mecanismo de segurança para restringir as permissões dos usuários por jogo. Hoje, todos os usuários têm acesso irrestrito a todos os jogos.

Referências Bibliográficas

Almeida, E.S, Álvaro, A., Garcia, V.C, Mascena, J.C.C.P, Burgério, V.A.A, Nascimento, L.M., Lucrédio, D. & Meira, S.L. (2007) "*CRUiSE: Component Reuse in Software Engineering*", C.E.S.A.R – Centro de Estudos e Sistemas Avançados do Recife, disponível em <cruise.cesar.org.br>, acesso em 29/02/2008.

Alves, L. (2004). "*Game Over: Jogos Eletrônicos e Violência*". Salvador, 2004. Tese de doutorado. 211p. UFBA.

Barroca, L., Gimenes, I.M.S. & Huzita, E.H.M. (2005). "Conceitos Básicos", in: "*Desenvolvimento Baseado em Componentes"*, Gimenes, I.M.S. & Huzita, E.H.M. (eds), Editora Ciência Moderna, Rio de Janeiro, 2005. ISBN 85-7393-406-9, pg. 57-103.

Buckland, M. (2005), "*Programming Game AI by Example*", Wordware Publishing.

Cerami, E. (2002), *"Web Services Essentials: Distributed Applications with XML-RPC, Soap, UDDI"*, O'Reilly, ISBN: 0596002246.

Chen, X. (2004). "*Developing Application Frameworks in .NET"*, Apress, 374 p., ISBN: 1590592883.

Clickteam (1994). "*Klik & Play*", disponível em <members.fortunecity.com/oponto/klik.html>, acesso em 28/08/2007.

Clickteam (1996). "*The Games Factory 2*" e "*Multimedia Fusion 2*", disponíveis em <www.clickteam.com>, acesso em 28/08/07.

Clua, E.W.G. & Bittencourt, J. R. (2004) "*Uma Nova Concepção para a Criação de Jogos Educativos*", SBIE'2004.

Clua, E.W.G.; Junior, C.L.L.; Nabais, R.J.M. (2002) "*Importância e Impacto dos Jogos Educativos na Sociedade*." In: I Workshop Brasileiro de Jogos e Entretenimento Digital. Proceedings. SBC: Fortaleza, 2002.

D'Souza, D.F. & Wills, A.C. (1998). *"Objects, Components and Frameworks with UML: The Catalysis Approach"*, Addison Wesley, ISBN 0-201-31012-0, 1998.

ECMA International (1997), "ECMAScript : A general purpose, cross-platform programming language", Standard ECMA-262.

Fayad, M. E. (1999) "*Building application frameworks: Object-oriented foundations of framework design*". 1. ed. Nova York: John-Wiley & Sons.

Fazenda, I.C.A. (1994) "*Interdisciplinaridade: História, Teoria e Pesquisa*". Campinas, SP: Papirus.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995) *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley Professional Computing Series, 395p., ISBN: 0201633612.

GarageGames (2007). "*Torque Game Builder*", disponível em <www.garagegames.com/products/torque/tgb>, acesso em 28/08/2007.

Gee, J.P. (2003). *"What Video Games Have to Teach Us About Learning and Literacy"*, Palgrave Macmillan.

Guizzardi, G. (2001) *"Desenvolvimento Para e Com Reuso: Um Estudo de Caso no Domínio de Vídeo sob Demanda"*, Dissertação de Mestrado, Departamento de Informática, UFES, Vitória, ES.

Harel, I. (1991). "*Children Designers*" Ablex Publishing. Norwood, NJ.

Ilha, P.C.A. & Cruz, D.M. (2005), "*Reality Simulation in Education: the SimCity in Brazilian High School"*. WORKSHOP BRASILEIRO DE JOGOS E ENTRETENIMENTO DIGITAL, São Paulo. Anais WJOGOS 2005. Porto Alegre: SBC, 2005, p.295-299.

Jia, X (2000) *"Object-Oriented Software Development Using Java: principles, patterns and frameworks"*. Addison-Wesley.

Jenkins, H. (2006). "*Convergence Culture: Where Old and New Media Collide"*. New York University Press.

Johnson, R. E. (1997) "*Frameworks = (Components + Patterns): How frameworks compare to other object-oriented reuse techniques*", Communications of the ACM, v. 40, n. 10.

Kafai, Y. B. (1995). "*Minds in play: Computer game design as a context for children's learning"*. Hillsdale, NJ: Erlbaum.

Krueger, C.W. (1992). *"Software Reuse"*, ACM Computing Surveys, Volume 4 Issue 2 p131-183.

Kafai, Y. B. (2001). "*The educational potential of electronic games: From games–to–teach to games–to–learn"*. Chicago: Playing by the Rules Cultural Policy Center, University of Chicago.

Lutz, M. & Ascher, D. (2003), "*Learning Python, Second Edition*", O'Reilly Media.

Michael, D & Chen, S. (2006). "*Serious Games: Games That Educate, Train, and Inform"*, Course Technology PTR, 312p., ISBN 1592006221.

MIT (2007). "*Scratch*", disponível em <scratch.mit.edu>, acesso em 28/08/2007.

Monroy-Hernandez, A. (2007). *"ScratchR: sharing user-generated programmable media"*, Proceedings of the 6th international conference on Interaction design and children, Pages: p. 167 – 168, Aalborg, Denmark.

Moore, J. M. & Bailin, S.C. (1991) *"Domain Analysis: Framework For Reuse"*, in Prieto-Diaz, R. & Arango, G. (eds.), *"Domain Analysis and Software System Modeling"*. LosAlamitos, CA: IEEE Computer Society Press, pp. 179-203.

Nokia (2007). "*Nokia predicts 25% of entertainment by 2012 will be created and consumed within peer communities*", disponível em <press.nokia.com/PR/200712/1172517_5.html>, acesso em 29/05/2008.

Parberry I. (2007), *"Game Development in Computer Science Education: From Outcast to Mainstream"*, Guest Editor's Introduction, Journal of Game Development, Vol. 2, No. 2, pp. 5-6, Feb. 2007.

Papert, S. (1980). *"Mindstorms: Children, computers and powerful ideas"*. NY: Basic Books.

Papert, S. (1993). *"The Children's Machine"*. New York: Basic Books.

Partnership for 21st Century Skills (2003). "*Learning for the 21st Century*", disponível em <www.21stcenturyskills.org>, acesso em 03/06/08.

Prensky, M. (2001). *"Digital Game-Based Learning"*. New York: McGraw Hill.

Prensky, M. (2006). *"Don't bother me mom, I'm learning!"*, Paragon House Publishers.

Resnick, M. (2002). *"Rethinking learning in the digital age"*. In G. Kirkman (Ed.). The global information technology report: Readiness for the networked world. Oxford: Oxford University Press.

Resnick, M. (2007). "*All I Really Need to Know (About Creative Thinking) I Learned (By Studying How Children Learn) in Kindergarten*". Proceedings of the ACM SIGCHI conference on Creativity & Cognition, Washington, DC.

Robertson, J. & Good, J. (2005), "*Story creation in virtual game worlds*", Communications of the ACM, v.48 n.1, janeiro de 2005.

Rollings, A. & Morris, D. (2004), "*Game Architecture and Design – A New Edition*", New Riders Publishing.

Szyperski, C. (1997). *"Component Software: Beyond Object-Oriented Programming"*, Addison-Wesley, ISBN 0-201-17888-5

Valente, J.A. (1999) *"O computador na sociedade do conhecimento"*, Campinas: UNICAMP/NIED, 156 p.

Werner, C.M.L. & Braga, R.M.M.B (2005) *"A Engenharia de Domínio e o Desenvolvimento Baseado em Componentes"*, in: Desenvolvimento Baseado em Componentes, Gimenes, I.M.S. & Huzita, E.H.M. (eds), Editora Ciência Moderna, Rio de Janeiro, 2005. ISBN 85-7393-406-9, pg. 57-103.

YoyoGames (2007). "*Game Maker*", disponível em <www.yoyogames.com>, acesso em 27/08/07.

Zagal, J et. al. (2005). *"Towards an Ontological Language for Game Analysis"*, proceedings of the Digital Interactive Games Research Association Conference (DiGRA 2005), Vancouver B.C., June, 2005.

Zerbst, S. & Düvel, O. (2004), "*3D Game Engine Programming*". Thomson Course Technology, Premier press.

Anexo A

Descrição dos Componentes

Este anexo fornece uma descrição detalhada de todos os componentes implementados na versão atual do Affinity. Cada componente é apresentado com as seguintes informações:

- Descrição: resume a funcionalidade do componente.
- Exemplo de uso: fornece um exemplo prático de utilização do componente.
- Dependências: indica todos os componentes que devem estar presentes na entidade ou no jogo para o funcionamento correto do componente.
- Único: indica se só pode haver uma instância do componente anexado à entidade ou ao jogo.
- Propriedades: descreve todas as propriedades do componente visíveis aos projetistas. As propriedades definem o comportamento do componente.
- Métodos: descreve todos os métodos do componente disponíveis para o projetista configurar nos eventos.

O restante do anexo está organizado da seguinte maneira: a Seção A.1 descreve todos os componentes de entidade e a Seção A.2 descreve todos os componentes de jogo.

A.1. Componentes de Entidade

Componentes de entidade acrescentam funcionalidades às entidades do jogo. Esta seção descreve todos os componentes de entidade presentes na versão atual do Affintiy.

A.1.1. StaticSprite

Descrição: desenha uma imagem estática na mesma localização da entidade dona.

Exemplo de uso: utilizado para fornecer uma representação visual para a entidade como, por exemplo, uma nave, um monstro, uma bola etc.

Dependências: nenhuma.

Único: não.

Propriedades:

Nome	Tipo	Descrição
TextureSource	string	Caminho da textura que será desenhada.
ColorFilter	Color	Filtro de cor aplicado na imagem.
Width	int	Largura da imagem (quando a propriedade TextureSource é alterada, a largura é atualizada automaticamente).
Height	int	Altura da imagem (quando a propriedade TextureSource é alterada, a altura é atualizada automaticamente).
IsBoudingBoxVisible	bool	Indica se deve ser exibida uma caixa em torno da entidade.
FlipHorizontal	bool	Indica se a imagem deve ser invertida horizontalmente.
FlipVertical	bool	Indica se a imagem deve ser invertida verticalmente.

A.1.2. PhysicComponent

Descrição: movimenta e colide a entidade dona com outras entidades.

Exemplo de uso: geralmente utilizado em conjunto com outros componentes.

Dependências: PhysicEngine.

Único: sim.

Propriedades:

Nome	Tipo	Descrição
Limit	Rectangle	Define um limite para a entidade se movimentar.
LinearDragCoefficient	float	Coeficiente de atrito.
Velocity	Vector2	Velocidade inicial da entidade

A.1.3. CollisionResponse

Descrição: Configura uma resposta a uma colisão com outra entidade ou com o limitador de espaço da entidade dona.

Exemplo de uso: no jogo breakout, esse componente é usado para destruir os blocos e detectar quando o jogador perdeu uma chance.

Dependências: PhysicComponent.

Único: não.

Propriedades:

Nome	Tipo	Descrição
ResponseType	ResponseTypes	- Assume um dos seguintes valores: Bounce: rebate a entidade na outra entidade que colidiu; Stop: para a entidade; Ignore: ignora a colisão e continua o movimento; Destroy: destrói a entidade. - O valor padrão dessa propriedade é Ignore
OtherEntityType	string	- Tipo de entidade sobre o qual o componente responde. - Se especificado a string “All”, o componente responde a uma colisão com qualquer entidade. - Se especificados “SpaceLimit”, “LeftSpaceLimit”, “RightSpaceLimit”, “TopSpaceLimit” ou “BottomSpaceLimit”, responde sobre o limite de espaço, com todos, o esquerdo, o direito, o superior e o inferior respectivamente. - O valor padrão dessa propriedade é “All”.
Actions	ActionSequencer	Configura uma seqüência de ações para serem executadas quando ocorrer a colisão.

A.1.4. KeyboardMoviment

Descrição: movimenta a entidade dona de acordo com as teclas pressionadas no teclado.

Exemplo de uso: no jogo breakout, esse componente é usado para movimentar a raquete.

Dependências: PhysicComponent.

Único: sim.

Propriedades:

Nome	Tipo	Descrição
Velocity	int	Velocidade aplicada na direção correspondente a tecla pressionada.
MoveUpKey	Keys	Tecla que movimenta a entidade para cima. Padrão: Up.
MoveDownKey	Keys	Tecla que movimenta a entidade para baixo. Padrão: Down.
MoveRightKey	Keys	Tecla que movimenta a entidade para direita. Padrão: Right.
MoveLeftKey	Keys	Tecla que movimenta a entidade para esquerda. Padrão: Left.

A.1.5. MouseFollowing

Descrição: Atualiza a posição da entidade dona com a posição do ponteiro do mouse.

Exemplo de uso: no jogo breakout, esse componente é usado para mover a raquete na direção horizontal.

Dependências: PhysicComponent.

Único: sim.

Propriedades:

Nome	Tipo	Descrição
VerticalFollowing	bool	Indica se a entidade segue a posição do mouse verticalmente. Padrão: true.
HorizontalFollowing	bool	Indica se a entidade segue a posição do mouse horizontalmente. Padrão: true.

A.1.6. RandomStart

Descrição: Inicializa a entidade dona com uma velocidade de magnitude constante e direção aleatória entre um ângulo mínimo e máximo.

Exemplo de uso: no jogo breakout, a bola sempre começa subindo em uma direção aleatória.

Dependências: PhysicComponent.

Único: sim.

Propriedades:

Nome	Tipo	Descrição
VelocityModule	float	Módulo da velocidade inicial.
StartMinAngle	float	Ângulo mínimo da velocidade. Padrão: 0.
StartMaxAngle	float	Ângulo máximo da velocidade. Padrão: 360.

A.1.7. Shoot

Descrição: dispara outra entidade a partir da posição da entidade dona.

Exemplo de uso: em jogos de tiro, o personagem dispara uma bala para atingir os adversários.

Dependências: PhysicComponent.

Único: não.

Propriedades:

Nome	Tipo	Descrição
ShootAngle	float	Ângulo de disparo.
ShootKey	Keys	Tecla usada para disparar.
ShootMouseButton	MouseButton	Botão do mouse usado para disparar.
ShootDelay	float	Intervalo de tempo mínimo entre dois disparos.
ShootVelocity	float	Magnitude da velocidade de disparo.
ShootEntity	string	Tipo de entidade que será disparado.
ShootActions	ActionSequencer	Evento disparado no momento do disparo.

A.2. Componentes de Jogo

Componentes de jogo acrescentam funcionalidades que não estão associadas a uma entidade específica. Esta seção descreve todos os componentes de jogo presentes na versão atual do Affintiy.

A.2.1. PhysicEngine

Descrição: responsável pela simulação (movimentação e colisão) de todos os corpos rígidos que são criados pelos componentes PhysicComponent.

Exemplo de uso: em alguns jogos, todas as entidades são afetadas pela força da gravidade.

Dependências: nenhuma.

Único: sim.

Propriedades:

Nome	Tipo	Descrição
Gravity	Vector2	Gravidade aplicada a todas as entidades que possuem um PhysicComponent anexado.

A.2.2. AudioEngine

Descrição: responsável pela execução dos efeitos sonoros e da trilha sonora do jogo.

Exemplo de uso: no jogo Breakout, o evento de colisão da bola com o bloco chama o método “Play” do componente AudioEngine para tocar um efeito sonoro.

Dependências: nenhuma.

Único: sim.

Propriedades:

Nome	Tipo	Descrição
Volume	float	Volume do jogo. Varia de 0 à 1. Padrão: 1.
Soundtrack	string	Caminho do arquivo da trilha sonora que é executada em loop.

Métodos:

Nome	Argumentos	Descrição
Play	audio:string	Executa o arquivo especificado.

A.2.3. Counter

Descrição: conta e mostra na tela valores numéricos.

Exemplo de uso: no jogo Breakout, o componente Counter é usada para contar o número de chances restantes do jogador ou o seu número de pontos.

Dependências: nenhuma.

Único: não.

Propriedades:

Nome	Tipo	Descrição
TextWriter	TextWriter	Referência para o componente do tipo TextWriter.
Value	int	Valor do contador.
OnChangeValue	ActionSequencer	Evento chamado quando o valor da propriedade “Value” é alterado.

Métodos:

Nome	Argumentos	Descrição
SetValue	points:int	Atribui um valor para o contador.
IncrementValue	points:int	Incrementa o valor do contador.

A.2.4. TextWriter

Descrição: mostra um texto na tela.

Dependências: nenhuma.

Único: não.

Propriedades:

Nome	Tipo	Descrição
Align	TextAlign (Left, Center, Right)	Alinhamento do texto.
Scale	float	Tamanho do texto.
Text	string	Texto que é desenhado.
Position	Vector2	Posição onde o texto é desenhado.
Color	Color	Cor do texto.

A.2.5. Timer

Descrição: gerencia um contador de tempo e mostra na tela a contagem atual.

Exemplo de uso: em alguns jogos, a fase dura um período fixo de tempo. Para isso, deve-se configurar a propriedade *Interval* com o tempo da fase, a propriedade *Reverse* com *true*, a propriedade *Loops* com 1 e o evento *OnTick* chamando o método *GameOver* ou *Win* do componente *LevelManager*.

Dependências: nenhuma.

Único: não.

Propriedades:

Nome	Tipo	Descrição
TextWriter	TextWriter	Referência para o componente do tipo TextWriter.
Format	string	Define o formato de exibição do tempo. Padrão: mm:ss
Interval	float	Intervalo de tempo em segundos entre cada loop.
OnTick	ActionSequencer	Evento disparado toda vez que a contagem do tempo completa um loop.
Loops	int	Quantidade de loops antes que o componente pare a contagem. O valor -1 significa que o componente contará para sempre.
Reverse	bool	Indica se a contagem mostrada será a inversa.

A.2.6. DuplicateArea

Descrição: cria novas instâncias de um tipo de entidade, de acordo com a área e com o intervalo de tempo especificado.

Exemplo de uso: em alguns jogos, as entidades do inimigo do jogador aparecem periodicamente para tentar destruí-lo.

Dependências: nenhuma.

Único: não.

Propriedades:

Nome	Tipo	Descrição
Entity	string	Nome do tipo de entidade da nova entidade.
Area	Rectangle	Retângulo que representa a área de criação da nova entidade. Ela aparecerá em qualquer posição dentro desse retângulo.
MinInterval	float	Intervalo mínimo de tempo (em segundos) para criação da nova entidade.
MaxInterval	float	Intervalo máximo de tempo (em segundos) para criação da nova entidade.

A.2.7. EntityMonitor

Descrição: monitora a criação e a remoção de entidades de um determinado tipo de entidade.

Exemplo de uso: no jogo Breakout, quando todos os blocos são destruídos, o jogo é pausado indicando a vitória do jogador. Para isso deve-se configurar, no evento *EntityRemoved*, a invocação do método *Win* do componente *LevelManager* com a condição da propriedade *Count* ser igual a 0.

Dependências: nenhuma.

Único: não.

Propriedades:

Nome	Tipo	Descrição
Entity	string	Nome do tipo de entidade monitorado.
EntityAdded	ActionSequencer	Evento executado toda vez que uma nova entidade do tipo especificado é criada.
EntityRemoved	ActionSequencer	Evento executado toda vez que uma nova entidade do tipo especificado é removida.
Count	int	Quantidade de entidade do tipo especificado há no cenário. Essa propriedade é somente para leitura.

A.2.8. LevelManager

Descrição: responsável pela manipulação das fases do jogo. Contém métodos para anunciar a vitória ou a derrota do jogador.

Exemplo de uso: no jogo Breakout, o jogador tem um número de chances que ele pode errar a bola. Caso as chances acabem, a derrota é anunciada. Por outro lado, caso o jogador destrua todos os blocos, a vitória é anunciada.

Dependências: nenhuma.

Único: sim.

Métodos:

Nome	Argumentos	Descrição
Pause	msg:string	Pausa a fase corrente.
GameOver	msg:string	Anuncia a derrota do jogador.

Win	msg:string	Anuncia a vitória do jogador.