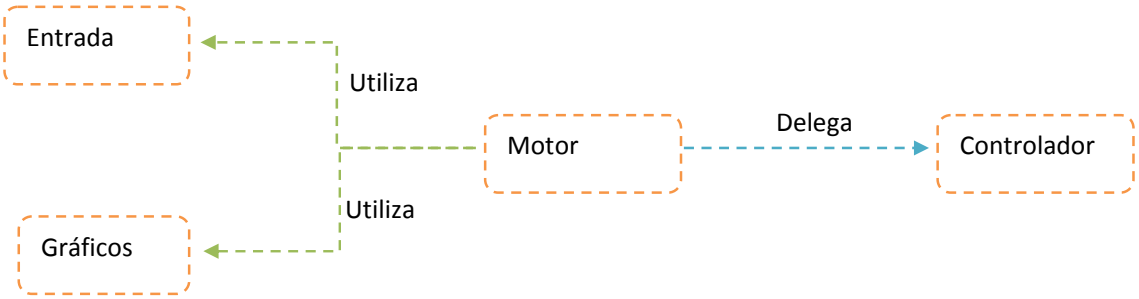
APÊNDICE D
O Projeto do javaPlay

Este apêndice documenta os princípios de projeto do motor especialmente desenvolvido para este livro, o motor 2D javaPlay.

D.1 Arquitetura Geral

O engine se divide em módulos responsáveis por tarefas específicas, chamados **sistemas**. Dentre essas tarefas temos: Controle de *loop* principal, desenho e verificação de *input* por parte dos jogadores, etc. A figura abaixo mostra a interação desses sistemas.



Tais sistemas possuem diferentes representações e podem ser definidos por um conjunto de classes ou por uma classe apenas. Estes sistemas são definidos apenas para fornecer uma separação de tarefas e uma organização de alto nível para o *engine*.

O sistema de aplicação é o sistema central. Ele se utiliza dos outros sistemas e delega tarefas a outros capazes de realizar lógica específica de jogo.

D.2 Sistema Motor

O sistema Motor é responsável pelo controle do *loop* principal de um jogo. Ele acessa outros sistemas que realizam tarefas específicas. O sistema de aplicação é responsável pela inicialização de um programa e dos outros sistemas, bem como do término do programa e finalização correta dos outros sistemas.

Ele define um conjunto de funções pré-definidas que permitem ao programador de aplicação controle sobre a inicialização, o término e as ações ocorridas para cada *frame* de jogo.

O sistema de aplicação possui uma classe principal *singleton* que permite com que as diferentes partes do sistema possuam visibilidade das classes mais importantes como o sistema gráfico e o sistema de entrada e saída. Essa classe se chama **GameEngine**. **GameEngine** possui os seguintes métodos:

```
public static getInstance()
public Keyboard getKeyboard()
public Mouse    getMouse()
public GameCanvas getGameCanvas()

public void addGameStateController(StateController,int controllerId)
public void removeGameStateController(int controllerId)
public void setNextGameStateController(int controllerId)
```

O método estático getInstance retorna a instância de *singleton* da aplicação. Dessa maneira, o acesso desse objeto, em basicamente qualquer ponto da aplicação, se torna trivial. getKeyboard e getMouse

retornam os objetos de controle de entrada de usuário. getGraphicsManager retorna o objeto GraphicsManager corrente, permitindo acesso a, por exemplo, o contexto de renderização corrente.
A inicialização e as funções executadas ao término da aplicação são associadas a uma interface denominada GameStateController.

D.3 Sistema de Estado

O sistema de estado é o principal responsável pela flexibilidade do sistema. Enquanto a classe de aplicação encapsula as funções de baixo nível associadas a plataforma e aos métodos globais de controle, o sistema de estado permite que o programador tenha um controle fino das ações que deverão ser apresentadas ao usuário final.

Jogos apresentam diferentes requisitos lógicos de acordo com o estado em que se encontram. Por exemplo, a lógica necessária para tratar o menu principal de jogo difere grandemente da lógica de jogo principal, onde o usuário de fato controla seu avatar. Para o sistema de aplicação, ambos estados são idênticos, uma vez que este tem como função apenas preparar o ambiente para a execução coerente de um novo *frame*. A lógica que irá ser processada nesse *frame* é baseada em um estado derivado da classe **GameState**.

A classe GameStateController é abstrata e permite a execução de lógica arbitrária. Como o sistema de aplicação, ela divide a execução em *time slices*, permitindo ao programador o controle de eventos específicos como processamento de *frame*. Os principais métodos dessa classe são:

```
public boolean load()
public boolean unload()
public boolean start()
public void step(int timeElapsed)
public void draw()
public boolean stop()
```

Os métodos load e unload são invocados quando o estado deve ser carregado ou descarregado da memória. Todos seus recursos devem passar pelo mesmo processo.

Os métodos start e stop são invocados quando o controlador de estado atual deve ser modificado mas seus recursos não devem ser liberados. Esse cenário é comum quando se deseja que uma lógica específica seja executada mas a troca deve ser rápida, não existindo a possibilidade de pausa para carregamento de recursos.

Os métodos step e draw permitem o controle fino da lógica associada a um *frame* específico. O sistema de aplicação invoca o método step passando o tempo em millisegundos decorrente entre o *frame* atual e o *frame* anterior, permitindo assim animações baseadas em tempo. Nesse método o programador deve atualizar o estado de todos os objetos de jogo relevantes e prepará-los para a renderização.

Após a invocação do método step, o sistema de aplicação chama o método draw do controlador atual. Este, por sua vez, permite ao programador renderizar seus objetos de forma customizada, ou repassar a uma entidade de alto nível como a classe Scene.

D.4 Objetos de Jogo

O engine provê também uma classe base utilizada para qualquer objeto que necessite de lógica específica de atualização.

A classe **GameObject** é uma classe abstrata que define um objeto com posição no espaço e a capacidade de receber mensagens de atualização e realizar operações de desenho customizadas. A classe GameObject é definida por:

```
public int x,y;
public void step(int timeElapsed)
```

```
public void draw()
```

É fácil perceber que essa classe por si só não apresenta muitas funcionalidades. Ela serve apenas como um ponto de partida. É de responsabilidade das classes derivadas adicionar atibutos, como um objeto sprite que representa o gráfico do avatar e métodos específicos.

D.5 Sistemas de Entrada

O sistema de entrada permite acesso ao estado do teclado e mouse para uma determinada aplicação. As principais classes desse sistema são Keyboard e Mouse. Os métodos publicos da classe Keyboard são:

```
public boolean keyPressed(byte keyCode)
public boolean keyReleased(byte keyCode)
public boolean keyDown(byte keyCode)
```

O método keyPressed permite verificar se uma tecla foi pressionada no *frame* corrente de execuação. keyReleased verifica se a tecla deixou de ser pressionada no *frame* corrente. keyDown verifica o estado da tecla independente do último *frame* em que este foi modificado.

Os métodos da classe Mouse são:

```
public Point mousePos()
public boolean isLeftButtonPressed()
public boolean isMiddleButtonPressed()
public boolean isRightButtonPressed()
public Point getMousePos()
```

O método getMousePos retorna um objeto do tipo Point com as coordenadas x e y da ultima localização do mouse antes de entrar no *frame* corrente. isLeftButtonPressed verifica se o botão esquerdo do mouse se encontra pressionado. isRightButtonPressed faz o mesmo para o botão direito.

D.6 Sistema Gráfico

O sistema gráfico deve se dividir em duas camadas básicas: Graphics Manager e a Scene Manager. A camada Graphics Manager é responsável pela interação do *engine* com as camadas de desenho da plataforma.

A camada GraphicsManager é definida para ser o mais leve possível, sendo capaz de permitir a integração dos métodos de desenho em um método controlado pelo sistema de aplicação ao invés de eventos assíncronos enviados pelo *runtime* Java.

Ela fornece acesso aos objetos AWT, permitindo que o usuário tenha acesso ao contexto gráfico corrente e possa utilizar os métodos de desenho encontrados nas classes Graphics e Graphics2D

A classe Scene define uma abstração de alto nível para criação de jogos baseados em mundos formados por blocos conhecidos como **Tiles**. Esse modelo de organização de mundo é o mais comum encontrado em jogos 2D. Essa classe divide o mundo em sub-camadas distintas:

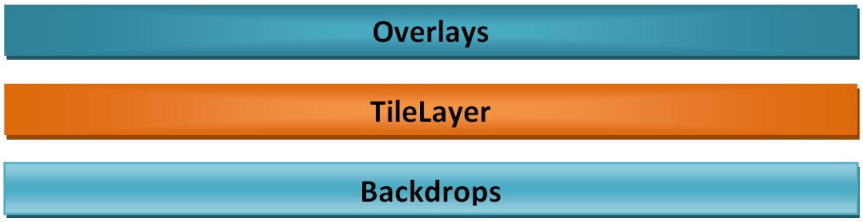
- Backdrops
- TileLayer
- Overlays.

Backdrops são os cenários de fundo onde o usuário possui pouca ou nenhuma interação, servindo na maioria das vezes como itens de adorno gráfico.

TileLayer é a sub-camada que apresenta o conjunto de blocos que define o mundo ao qual um avatar é inserido. Ela fornece métodos para detecção de colisão com os blocos do mundo.

A sub-camada de Overlays representa o local onde os sprites dos objetos dinâmicos de jogo, controlados pelo jogador, ou inteligência artificial são desenhados. Esses objetos são representados por sprites. Um overlay pode ser entendido como uma camada inicialmente transparente onde os sprites do avatar, inimigos, etc são desenhados.

A forma como Scene organiza suas sub-camadas é representada na figura abaixo:



A classe é capaz de realizar operações de *blending* das camadas, e permite a renderização em ordem correta de profundidade dos objetos: objetos de overlay se situam à frente dos objetos de tile que por sua vez estão à frente dos backdrops.

Os métodos encontrados nessa classe são:

```
public static Scene loadFromFile(string sceneFile)
public void addOverlay(GameObject overlay)
public void removeOverlay(GameObject overlay)

public Vector getTilesFromRect(Point min, Point max)

public void draw()
public void step(timeElapsed)
```

Os métodos addOverlay, overlays e removeOverlay permitem a inserção, deleção e busca de objetos de overlay respectivamente. Métodos análogos são disponíveis para backdrops.

O método step(int timeElapsed) é invocado pelo StateController e invoca step de cada um dos objetos de jogo associados ao SceneManager. O método draw por sua vez desenha os backdrops, conjunto de tiles e conjunto de overlays.

O método getTilesFromRect recebe 2 posições representando um retângulo e retorna os tiles ocupados por esse retângulo. Com isso é possível saber quais são os tiles que um determinado objeto ocupa e, através dessa informação, decidir se há colisão, se é necessário algum tipo de processamento especial, etc.

D.7 Sprites

A classe Sprite representa os principais objetos gráficos a serem manipulados em jogos 2D. Os sprites representam em geral todas as poses que um objeto pode ficar em suas animações. Sem sprites animados não seriam possível realizar as complexas animações de personagens vistas em jogos. A classe sprite possui a seguinte interface:

```
public x,y;
public int setCurrAnimFrame(int frame)
public void draw(Graphics g)
public Sprite clone()
```

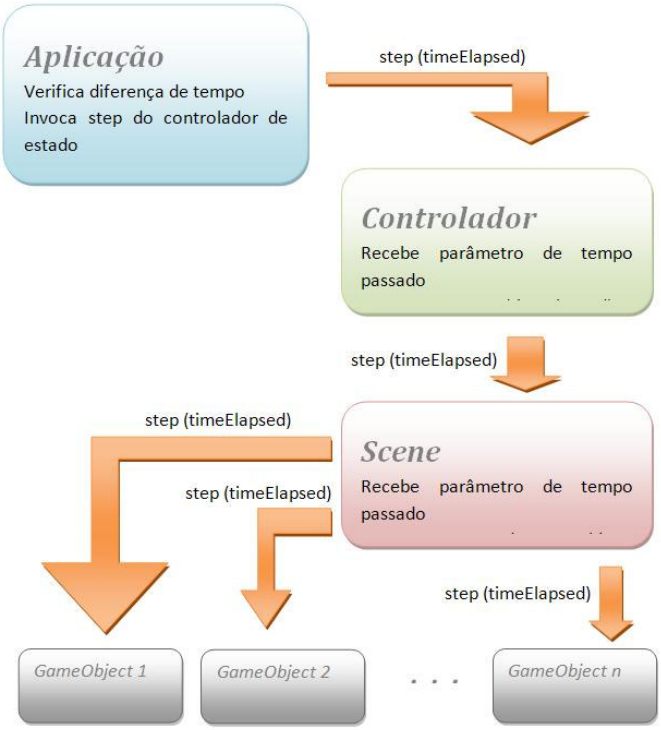
Os atributos x e y definem a posição do sprite no canvas de desenho. getCurrentAnimFrame e setCurrentAnimFrame definem o *frame* atual de animação. getAnimFrameRect retorna o retângulo ao qual o *frame* corrente pertence. Por exemplo, se cada frame de animação possui altura 90 pixels e

largura 20 pixels, os extremos do retângulo associado ao frame 2 (assumindo que a contagem de *frames* começa em 0) é [(40,0),(60,90)].

Como comodidade, é fornecido um método draw que desenha o *frame* atual de animação com origem no ponto (x,y) no contexto de renderização associado ao parâmetro g.

D.8 Exemplo de Loop

O *loop* principal do *engine* se inicia dentro da classe MainApplication que possui controle da *thread* principal de jogo. Ela verifica a diferença de tempo entre o *frame* anterior e o *frame* atual em milisegundos, passa a execução para o método step do controlador de estado corrente. O controlador então pode executar sua lógica de atualização. Se o controlador tiver um objeto de alto nível associado (como, por exemplo, uma instância da classe Scene), ele deve invocar o método step de cena que por sua vez irá atualizar cada um de seus GameObjetcs.



É importante lembrar que o modelo aqui utilizado é semelhante ao paradigma de *time sharing*: Cada objeto recebe um "quantum" de tempo e este deve utilizar esse parâmetro para atualizar seu estado corretamente. Esse procedimento é repetido para cada objeto até que todos tenham sido atualizados. Tal lógica difere de atualizações feitas através de iterações tradicionais em laço do tipo *while* ou *for*, no qual um objeto realizaria toda sua atualização sequencialmente.