# Uma Proposta de Motor de Jogos Baseado em um Conjunto Simplificado de Features de Jogos Digitais

**Filipe Macêdo Borges Boaventura, Victor Travassos Sarinho**

Departamento de Exatas
Universidade Estadual de Feira de Santana – BA – Brasil

{fmbboaventura,vsarinho}@gmail.com

***Abstract.*** *This paper proposes a simplified model of representative classes for digital game features, capable of representing casual games regardless of their implementation resources. The* ***M****inimal* ***En****gine for* ***Di****gital* ***Ga****mes (****MEnDiGa****) presents minimal features in a representative hierarchy of spatials and game elements, along with a basic support of behaviors and events relating to such features of the game business. In order to prove the MenDiGa viability, a clone of the classic game Pitfall is under development and will also be presented in this paper.*

***Resumo.*** *Este artigo propõe um modelo simplificado de classes representativas de features de jogos digitais, capazes de representar jogos casuais independente de seus recursos de implementação. O Motor Mínimo para Jogos Digitais (****M****inimal* ***En****gine for* ***Di****gital* ***Ga****mes - ****MEnDiGa****) apresenta features mínimas em uma hierarquia representativa de espaços e de elementos de jogos, juntamente com um suporte básico de comportamentos e de eventos relativos a tais recursos do negócio do jogo. Para comprovar a viabilidade do MEnDiGa, um clone do jogo clássico Pitfall se encontra em desenvolvimento e também será apresentado neste artigo.*

## 1. Introdução

Na abordagem de desenvolvimento de jogos baseada em game engines (motores de jogos), o motor de jogo representa "a coleção de módulos de código de simulação que não especificam diretamente o comportamento do jogo (lógica do jogo) ou o ambiente do jogo (dados do nível)" [Lewis e Jacobson 2002].

De acordo com BinSubaih e Maddock [2008], embora os motores de jogos sejam reutilizáveis em vários projetos de diferentes jogos, eles geram uma relação de dependência muito grande do negócio do jogo com os recursos de implementação fornecidos pelo motor escolhido. Neste sentido, BinSubaih e Maddock [2008] propuseram o conceito de *g-factor* na tentativa de separar objetos de negócio do jogo da implementação propriamente dita. Contudo, BinSubaih e Maddock [2008] não descreveram como tais objetos poderiam ser definidos no intuito de garantir um repositório de objetos de negócio para jogos.

Buscando identificar características comuns e reusáveis no domínio dos jogos digitais, Sarinho e Apolinário [2008] propuseram o modelo de features ***N****arrative*, ***E****ntertainment*, ***S****imulation* e ***I****nteraction* (**NESI**). Trata-se de um modelo baseado na literatura referente a conceitos de jogos digitais capaz de representar o *g-factor* (negócio do jogo) no projeto de jogos digitais diversos [BinSubaih e Maddock 2008].

No intuito de mostrar a viabilidade do uso de features na produção real de jogos digitais, Sarinho e Apolinário também propuseram o modelo **G**ameSystem, **D**ecisionSupport e **S**ceneView (**GDS**) [Sarinho e Apolinário 2009]. Neste modelo, cada feature principal descreve configurações genéricas e aspectos comportamentais de um jogo, sendo estas focadas em aspectos de implementação identificados na literatura de jogos digitais [Sarinho e Apolinário 2009]. A partir do modelo GDS, definiu-se também uma abordagem de produção generativa de jogos, tendo com base configurações de features do modelo GDS proposto.

Embora os modelos NESI e GDS sejam capazes de representar jogos digitas sem depender da estrutura de motores de jogos, o excesso de features propostas dificulta o desenvolvimento de jogos mais simples como os jogos casuais por exemplo. Uma prova desta dificuldade foi obtida com a produção do **F**eature-based **En**vironment for **Di**gital **Ga**mes (**FEnDiGa**) [Sarinho, Apolinário e Almeida, 2011], um motor baseado na integração de features NESI e GDS definidas via **O**bject **O**riented **F**eature **M**odeling (**OOFM**) [Sarinho e Apolinário 2010; Sarinho, Apolinário e Almeida, 2012, Sarinho, Apolinário e Almeida, 2013].

Este artigo propõe uma implementação simplificada dos modelos de features NESI e GDS propostos para jogos digitais denominada ***M**inimal **En**gine for **Di**gital **Ga**mes* (**MEnDiGa**). Seu objetivo é garantir a representação e implementação de features mínimas necessárias para o desenvolvimento de jogos casuais, bem como permitir a portabilidade do *g-factor* [BinSubaih e Maddock 2008] alcançada nos modelos NESI e GDS previamente propostos.

## 2. Modelando MEnDiGa

Apresentado originalmente por Kang et al. [1990] como parte da **F**eature **O**riented **D**omain **A**nalysis (**FODA**), a modelagem de features é utilizada para identificar propriedades do sistema durante a análise de domínio. Segundo eles, "um modelo de features representa as features padrões de uma família de sistemas e as relações entre elas" [Kang et al 1990]. Ainda segundo Kang et al [1990], features são aspectos ou características de um domínio que são visíveis ao usuário. Elas são usadas para identificar semelhanças ou diferenças entre os produtos de uma linha de produtos [Antkiewicz e Czarnecki 2004].

Partindo de FODA, tem-se no MEnDiGa um jogo digital formado por coleções de três features principais: *Spatial*, *Behavior* e *Observer* (Figura 1). Tratam-se de features que, assim como no modelo GDS, representam comportamentos e configurações genéricas do negócio do jogo [Sarinho e Apolinário 2009].

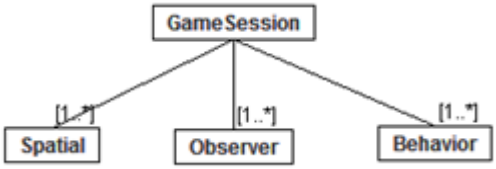


**Figura 1. Diagrama de features do MEnDiGa**

Com relação à feature *Spatial* (Figura 2), trata-se de uma coleção de features *Node* que representam os elementos do jogo, a qual pode ser estendida a um *Environment* e conter também uma coleção de features *Location*. Cada feature *Node* é situada por uma *CurrentLocation* e pode conter ao mesmo tempo informações de

*AudioNode* e *GraphicNode*, semelhante ao *SceneNode* do modelo GDS [Sarinho e Apolinário 2009].

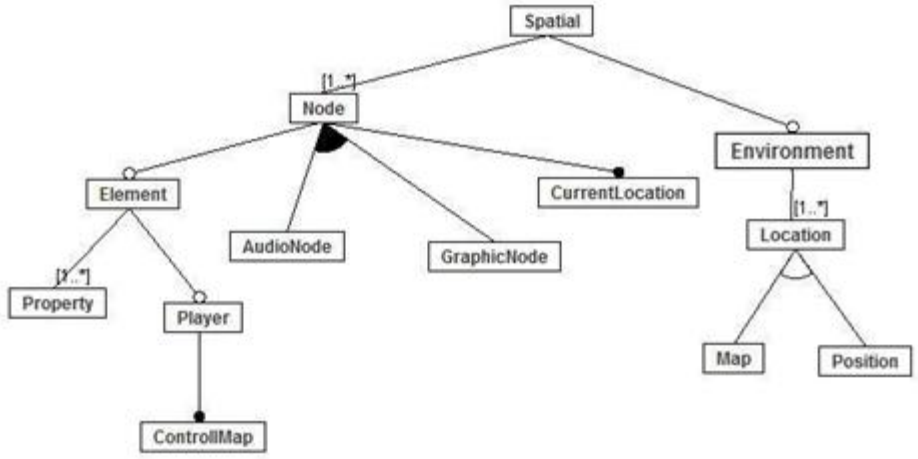


**Figura 2. Diagrama de subfeatures da feature *Spatial***

*AudioNode* representa informações sobre efeitos sonoros ou arquivos de áudio a serem usados posteriormente em um jogo digital. Já o *GraphicNode* representa configurações relacionadas a modelagem gráfica de um certo *Node*, tais como altura, largura e caminho de um arquivo de imagem que deve ser renderizado. No caso de *sprite sheets* (imagem contendo vários quadros de uma personagem 2D – Figura 3), *GraphicNode* permite identificar quais células de um *sprite sheet* irão formar uma animação.

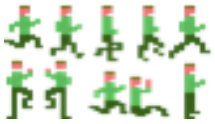

**Figura 3. Exemplo de sprite sheet**

Features *Element* são especializações de features *Node*, contendo uma ou mais features *Property* representadas por um nome identificador e um valor representativo corrente (*Velocidade*: 50 m/s; *Life*: 2 vidas). Um *Element* que possui comportamentos (features *Behavior*) ativados pelo usuário é definido como *Player*. Comportamentos do *Player* (*Jump* e *Crouch*, por exemplo) são relacionados à comandos default (*MOVE_UP, MOVE_DOWN*, entre outros) definidos na sua feature *ControlMap*.

Com relação às features *Observer*, estas são responsáveis por executar features *Behavior* de acordo com a avaliação de seus recursos de monitoração. Por exemplo, um *InputObserver* pode ser configurado para monitorar o acionamento do comando *BUTTON1* por um dispositivo de entrada, executando as features *Behavior* do *Player* relacionadas logo após o acionamento do mesmo. Já um *CollisionObserver* pode disparar a execução de um *IncreaseScoreBehavior* quando o *Player* colide num item ou um *LoseLifeBehavior* quando colide com um inimigo.

## 3. Implementando MEnDiGa

Tendo como base o modelo hierárquico de features inicialmente proposto para o MEnDiGa, implementou-se um framework [Fayad, Schmidt, e Johnson 1999] Java [Oracle 2014] capaz de configurar features de jogos digitais definidas pelo mesmo.

Trata-se de uma etapa de projeto de domínio de software [Czarnecky 2004], onde a estrutura do modelo de features proposto é utilizada como base para a definição de um conjunto de classes capaz de configurar jogos digitais diversos.

Assim, para o MEnDiGa framework, tem-se na classe *GameSession* a responsabilidade de configurar as demais classes *Spatial*, *Observer* e *Behavior* representativas de features de um jogo digital modelado. *GameSession* é uma classe base do MEnDiGa framework, o qual é organizado nos seguintes subpacotes (Figura 4):

- ***spatials***: representando a coleção de espaços e elementos do jogo nas classes *Spatial, Environment e Node*;
- ***nodes***: definindo as especializações e componentes internos do *Node* nas classes *AudioNode*, *GraphicNode*, *Element* e *Player*;
- ***locations***: definindo os componentes internos de um *Environment* nas classes *Location*, *Map* e *Position*;
- ***behaviors***: definindo as estruturas abstratas na classe *Behavior* a serem especializadas em ações executadas por um jogo;
- ***observers***: definindo estruturas abstratas (*Observer*, *InputObserver*, *CollisionObserver* e *CollisionGroupObserver*) de tratamento de eventos acionados em um jogo; e
- ***adapters***: declarando classes concretas que renderizam objetos instanciados do MEnDiGa framework em motores de jogos distintos.

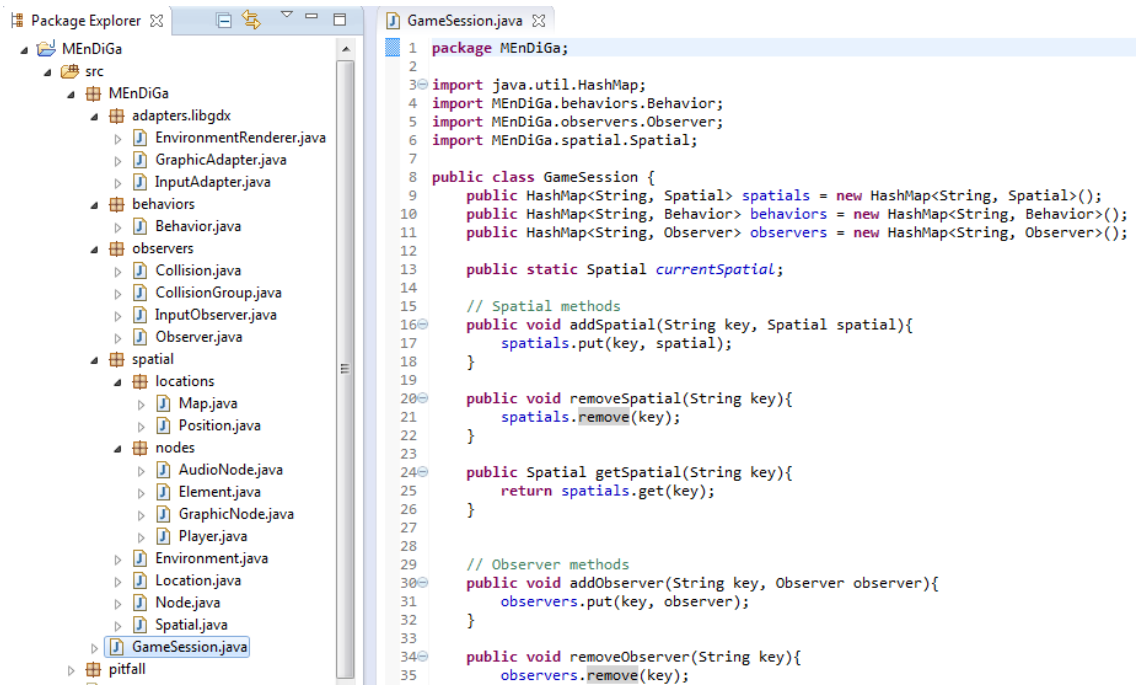


**Figura 4. Classe *GameSession* e subpacotes do MEnDiGa framework.**

Com relação às classes definidas para cada subpacote do MEnDiGa framework, tem-se na classe *Spatial* uma coleção de objetos *Node* instanciados, os quais são compostos por instâncias de *AudioNode*, *GraphicNode* e *CurrentLocation* específicas. Já a classe *Element*, esta é implementada como sendo uma subclasse de *Node* que possui uma coleção de instâncias *Property* representadas em uma estrutura *HashMap*.

Definiu-se a classe *Player* como sendo uma subclasse de *Element*, sendo esta composta por um *controlMap* que contém a coleção de instâncias *Behavior* relacionadas à constantes representativas de comandos de um usuário. *Environment* também foi definido como uma especialização do *Spatial*, sendo esta composta por uma coleção de *locations* do tipo *Map* ou *Position* (o espaço do jogo).

Para cada instância de *Observer* tem-se uma coleção de instâncias *Behavior*, as quais são executadas de acordo com a forma de avaliação programada para cada possível evento de um jogo. Cada *Behavior* é programado para executar uma determinada ação do jogo, modificando atributos de instâncias de *nodes*, alterando o *Spatial* atual do *GameSession* em cada ciclo de renderização do jogo.

Com relação às classes definidas em *adapters,* estas são responsáveis pela execução dos objetos MEnDiGa configurados nos recursos de implementação de um motor de jogo escolhido. Para cada motor de jogo trabalhado, um novo conjunto de classes *adapters* é modelado uma única vez, garantindo assim a portabilidade de um jogo MEnDiGa para diferentes motores de jogos existentes.

## 4. Estudo de Caso –"Pitfall!"

Uma versão do jogo "Pitfall!" [Kohler 2010] está sendo desenvolvida para demonstrar a viabilidade de produção de jogos com o MEnDiGa. Embora ainda seja um trabalho em progresso, a explanação de seu desenvolvimento permite o uso e as configurações de classes do MEnDiGa framework.

### 4.1. Modelando o jogo "Pitfall!"

No "Pitfall!" original, o personagem percorre vários cenários em busca de tesouros. Nestes cenários, o jogador encontra diversos obstáculos e armadilhas como poços, barris de madeira e até animais selvagens. O jogador pode pular, subir/descer escadas e se balançar em cipós para superar obstáculos. Para cada "ferimento" sofrido, o jogador é penalizado com reduções na pontuação ou no número de vidas correntes. O jogo termina ou com vitória após coletar todos os tesouros em um certo limite de tempo, ou com derrota caso falhe em superar os obstáculos.

Na versão MEnDiGa do "Pitfall!", os cenários do jogo são representados por instâncias de *Environment*, onde cada uma conterá uma possível instância de *Map* para a renderização do ambiente do jogo. *Map* contém informações sobre a geometria gráfica do *Environment*, localização de poços e escadas, obstáculos e armadilhas renderizadas, etc. Para localizar os *nodes* no *Map* será utilizada uma instância de *Position*, a qual contém as coordenadas cartesianas de um *Node* e a informação de profundidade usada para definir a sobreposição de imagens no *Environment*.

Os tesouros, animais e os troncos de madeira serão representados por especializações da classe *Element* ou *Node*. Objetos estacionários como a fogueira e a cobra que não possuem propriedades (*Property*) aparentes, serão representados por subclasses da classe *Node*. Já os objetos como o escorpião e os barris que demonstram possuir uma propriedade como a velocidade, estas serão representados por subclasses da classe *Element*. Este mapeamento acima descrito de elementos do jogo "Pitfall!" para artefatos MEnDiGa, propostos dentro da ótica de modelagem da subfeature *Spatial*, está ilustrado na Figura 5.

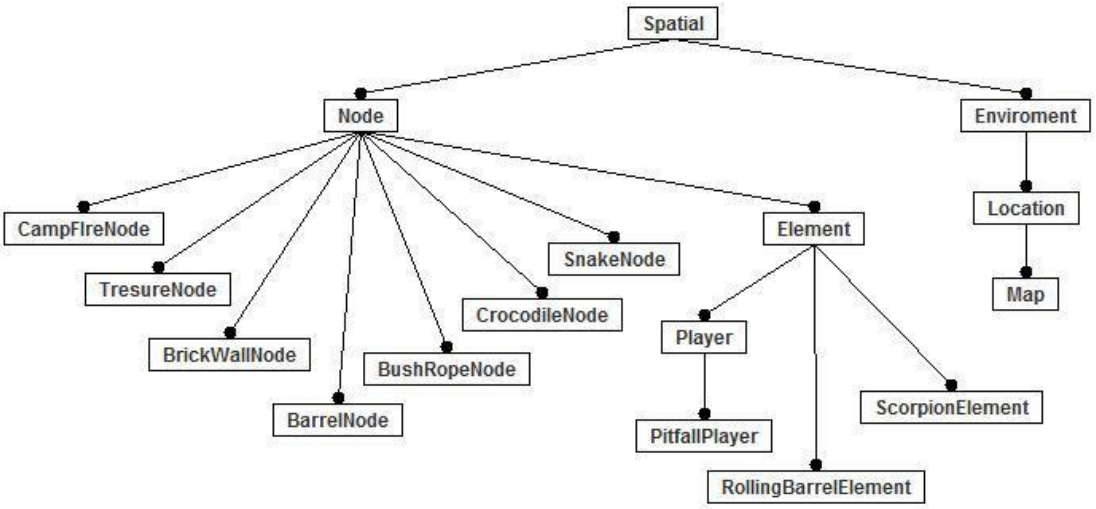


**Figura 5. Diagrama da subfeature *Spatial* para o jogo "Pitfall!"**

Considerando o tratamento de eventos de um jogo por parte dos recursos MEnDiGa desenvolvidos, tem-se uma série de colisões entre elementos do "Pitfall!" e eventos gerados pelo jogador que precisam ser tratadas por instâncias *Observer* específicas. O diagrama contendo algumas destas features *Observer* de colisão e de monitoramento de ações do jogador, que serão implementadas para a versão MEnDiGa do jogo "Pitfall!", pode ser visto na Figura 6.

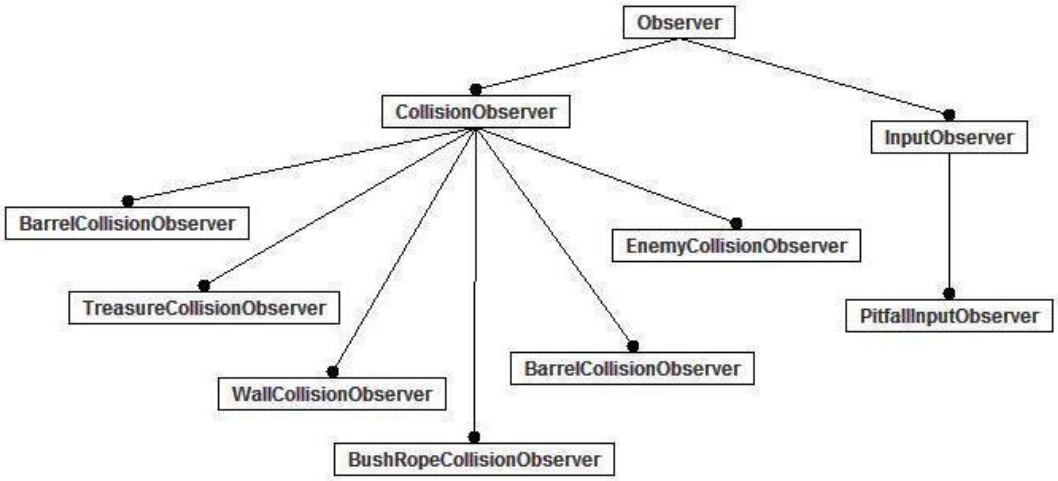


**Figura 6. Diagrama da subfeature *Observer* para o jogo "Pitfall!"**

Considerando as ações responsáveis pela alteração das instâncias *Spatial* de um jogo MEnDiGa, ativadas ou pelo personagem jogável (andar, pular, escalar, etc.) ou pela dinâmica do jogo em si (aumentar/diminuir pontuação, ganhar/perder vida, etc.), estas são representadas no MEnDiGa por objetos *Behavior*. Algumas destas features *Behavior* de jogabilidade e dinâmica do jogo, que serão implementadas para a versão MEnDiGa do jogo "Pitfall!", estão ilustradas na Figura 7. Vale salientar que tais instâncias *Behavior* são executadas no MEnDiGa por objetos *Observer* quando da confirmação de um evento monitorado (*BUTTON1_PRESS -> JUMP*, por exemplo).

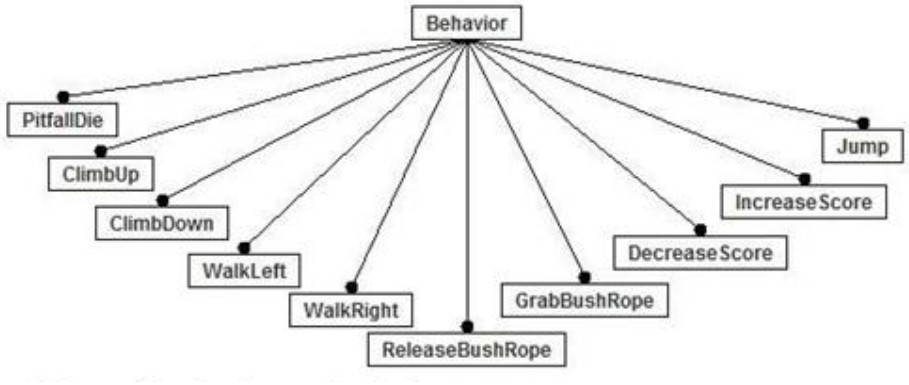


**Figura 7. Diagrama da subfeature *Behavior* para o jogo "Pitfall!"**

## 4.2. Implementando o Jogo "Pitfall!"

Com a identificação e produção dos componentes necessários para a criação do "Pitfall!" via recursos MEnDiGa (Figuras 5, 6 e 7), segue-se para a etapa final de produção do jogo que consiste na configuração de tais componentes para caracterizar a dinâmica do "Pitfall!" e na adaptação dos mesmos para um motor de jogo específico.

Neste sentido, a classe *PitfallGame* especializa a classe *GameSession* para configurar o jogo "Pitfall!". Esta classe é programada para instanciar e configurar o *Environment* inicial, o *Map* contendo as informações do arquivo de mapa a ser renderizado, o *PitfallPlayer* e seus comandos (*MOVE_LEFT->WalkLeft*, *MOVE_RIGHT->WalkRight*), o *PitfallInputObserver* e os *behaviors* disponíveis.

*PitfallGame* procura adicionar *Map* e *PitfallPlayer* ao *Environment* inicial do jogo (*currentSpatial* da *GameSession*), juntamente com os demais elementos do respectivo *Environment*. Classes *Observer* e *Behavior* também são instanciadas no *PitfallGame,* a exemplo do *PitfallInputObserver* que recebe como parâmetro do construtor o *PitfallPlayer* corrente da partida do jogo.

*PitfallPlayer* também é responsável por configurar as características do personagem jogável, tais como: nome do jogador, posição inicial (*Position*), *properties* como *speed* e *jumpStrength* (ambas com valores numéricos), e configurações de representação gráfica (*GraphicNode*). As *properties* são utilizadas por instâncias *Behavior* para alterar o *position* atual do *PitfallPlayer*. Já o *GraphicNode*, este contém informações sobre o *sprite sheet* escolhido, o que permite compor animações gráficas para o personagem via escolha e temporização de sprites a serem renderizados, conforme a lógica de posicionamento e configuração ilustrada nas Figuras 8 e 9.

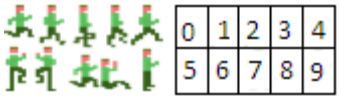


**Figura 8. Sprite sheet e seus índices para o personagem do "Pitfall!"**

Com relação ao *PitfallInputObserver*, este foi configurado para avaliar os comandos ativados pelo usuário e executar o *Behavior* apropriado (contido no *controlMap* do *Player*) de acordo com o estado atual do *PitfallPlayer*. De fato, certos comandos não devem ser executados a qualquer momento segundo a dinâmica do jogo, a exemplo de *WalkLeft* e *WalkRight* cujo *PitfallPlayer* deve ser incapaz de executar enquanto estiver executando o *Behavior Jump*.

```
this.graphicNode = new GraphicNode(
        //Caminho do arquivo de imagem contendo os frames
        "sheet4.png",
        //Altura e largura da área da imagem que será utilizada
         80, 44,
         // Numero de linhas e colunas
         2, 5
        );
// Redimensionando os quadros
this.graphicNode.resize(48, 56);

// Define o quadro atual do personagem do "Pitfall!"
this.graphicNode.setCurrentFrame(9);

// Definindo uma animação
this.graphicNode.defineAnimation(
        WALK_ANIMATION, // Chave da animação
        new int[]{0, 1, 2, 3, 4}, // Quadros
        0.07f, true); // Delay entre os quadros e flag de looping
```

**Figura 9. Configurando o sprite sheet no Graphic Node do PitfallPlayer**

### 4.3. Adaptando o Jogo "Pitfall!"

Com a finalização de uma proposta inicial de configuração de recursos MEnDiGa para o jogo "Pitfall!", o passo seguinte consiste na integração das classes de configuração "Pitfall!" com as classes de adaptação do MEnDiGa (pacote *adapters* na Figura 4). Tais classes de adaptação buscam estabelecer um padrão de integração entre as configurações MEnDiGa de um jogo qualquer com os recursos de implementação de um motor de jogo desejado.

LibGDX [2014] vem sendo utilizado neste projeto como motor de jogo alvo para as classes de adaptação MEnDiGa. Trata-se de um framework open-source que apresenta uma boa aceitação no mercado para a produção profissional de jogos Java multiplataforma. Dentre as classes de adaptação disponíveis no MEnDiGa para a LibGDX, podemos destacar: *EnvironmentRenderer, GraphicAdapter* e *InputAdapter*.

Para cada instância de *Environment* no *PitfallGame*, um *EnvironmentRenderer* é criado. Ele é responsável por carregar e gerenciar a renderização dos elementos do jogo especificados nas configurações das instâncias *Map* e *GraphicNode* correspondentes. Ele também segue a ordem de renderização de instâncias *GraphicNode* conforme a profundidade informada em cada *Node* correspondente (componente z do *Position*).

Para cada *GraphicNode* a ser renderizado, um *GraphicAdapter* é criado com base nas informações do *GraphicNode* para carregar e animar o arquivo de imagem especificado. O *sprite sheet* é dividido e seus quadros são armazenados em um *array* de índices de *sprites*, permitindo assim a manipulação gráfica ilustrada na Figura 8. No processo da adaptação, as informações de animação definidas no *GraphicNode* são utilizadas para criar *Animations* – objetos do LibGDX capazes de gerenciar animações.

O *InputAdapter* é a classe responsável por notificar o *PitfallInputObserver* sobre os eventos de entrada. Ela implementa o *InputProcessor*, interface fornecida pelo LibGDX para receber eventos de entrada do teclado, *touchscreen* ou mouse. O *InputAdapter* possui um *adaptedControlMap* que faz a correspondência entre os comandos do MEnDiGa e as constantes representativas de teclas do LibGDX. Se a tecla

pressionada ou liberada faz correspondência com algum comando do MEnDiGa, o *InputAdapter* notifica qual comando foi acionado ou liberado ao *PitfallInputObserver*, que executa um *playerBehavior* apropriado ao evento.

Para efetuar a configuração final de conversação entre os recursos MEnDiGa do jogo e a LibGDX, criou-se a classe *PitfallLibGDX*. Ela trabalha com a interface *AplicationListener* da LibGDX, que contém os métodos chamados durante a execução da aplicação (criação, renderização, resumo). A classe *PitfallLibGDX* implementa essa interface, sobrescrevendo o método *create* para que o *PitfallGame* e os *adapters* necessários sejam criados no momento que a *Aplication* é lançada. Associações entre instâncias *Observer* do *PitfallGame* e do pacote *adapter* (*InputAdapter*, por exemplo) também são configuradas durante a criação de *PitfallLibGDX*.

Como resultado final, basta executar a *PitfallLibGDX* para que a LibGDX inicie seus processos de inicialização, renderização e atualização constante dos recursos MEnDiGa para a exibição do jogo final modelado.

## 5. Conclusão

Este artigo apresentou o MEnDiGa, uma proposta de motor de jogo baseado na simplificação dos modelos de features NESI e GDS para jogos casuais. Para tal, descreveu-se o modelo de features MEnDiGa proposto bem como sua implementação no MEnDiGa framework, estrutura esta responsável pelo projeto da variabilidade identificada para jogos casuais.

Uma implementação de um clone do clássico "Pitfall!" também foi demonstrada neste artigo. Trata-se de uma etapa importante deste projeto na tentativa de mostrar a viabilidade dos artefatos MEnDiGa produzidos na geração de jogos digitais concretos. Apesar do "Pitfall!" ter sido desenvolvido para a antiga plataforma Atari, este apresenta mecânicas, dinâmicas e estéticas similares a diversos jogos digitais da atualidade, garantindo assim uma validação eficaz e eficiente das potencialidades do MEnDiGa na produção de jogos digitais diversos.

Como trabalhos futuros para o MEnDiGa, pretende-se implementar diferentes jogos casuais da atualidade, produzindo assim uma coleção de features e componentes reutilizáveis capazes de garantir ao MEnDiGa o status de Linha de Produto de Jogos Digitais Casuais. Com a contínua evolução do MEnDiGa, pretende-se também implementar hierarquias de features para diferentes categorias de jogos digitais, bem como criar novas classes de adaptação do MEnDiGa para diferentes motores de jogos disponíveis.

## 6. Agradecimentos

Os autores gostariam de agradecer à Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB), por financiar o projeto de pesquisa e conceder a bolsa de iniciação científica ao graduando e coautor deste trabalho, Filipe Macêdo Borges Boaventura, tornando possível a realização do mesmo.

## Referências

Lewis, M. and Jacobson, J. (2002) "Games engines in scientific research", *Communications of the ACM, vol. 45, no. 1,* p 21.

BinSubaih, A. and Maddock, S. (2008) “Game Portability Using a Service-Oriented Approach”, In: International Journal of Computer Games Technology, Volume 2008, Article ID 378485, 7 pages. Hindawi Publishing Corporation, doi:10.1155/2008/378485.

Sarinho, V. and Apolinário, A. (2008) “Feature Model Proposal for Computer Games Design”, In: Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment, Belo Horizonte, p. 54-63.

Sarinho, V. and Apolinário, A. (2009) “A Generative Programming Approach for Game Development”, In: Proceedings of the VIII Brazilian Symposium on Computer Games and Digital Entertainment, Rio de Janeiro, p. 9-18.

Kang, K., Cohen, S., Hess, J., Novak, W. and Peterson, S. (1990) “Feature-Oriented Domain Analysis (FODA): Feasibility Study”, *CMU/SEI-90-TR-21, SEI, USA*.

Antkiewicz, M. and Czarnecki, K., (2004). “FeaturePlugin: feature modeling plug-in for Eclipse” *In: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange, October 24th to 28th, 2004, Vancouver,Canada, pp 67-72*. New York, USA, ACM Press.

Kohler, Chris (2010). "Pitfall! creator David Crane named videogame pioneer". *Wired*. Retrieved 7 November 2010.

Fayad, E. Schmidt, C and Johnson R. (1999) “Building Application Frameworks Object-Oriented Foundations of Framework Design”. John Wiley Sons.

Czarnecki, K. (2004) “Overview of Generative Software Development”. In: Proceedings of Unconventional Programming Paradigms (UPP), pp 313-328, Springer-Verlag.

Java. Oracle Technology Network. Available from: http://www.oracle.com/technetwork/java/index.html [Accessed 30 March 2014].

LibGDX. Desktop/Android/BlackBerry/iOS/HTML5 Java game development framework. Available from: http://libgdx.badlogicgames.com [Accessed 30 March 2014].

Sarinho, V. and Apolinário, A. (2010) “Combining Feature Modeling and Object Oriented Concepts to Manage the Software Variability”. In: IEEE International Conference on Information Reuse and Integration (IRI), Las Vegas, USA, pp. 344-349.

Sarinho, V. Apolinário, A and Almeida, E. (2011) “A Feature-based Environment for Digital Games.” In: 1st Workshop on Game Development and Model-Driven Software Development, in conjunction with 10th International Conference on Entertainment Computing (ICEC), Vancouver.

Sarinho, V. Apolinário, A and Almeida, E. (2012) “OOFM - A Feature Modeling Approach to Implement MPLs and DSPLs”. IEEE IRI 2012, August 8-10, Las Vegas, Nevada, USA, pp 740-742.

Sarinho, V. and Apolinário, A. (2013) “Detailing the UML Profile of the OOFM Technique”. In: Proceedings of 3rd Brazilian Workshop on Model Driven Development (WB-DSDM 2012), v. 8, pp 25-32, 2012, ISSN:2178-6097.