UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SYMBG(R)AF – SYMBIAN GAMES FRAMEWORK

por

Tiago Guedes Ferreira Barros

Trabalho de Graduação

Recife, agosto de 2003.

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TIAGO GUEDES FERREIRA BARROS

SYMBG(R)AF – SYMBIAN GAMES FRAMEWORK

Este trabalho foi submetido à Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para conclusão da disciplina de Trabalho de Graduação em Engenharia de Software e obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. André Luís M. Santos

Co-Orientador: Prof. Dr. Geber Lisboa Ramalho

Recife, agosto de 2003.

Dedicatória

Dedico este trabalho aos meus pais. E isto{e eles} é{são} tudo.

Agradecimentos

Agradeço a Deus e aos meus pais, pela vida.

Gostaria de agradecer imensamente a todos que contribuíram para este trabalho. Agradeço à minha família, pelo apoio, compreensão e incentivo, não só durante o curso, mas em todos os momentos importantes de minha vida.

Agradeço também aos meus orientadores e amigos André Santos e Geber Ramalho, e a todos os Mestres que contribuíram para a minha formação acadêmica, pessoal e profissional.

Não poderia também deixar de agradecer a Paulo Abadie (Paulinho), meu primo e amigo que ajudou bastante na revisão deste texto, e a Ivo Frazão, que desenvolveu jogos e um *framework* em BREW e me ajudou bastante neste trabalho.

Gostaria também de agradecer aos amigos Brainer, Ângelo, Pedro Aléssio (Pedrão), Cláudio Takahasi e Talita pelo seu apoio e incentivo.

Por fim agradeço a todos os meus amigos, que contribuíram de todas as formas para a realização deste trabalho.

Que a força do medo que tenho
não me impeça de ver o que anseio...
Que a arte me aponte uma resposta
mesmo que ela não saiba
e que ninguém a tente complicar
pois é preciso simplicidade pra fazê-la florescer.
{Porque metade de mim é a criação
e na outra habita o criador.}

(adaptado de Metade, de Oswaldo Montenegro.
...Porque na natureza, tudo se transforma.)

Sumário

Lista de Figuras

Resumo

O desenvolvimento de jogos para celulares vem alcançando um espaço considerável na indústria[35]. Atualmente, vários modelos de celulares permitem que se instalem outros programas, o que impulsionou bastante o desenvolvimento de jogos nestes dispositivos. Apesar disto, o desenvolvimento de jogos é uma tarefa complexa que envolve diversas áreas do conhecimento e que em geral apresenta vários desafios. De forma a reduzir os riscos e controlar a complexidade destes desafios, são utilizadas diversas ferramentas durante o desenvolvimento do jogo. Um dos tipos de ferramentas, fundamental atualmente, são os *frameworks*. Um *framework* é um conjunto de componentes de software organizado e projetado para a construção de aplicações de um determinado domínio específico, implementando as operações comuns a estas aplicações. Com isto, é possível diminuir o *time-to-market* para desenvolver jogos com menores tempo e custo. Este trabalho visa a construção de um *framework* para o auxílio ao desenvolvimento de jogos para o sistema operacional Symbian [36], de forma a permitir que o desenvolvimento dos jogos seja mais rápido e otimizado.

Palavras-chave: Jogos eletrônicos, *Framework*, Symbian, Telefones Celulares.

Introdução

Com o crescimento da tecnologia de telefonia celular, estes dispositivos, que antes eram centrados na transmissão de voz, passaram cada vez mais a focar seus objetivos em processamento e transmissão de dados. Rodar aplicações, tirar fotos, receber e-mails ou navegar na *web* estão se tornando atividades comuns aos celulares atuais. Devido ao seu grande poder de conectividade e a sua mobilidade, os celulares também se tornaram uma potencial plataforma para o desenvolvimento de jogos.

A ubiqüidade computacional proporcionada por estes dispositivos realmente muda significativamente a forma de jogar. Poder jogar a qualquer hora, em qualquer lugar, e interagir com adversários humanos aumenta bastante o interesse e a diversão dos jogos eletrônicos. Imagine a possibiliadade de se estar jogando RPG com uma pessoa você não conhece, mas que entrou no jogo devido à proximidade dos dispositivos, e agora faz parte do jogo e pode interagir com você. Este tipo de interação agora é possível com estes novos dispositivos.

Com isto, os dispositivos móveis vem alcançando um espaço considerável como plataforma de desenvolvimento de jogos. A NOKIA está apostando em um celular completamente voltado para jogos, o N-Gage[18], que roda o sistema operacional Symbian. Portanto, podemos perceber a tecnologia está evoluindo cada vez mais para acompanhar esta nova demanda dos jogadores.

Entretanto, desenvolver um jogo não é uma tarefa trivial, pois envolve o conhecimento profundo de diversas áreas da computação. Em ambientes restritos, com limitações de memória e processamento, esta tarefa é ainda mais difícil. No entanto, existem várias operações que fazem parte do desenvolvimento, como detecção de colisão, animação de sprites, renderização, que são necessárias a todo jogo.

A partir da identificação destes elementos comuns, podemos unificá-los em um componente de software, que seja genérico o suficiente para permitir sua utilização pelos mais variados tipos de jogos, mas ao mesmo tempo seja extremamente otimizado para garantir uma boa performance do jogo. Esse é o papel de um *framework*.

O objetivo deste projeto é elicitar os requisitos de um *framework* de jogos para o sistema operacional Symbian (Symbian OS), visando desenvolver um protótipo do mesmo. O desenvolvimento de aplicações para Symbian deve respeitar um conjunto de práticas (*code idioms*) para suprir a falta de um sistema de gerenciamento de exceções, limitação de memória e

poder de processamento. Além disto, existe um *framework* para o desenvolvimento de aplicações para Symbian, o que torna ainda mais desafiadora a tarefa de implementar um *framework* de jogos: deve-se implementá-lo respeitando todas as restrições da arquitetura e os padrões de projeto impostos pelo *framework* de aplicações e fazer com que as funcionalidades básicas de um *framework* de jogos sejam implementadas.

No capítulo dois, é visto o contexto das tecnologias envolvidas no desenvolvimento de jogos para celulares. No capítulo três é explicado o processo de desenvolvimento de jogos, bem como a utilização dos frameworks para aumentar a efetividade do desenvolvimento. O capítulo quatro trata do desenvolvimento de aplicações para o Symbian OS, bem como todas as retrições que devem ser seguidas e o suporte de Symbian ao desenvolvimento de jogos. O capítulo cinco trara do SymbG(r)aF, descrevendo a metodologia utilizada para o seu desenvolvimento, os seus requisitos, sua arquitetura e os detalhes da sua implementação. No capítulo seis são apresentadas as conclusões do desenvolvimento deste trabalho, as principais dificuldades encontradas bem como sugestões para a extensão deste trabalho.

Estado da arte

O desenvolvimento de jogos para celulares é bastante recente, surgindo junto com as tecnologias celulares da segunda geração: TDMA e GSM. Antes disto, os celulares eram utilizados apenas para comunicação por voz. Com as tecnologias digitais, o software dos celulares evoluiu bastante e passaram-se a criar várias aplicações para estes dispositivos, e com elas, inevitavelmente, vieram os jogos.

Inicialmente, os jogos eram criados pelos fabricantes dos celulares e cada dispositivo saia de fábrica com um conjunto de jogos predefinidos. Os primeiros jogos criados foram Snakes, Memória, etc. Depois, passou-se a desenvolver jogos em cima de SMS, e isso permitiu pela primeira vez que houvessem jogos *multiplayer*. Com GPRS, tornou-se possível a comunicação de dados entre os celulares e isto foi utilizado para acesso à internet. Então, não demorou para isto ser utilizado pelos jogos também.

Com a evolução tecnológica, tornou-se possível instalar aplicações nos celulares. Isto mudou radicalmente o futuro dos jogos nestes dispositivos, pois agora não era só o fabricante de celulares que desenvolvia jogos, mas outros programadores e empresas de desenvolvimento de jogos passaram a investir nesta nova "plataforma". A seguir serão apresentadas as principais formas de desenvolvimento de aplicações para celulares.

2.1. J2ME

A primeira iniciativa no sentido de desenvolvimento de aplicações para celulares foi *Java 2 Micro Edition – J2ME*[1], uma versão de JAVA para dispositivos com limitações de memória e processamento, como celulares, PDA´s e sistemas embarcados em geral. Para rodar nestes dispositivos, a liguagem precisaria ser bastante simplificada e otimizada, removendo os componentes que não estão presentes ou não são utilizados nestes dispositivos. Sua arquitetura[2] foi dividia em três camadas: a máquina virtual propriamente dita, os perfis e as configurações, conforme pode ser visualizado na Figura 1.

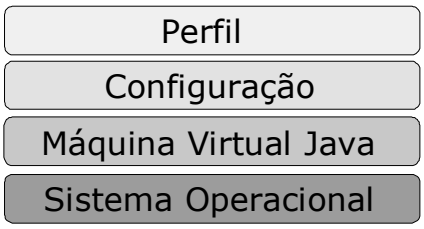

Figura 1: Camadas da arquitetura de J2ME

A máquina virtual de J2ME é a *KVM – Kilobyte Virtual Machine*, e é a base da arquitetura, devendo ser implementada para cada dispositivo, criando uma camada de software que abstrai o sistema operacional e garante a portabilidade.

Sobre a máquina virtual são definidas as configurações, que são um conjunto mínimo de classes que provêem as funcionalidades básicas para as categorias de dispositivos. Atualmente existem 2 configurações:

- *CLDC – Connected Limited Device Configuration*, projetada para dispositivos com conecções de rede intermitentes, processadores mais lentos e memória limitada, como os telefones celulares e alguns PDA's;
- *CDC – Connected Device Configuration*, configuração para dispositivos com mais memória e poder de processamento, e uma conexão com uma maior largura de banda, como set-top boxes, sistemas de telemetria de veículos e PDA's mais poderosos.

Sobre a configuração, são definidos os perfis, API's de alto-nível que disponibiliza um ambiente de execução completo e direcionado para os dispositivos alvo. O perfil utilizado para telefones celulares é o *MIDP – Mobile Information Device Profile*. Este perfil provê as principais fuincionalidades requeridas pelas aplicações para celulares, como interface com o usuário, conectividade, manipulação de arquivos e gerenciamento da aplicação em geral.

A versão 1.0 do MIDP não foi projetada visando o desenvolvimento de jogos, por isto este perfil apresenta uma série de restrições que precisam ser contornadas pelo desenvolvedor, como falta de suporte a transformações geométricas, som e gerenciamento dos objetos do jogo. Recentemente foi lançada a versão 2.0 do MIDP. Esta versão provê uma API para o desenvolvimento de jogos (Game API) que estende as funcionalidades gráficas do MIDP e

implementa algumas funcionalidades como mapa de *tiles*[1], *layers* e acesso direto aos pixels da imagem.

2.2. BREW

Seguindo este caminho, a Qualcomm desenvolveu o *BREW – Binary Runtime Environment for Wireless* (Ambiente de desenvolvimento Binário para Dispositivos Móveis)[3]. Neste ambiente, as aplicações são desenvolvidas em C/C++, compiladas e rodam em um chip separado do processador principal do dispositivo, o que possibilita uma performance melhor em relação a J2ME.

BREW oferece uma API para acesso às várias funcionalidades dos dispositivos, incluíndo a tela, rede, teclado, som, etc. O uso desta API se dá no acesso às interfaces (em vez de objetos), que possuem um conjunto de funções associadas para acessar suas funcionalidades. Estas interfaces possuem um identificador único e implementam o padrão de projeto *reference counting*[4], para gerenciar a alocação e desalocação de memória. A programação em BREW é orientada a eventos e feita de forma procedural, onde os objetos são passados como parâmetros para as funções da interface.

BREW possui um rasoável suporte para o desenvolvimento de jogos. É possível desenhar primitivas gráficas e imagens, mas não é permitido acessar os pixels da imagem nem realizar transformações como rotação e escala. Também dá suporte a som, criação de temporizadores, conectividade e persistencia de dados.

Além disto, a Qualcomm também oferece um modelo de negócio que facilita a entrega e venda das aplicações: os desenvolvedores criam as aplicações, submetem a Qualcomm para que sejam testadas e depois enviam às operadoras para que sejam comercializadas. A desvantagem deste sistema é que os desenvolvedores ficam presos a este modelo, dependendo da Qualcomm e das operadoras para fazer a distribuição dos seus jogos. Sem falar que para desenvolver em BREW, é necessário ser um desenvolvedor registrado, devendo pagar uma série de taxas para registro, aquisição do kit de desenvolvimento e etc.

---

[1] Cenário composto por um conjunto de figuras repetidas, os *tiles,* o que ajuda em várias operações do jogo.

2.3. Symbian OS

Estabelecida como uma empresa independente em 1998, Symbian surgiu como um consórcio formado pelas maiores empresas da indústria de comunicação sem fio: Nokia, Panasonic, Motorola, Psion, Samsung Electronics, Siemens e Sony Ericsson.

A intenção destas empresas foi desenvolver um sistema operacional para dispositivos móveis que seja aberto e padrão, a partir do sistema operacional EPOC, para PDA's: assim surgiu o Symbian OS. Devido a estas características, tornou-se possível o desenvolvimento de aplicações, por parte de terceiros, para os dispositivos que possuíssem este sistema operacional. Estas aplicações são desenvolvidas em código nativo, para ser executado pelo processador do dispositivo, diferentemente de J2ME e BREW.

Também é importante notar que, por ser um sistema operacional, o desenvolvimento de aplicações para Symbian difere bastante de J2ME e BREW. Inclusive já existe uma KVM que roda em cima de Symbian, possibilitando o desenvolvimento de aplicações em J2ME para este sistema operacional.

A API do Symbian OS fornece um *framework* de desenvolvimento de aplicações. Desta forma, é necessário apenas herdar algumas classes reimplementado alguns métodos. Entretanto, para satisfazer as restrições do desenvolvimento para sistemas embutidos, deve-se implementar uma série de práticas, definidas como *Symbian Coding Idioms*[1], que serão discutidas posteriormente.

2.4. Comparação entre as plataformas

O documento *J2ME & Symbian OS: A Platform Comparison*[5] faz uma comparação entre o Symbian OS e J2ME. Apesar de serem esferas diferentes, pois J2ME é executada em uma máquia virtual que roda sobre um sistema operacional e Symbian é um sistema operacional, esta comparação foi feita sob o ponto de vista do desenvolvedor, visando verificar o que cada plataforma oferece para o desenvolvimento de jogos. Ivo Frazão, no desenvolvimento de um

[1] *Code Idiom*: Padrão de projeto (*design pattern*) específico para uma linguagem ou plataforma de desenvolvimento.

framework de jogos para BREW[6], complementa esta comparação introduzindo as características de BREW. Ele define uma tabela comparativa a qual foi reproduzida abaixo:

	J2ME	Symbiam OS	BREW
Tamanho permitido para as aplicações	Poucos kilobytes (em geral, 64KB)	Alguns megabytes	Alguns kylobytes, limitado apenas pela memória do celular
Penetração no Mercado	Grande e em crescimento	Pequeno e em crescimento	Grande nos Estados Unidos e em crescimento
Instalação por download pela rede telefônica	Sim	Possível, mas evitado por causa do tamanho das aplicações	Sim
Executa como código-nativo	Não	Sim	Sim
Linguagem de programação principal suportada	Java	C++	C/C++
Suporte a outras linguagens de programação	Não	Sim, inclusive Java	Sim, inclusive Java
Comunicação através de *sockets*	Sim, mas não é implementado em todos aparelhos	Sim	Sim
Conexão via Infravermelho ou Bluetooth	Não	Sim	Não
Animação 2D	Sim	Sim	Sim
Animação 3D	Não	Sim	Não
Exibe Vídeos	Não	Sim	Não
Suporte a MIDI e WAV	Sim	Sim	Sim
Acesso a SMS	Não	Sim	Sim
Acesso à agenda e calendário do aparelho	Não	Sim	Sim
Efetua chamadas telefônicas	Não	Sim	Sim
API específica para Jogos	Sim	Não	Não

Tabela 1: Comparação entre J2ME, Symbian e BREW

Desenvolvimento de jogos: o uso de *frameworks*

Desenvolver um jogo é uma tarefa extremamente complexa, pois envolve diversas áreas do conhecimento e que, em geral, apresenta grandes desafios. A concepção de um jogo inicia-se com a criação do ***Game Design***[7, 8], documento que irá descrever toda a estrutura e componentes do jogo (i.e. a história do jogo, roteiro, personagens com suas habilidades e características, armas, itens, níveis, objetivos, adversários, etc) como também as características inerentes à tecnologia e à plataforma em que o jogo será desenvolvido.

Portanto, podemos concluir que o desenvolvimento de um jogo consiste em implementar o seu *game design* como um programa de computador. Entretanto, além das características presentes no *game design*, o desenvolvedor de jogos deve implementar também todos os componentes de software responsáveis por fazer o jogo funcionar. Isto inclui: gerenciamento das entradas do usuário, renderização da tela, modelagem e animação dos objetos do jogo, temporização, etc. o que consome uma parcela considerável do tempo de implementação de um jogo.

Estes componentes de software são comuns a todos os jogos e não fazem parte do *game design*, entretanto precisam ser implementados. Com o resultado da utilização cada vez maior da engenharia de software no desenvolvimento de jogos, estes componentes comuns passaram a ser reutilizados entre o desenvolvimento dos jogos e deram origem aos *frameworks*.

Um *framework* é um conjunto de componentes de software organizado e projetado para a construção de aplicações de um determinado domínio específico. É importante notar que um *framework* de jogos não é um conjunto de funções[1], ou uma API, relativas a um determinado objetivo como, por exemplo, mostrar imagens na tela. Ele também não é um produto pronto. O *framework* deve ser integrado junto com a implementação do *game design*, para formar o jogo completo. Então, podemos perceber que, o objetivo do *framework* é fazer com que os seus usuários, no caso os desenvolvedores de jogos, possam implementar "apenas" o *game design* do seu jogo.

Entendido o que é um *framework* e qual o objetivo ao implementar um, veremos a seguir quais as maiores dificuldades e riscos e também a utilização dos *frameworks* de jogos nos dispositivos móveis.

---

[1] Este é o conceito das bibliotecas de ligação dinâmica, ou DLL.

3.1. Riscos e Dificuldades no desenvolvimento de *frameworks*

O problema da construção de sistemas complexos como *frameworks* e motores[1] de jogos (do inglês *Game Engine*) é o controle dos riscos e dos problemas de desenvolvimento. Neste caso, existem diversos aspectos que devem ser considerados e não é raro que o projeto, por falta de uma boa estrutura e planejamento, fracasse como vítima de sua própria complexidade.

A confirmação disso pode ser obtida pelo infeliz resultado do jogo *Golgotha*[9], que era um projeto comercial para o desenvolvimento de um jogo com motor proprietário 3D. Neste projeto, doze pessoas estiveram envolvidas durante dois anos de trabalho, e quinhentos mil dólares foram gastos, mas o projeto foi encerrado incompleto devido a sua não conclusão após o período preestabelecido, sendo então o código fonte liberado para o público[28]. Isso demonstra o enorme esforço que deve ser empregado para tal tarefa, mesmo que sejam utilizados extensamente os princípios de engenharia de software. Atualmente um grupo de desenvolvedores de jogos está tentando retomar o projeto, com o *Golgotha Forever*[10].

Portanto, é importante, ao se projetar um *framework* ou motor de jogos, que se tenha em mente seus objetivos e propósitos, para evitar estes tipos de prejuízos.

3.2. *Frameworks* para dispositivos móveis

Desenvolver jogos para dispositivos móveis é uma tarefa ainda mais desafiadora do que desenvolver jogos para PC´s ou consoles. Além da dificuldade de desenvolver o jogo em si, o desenvolvedor deve se ater a vários problemas inerentes a estes dispositivos, como limitação de memória, recursos e processamento, tamanho da tela e entrada (teclado) não apropriada. Desta forma, o uso de *frameworks* de jogos vem para capturar toda a *expertise* do desenvolvedor

[1] Os motores diferem dos *frameworks* pois já são aplicações quase completas, para um domínio bem mais específico do que o de um *framework*. Os motores de jogos já rodam sozinhos, bastando ao desenvolvedor acrescentar as regras do jogo.

naquela plataforma, podendo prover um conjunto de componentes projetados e otimizados para estes dispositivos. Além disto, a utilização de *frameworks* visa reusabilidade de código, dos elementos inerentes ao jogo e das características da plataforma.

O desenvolvimento de frameworks de jogos para celulares está começando a crescer. Para o Symbian OS, dois grandes projetos se destacam: o X-Forge[11] e o Mophun[12].

O X-Forge é um motor completo para desenvolvimento de jogos 3D em diversas plataformas, como: Symbian OS™, Palm OS, Microsoft Windows® CE.NET, Microsoft Smartphone, Pocket PC, Mobile Linux. É formado por uma camada que abstrai o sistema operacional, o *X-Forge Core*, provendo acesso aos diversos componentes do dispositivo, como gráficos, som, rede, teclado, etc. e o *X-Forge Game Engine*, que provê os elementos necessários a um jogo, como IA, detecção de colisões, modelagem física, áudio 3D, multiplayer, Gráficos, etc.

O Mophun propõe a construção de uma máquina virtual sobre o sistema operacional, a qual oferece uma API para o desenvolvimento de jogos 2D e 3D. Desta forma, o desenvoledor cria Gamelets[1] (em C/C++ ou Assembly) que são enviados via SMS para o telefone com o Mophun instalado. Então o usuário pode rodar os jogos a partir do Mophun. Portanto, funciona de forma semelhante a BREW, mas sem ter um chip dedicado para rodá-lo.

Além disto, o Centro de Informática da UFPE também produziu alguns *frameworks* para celulares além do SymbG(r)aF: o wGEM[13], *framework* de jogos em J2ME, desenvolvido a partir de uma tese de mestrado e um *framework* de jogos em BREW[6], desenvolvido a título de trabalho de graduação.

Entretanto, o uso de *frameworks* não traz só vantagens. A seguir, será feita uma análise do uso de *frameworks* para dispositivos móveis, listando suas desvantagens e principais vantagens.

3.2.1. Desvantagens

Uma das grandes desvantagens no uso dos *frameworks* ocorre devido ao fato de que os jogos podem ser completamente diferentes uns dos outros. Desta forma, utilizar uma arquitetura muito geral, que busque atender a todos os tipos de jogos, fatalmente irá causar um *overhead* que poderá inviabilizar a construção de jogos.

---

[1] Termo proveniente do Applet de JAVA, para identificar um Applet de jogo.

Outro problema ocorre justamente na situação contrária: uma arquitetura extremamente especialista servirá apenas para um determinado tipo de jogo, o que foge ao propósito do *framework* de ser reusável.

Portanto, podemos perceber que ao se utilizar um *framework*, um dos primeiros problemas é a sua escolha.

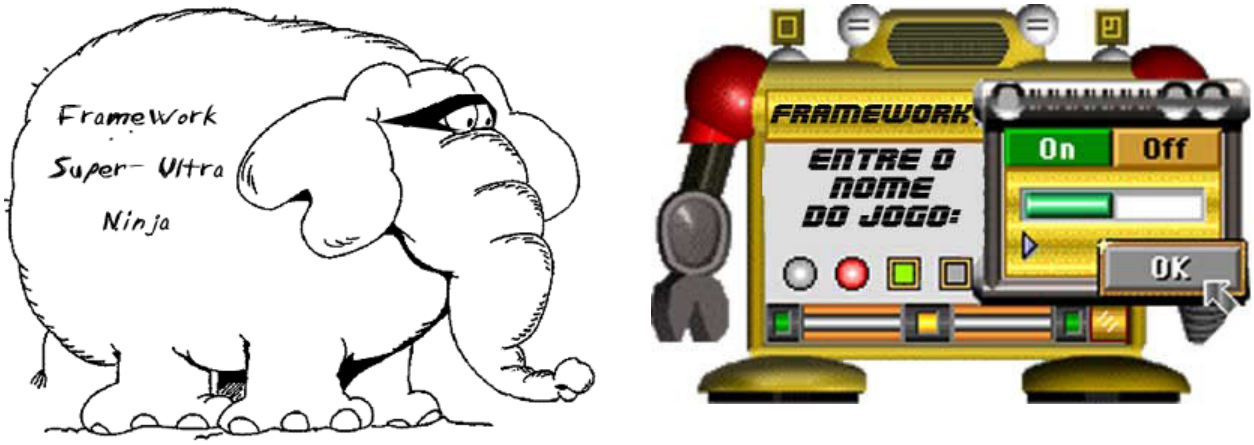

Figura 2: *Frameworks* muito genéricos ou muito específicos

Além disto, ao se utilizar um *framework*, estamos inserindo mais uma camada de abstração entre o código do jogo e a plataforma, de modo que isto gera um *overhead*, por mais que ele seja otimizado. Outro fator que influencia no uso de *frameworks* é o seu custo. Devido à sua complexidade de implementação, podem custar um valor considerável. Além do custo financeiro, dependendo da complexidade do *framework*, um custo adicional em tempo de desenvolvimento e treinamento pode ser necessário.

Apesar destas desvantagens, a utilização de *frameworks* no desenvolvimento de jogos está cada vez mais intensa, pois, a maioria destas desvantagens pode ser contornada. A seguir veremos as principais vantagens do seu uso.

3.2.2. Vantagens

Uma das principais vantagens no uso de *frameworks* é quanto ao tempo de desenvolvimento. Uma vez que os componentes comuns de um jogo já estão implementados, o seu desenvolvimento é mais rápido e pode ser focado no *game design*, justamente a parte da implementação que é diferente entre os jogos.

Seguindo este pensamento, devemos notar também que o *framework* visa suprir deficiências da API da plataforma de desenvolvimento. Operações não implementadas na API da plataforma devem ser implementadas pelo *framework*, possibilitando ao desenvolvedor utilizá-las no seu jogo.

Os *frameworks* também são especialmente bons caso sejam produzidos vários jogos similares. Isto faz com que o desenvolvedor torne-se familiar ao tipo do jogo e ao *framework* aumentando consideravelmente a sua produtividade.

Após verificar a necessidade de uso dos *frameworks*, será visto no próximo capítulo quais as restrições e diferenças no desenvolvimento de aplicações para o Symbian OS.

O Symbian OS

4.1. Desenvolvimento de aplicações para Symbian OS

O sistema operacional Symbian permite o desenvolvimento de aplicações em C++, as quais são instaladas no dispositivo em forma de uma biblioteca de ligação dinâmica, ou DLL. Isto é necessário para que não seja preciso reiniciar o dispositivo toda vez que uma aplicação for instalada. Desta forma, as "aplicações DLL" devem prover uma função de acesso externo que serve justamente para instanciar a aplicação, permitindo que a mesma seja executada.

O Symbian OS pode ter vários tipos de interface gráfica com o usuário, as mais comuns são a UIQ e a Series 60, discutidas a seguir:

- UIQ: interface para dispositivos que possuem uma tela com dimensões entre 4x6 cm a 6x8 cm e tela sensível ao toque (*touch screen*). O Sony Ericsson P800[14] é um exemplo de dispositivo com esta interface, como ilustrado na Figura 3. Maiores informações podem ser encontradas em [15].

- Series 60: de acordo com [16], a plataforma "Series 60" é um padrão de interface com o usuário criada pela Nokia, mas aberta a todos os dispositivos que utilizem o Symbian OS. É voltada para *smartphones*, que possuem uma tela de 176x208 Pixels. O Nokia 3650[17] e o Nokia N-Gage[18] (mostrado na Figura 3) são exemplos de telefones com esta interface.

Figura 3: O Sony Ericsson P800 (UIQ) e o Nokia N-Gage (Series 60)

Estes dois modelos de interface com o usuário estendem de um mesmo *framework* de controle e interface com o usuário, o UIKON (User Interface and Control Framework), que é a base do *framework* para desenvolvimento de aplicações[19] para o Symbian OS. Para cada tipo de interface com o usuário é criada uma extensão do UIKON: a plataforma Series 60 utiliza o AVKON, enquanto o UIQ utiliza o QIKON. Podemos encontrar maiores detalhes sobre as diferenças entre estas extensões e como fazer o *porting* de uma plataforma para a outra em [20].

Para utilizar o *framework* de aplicações Symbian, o desenvolvedor deve implementar as seguintes classes[21]:

(i) Classe da Aplicação:

O ponto de entrada do programa. Esta classe provê a inicialização da aplicação, sendo a primeira classe a ser instanciada pelo *framework* e é responsável por instanciar a classe de documento.

(ii) Classe de Documento:

Responsável por gerenciar a parte persistente (arquivos) da aplicação. É possível definir ou associar quais os tipos de arquivos (extensões) são suportados e gerenciados por uma determinada aplicação. É importante notar que nem todas as aplicações possuem documentos associados. No entanto esta classe deve existir, pois instancia a classe de interface com o usuário.

(iii) Classe de Interface com o Usuário (UI):

Esta classe provê o gerenciamento da interface com o usuário, realizando o tratamento de eventos, tais como: pressionamento de teclas, posicionamento do ponteiro (no caso de *touch screens*), abertura/fechamento de arquivos, etc. Também é responsável pela criação da(s) classe(s) de visualização.

(iv) Classe de Visualização (View):

Esta classe representa tudo o que está sendo mostrado na tela, recebendo da classe de UI os comandos para desenhar e atualizá-la. É importante salientar que uma aplicação pode ter várias *views*, sendo a classe de UI responsável por gerenciá-las e definir qual *view* estará ativa.

(v) Motor ou núcleo da aplicação (ApplicationEngine):

Esta classe, implementada pelo desenvolvedor, é o que contém toda a lógica da aplicação, implementa seus algoritmos e provê os pontos de acesso para a classe de UI.

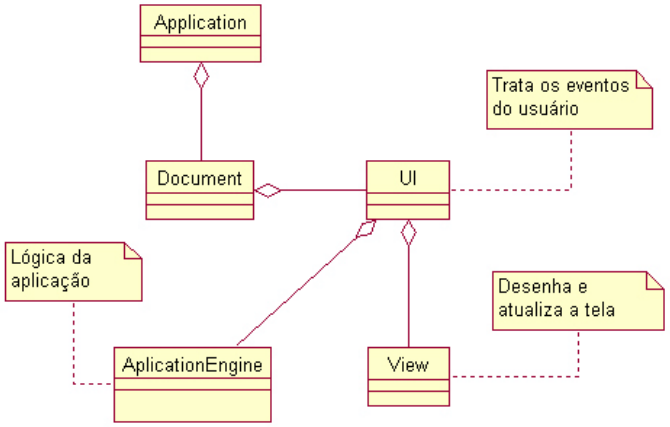

Figura 4: diagrama de classes do UIKON framework.

As classes de aplicação, documento, UI e *view* que devem ser desenvolvidas, herdam de classes correspondentes do UIKON (ou de uma de suas extensões: AVKON para Series 60 e QIKON para UIQ) e possuem as suas funcionalidades básicas implementadas. Entretanto a classe núcleo da aplicação (ApplicationEngine) não possui correspondente no UIKON, devendo ser criada pelo desenvolvedor.

No decorrer do desenvolvimento de aplicações para o SymbianOS, é bastante comum que os desenvolvedores implementem o padrão de projeto MVC – Model/ View /Controller[21], onde a classe ApplicationEngine corresponde ao Model, a classe *view* corresponde à View e a classe UI corresponde ao Controller. Abaixo segue uma descrição do papel de cada classe neste padrão:

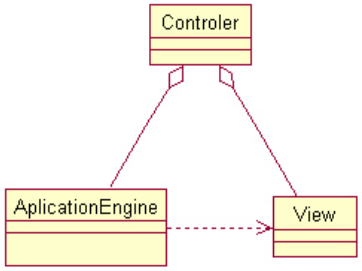

Figura 5: Diagrama do padrão de projeto MVC

Model (ApplicationEngine)

- Encapsula o estado da aplicação.
- Ponto de acesso às funcionalidades.
- Notifica as mudanças à View.
- Responde a consultas da View.

View

- Renderiza (desenha) a aplicação.
- Recebe pedidos de atualizações do Model.
- Envia as entradas do usuário ao Controler.

Controler (UI)

- Define o comportamento da aplicação.
- Mapeia as ações do usuário em alterações do Model.
- Seleciona e controla a View ativa.

Também é importante salientar que o Symbian OS não é C++ padrão, não fornecendo suporte a alguns recursos da linguagem, tais como a *STL – Standard Template Library*

(Biblioteca de classes template padrão) e *RTTI – Run-Time Type Information* (Informação de tipos em tempo de execução).

Além de implementar as classes e seguir a estrutura do *framework* de aplicações Symbian, ilustradas até este ponto, também é necessário seguir uma série de práticas. Estas práticas serão abordadas a seguir.

4.2. Symbian Coding Idioms

Os *Symbian Coding Idioms*, são uma série de práticas que devem ser implementadas para satisfazer restrições de limitação de memória, recursos do sistema (processamento) e tratamento de exceções.

Para entender melhor a necessidade destas práticas, veremos quais os problemas que podemos ter ao alocar memória dinamicamente, para a construção de objetos em tempo de execução.

4.2.1. A necessidade do *symbian code idioms*: Problemas com alocação de memória

No desenvolvimento de aplicações para sistemas portáteis, o uso racional de memória é extremamente importante. Estes dispositivos possuem uma memória escassa e, como são projetados para rodar durante meses ou anos sem serem desligados, *memory leaks*[1] são bem mais prejudiciais: um programa que gere um *memory leak* irá inutilizar parte da memória cada vez que for executado durante o período que o dispositivo permanecer ligado. Portanto, podemos perceber que exceções de falta de memória (*out-of-memory*) podem ocorrer com freqüência bem maior do que em aplicações para PC's.

Uma forma natural de resolver este problema seria verificar, a cada alocação dinâmica de memória, se o retorno da alocação é NULL, indicando que a memória solicitada não foi alocada.

---

[1] *Memory Leak*: Memória alocada que não está sendo referenciada por nenhuma variável no programa e, portanto, é perdida pois não pode ser desalocada.

Entretanto, isto gera um *overhead* ao adicionar as instruções para a verificação, e não pode ser feito para construtores, pois os construtores não retornam valor. Neste caso, poderiam ser construídos objetos incompletos, com alguns membros possivelmente não alocados, o que causaria erros em tempo de execução (exceções).

Como o Symbian OS não suporta exceções, a falha na alocação fará com que a aplicação seja terminada. No entanto, a memória atribuída às variáveis automáticas[1] que foi alocada dinamicamente não é liberada nesta situação, gerando *memory leaks*. Para evitar este problema, deve-se usar o padrão de projeto *Two-phase constructor*[21].

4.2.2. O padrão de projeto Two-phase Constructor

Este padrão de projeto, que é o mais importante do *Symbian Coding Idioms*, consiste em construir um objeto em 2 fases, seguindo as seguintes regras:

(i) Todas as funções que podem gerar exceções de falta de memória devem terminar com "L". Desta forma, é possível saber quais as funções que podem gerar estas exceções. Então, qualquer exceção gerada é propagada de volta através da pilha de chamada de funções até que seja tratada através de um TRAP. Note que é extremamente raro o desenvolvedor usar um TRAP para tratar estas exceções. O *framework* de aplicações do Symbian OS já utiliza TRAP nas posições de código corretas e provê código para tratar as exceções.

(ii) O *framework* de aplicações do Symbian OS provê um sistema de desalocação de memória no caso de uma exceção ocorrer. Toda memória alocada que estiver sendo referenciada através de uma variável automática deverá ser colocada na *Cleanup Stack* (pilha que serve para desalocação em caso de falha). Caso ocorra alguma exceção, a memória alocada por todas as variáveis que estão na *Cleanup Stack* é liberada. Antes de serem destruídos, estes objetos devem ser retirados da *Cleanup Stack*.

[1] Variáveis locais, alocadas na pilha.

(iii) Nenhum construtor ou destrutor de classe pode gerar uma exceção. Portanto se um construtor tiver código que possa gerar uma exceção, todas estas instruções deverão ser colocadas em um método separado, denominado `ConstructL()`, que deve ser chamado logo após a criação do objeto. Além disto, o operador `new()` é sobrescrito para receber um parâmetro `Eleave`. Chamado com este parâmetro, o operador gera uma exceção caso não consiga alocar a memória para o objeto.

Então, para automatizar este processo de construção, as classes que alocam memória automaticamente implementam os métodos estáticos `NewL()` e `NewLC()`. O método `NewL()` simplesmente cria o objeto e gera uma exceção em caso de falta de memória. O método `NewLC()` cria o objeto, gera uma exceção em caso de falta de memória e empilha o objeto na *Cleanup Stack*. Desta forma, devemos chamar o método `NewL()` caso o objeto criado for atribuído a um atributo de classe ou `NewLC()` se o objeto for atribuído a uma variável automática.

```
// Método para alocar o objeto e colocá-lo na cleanup stack
ClasseComposta* ClasseComposta::NewLC()
{
    ClasseComposta * self = new (ELeave) ClasseComposta;
    CleanupStack::PushL(self);  //self é uma variável automática
                                // logo deverá ser colocada na
                                //CleanupStack
    self->ConstructL(); // instruções de inicialização que podem
                        // gerar exceções
    return self;
}

// Método para alocar o objeto
ClasseComposta * ClasseComposta::NewL()
{
    ClasseComposta * self = new (ELeave) ClasseComposta;
    CleanupStack::PushL(self);
    self->ConstructL();
    CleanupStack::Pop(); //remove self da CleanupStack
    return self;
}
```

Código 1: Exemplo do código dos métodos NewL() e NewLC().

Após verificar tudo o que é necessário para o desenvolvimento de aplicações para o Symbian OS, veremos, a seguir, qual o suporte dado por Symbian ao desenvolvimento de jogos.

4.3. Suporte de Symbian a jogos

Nesta seção, iremos discutir quais as funcionalidades necessárias para o desenvolvimento de jogos que estão (e as que não estão) presentes na sua API.

4.3.1. Gerenciamento de I/O

O Symbian OS permite receber os eventos das teclas do telefone. Quando uma tecla é pressionada, são gerados três eventos: `EeventKeyDown`, indicando que a tecla foi pressionada, depois é gerado o evento `EEventKey`, e em seguida o evento `EEventKeyUp` indicando que a tecla foi liberada. Se uma tecla permanecer pressionada por mais de 0,8 segundos, outro evento `EEventKey` será gerado, indicando um pressionamento longo de tecla. Se esta mesma tecla permanecer pressionada por um tempo superior a este, serão gerados eventos `EEventKey` a cada 0,25 segundos, indicando repetição da tecla. É importante salientar que estes valores de tempo são valores padrão, e podem ser alterados para os valores desejados pela aplicação. No entanto, isto é uma configuração global para todo o telefone, então deve ser restaurada sempre que a aplicação perder o foco.

O pressionamento simultâneo da maioria das teclas é bloqueado. Apenas as teclas de *Power* e *Edit* não o são. No entanto, o Symbian OS provê uma API para alterar estas configurações, permitindo o pressionamento simultâneo de teclas, que é um recurso essencial para jogos.

Quanto à tela, não existe um padrão único. Os celulares da *Series 60* utilizam uma tela de cristal liquido com dimensões de 176x208 pixels e 4096 cores. Já os celulares da plataforma UIQ possuem atualmente duas configurações padrão: a configuração de *communicator*, com 240x320 pixels e a configuração de *smartphone*, com 208x320 pixels, ambas com 4096 cores e tela sensível ao toque.

4.3.2. Processamento Gráfico

A API do Symbian OS oferece diversas primitivas para desenho e processamento gráfico, especificadas em sua GDI (*Graphics Device Interface*). Para isto, a GDI disponibiliza um *device context*, classe que abstrai o dispositivo de saída (que pode ser a própria tela ou um bitmap na memória, por exemplo). Esta classe possui métodos para fazer desenho de primitivas gráficas como pontos, linhas, retângulos, etc e cópia de bitmaps (*blitting*) com ou sem máscara, dentre outras operações. Também é possível ter acesso à manipulação direta de pixels, o que permite a utilização de algoritmos que necessitem alterar ou acessar o conteúdo gráfico de uma imagem.

Entretanto, não suporta inverter um bitmap vertical ou horizontalmente (*flipping*) e nem desenhar em um buffer na memória e depois copiar este buffer para a tela automaticamente (*double buffering*), de forma que o desenvolvedor deve escrever o código para implementar isto.

O formato de arquivo de imagens utilizado pelo Symbian OS é o MBM (*multi bitmap file*). Os arquivos MBM são gerados através de uma ferramenta que converte bitmaps do Windows nos bitmaps do EPOC, encapsulando vários bitmaps em um mesmo MBM. A API do Symbian OS provê, então, diversas funções para manipulação destes arquivos.

4.3.3. Processamento de som

O Symbian OS provê uma API para a execução e edição de arquivos de áudio. Esta API suporta diversos formatos de arquivos de áudio, como WAV e WVE e permite que os desenvolvedores criem plug-ins para suportar seus próprios formatos. Esta API é dividida em três interfaces: o *audio sample editor*, que permite editar e gravar arquivos de áudio, o *audio sample player,* para a execução destes arquivos e o *audio tone player*, para tocar tons sintetizados, no estilo MIDI.

Além disto, cada aplicação pode definir quais os tons que serão tocados quando uma tecla é pressionada. Esta funcionalidade pode ser desabilitada, evitando que se consuma tempo de processamento e permitindo que os sons sejam gerenciados através do próprio jogo.

4.3.4. Conectividade

Por ser desenvolvido para celulares, o Symbian OS permite acesso a todas as formas de conexão destes dispositivos. Sua API fornece acesso a:

- GPRS: é possível criar conexões via sockets TCP e UDP e acessar a internet.
- Bluetooth: podemos criar sockets *pear-to-pear* (entre dois telefones) através de Bluetooth, o que para jogos é extremamente interessante, pois este tipo de conexão permite a criação de jogos *multiplayer* sem que o jogador tenha que pagar taxas de conexão (como no caso de GPRS). A desvantagem é que o alcance da conexão Bluetooth é pequeno (aproximadamente 10m).
- SMS: o Symbian OS oferece uma API de acesso às funcionalidades de envio de mensagens através da rede de telefonia. Os primeiros jogos *multiplayer* criados para celulares usavam comunicação baseada na troca de mensagens, e esta API pode ser usada para o desenvolvimento de jogos neste estilo.

4.3.5. Ambientes de desenvolvimento

A Symbian disponibiliza um kit de desenvolvimento (*SDK – Software Development Kit*) para os desenvolvedores. Este SDK pode ser integrado a várias IDE's[1]. A Borland disponibiliza o Borland C++ Builder 6.0 com um plugin que integra o SDK de Symbian e permite o desenvolvimento e teste nesta IDE. Também é possível integrar o SDK de Symbian com o Microsoft Visual Studio 6.0 e o Metrowerks CodeWarrior.

Portanto, podemos perceber que a API do Symbian OS provê as funcionalidades essenciais para o desenvolvimento de jogos, o que torna possível o desenvolvimento de um *framework* de jogos sobre esta API.

Desta forma, foi construído o SymbG(r)aF, *framework* que deve fornecer os recursos utilizados na maioria dos jogos, encapsulando a API do Symbian OS e implementando algumas funcionalidades necessárias que não estão disponíveis, como *flipping*, por exemplo.

No capítulo seguinte será mostrada a metodologia utilizada para o desenvolvimento deste *framework*, bem como os requisitos e a arquitetura do SymbG(r)aF.

---

[1] IDE – Ambiente Integrado de Desenvolvimento (Integrated Development Environment)

SymbG(r)aF

5.1. Metodologia de desenvolvimento

O desenvolvimento do SymbG(r)aF foi feito baseado em vários documentos[19, 21-25] disponibilizados pelo *Forum NOKIA*[26] e pela *Symbian Developer Network*[27]. Estes documentos forneceram as informações sobre o desenvolvimento de aplicações para o Symbian OS, os *Coding Idioms*, etc.

A concepção e desenvolvimento do SymbG(r)aF foram realizados de acordo com a seguinte metodologia:

(i) Desenvolvimento de um jogo para o Symbian OS: o *SeaHunter*, para que fosse possível capturar detalhes de implementação e entender melhor o desenvolvimento para este sistema operacional, bem como compará-lo com implementações de outros jogos para Symbian (abordagem *Bottom-Up*).

(ii) Análise dos elementos comuns aos jogos, a partir de outros *frameworks*[13, 28]: foi verificado o que seria aplicável ao Symbian OS de acordo com a experiência adquirida com o desenvolvimento do *SeaHunter*.

(iii) Análise dos elementos de um *game design:* feita de acordo com a taxonomia proposta por Pedro Aléssio[29], para identificar quais destes elementos são comuns aos jogos e implementáveis por um *framework* (abordagem *Top-Down*).

Desta forma, visa-se evitar os erros das duas abordagens:

- do tipo *Top-Down*, em que se termina construindo um *framework* genérico demais para tentar abranger o maior número de jogos possível e o seu uso fica impraticável devido a questões de performance e usabilidade;
- e do tipo *Bottom-Up*, em que o *framework* fica extremamente especializado no tipo de jogo que é utilizado para sua construção.

A seguir, analisaremos as abordagens utilizadas para a elicitação dos requisitos do SymbG(r)aF, bem como os seus resultados.

5.2. Levantamento dos requisitos do SymbG(r)aF

5.2.1. Abordagem *Bottom-Up*: desenvolvendo um jogo para o Symbian OS

Para poder elicitar melhor os requisitos do *framework* e diminuir os riscos de criar um "elefante branco" (ver Figura 2, na página 11), foi desenvolvido um jogo para o Symbian OS. Este jogo deveria ser simples, mas ao mesmo tempo deveria conter os recursos utilizados na maioria dos jogos, para que pudéssemos perceber como seria a implementação destes recursos sobre a API do Symbian OS.

Então, foi desenvolvido o *SeaHunter*, um jogo de coletar itens e atirar em inimigos, que já havia sido desenvolvido para J2ME, pela equipe do C.E.S.A.R.[1] (ilustrado na Figura 6). Este jogo foi escolhido, pois, como já havia sido implementado para celulares, foi possível ter acesso a todos os gráficos do jogo, bem como ao algoritmo de sua lógica, proporcionando focar o seu desenvolvimento na utilização da API do Symbian OS e não em detalhes pertinentes à lógica específica do jogo.

(a) (b)

Figura 6: o SeaHunter, implementado em J2ME(a) e para o Symbian OS(b)

[1] Centro de Estudos e Sistemas Avançados do Recife

Através da implementação deste jogo conseguimos perceber algumas diferenças na programação para o Symbian OS, como, por exemplo, como implementar os *Symbian Coding Idioms*, fazendo o uso da *cleanup stack* e como implementar o padrão de projeto *Two-phase Constructor*.

Além disto pudemos perceber alguns recursos da API do Symbian OS desconhecidos anteriormente como os *Descriptors,* os *Timers* e os *Sprites*.

O Symbian OS faz uso de *Descriptors*, no lugar das *strings* convencionais. Os *Descriptors* são um conjunto de classes que abstraem *strings*, fazendo o uso da mínima memória necessária, bem como fornecendo um conjunto de métodos para manipulação e alocação dos mesmos.

Como o Symbian OS é um sistema operacional não preemptivo e dirigido a eventos, os jogos não podem implementar sua rotina principal como um loop infinito. Desta forma, temos que utilizar temporizadores, que chamam o loop principal do jogo quanto o seu tempo expira.

Symbian também possui uma classe pronta, a `CSprite`, para representar os *sprites* do jogo. Os principais atributos desta classe são o conjunto de imagens e máscaras, a taxa de repetição dos frames (*frame rate*) e a posição na tela. Uma vez definido isto, A API se encarrega de desenhar os *frames* no tempo correto. Entretanto, esta classe se utiliza de uma imagem para representar cada *frame*, o que consome mais memória por ter que criar vários objetos, e não podem ser definidos tempos diferentes entre a transição de um *frame* para o outro.

Além da implementação do *SeaHunter*, foram analisados os códigos exemplo dos jogos *Battleships*[30] (código publicado no livro *Symbian OS C++ for Mobile Phones*[31]) e *Asteroids*[32], publicado na *Symbian Developer Network*. Desta forma, foi possível capturar os elementos comuns aos jogos, bem como a experiência dos seus autores, pois o *Simon Douet*, autor do *Asteroids*, é Engenheiro de Software Sênior da Symbian e o *Richard Harrisson*, autor do *Battleships*, possui 10 anos de experiência no desenvolvimento para Symbian e atualmente é Autor Sênior da *Symbian Press*.

Nestas implementações, as considerações mais importantes dizem respeito à arquitetura implementada pelos autores.

Nos dois jogos, os autores não utilizaram a classe `CSprite`, definida na API de Symbian, preferindo realizar suas próprias implementações. O jogo *Battleships* foi implementado

seguindo o padrão de projeto MVC (visto na seção 4.1), e possui suporte a som, implementando seu gerenciamento na classe de *view*.

O jogo *Asteroids* implementa os *sprites* do jogo como uma máquina de estados, seguindo o padrão de projeto *State*[33]. Neste padrão, cada estado do *sprite* é representado por uma classe, que é responsável por definir seu comportamento e responder aos eventos.

5.2.2. Abordagem Top-Down: implementando a taxonomia dos elementos dos jogos eletrônicos

De acordo com os diversos *frameworks* e motores de jogos analisados [7, 13, 28] pudemos identificar quais os módulos seriam necessários a um *framework*. Abaixo segue a arquitetura proposta por Carlos André[7] para um *framework* de jogos para PC's.

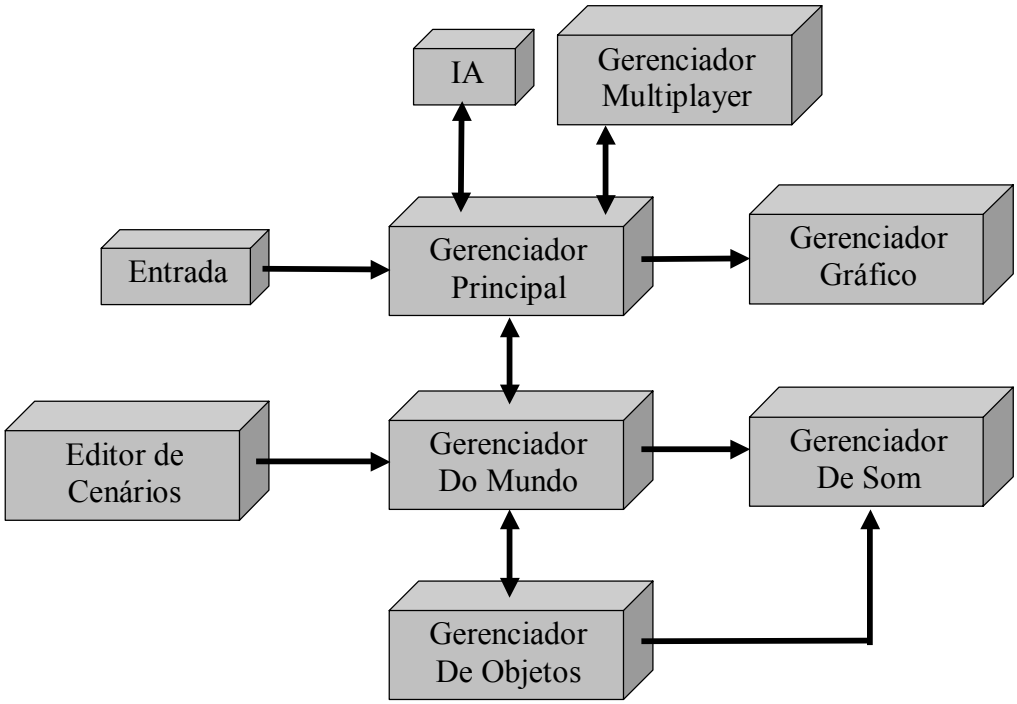

Figura 7: Arquitetura de um *framework* de jogos para PC

Nesta arquitetura temos os seguintes papéis:

- Interface de Entrada: responsável por capturar os eventos de entrada (teclado, mouse, etc.) e repassá-los para o gerenciador do mundo.

- Gerenciador gráfico: Gerencia o estados dos pixels da tela a partir da descrição dos diversos objetos que formam a cena.
- Gerenciador de Som: responsável pela execução dos sons do jogo.
- Gerenciador de IA: realiza certas ações do jogo de acordo com o estado do jogo e algumas regras.
- Gerenciador Multiplayer: faz o gerenciamento da conexão e troca de informações entre os computadores conectados.
- Gerenciador de Objetos: responsável pela interação dos objetos e com outros e com o cenário. Podemos ter vários gerenciadores de objetos, um para cada grupo de objetos similares do jogo.
- Gerenciador do Mundo: gerencia o estado atual do jogo, interagindo com os outros gerenciadores para informar o que deve ser feito.
- Gerenciador Principal: é a ponte de troca de informações e provê um ponto de acesso único às funcionalidades do jogo.

Devemos notar que esta arquitetura considera a implementação de um motor de jogos para PC's, sendo, portanto, uma arquitetura que necessita de bastante recursos em termos de memória e processamento para ser implementada para celulares. Na prática, alguns destes gerenciadores podem ser fundidos para diminuir o tamanho da implementação. Isto será mostrado posteriormante, após a análise dos elementos comuns aos jogos.

Em paralelo a esta análise da arquitetura, uma análise dos elementos comuns aos jogos, identificados por *Cris Crawford* em *The Art of Computer Game Design*[8], tornou possível identificar quais destes elementos seriam passíveis de serem implementados em um *framework*.

Crawford identificou 4 elementos comuns a todos os jogos: **Representação**, **Interação**, **Conflito** e **Segurança**.

Pedro Aléssio, em *Designing Game Design*[29], faz uma taxonomia destes elementos citados por *Crawford*, subdividindo-os para que seja possível projetar um jogo, que é um sistema complexo, a partir de seus elementos básicos. Aléssio discorda de *Crawford* na questão da Segurança, alegando que este é um elemento que não faz parte do desenvolvimento dos jogos eletrônicos, pois sempre está presente no ambiente em que se joga, sendo mais adequado a jogos esportivos como atletismo, futebol, etc.

Também é possível notar que o Conflito é um elemento inerente ao jogo, e cada jogo possui seu conflito específico, não podendo ser implementado de forma genérica, pois a implementação do conflito se confunde com a implementação das regras do jogo.

Portanto, dos 4 elementos comuns aos jogos, a **Representação** e a **Interação** foram identificados como os elementos possíveis de serem implementados de forma genérica a todos os jogos.

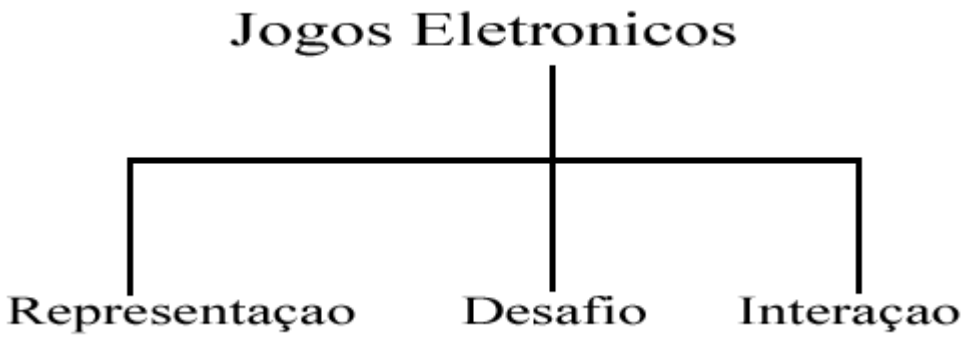

Figura 8: Elementos de um game design.

A seguir será analisada esta subdivisão dos elementos de um jogo para que seja possível identificar quais destes elementos básicos são passíveis de serem implementadas por um *framework*.

5.2.2.1. Representação

Aléssio subdivide a **Representação** em dois elementos: **Personagens** e **Mundo**.

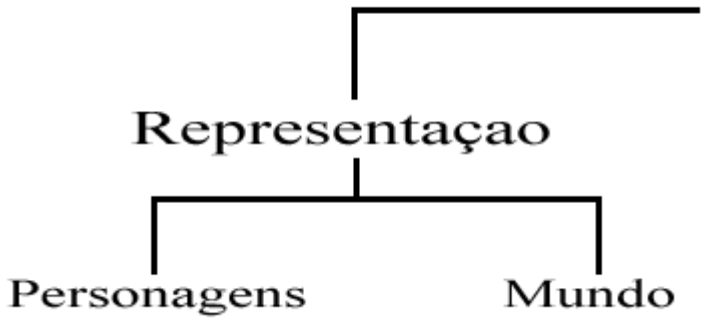

Figura 9: Representação é dividida em Personagens e Mundo.

Os **Personagens** possuem habilidades e fraquezas e podem utilizar ferramentas para conseguir seu objetivo. Desta forma, podemos considerar personagens e ferramentas como **objetos** do jogo, cada um com suas características individuais, mas ao mesmo tempo com atributos que são comuns a todos os objetos, como posição, tamanho, imagem representativa, etc. Então o *framework* deverá prover um conjunto de classes responsável por representar estes objetos. Também é importante notar que estes objetos sofrem mudanças durante o jogo nas suas características (posição, tamanho, imagem) de forma que o *framework* deverá prover um mecanismo para tratar adequadamente estas mudanças.

Segundo Aléssio, o **Mundo** é composto por **Cenário** e **Leis**.

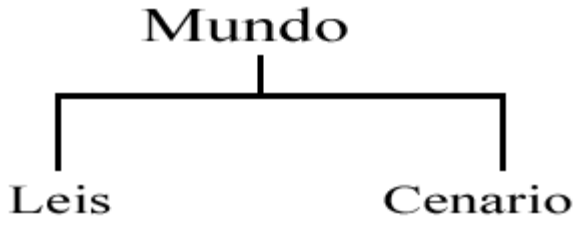

Figura 10: Mundo é subdividido em Leis e Cenário.

O **Cenário** é toda a representação do ambiente físico onde acontece o jogo. Esta representação pode ser feita de diversas formas em uma tela: 2D, 2 1/2D e 3D. Como estamos

desenvolvendo para dispositivos com recursos limitados, iremos focar a representação de nosso cenário em 2D, por ser a representação que consome menor processamento e memória, bem como ser a representação mais fácil de ser implementada.

Portanto, o *framework* deverá prover mecanismos de representar este cenário e fazer a interação entre o cenário e os objetos, bem como prover funcionalidades inerentes aos cenários 2D, como o rolamento automático da tela (*scrolling*).

Quanto às **Leis**, elas podem ser divididas em Naturais, Sociais e Artificiais. Dentre estas, o que pode ser implementado de forma comum aos jogos são as leis naturais da física. Portanto o *framework* deverá prover mecanismos para modelar fisicamente seus objetos, detectando colisão, realizando movimentação e etc. É importante que haja um balanceamento entre a complexidade da modelagem física e o tempo consumido de processamento, para que o jogo não fique com uma modelagem irreal nem impraticável de ser jogado devido ao tempo consumido nesta atividade.

5.2.2.2. Interação

A interação com o jogo pode ser dividida em 2 partes: **Possibilidade** e **Acesso**.

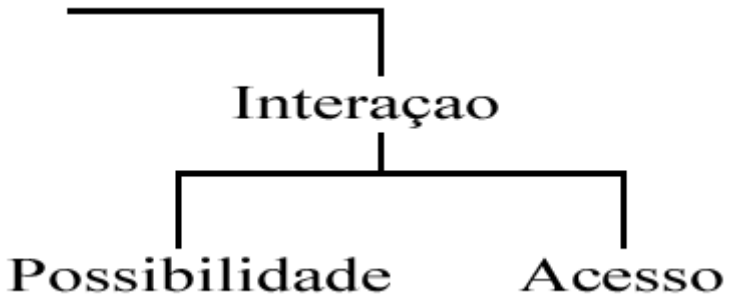

Figura 11: A interação é dividida em Possibilidade e Acesso

A possibilidade consiste em tornar possível o jogador realizar alguma ação. Como isto está intrinsecamente relacionado ao *game design*, não pode ser implementado por um *framework*.

Já o **Acesso**, ou Interface, é justamente a forma com a qual o usuário interage com a **Possibilidade**. Esteticamente falando, existem várias formas de Interface do jogador com o jogo,

o que impossibilitaria uma generalização. Entretanto, do ponto de vista funcional, pode-se perceber que o jogador realiza entradas (***input***) e espera por uma reação do jogo (***output***) às suas entradas. Então o *framework* deve prover um ponto em que se possa capturar as entradas do usuário através das teclas e do ponteiro da tela sensível ao toque. Deve também prover formas de responder às ações do jogador. Esta resposta já está de certa forma mapeada na **Representação** do jogo. Entretanto, as respostas do jogo não devem ser apenas visuais, o *framework* deve prover respostas sonoras também. Portanto, o *framework* também deve implementar um módulo que torne possível e facilite a execução de sons.

Além da interação entre o jogador e o jogo, pode existir **interação entre mais de um jogador**, no caso de jogos *multiplayer*. Consequentemente, um *framework* deverá dar acesso à camada de conectividade do Symbian OS, possibilitando conexão através de GPRS e Bluetooth.

Então, através das duas abordagens utilizadas, pudemos identificar quais os principais requisitos para a implementação do SymbG(r)aF. Na Figura 12, é proposta uma arquitetura de *framework* para celulares, onde temos o gerenciador de jogo, do mundo e o gerenciador principal fundidos em um só. Além disto, outros gerenciadores foram modificados para permitir uma melhor implementação.

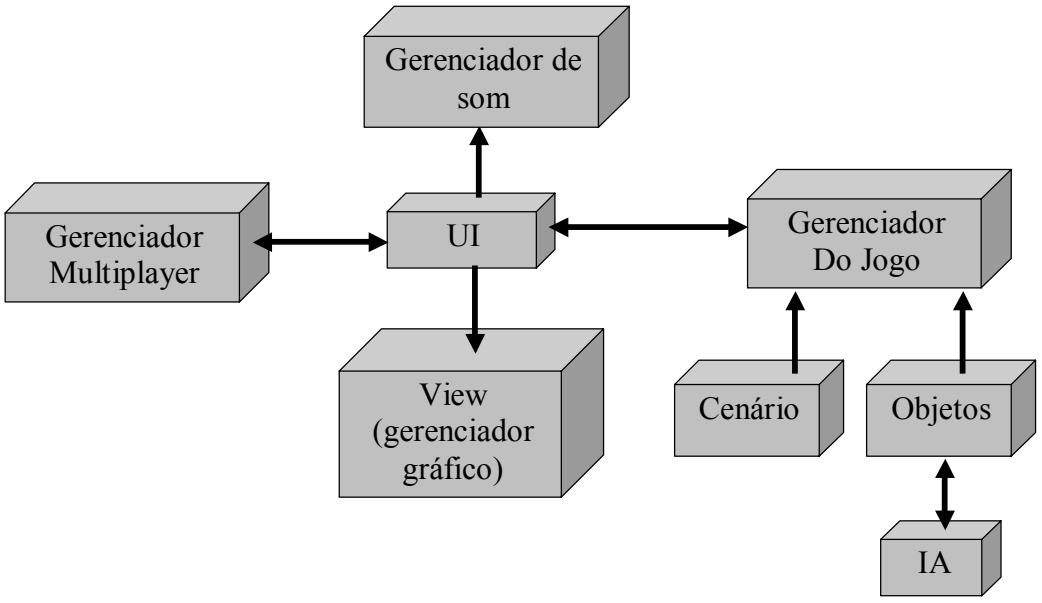

Figura 12: Arquitetura proposta de um *framework* de jogos para celulares

Nesta arquitetura, temos o Gerenciador do Jogo, responsável por guardar o estado do jogo e enviar comandos a UI, como também criar e gerenciar os objetos. Este gerenciador assume a função de tratar a interação entre os objetos e cenário e fazer o controle da tela. A tela é representada pela *View*, que é responsável por desenhar os objetos e o cenário. O Gerenciador de Som é responsável por responder aos eventos de som. A *UI – User Interface* é responsável por fazer o gerenciamento dos eventos de entrada, repassá-los ao gerenciador do jogo e enviar eventos do jogo para a *view* e para os outros gerenciadores. Desta forma, temos uma arquitetura bem mais simples, que pode ser implementada em um celular sem perda significativa de performance. A seguir iremos descrever a arquitetura do SymbG(r)aF, que foi baseda na arquitetura proposta para um *framework* de jogos para celulares.

5.3. Arquitetura do SymbG(r)aF

A arquitetura do SymbG(r)aF foi definida de acordo com os requisitos identificados na seção 5.2, e foi decidido que seria um *framework* de arquitetura aberta, ou seja, os desenvolvedores tem acesso ao código fonte do SymbG(r)aF e podem fazer modificações.

O objetivo foi desenvolver um protótipo do *framework*, onde seria tratada apenas a representação dos objetos do jogo e a interação entre eles. Desta forma, o foco foi a implementação gráfica, gerenciamento dos objetos do jogo, detecção de colisão e implementação do cenário, não sendo implementados o suporte a execução de sons e a comunicação.

Primeiro definimos como o SymbG(r)aF iria interagir com o *framework* de aplicações do Symbian OS. Resolvemos implementar o padrão de projeto MVC (descrito na seção 4.1).

Foi criada uma classe que seria o ponto de entrada do SymbG(r)aF, o `CSGFGameEngine`, representando o *Model* no padrão de projeto MVC. Esta classe é responsável pela criação e gerenciamento dos objetos do jogo (*sprites*), do cenário e dos recursos (animações e estados dos *sprites*), além da detecção de colisão.

A classe de UI, `CSGFGameUI`, foi herdada da classe `CEikAppUi` do UIKON, sendo responsável por instanciar o *Game Engine*, receber e tratar eventos e criar as *Views* do jogo. Esta classe foi implementada como uma máquina de estados com três estados pré-definidos: `EStartingGame`, `EPlayingGame` e `EEndGame`, que correspondem aos estados de início e configuração do jogo, o estado do jogo e o estado fim do jogo, respectivamente. É importante

notar que esta implementação não impede que o desenvolvedor crie mais estados ou *views*, visto que o código do *framework* pode ser alterado. Estes estados foram pré-definidos por serem o mínimo necessário para a criação de um jogo.

A cada estado está associada uma instância da *view,* que deve herdar da classe `CCoeControl`, e corresponde a representação da tela neste estado. Esta classe deve implementar as funções de criação de bitmaps, deve possuir um *Container* para gerenciá-los e implementar alguns recursos indisponíveis pela API de Symbian, como *flipping* e *double buffering*. Na figura Figura 13 podemos ver o diagrama de classes completo do SymbG(r)aF.

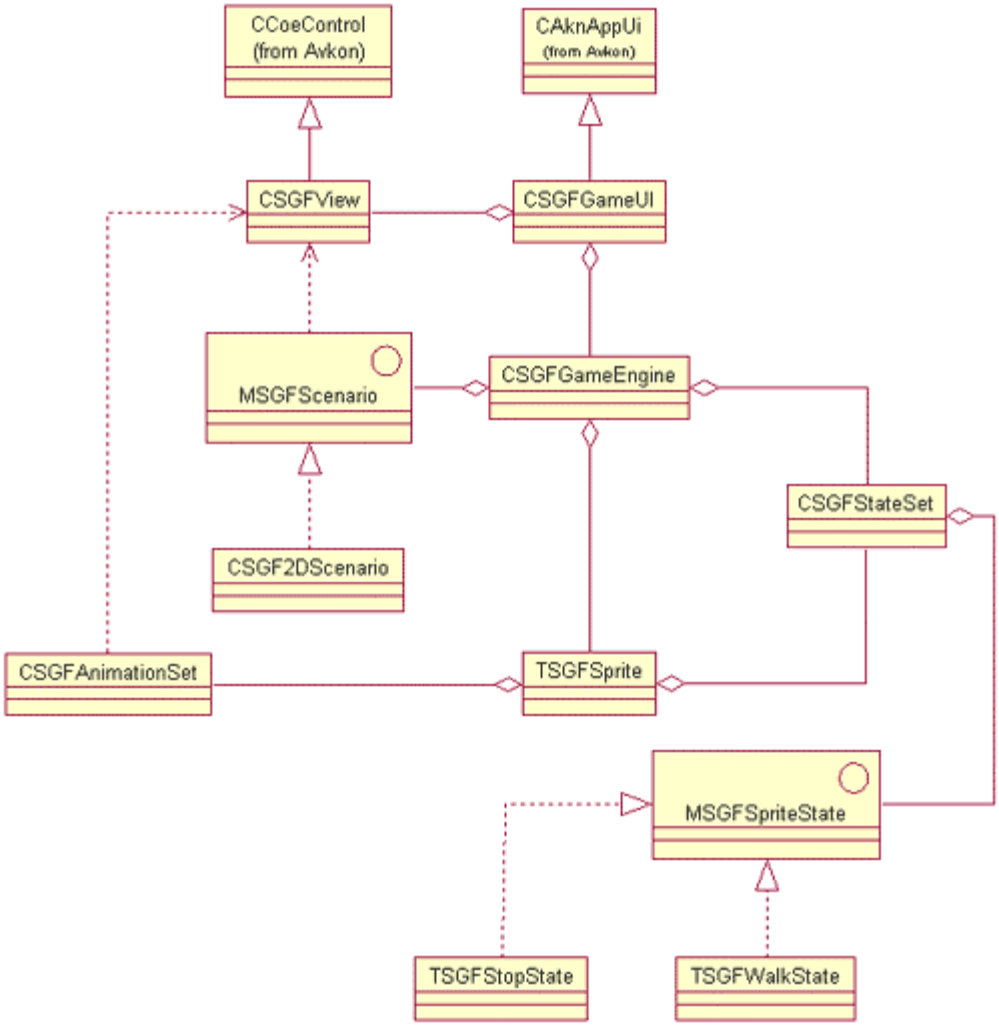

Figura 13: Diagrama de classes do SymbG(r)aF

Na próxima seção, veremos alguns detalhes de como o SymbG(r)aF foi implementado, bem como a descrição das suas principais classes.

5.4. Detalhes de Implementação

A implementação do SymbG(r)aF foi feita baseada no desenvolvimento de outro jogo, o Little Fighter. Este jogo consiste em um personagem que deve coletar itens e destruir inimigos. A idéia foi implementar este jogo e usá-lo para criar as classes e os módulos do *framework*, fazendo *refactoring*[34] para generalizar as necessidades identificadas em cada módulo do jogo. Desta forma, pode-se testar a implementação do *framework* gradativamente, enquanto temos sempre uma versão "rodando".

Como exemplo desta metodologia de implementação, será mostrado como foi o desenvolvimento dos *sprites* do jogo. Os *sprites* foram implementados como máquinas de estados, utilizado o padrão de projeto *State*. Entretanto, primeiro foi implementado um *sprite* dentro da classe de *view*. Então começou-se a fazer *refactoring* no código para generalizar este *sprite* e criar as classes correspondentes. Antes de implentar a classe `TSGFSprite`, foram implementadas as classes de quem esta dependia. A primeira classe implementada foi a `CSGFAnimationSet`, que representa um conjunto de animações do *sprite*. Esta classe possui uma imagem, com todos os *frames* de todas as animações do *sprite* e um vetor de animações. Cada animação é um vetor da classe `TSGFFrame`, que é um retângulo que indica qual parte da imagem da animação corresponde ao *frame*. Esta classe possui métodos para setar a aniamção corrente e pegar o *frame* desejado desta animação.

Com esta classe implementada e testada, passou-se a implementar a classe `TSGFSprite`. Primeiro, os atributos do *sprite*, que antes pertenciam a classe de *view*, foram agrupados na classe `TSGFSprite`. Esta classe passou também a possuir um objeto do tipo `CSGFAnimationSet`. Com isto implementado, começou-se a implementar os estados do *sprite*. Foi criada uma classe `MSGFSpriteState`, classe abstrata com os métodos que cada estado deve implementar. Depois, o código correspondente a cada estado do sprite (parado, andando, pulando, etc) foi movido para a classe correspondente, dando origem as classes `TSGFStopState`, `TSGFWalkState`, `TSGFJumpState`, etc. Então, este novo objeto, o `TSGFSprite` passou a

ser utilizado no jogo. Seguindo esta abordagem, foram construídas todas as classes do *framework*. Na Figura 14 podemos visualizar o jogo implementado com o SymbG(r)aF.

Figura 14: Little Fighter implementado com o SymbG(r)aF

Existiram várias dificuldades na implementação do SymbG(r)aF. Uma das dificuldades principais foi o fato das aplicações em Symbian serem criadas como DLL's. Devido a isto, as aplicações não podem ter nenhuma variável global, ou atributo estático de classe. Isto inviabiliza a construção do padrão de projeto *Singleton*[33], amplamente utilizado para criar os diversos gerenciadores do *framework*. Para contornar este problema, devemos passar os objetos necessários como parâmetro para os métodos desejados. No caso dos *SpriteStates*, como eles podiam ser compartilhados por vários *sprites*, foi implementado o padrão de projeto *FlyWeight*[33], com algumas modificações para que a criação dos estados fosse realizada por uma classe *container* (`CSGFStateSet`).

5.5. Estudo de caso

A implementação do jogo Little Fighter foi extremamente útil para o desenvolvimento do SymbG(r)aF. Este jogo, apesar de simples, requer a maioria dos recursos utilizados em jogos. Para o seu desenvolvimento foram utilizados os seguintes recursos do SymbG(r)aF:

- ***Sprites* e animações**: cada objeto do jogo foi implementado a aprtir da classe CSGFSprite do SymbG(r)F. Isto também permitiu a implementação de várias animações por sprite.
- **Cenário**: o jogo utilizou uma simplificação de um mapa de *tiles*, implementado pelo SymbG(r)aF.
- **Entrada**: a classe de UI do SymbG(r)aF realiza toda a configuração do teclado do celular para que seja possível o pressionamento de mais de uma tecla simutaneamente, bastando para o desenvolvedor mapear as teclas pressionadas em eventos do jogo.
- **Saída**: O SymbG(r)aF provê um timer para atualização da tela. Desta forma, só é necessário implementar o método para desenhar os objetos na ordem desejada.
- **Detecção de colisão**: o gerenciador de jogo do SymbG(r)aF provê a detecção de colisão entre os objetos e o cenário, através da técnica de *bounding box*[1]. O que deve ser implementado são os métodos para aplicar a reação a colisão, que vai depender do comportamento de cada objeto.
- **Mecanismo do jogo**: o gerenciador de jogo já provê os métodos para implementar o loop principal do jogo, que consiste em: pegar entrada do usuário, enviar eventos para os objetos do jogo, atualizar os objetos, verificar condições do jogo (fim de jogo, etc) e atualizar a tela. Isto é implementado através dos Timers do Symbian OS.

Estes recursos influenciaram bastante na implementação do jogo, garantindo uma melhor qualidade e um menor tempo de desenvolvimento. O que foi conseguido com o SymbG(r)aF foi que o desenvolvedor do jogo implementasse apenas o código correspondente às funcionalidades do seu jogo, ou seja, o *game design*. O SymbG(r)aF mostrou-se extremamente eficiente no desenvolvimento de jogos do tipo arcade, como o Little Fighter e o SeaHunter, e jogos que

[1] Detecção de colisão na qual os objetos são envoltos por um retângulo e é verificada a colisão entre estes retângulos.

requerem *sprites* animados movendo-se pela tela, como *BreakOut, PacMan, Asteroids, RiverRaid, Smurfs, Zaxxon*, etc.

O SymbG(r)aF possui recursos excessivos para implementar jogos que não requeiram animações, detecção de colisão, etc, como os jogos de carta, no estilo paciência ou poker. Mas por ser extremamente modularizado e possuir arquitetura aberta, apenas uma parte dos seus módulos pode ser utilizada, evitando que se tenha código desnecessário no jogo.

No capítulo seguinte serão apresentadas as conclusões obtidas com este trabalho, bem como as dificuldades e os trabalhos futuros.

Conclusões

Com a implementação do SymbG(r)aF, foi possível perceber a importância de um *framework* para jogos. De acordo com a proposta de desenvolvimento, foram cobertos todos os requisitos do desenvolvimento do *framework*, que incluíam todo o gerenciamento gráfico e do jogo, bem como restrições de performance, modularização e reusabilidade.

A implementação de jogos seguindo a arquitetura do SymbG(r)aF é extremamente simples, facilitando o desenvolvimento. Do tempo de desenvolvimento do Little Fighter (que foi desenvolvido junto com o SymbG(r)aF) apenas 40% foi gasto no desenvolvimento do jogo propriamente dito.

A performance do *framework* é extremamente satisfatória, não impactando na jogabilidade dos jogos implementados com o mesmo. O jogos desenvolvidos para este sistema operacional não tiveram nenhum problema de performance, mesmo com um *frame rate* fixo (gerado por um temporizador). Como o emulador do Symbian é mais lento do que os dispositivos reais, foi necessário inclusive reduzir o *frame rate* ao instalar o jogo no dispositivo para que ficasse com uma boa jogabilidade.

6.1. Dificuldades Encontradas

Algumas dificuldades foram encontradas no desenvolvimento do *framework*. Apesar de existirem vários documentos sobre o desenvolvimento de aplicações em Symbian, comparações, padrões de projeto, etc, a sua API é bastante extensa e não possui uma documentação abrangente. Isto faz com que o desenvolvimento de aplicações para o Symbian OS não seja uma atividade trivial.

Outra dificuldade no desenvolvimento foi o fato das aplicações para o Symbian OS serem desenvolvidas como DLL's, sem a possibilidade de declarar variáveis estáticas ou globais. Isto impediu a utilização de alguns padrões de projeto extremamente utilizados no desenvolvimento de jogos, como o *Singleton*[33].

A complexidade do desenvolvimento de um *framework* de jogos também foi um desafio. A necessidade de prover uma solução geral e ao mesmo tempo otimizada, associada as restrições

de programação para o Symbian OS aumentaram a dificuldade do desenvolvimento deste *framework*.

6.2. Trabalhos futuros

Como continuação deste trabalho, pode-se extender este *framework* para implementar os módulos que ainda não são suportados:

- Implementar um gerenciador de jogos *multiplayer* que suporte conexões GPRS e Bluetooth, bem como um protocolo para a comunicação.
- Implementar um gerenciador de som responsável por criar os recursos de som e responder aos eventos para reproduzí-los.
- Implementar outros tipos de cenário 2D.
- Implementar um gerenciador de IA.

Além disto, uma melhoria extremamente importante seria unificar os *frameworks* desenvolvidos para J2ME[13] e BREW[6] com o SymbG(r)aF, de forma a oferecer uma API semelhante que facilitasse o *porting* de uma plataforma para outra. Com isto poderia ser gerada uma ferramenta para automatizar este processo, gerando código para uma plataforma a partir do código de outra.

7. Referências

[1] SUN *J2ME - Java 2 Micro Edition*, disponível em http://java.sun.com/j2me/ (15/06/2003).

[2] SUN *J2ME Datasheet*, disponível em http://java.sun.com/j2me/docs (16/06/2003).

[3] Qualcomm *BREW - Binary Runtime Environment for Wireless*, disponível em http://www.qualcomm.com/brew/ (15/06/2003).

[4] Muldner, T. (2002). *C++ Programming with Design Patterns Revealed*, Addison-Wesley.

[5] NOKIA (2003).*J2ME & SymbianOS: A Platform Comparison*, Forum Nokia, disponível em http://www.forum.nokia.com/documents/ (20/06/2003).

[6] Nascimento, I.F. (2003). *Desenvolvimento de um Framework para Jogos Sobre a Plataforma BREW*, Trabalho de Graduação, Centro de Informática, Universidade Federal de Pernambuco.

[7] Ramalho, G. *Página da disciplina de jogos*, centro de Informática, disponível em http://www.cin.ufpe.br/~game (14/06/2003).

[8] Crawford, C. (1982).*The Art of Computer Game Design*, Eletronic Version, disponível em http://www.vancouver.wsu.edu/fac/peabody/game-book/Coverpage.html.

[9] Clark, J. *Golgotha*, disponível em http://jonathanclark.com/golgotha/ (20/06/2003).

[10] *Golgotha Forever*, disponível em http://sourceforge.net/projects/golgotha/ (20/06/2003).

[11] Fathammer *X-Forge*, disponível em http://www.fathammer.com/x-forge/index.shtml (02/07/2003).

[12] Synergenix *Mophun*, disponível em http://www.mophun.com/ (02/07/2003).

[13] Pessoa, C.A.C. (2001). *wGEM: um Framework de Desenvolvimento de Jogos para Dispositivos Móveis*, Dissertação de Mestrado, Centro de Informática, Universidade Federal de Pernambuco.

[14] *Sony Ericsson P800*, disponível em http://www.sonyericsson.com/P800/ (18/06/2003).

[15] UIQ *SymbianOS User Interface Plataform*, disponível em http://www.uiq.com/ (20/06/2003).

[16] NOKIA *Series 60 User Interface Plataform*, disponível em http://www.nokia.com/series60 (20/06/2003).

[17] *Nokia 3650*, disponível em http://www.nokia.com/nokia/0,,2273,00.html (14/06/2003).

[18] *Nokia N-Gage game deck*, disponível em http://www.n-gage.com/ (14/06/2003).

[19] NOKIA (2002).*Introduction to Series 60 Applications for C++ Developers*, disponível em http://www.forum.nokia.com/documents/ (11/06/2003).

[20] Jode, M.d. (2002).*Porting Applications from Series 60 to UIQ*, disponível em http://www.symbian.com/developer/techlib/papers/cpp_porting.html (20/06/2003).

[21] NOKIA (2002).*Designing C++ Applications for Series 60*, disponível em http://www.forum.nokia.com/documents/ (11/06/2003).

[22] Kanner, J. (2002). *Series 60 and Symbian OS based smart phone as a multiterminal game platform*, Master of Science Thesis, Tampere University of Tecnology.

[23] NOKIA (2002).*Coding Idioms for Symbian OS*, disponível em http://www.forum.nokia.com/documents/ (14/06/2003).

[24] NOKIA (2002).*Series 60 Application Framework Handbook*, disponível em http://www.forum.nokia.com/documents/ (14/06/2003).

[25] Weinstein, A. (2002).*Symbian OS C++ for Windows C++ programmers*, disponível em http://www.symbian.com/developer/techlib/papers/windows_symbianOS/Windows_SymbianOS.pdf (21/06/2003).

[26] *Forum NOKIA*, disponível em http://www.forum.nokia.com/.

[27] *Symbian Developer Network*, disponível em http://www.symbian.com/developer/.

[28] Madeira, C. (2001). *FORGE V8: Um framework para o desenvolvimento de jogos de computador e aplicações multimídia*, Dissertação de Mestrado, Centro de Informática, Universidade Federal de Pernambuco.

[29] Aléssio, P.M. (2003). *Designing Game Design: Uma nova abordagem em jogos eletronicos*, Trabalho de Graduação, Departamento de Design, Universidade Federal de Pernambuco, Recife.

[30] Harrison, R. (2003).*Battleships source code.*, Symbian OS C++ for Mobile Phones, Symbian Press., disponível em http://www.symbian.com/books/scmp/scmp-source.html (08/07/2003).

[31] Harrison, R. (2003). *Symbian OS C++ for Mobile Phones*, Symbian Press.

[32] Douet, S. (2003).*Asteroids game*, Symbian Developer Network, disponível em http://www.symbian.com/developer/techlib/apps/asteroids.html (08/07/2003).

[33] Erich Gamma, R.H., Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

[34] Fowler, M. (2000). *Refactoring: Improving the Design of Existing Code*, Addison Wesley.

[35] Interactive Digital Software Association. (2003). *Essential facts about the computer and video game industry - 2003 Sales, Demographics and Usage data.*

[36] Symbian OS. Disponível em http://www.symbian.com