FunGEn – um Motor para Jogos em Haskell

ANDRÉ WILSON BROTTO FURTADO[1]
ANDRÉ LUÍS DE MEDEIROS SANTOS[1]

[1]Universidade Federal de Pernambuco – Centro de Informática
{awbf,alms}@cin.ufpe.br

Resumo

O uso de ferramentas para tornar o desenvolvimento de jogos um processo automatizado, padronizado e rápido tem se tornado uma atividade cada vez mais relevante. Neste trabalho, mostramos que linguagens funcionais podem ser uma alternativa concreta para implementar este tipo de ferramenta, apresentando um motor para jogos desenvolvido em e para uma linguagem funcional.

Palavras-Chave: *motor para jogos, Haskell, OpenGL, HOpenGL*

1 Introdução

O processo utilizado no desenvolvimento de jogos de computador, não apenas quando estes surgiram como também ainda alguns anos depois, estava orientado por uma regra bastante específica: a eficiência do jogo deveria ser um objetivo mantido a todo custo, mesmo que isso implicasse em abrir mão de algumas boas características de projeto como modularidade, reusabilidade, extensibilidade e flexibilidade.

O motivo desta preocupação estava baseado principalmente no fato de que as máquinas da época sofriam de uma séria limitação de performance, fazendo com que mesmo os jogos mais simples exigissem muito das mesmas.

Depois de um certo tempo, a evolução tecnológica disponibilizou aos jogos uma considerável melhoria de performance, permitindo o crescimento do tamanho, da complexidade e dos recursos necessários para se desenvolver um jogo. Desse modo, as boas técnicas de programação e os conceitos pregados pela Engenharia de Software foram se tornando cada vez mais viáveis e necessários para o desenvolvimento de um jogo.

Um dos frutos dessa nova realidade foi a criação de motores para jogos (*game engines*) que proporcionam uma redução significativa no ciclo de vida de desenvolvimento deste tipo de aplicação. Os motores fizeram com que a etapa de programação do jogo fosse um processo mais automatizado, permitindo uma concentração maior nas atividades mais específicas do jogo, como seu roteiro, o comportamento de seus personagens, seus cenários e sua trilha sonora, entre outras.

Oferecendo todo um conjunto de funcionalidades ao programador, um motor permite que ele especifique, em alto nível, ***qual*** será o comportamento de seu jogo, sem necessariamente precisar saber ***como*** esse comportamento será implementado. Funcionalidades específicas para detecção de colisão ou gerenciamento do mapa de fundo são exemplos claros de como um motor pode facilitar o trabalho de um programador de jogos. Quando usado em conjunto com um *editor de cenários*, o motor se apresenta como uma ferramenta ainda mais útil e desejável.

2 Linguagens Funcionais

Historicamente, os mais populares motores para jogos foram desenvolvidos em linguagens imperativas ou orientadas a objeto, por uma questão de performance. Entretanto, o amadurecimento das linguagens funcionais, caracterizadas por seu forte embasamento teórico, alto nível de abstração, sistema de tipos forte, polimorfismo, tipos de dados algébricos e funções de alta-ordem, pode revelar uma

alternativa concreta para a implementação deste tipo de ferramenta.

O uso de linguagens funcionais se apresenta como uma opção para a implementação de soluções que levariam muito mais tempo para serem concebidas em outros tipos de linguagem. Uma significativa melhoria na performance das linguagens funcionais, observada nos últimos anos, aliada à integração deste tipo de linguagem com outras tecnologias, permitiu que o uso deste paradigma de programação fosse expandido a áreas antes restritas apenas a linguagens imperativas ou orientadas a objeto. Como se deseja mostrar neste trabalho, ferramentas para agilizar o desenvolvimento de jogos podem estar perfeitamente incluídas nesse grupo.

O principal benefício da utilização de linguagens funcionais para o desenvolvimento de uma aplicação consiste no resultado do código-fonte necessário para sua implementação: ele é elegante (alto nível), coeso e, o que é mais importante, pelo menos uma ordem de magnitude menor do que o código-fonte desenvolvido em outros tipos de linguagem para uma mesma aplicação.

Para o motor desenvolvido neste projeto, foi utilizada a linguagem de programação ***Haskell***, uma das mais populares no mundo da programação funcional. Esta linguagem mostrou-se madura o suficiente para satisfazer os requisitos necessários para a construção de um motor, além de facilitar a incorporação de características como modularidade, reusabilidade e concisão de código, propriedades inerentes às linguagens funcionais.

A biblioteca gráfica ***HOpenGL*** (*Haskell Open Graphics Library*, ou simplesmente *Haskell OpenGL*) constitui o alicerce do motor. Na verdade, a *HOpenGL* não é realmente uma biblioteca, e sim um *binding* (ligação ou ponte) em *Haskell* para a popular biblioteca gráfica *OpenGL*. Em outras palavras, a *HOpenGL* torna possível que programadores *Haskell*, desenvolvendo aplicações nesta linguagem, chamem rotinas específicas da *OpenGL* (geralmente escritas em C/C++).

Pode-se afirmar, portanto, que há no motor desenvolvido uma formidável combinação entre facilidade de programação, provida pela linguagem *Haskell*, com poder de representação gráfica, correspondente às técnicas oferecidas pela *(H)OpenGL*. Em outras palavras, o cenário criado permitiu unir a produtividade à funcionalidade.

3 Uma Visão Geral do FunGEn

O foco deste trabalho, o ***FunGEn*** (**Fun**ctional **G**ame **En**gine), é um motor para jogos bidimensionais desenvolvido através da *HOpenGL*, na linguagem funcional *Haskell*, destinado ao desenvolvimento de jogos nesta mesma linguagem. Como o projeto completo de um motor para jogos tridimensionais é um desafio muito mais complexo do que motores bidimensionais, foi considerado o desenvolvimento inicial de uma solução 2D, para posteriormente checar a viabilidade de alternativas 3D.

Entre algumas das funcionalidades e facilidades oferecidas pelo ***FunGEn*** a um programador de jogos, inclui-se o gerenciamento completo dos objetos do jogo (inicialização, atualização, localização, remoção, agrupamento e renderização). A renderização é um dos pontos a ser destacado neste tópico: um objeto do jogo pode estar associado a uma simples primitiva gráfica (como círculos, quadrados ou triângulos) ou ser representado por *sprites* animadas, cujas figuras (quadros) são carregados de arquivos gráficos. O intervalo de tempo entre cada quadro da animação pode ser definido pelo programador, ao qual também são oferecidas algumas interessantes opções, como *stretch* (distorção) e auto-espelhamento, para lidar de uma forma mais natural com o desenho seus objetos.

O código abaixo mostra como associar uma *sprite* a um determinado objeto. Em primeiro lugar, é definido como carregar o arquivo gráfico (considerado pelo motor um `GameBitmap`):

```
rosquinhas = GameBitmap {
   bitmapName = "donuts.bmp",
   unitSize = (64,64),
   dimensions = (2,2),
   transparent = Just [magenta]}
```

Em um `GameBitmap`, é especificado o nome do arquivo gráfico, o tamanho dos *tiles* que ele contém, a quantidade horizontal e vertical de *tiles* no arquivo e, por fim, as "cores transparentes" do mesmo. O motor, partindo dessas especificações, constrói uma lista de texturas e as disponibiliza ao programador.

O próximo passo consiste em determinar como se dará a ordem e tempo de animação de cada textura criada. Isso é especificado através de uma lista de pares (textura,tempo), que o motor trata como uma `Sequence`:

```
objSequence=[(0,10),(1,20),(2,10),(3,20)]
```

Finalmente, cria-se o *sprite* propriamente dito, informando seu tipo (*sprite* de texturas, nesse caso), seu `GameBitmap` e sua `Sequence`:

```
sprite = Textured rosquinhas
         objSequence True
```

O booleano especifica se a seqüência da animação será cíclica ou não (chamando o evento `animationEnded` do objeto, nesse último caso). Através deste exemplo, pode-se observar como o programador, com o uso de poucas linhas de código e trabalhando em alto nível, pode especificar a maneira pela qual os objetos do seu jogo são desenhados.

A detecção de colisão entre objetos é outra funcionalidade oferecida pelo motor ao programador, que desta forma não precisa se deter em cálculos envolvendo posição, tamanho ou velocidade dos objetos para checar se eles se chocaram ou não.

O gerenciamento do mapa de fundo é mais um tópico que visa a simplificação do trabalho do programador. O mapa pode ser especificado por cores, texturas simples ou até construções mais complexas, como um mapa de *tiles* em que informações específicas podem estar associadas a cada *tile* (como a dificuldade de transposição de um terreno, por exemplo). Algumas funcionalidades em torno do mapa de fundo, como auto-rolamento (*scrolling*), também são oferecidas ao programador.

O controle das entradas fornecidas pelo jogador, provenientes do mouse ou teclado, é realizado de modo bastante intuitivo pelo ***FunGEn***: cada tecla (ou botão do mouse) é associada a um evento e a uma ação. O evento indica um dos três estados possíveis para a tecla (ou botão): pressionada, liberada ou mantida continuamente pressionada. A ação, por sua vez, é definida pelo programador e indica o que deve acontecer no jogo em resposta à tecla e evento correspondentes.

Algumas funcionalidades adicionais são oferecidas para o gerenciamento de entradas provenientes do mouse: o programador pode questionar se o clique (ou liberação) do botão ocorreu dentro ou fora de alguma região específica da tela. Esta região, definida em alto nível pelo programador, pode ter o formato de um círculo ou de um polígono qualquer.

Por fim, é permitida ao programador a configuração de alguns comportamentos mais gerais do jogo, como a determinação de sua taxa de execução (em milisegundos), a configuração dos parâmetros de sua janela (ou tela cheia) e a definição do fluxo de eventos básico que ocorrerá a cada ciclo de jogo.

4 Analisando Alguns Resultados

O primeiro benefício observado na programação de jogos utilizando o ***FunGEn*** é uma redução significativa do número de linhas de código. Um jogo simples, como o popular *Pong*, foi desenvolvido com o auxílio do motor em apenas 50 linhas. Sem o motor, este número salta para 200 linhas. Se, além disso, uma linguagem funcional não fosse utilizada, este número aumentaria ainda mais. (A contagem do número de linhas de código foi realizada, obviamente, sem incluir linhas em branco ou comentários. Além disso, não houve a tentativa de induzir o código feito em *Haskell* a ser o menor possível, preservando seu estilo e legibilidade.)

O jogo *Donuts*, exemplo que acompanha a biblioteca gráfica *DirectX*, da *Microsoft*, também foi desenvolvido em *Haskell* com e sem o auxílio do ***FunGEn***. A versão original do jogo, feita em *C++*, utiliza 2000 linhas de código. A versão em Haskell sem o ***FunGEn*** foi concluída em 810 linhas. Finalmente, utilizando-se o motor, este número caiu para 325. Além do *Donuts*, o jogo *Worms* ou *Snake* (bastante popular em telefones celulares) foi desenvolvido

com o auxílio do ***FunGEn*** e comparado a uma versão já existente, implementada na linguagem funcional *Clean*, através da biblioteca para jogos *CGL*. O código em Haskell demandou 350 linhas de código, ao passo que o código em *Clean* necessitou de 1500.

Por ser baseado numa biblioteca gráfica independente de plataforma (*OpenGL*), o próprio ***FunGEn*** também é independente: o mesmo código-fonte poderia ser compilado nas plataformas *Windows*, *UNIX* ou *MacOS*, por exemplo. Além disso, é interessante ressaltar que jogos implementados através do ***FunGEn*** herdam da *OpenGL* algumas importantes características como robustez e estabilidade.

O fato do processamento gráfico ser feito em outra linguagem (*C++*), e não em *Haskell*, devido ao *binding* provido pela *HOpenGL*, permite um bom desempenho dos jogos desenvolvidos. Alguns testes foram realizados em relação a essa questão: a taxa de quadros por segundo do jogo *Donuts* foi medida tanto na sua versão original como na versão que utiliza o ***FunGEn***. Em um Pentium III de 500 MHz e 128 MB de RAM, a versão em *C++* apresentou uma taxa média de 59 quadros por segundo. Na versão em *Haskell* essa taxa caiu para 42 quadros por segundo, perfeitamente satisfatória para este tipo de jogo. Em um Pentium III de 900 MHz e 256 MB de RAM, as duas versões apresentaram o mesmo comportamento: uma taxa de 68 quadros por segundo.

5 Conclusão

O uso do ***FunGEn*** aparenta ser uma boa escolha para programadores que desejam desenvolver jogos com alta produtividade e performance adequada. Obviamente, jogos implementados com o auxílio de motores desenvolvidos em outras linguagens tendem a ser mais eficientes, mas o programador pode sofrer uma séria perda em flexibilidade e extensibilidade se o projeto do jogo se tornar muito grande.

Apesar do ***FunGEn*** não ser um motor para jogos completo, sua extensibilidade encoraja a adição de novas funcionalidades sem grandes modificações. Além disso, soluções para muitas das funcionalidades que o ***FunGEn*** ainda não oferece, como jogos em redes, já se apresentam bastante maduras em *Haskell*.

Finalmente, acredita-se que características desejáveis para um software de qualidade podem ser atingidas com o uso do ***FunGEn***, visto que ele produz jogos com um código-fonte extensível, modularizado e reusável. Estas características também poderiam ser obtidas em uma abordagem orientada a objeto, mas a elegância, coesão e alto-nível do código se apresentam mais evidentes no paradigma funcional. Isto se aplica também ao código do próprio motor, facilitando sua manutenção e documentação.

A decisão de tornar o ***FunGEn*** uma ferramenta de código aberto, por fim, encoraja seu contínuo desenvolvimento. Os autores deste projeto esperam que o motor contribua não apenas para que linguagens funcionais sejam encaradas pela comunidade de programadores como uma alternativa concreta para a implementação de jogos, como também para tornar o desenvolvimento de jogos um processo mais automatizado, eficiente, produtivo e menos sujeito a erros.

6 Referências

1. FunGEn: A game Engine for Haskell, http://www.cin.ufpe.br/~haskell/fungen (11/08/02)
2. HOpenGL - An OpenGL/GLUT binding for Haskell, http://haskell.org/HOpenGL (14/07/02)
3. The Clean Programming Language, http://www.cs.kun.nl/~clean/ (10/08/02)
4. Madeira, C. A. G. FORGE V8: um *framework* para o desenvolvimento de jogos de computador e aplicações multimedia. Tese de Mestrado, Universidade Federal de Pernambuco, pp. 1-5
5. Pessoa, C. A. P. wGEM: um *Framework* de Desenvolvimento de Jogos para Dispositivos Móveis. Tese de Mestrado, Universidade Federal de Pernambuco, pp. 1-13
6. MSDN Library Archive, Space Donuts Sample, http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/ddraw7/directdraw7/ddsamp_57zr.asp (14/08/02)
7. Wiering, M. Clean Game Library, http://tilestudio.sourceforge.net/cgl/thesis/ (23/07/02)
8. Haskell , http://www.haskell.org (11/08/02)