Compatibilizando MIDP 1.0 e MIDP 2.0 para o desenvolvimento de jogos em dispositivos móveis

TARCISIO CAMARA
GEBER RAMALHO
ANDRÉ SANTOS

Universidade Federal de Pernambuco – Centro de Informática
{tpc, glr, alms}@cin.ufpe.br

Abstract

Esse trabalho apresenta uma forma de utilizar o suporte para jogos do MIDP 2.0 em dispositivos móveis compatíveis apenas com o MIDP 1.0. Dessa forma, jogos desenvolvidos em uma versão podem ser facilmente portados para outra, mantendo a lógica do jogo inalterada.

Keywords: *MIDP, J2ME, Game tool*

1 Introdução

MIDP (*Mobile Information Device Profile*) [3] é parte da plataforma J2ME (*Java 2 Platform, Micro Edition*) [1] e define o ambiente de aplicação Java para dispositivos móveis de informação (*mobile information devices - MIDs*) como telefones celulares, pagers e PDAs (*Personal Digital Assistant*).

Cada versão do MIDP [6][7] especifica um conjunto de bibliotecas, APIs (*Application Programing Interfaces*) e recursos de hardware que os dispositivos devem disponibilizar para serem ditos compatíveis. Assim, aplicações para MIDP (também chamadas *MIDlets*), desenvolvidas para uma certa versão da especificação, têm sua portabilidade garantida entre todos os dispositivos compatíveis.

A versão 1.0 do MIDP [6], lançada em setembro de 2000, foi rapidamente absorvida pelo mercado e implementada por diversos fabricantes de MIDs. A quantidade de jogos para esse ambiente logo se tornou expressiva, consagrando-se como uma das aplicações preferidas pelos usuários. Diante da demanda, um pacote inteiro de classes, dedicados ao desenvolvimento de jogos, foi incluído na nova API do MIDP, versão 2.0 [7], lançada em novembro de 2002.

Infelizmente, por ter sido recentemente liberada, poucos dispositivos implementam essa nova versão do MIDP. Além disso, inúmeros aparelhos que já estão no mercado são compatíveis apenas com a versão antiga, e muitos de seus usuários não pretendem trocar de aparelho de imediato. Assim, muitos jogos ainda são desenvolvidos para MIDP 1.0, sem o benefício do novo pacote; e ainda, jogos que pretendam atingir ambos os mercados, podem acabar tendo várias versões com arquitetura interna radicalmente diferente.

Para resolver esse problema, esse trabalho apresenta uma forma prática para portar o pacote para jogos do MIDP 2.0 para o MIDP 1.0, a partir da implementação de referência (*Reference Implementaion* - RI) do MIDP 2.0 [8], destinada aos fabricantes de dispositivos que desejam implementar o padrão. Adotaremos o termo "porte do pacote" inspirado na atividade de porte da RI entre dispositivos, referenciada em sua documentação [14]. Em nosso processo de porte, o código original do pacote é extraído e submetido a uma série de adaptações e decisões de projeto para compatibilizá-lo com a versão 1.0 do MIDP, como mostra a Figura 1.

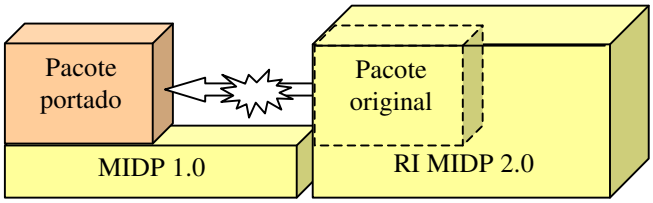

Figura 1 Processo de porte do pacote para jogos

Por ser baseado na implementação de referência, o pacote portado assegura automaticamente toda a estrutura hierárquica das classes e a interface de todos os métodos do pacote original, tornando-o praticamente transparente para a aplicação. Dessa forma, além de acelerar o desenvolvimento de jogos para MIDP 1.0, o pacote também facilita o porte de jogos entre as duas plataformas, isolando a lógica do jogo. A Figura 2 representa esse efeito.

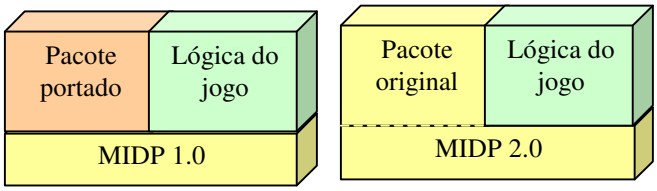

Figura 2 Jogos projetados para ambas as versões do MIDP

De fato, devido à compatibilidade retroativa, *MIDlets* implementados para MIDP 1.0 podem ser executados no ambiente MIDP 2.0 sem alteração [7]. No entanto, com a utilização do pacote portado, uma versão específica para MIDP 2.0 pode ser facilmente gerada utilizando o pacote original nativo, tornando-se menor e possivelmente mais rápida.

De forma similar, jogos implementados originalmente para MIDP 2.0 e que utilizam o pacote nativo, podem ser facilmente adaptados para MIDP 1.0, utilizando o pacote portado. Logicamente, demais recursos presentes apenas na plataforma MIDP 2.0 devem ser excluídos ou contornados.

Na seção 2 desse trabalho, vamos apresentar o pacote original como especificado pela API do MIDP 2.0. Na seção 3, apresentaremos a implementação de referência e suas restrições de utilização. Na seção 4, discutiremos os passos necessários ao processo de porte, apresentando as dificuldades encontradas e as soluções implementadas. Por fim, na seção 5, relatamos uma experiência de utilização do pacote portado num jogo completo, o *Istari*.

2 O pacote para jogos do MIDP 2.0

O pacote para jogos, incluído no MIDP 2.0 em `javax.microedition.lcdui.game`, contém cinco classes: `GameCanvas`, `LayerManager`, `Layer`, `Sprite` e `TiledLayer`. Apesar do número reduzido de classes, elas provêem algumas das principais técnicas para o desenvolvimento de jogos 2D, como *off-screen buffer* [15], janela de visualização, *scroll*, mapas de *tiles* estáticos e animados, *sprites* animados[15] e detecção de colisão por *bounding box* e por *pixel* [16].

A Figura 3 demonstra o relacionamento entre as cinco classes, sua dependência de classes externas e a relação entre os pacotes. Alguns dos principais métodos das classes também foram explicitados, assim como alguns atributos e relacionamentos privados em benefício da expressividade do diagrama. Como o pacote portado possui a mesma estrutura e interface do pacote original, a Figura 3 também o representa, inclusive exibindo a classe `Util` adicionada durante o processo de porte.

No restante dessa seção, vamos apresentar uma rápida descrição das classes, como especificadas pela API do pacote original. Para maiores detalhes, consulte a documentação completa incluída com o MIDP 2.0 [3][7].

A classe `GameCanvas` estende a classe `Canvas` convencional adicionando a capacidade de leitura, a qualquer momento, da situação das teclas de jogos (*game keys*); e o recurso de gráfico secundário (*off-screen buffer*), evitando que imagens parciais sejam exibidas durante a composição de cada cena do jogo.

A classe `LayerManager` gerencia uma seqüência de `Layers` e permite a definição de uma janela de visualização que pode ser movida para implementar o *scroll* do jogo. Os objetos `Layer` armazenados são desenhados automaticamente na ordem correta e respeitando a janela de visualização definida.

A classe `Layer` representa um elemento visual do jogo, contendo posição espacial, largura, altura e um atributo indicando se está atualmente visível ou não. A classe `Layer` é estendida pelas classes `Sprite` e `TiledLayer`.

A classe `TiledLayer` implementa um mapa de *tiles* retangulares, ou seja, uma matriz de células retangulares onde cada célula pode ser preenchida com uma pequena imagem (*tile*). Essa técnica permite que grandes cenas sejam compostas a partir de pequenas imagens.

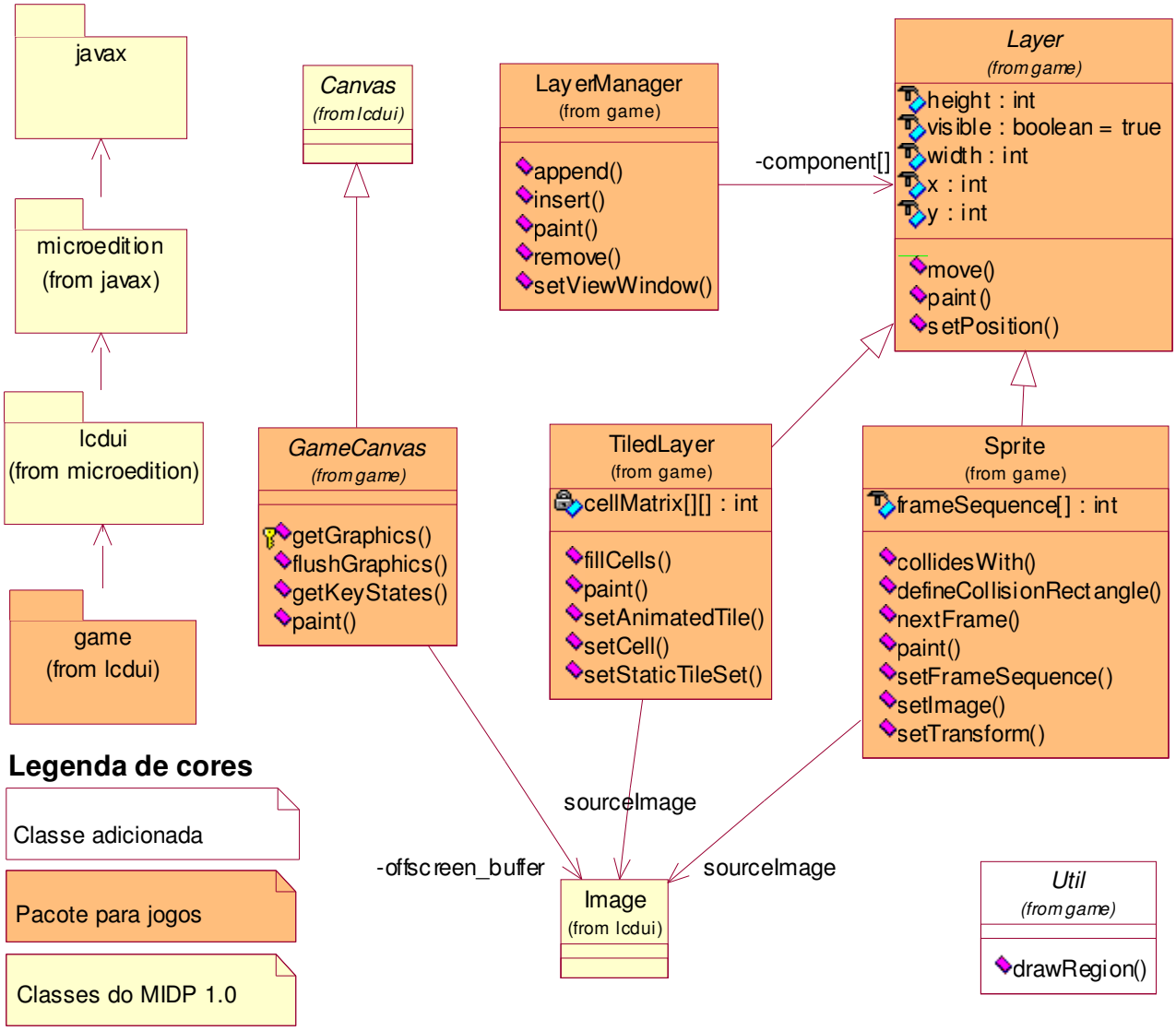

Figura 3 Pacote para jogos do MIDP 2.0, portado para MIDP 1.0

Todos os possíveis *tiles* são armazenados em uma única imagem (*TileSet*). Cada célula recebe o índice do *tile* correspondente. No caso geral, um índice referencia sempre um mesmo *tile* (mapa estático), no entanto, um mecanismo adicional pode alterar o *tile* referenciado por um certo índice, permitindo que várias células sejam animadas facilmente.

A classe `Sprite` implementa uma seqüência de imagens (*frames*) que, quando exibidas em uma certa ordem, produzem uma animação. Todos os possíveis *frames* devem ser passados em uma única imagem, no entanto, algumas transformações, como espelhamento e rotação, podem ser aplicadas sobre eles expandindo as possibilidades de exibição.

A colisão entre dois *sprites* pode ser testada de duas formas: (1) apenas comparando-se uma região retangular (*collision rectangle*), ou (2) avaliando a transparência dos *pixels*. Infelizmente, o recurso de leitura de *pixels* não está disponível no MIDP 1.0, como será discutido mais adiante.

3 A implementação de referência

As implementações de referência são projetadas para servirem como ponto de partida para os fabricantes de dispositivos que desejem implementar uma máquina virtual em seus produtos.

A implementação de referência para MIDP 2.0 inclui a maior parte do código dos pacotes escritos em Java puro, para facilitar o porte entre dispositivos. Entretanto, alguns métodos críticos são implementados com chamadas nativas a funções escritas em linguagem C. O *download* pode ser realizado a partir do *site* da Sun [8], mediante a devida licença de utilização.

Fechando o escopo no pacote para jogos, todas as cinco classes descritas, assim como todos os seus métodos, estão disponíveis em linguagem Java [14]. Infelizmente, em alguns casos o código depende de métodos nativos implementados em classes externas ao pacote, notadamente métodos que implementam recursos de entrada e saída e de manipulação de imagens, por razões de desempenho ou dependência do hardware. Como a API *Java Native Interface* (JNI) [9] não está disponível a aplicações J2ME [2][4][5], essas dependências a métodos nativos devem ser contornadas.

4 O processo de porte do pacote

Nessa seção, discutiremos diversos pontos que merecem atenção para que o porte do pacote para jogos seja bem sucedido. Eventualmente, métodos de ambas as versões do MIDP são referenciados para contextualização do problema ou compreensão da solução proposta. Para maiores detalhes sobre esses métodos, deve ser consultada a documentação da API [3][6][7].

4.1 Cópia dos arquivos fonte

Os arquivos fonte originais das cinco classes do pacote podem ser encontrados no subdiretório `src\share\classes` da implementação de referência. O primeiro passo consiste na cópia direta desses arquivos para o código da aplicação. A compilação nesse momento deve informar 16 erros.

4.2 Acréscimo no tamanho da aplicação

Por passar a integrar o código da aplicação, fatalmente os arquivos do pacote produzirão um acréscimo no seu tamanho final. Em nossos experimentos, todas as classes do pacote portado representaram cerca de 11 Kb do tamanho da aplicação, depois de compiladas e compactadas com as ferramentas `javac` e `jar`, ambas fornecidas como parte do *Java 2 SDK Standard Edition 1.4.1* [10]. Em um ambiente reduzido como o MIDP, esse tamanho ainda pode ser relevante, representando cerca de 20% do tamanho médio do código de um *MIDlet*.

Assim, recorremos ainda a algumas ferramentas especializadas na redução do tamanho das aplicações, conhecidas como *obfuscators* [11]. Elas atuam detectando e extraindo campos, métodos e até classes inteiras que não tenham sido utilizadas pela aplicação. Logicamente, o desempenho dessas ferramentas está vinculado à utilização que a aplicação faz do pacote. Em nosso jogo de exemplo, onde muitos recursos foram utilizados, a contribuição das classes do pacote para o tamanho final da aplicação foi de apenas 5 Kb, usando o *obfuscator* ProGuard, versão 1.6.2 [12].

4.3 Nome do pacote portado

O nome do pacote portado, que se reflete no diretório de destino para onde os arquivos serão copiados, também merece uma certa atenção. Para sua máxima transparência, o nome original `javax\microedition\lcdui\game` poderia ser mantido sem nenhum conflito com a API do MIDP 1.0.

Infelizmente, essa decisão pode incorrer em alguns transtornos. Por exemplo, alguns *obfuscators*, como o *DashO* da *PreEmptive* [13], usam os nomes dos pacotes para identificar classes de sistema que devem ser ignoradas durante o processo. Nesse caso, o pacote pode acabar sendo considerado como de sistema e excluído do arquivo resultante.

Deixamos essa decisão a cargo do projetista que deve considerar fatores como os objetivos do projeto, a necessidade de compartilhamento de código e as ferramentas de desenvolvimento envolvidas.

4.4 Gráfico secundário (*off-screen buffer*)

O recurso *off-screen buffer*, implementado pelo `GameCanvas`, disponibiliza, através do método `getGraphics`, um objeto gráfico secundário onde a aplicação pode preparar cada quadro da cena do jogo sem que imagens parciais sejam exibidas. Quando todo o quadro tiver sido desenhado, o gráfico pode ser exibido ao usuário mediante chamada a um dos métodos `flushGraphics`.

O código original implementa esse recurso, delegando a funcionalidade dos métodos `flushGraphics` a classes de suporte, integradas com a porção nativa da implementação de referência. Emulamos esse comportamento com o seguinte código, usando apenas métodos herdados do `Canvas` convencional:

```
{
    repaint();
    serviceRepaints();
}
```

A chamada ao método `repaint` força uma chamada assíncrona ao método `paint`, que por sua vez, transfere o gráfico secundário para o gráfico do `display`. A documentação original dos métodos `flushGraphics` também exige que ele só retorne após o gráfico ter sido totalmente apresentado. Esse comportamento é garantido pela chamada ao método `serviceRepaints`.

Note que, nessa implementação, se o método `paint` for sobrescrito (prática indicada no MIDP 1.0), o gráfico secundário não será exibido automaticamente. Nesse caso, apenas o gráfico gerado pelo código sobrescrito será visível. Para evitar essa preocupação adicional, recomendamos que o `GameCanvas` sempre seja desenhado mediante `getGraphics` e `flushGraphics`. Se o mecanismo anterior do MIDP 1.0 realmente precisar ser utilizado (sobrescrevendo o método `paint`) sugerimos considerar a utilização de um `Canvas` convencional.

4.5 Modo tela cheia (*full screen mode*)

Internamente, a inicialização da classe `GameCanvas` também depende de chamadas nativas para determinar o tamanho real da tela, chamado *full screen mode*. Essas dimensões são usadas pelo construtor do `GameCanvas` como parâmetros para criação do gráfico secundário.

Esse recurso foi incluído porque, na nova especificação do MIDP 2.0, a área visível do `Canvas` (recuperada pelos métodos `getWidth` e `getHeight`) pode alterar durante a execução. Como isso não é possível no MIDP 1.0, o tamanho real da tela é igual à área visível do `Canvas` que, por sua vez, é compartilhada por todas as instâncias.

Assim, a implementação do `GameCanvas` no MIDP 1.0 pode utilizar os próprios métodos `getWidth` e `getHeight` para determinar o tamanho da tela.

4.6 Estado das teclas de jogos (*key states*)

A classe `GameCanvas` também tem a capacidade de recuperar, a qualquer momento, a situação das teclas de jogos (*game keys*), através do método `getKeyStates`.

O código original da implementação de referência para o método `getKeyStates` delega sua funcionalidade a métodos nativos. Emulamos esse comportamento adicionando um atributo ao `GameCanvas` para "memorizar" as teclas pressionadas entre as chamadas ao método `getKeyStates`. O atributo é atualizado com o mapeamento das teclas capturadas pelo método `keyPressed`, que é parte do mecanismo convencional do MIDP 1.0 para monitoramento de eventos do teclado.

A documentação original do `GameCanvas` sugere que ambos os mecanismos podem ser utilizados ao mesmo tempo, inclusive podendo-se desativar um deles no construtor da classe. Infelizmente, como precisamos implementar o novo mecanismo a partir do antigo, a utilização simultânea no MIDP 1.0 torna-se complicada. Novamente, recomendamos que apenas um dos mecanismos seja usado por vez, no entanto, caso a utilização simultânea seja realmente necessária, a execução do código do método `keyPressed` definido aqui deve ser garantida.

Adicionalmente, a especificação original do método `getKeyStates` foi projetada para tratar o pressionamento simultâneo de teclas, recurso de hardware não garantido pelo MIDP 1.0. Note que a implementação proposta aqui ainda é capaz de simular esse comportamento

capturando várias teclas pressionadas seqüencialmente entre as chamadas ao método `getKeyStates`.

4.7 Detecção de colisão por *pixel* (*pixel level*)

A colisão entre dois *sprites* pode ser testada apenas comparando-se uma região retangular, chamada retângulo de colisão (*collision rectangle*). Essa técnica, conhecida como *bounding box*, é extremamente rápida e muito utilizada, no entanto, pode apresentar um efeito visual insatisfatório [16]. Nesses casos, um teste mais elaborado pode avaliar exaustivamente a transparência dos *pixels* das imagens detectando a sobreposisão de *pixels* opacos. Essa técnica é conhecida como detecção de colisão por *pixel*.

A técnica que deve ser utilizada é informada através de um parâmetro adicional, chamado `pixelLevel`, presente nos métodos `collidesWith` da classe `Sprite`. Infelizmente, o recurso de leitura de *pixels* não está disponível na especificação do MIDP 1.0, impossibilitando a detecção de colisão por *pixel*. Assim, todo o código original dedicado ao tratamento do parâmetro `pixelLevel` pode ser substituído pelo lançamento de uma exceção (por exemplo, `IllegalArgumentException`) caso o recurso seja solicitado (`pixelLevel == true`).

De forma geral, o uso de exceções para sinalizar incapacidade do hardware provê inclusive uma oportunidade para que a aplicação diferencie seu comportamento de acordo com o ambiente de execução, apenas tratando as exceções. No entanto, caso essa ocorrência seja muito freqüente, uma alternativa mais adequada para a aplicação seria a verificação prévia de qual versão do MIDP está em execução, através da propriedade de sistema *microedition.profiles*.

4.8 Cópia de regiões de imagens (*draw region*)

As classes `Sprite` e `TiledLayer` necessitam de um recurso onde uma região de uma imagem fonte seja desenhada em uma certa posição de um objeto gráfico (`Graphics`). Isso porque, ambas as classes utilizam uma imagem única como fonte de pequenas imagens representando *frames*, no caso de *sprites*, ou *tiles*, no caso de mapas de *tiles*.

Esse recurso é implementado pelo novo método `drawRegion` incluído na classe `Graphics`, apenas na API do MIDP 2.0. Por razões de desempenho, seu código é definido pela implementação de referência como nativo.

Como a classe `Graphics` já está presente no MIDP 1.0, não podemos redefini-la, pois a classe já é fornecida pelo próprio ambiente de execução. A solução foi definir uma nova classe, com o nome `Util`, para abrigar uma implementação estática do método `drawRegion` de forma que um parâmetro adicional representa o objeto `Graphics` ao qual o método teria sido chamado. Assim, seja `g` um objeto da classe `Graphics`, todas as chamadas no formato `g.drawRegion(...)` podem ser convertidas para `Util.drawRegion(g, ...)`.

De forma geral, o comportamento do método, como definido anteriormente, pode ser reproduzido por uma operação de *clipping* no objeto gráfico de destino, selecionando a região onde a imagem será desenhada, seguida por uma chamada ao método `drawImage`, devidamente posicionada para que a região de origem sobreponha a região de destino.

4.9 Transformações sobre imagens (*draw region*)

Adicionalmente, a especificação do método `drawRegion` ainda permite que algumas transformações, como espelhamento e rotação, possam ser aplicadas sobre a região da imagem copiada. Essas transformações são definidas e utilizadas pela classe `Sprite` como alternativas de exibição, no entanto, também podem ser aplicadas diretamente a qualquer imagem através de chamadas ao método `drawRegion`.

A transformação nula é emulada como caso especial desenhando toda a imagem de uma vez (conforme descrito na seção 4.8); todas as outras transformações foram emuladas desenhado-se *pixel* a *pixel*. Para facilitar essa tarefa, utilizamos uma reimplementação estática de dois métodos privados, chamados `getTransformedPtX` e `getTransformedPtY`, já definidos na classe `Sprite`, que realizam um mapeamento de um *pixel* quando aplicado a uma certa transformação.

Logicamente, o desempenho da transformação nula, implementada com cópia

direta, é muito superior ao das demais transformações. Por essa razão, recomendamos uma certa cautela na utilização destas em situações iterativas, como a animação de objetos.

4.10 Resumo das restrições de utilização

A Tabela 1 resume os recursos que sofreram alguma restrição de utilização, decorrentes das adaptações e comentários apresentados anteriormente, assim como as classes envolvidas na sua implementação.

Recurso	Restrições	Classes
Nome do pacote portado	Se o nome do pacote original for mantido, algumas ferramentas podem se confundir, ao considerá-lo um pacote de sistema.	Todas
Gráfico secundário	Se o método `paint` do `GameCanvas` precisar ser sobrescrito, o recurso requer que o código do método `paint` original continue sendo executado.	`GameCanvas`
Modo de tela cheia	Recurso irrelevante no MIDP 1.0.	`GameCanvas`
Estado das teclas de jogos	Se o método `keyPressed` do `GameCanvas` precisar ser sobrescrito, o recurso requer que o código do método `keyPressed` original continue sendo executado.	`GameCanvas`
Detecção de colisão por *pixel*	Recurso não disponível no MIDP 1.0. Na tentativa de utilização, uma exceção é lançada pelos métodos `collidesWith`.	`Sprite`
Cópia de regiões de imagens	Se apenas a transformação nula for aplicada, o recurso apresenta um bom desempenho, próximo a uma chamada `drawImage`.	`TiledLayer`, `Sprite`, `Util`
Transformações de imagens	A emulação em software das transformações especiais possui um desempenho precário. O recurso deve ser utilizado com cautela.	`Sprite`, `Util`

Tabela 1 Resumo de restrições de utilização do pacote portado

5 Uma experiência de utilização

O pacote portado foi utilizado com sucesso em um jogo completo. *Istari* é um jogo de tabuleiro, por turno, com características de RPG (*Role Playing Game*), onde dois magos lutam entre si, cada um comandando um exército de criaturas, evocadas ao longo da partida, na intenção de eliminar o mago adversário.

Figura 4 Telas do jogo Istari

A classe `TiledLayer` foi utilizada para representação do tabuleiro. Cada criatura é representada por um objeto da classe `Sprite`. Entre os ciclos de turnos, as criaturas são animadas representando os movimentos selecionados pelos jogadores. A transformação de espelhamento foi utilizada para confrontar as faces dos personagens durante as batalhas. Os recursos de *off-screen buffer* e *game key states* foram utilizados em substituição aos mecanismos tradicionais do MIDP 1.0.

Mesmo tendo utilizado quase todos os recursos disponíveis, a contribuição das classes do pacote para o tamanho final da aplicação foi de apenas 5 Kb, quando aplicado o *obfuscator* ProGuard, versão 1.6.2 [12].

O pacote portado manteve o nome original e reproduziu perfeitamente o comportamento esperado, tornando sua utilização no MIDP 1.0 quase transparente para o programador. O jogo

pôde ser portado para o MIDP 2.0, apenas suprimindo-se as classes do pacote portado.

6 Conclusões

Esse trabalho apresentou como é possível portar o pacote para jogos do MIDP 2.0 para o MIDP 1.0, a partir do código disponibilizado pela implementação de referência da nova versão.

Durante o processo de porte do pacote, identificamos que apenas um pequeno conjunto de recursos sofreu alguma restrição de utilização. Nossos experimentos mostram que, apesar dessas restrições, a utilização do pacote no MIDP 1.0 é viável e estável, permitindo que jogos desenvolvidos em uma versão da plataforma possam ser facilmente portados para outra.

Assim, o porte do pacote para jogos do MIDP 2.0 para o MIDP 1.0 se mostrou uma boa estratégia para compatibilizar o desenvolvimento de jogos entre as versões da plataforma MIDP.

7 Agradecimentos

Gostaríamos de agradecer a Hugo Henrique Reis Raposo, Igor Azevedo Sampaio e Rafael Palermo de Araújo pelas valiosas contribuições no projeto e implementação do jogo Istari.

8 Referências

1. J2ME - Java 2 Platform, Micro Edition, Sun Microsystems, http://java.sun.com/j2me/ (08/08/2003)
2. CLDC - Connected Limited Device Configuration, Sun Microsystems, http://java.sun.com/products/cldc/ (08/08/2003)
3. MIDP – Mobile Information Device Profile, Sun Microsystems, http://java.sun.com/products/midp/ (08/08/2003)
4. JSR-000030 J2ME Connected, Limited Device Configuration http://jcp.org/aboutJava/communityprocess/final/jsr030/index.html (08/08/2003)
5. JSR-000139 Connected Limited Device Configuration 1.1 http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html (08/08/2003)
6. JSR-000037 Mobile Information Device Profile (MIDP) http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html (08/08/2003)
7. JSR-000118 Mobile Information Device Profile 2.0 http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html (08/08/2003)
8. MIDP Reference Implementation - Version 2.0 FCS http://wwws.sun.com/software/communitysource/j2me/midp/download20.html (08/08/2003)
9. JNI – Java Native Interface, Sun Microsystems, http://java.sun.com/products/jdk/1.2/docs/guide/jni/ (08/08/2003)
10. Java 2 SDK Standard Edition 1.4.1, http://java.sun.com/j2se/1.4.1/download.html (08/08/2003)
11. Support for Bytecode Obfuscators http://java.sun.com/j2me/docs/wtk2.0/bc_html/Ap_obfuscator.html (08/08/2003)
12. Proguard, Eric Lafortune, Luciad, http://proguard.sourceforge.net/ (08/08/2003)
13. DashO-EE (Embedded Edition), PreEmptive, Sollutions, http://www.preemptive.com/tools/ (08/08/2003)
14. Porting The Game Package, MIDP Reference Implementation - Version 2.0 FCS, Livro Porting MIDP, Cap. 6, pg. 52.
15. Gamedev.net – Game Dictionary, http://www.gamedev.net/dict/ (18/08/2003)
16. Gamedev.net – Collision Detection Algorithm, 1998, http://www.gamedev.net/reference/articles/article754.asp (18/08/2003)