

UNIVERSIDADE PAULISTA
BACHAREL EM CIÊNCIA DA COMPUTAÇÃO

DIEGO FREIRE DE ALMEIDA, N8059D2

KAIKY DE LARA SALES, N9218H8

LEONARDO DE SOUZA RODRIGUES, F344HB2

NICOLAS PIMENTA DA SILVA, N863579

VITOR DOS SANTOS ROSA, G521CE3

EcoRunner: uma implementação lúdica, didática e conscientizadora

Jundiaí – SP

2023

DIEGO FREIRE DE ALMEIDA, N8059D2

KAIKY DE LARA SALES, N9218H8

LEONARDO DE SOUZA RODRIGUES, F344HB2

NICOLAS PIMENTA DA SILVA, N863579

VITOR DOS SANTOS ROSA, G521CE3

EcoRunner: uma implementação lúdica, didática e conscientizadora

Artigo de atividades supervisionadas,
apresentado ao Curso de Ciência da
Computação para composição de nota
referente ao 3º semestre letivo.

Orientador: Prof. Nathan Cirillo E Silva

Jundiaí – SP

2023

INDÍCE

1 OBJETIVO E MOTIVAÇÃO DO TRABALHO	4
2 INTRODUÇÃO	5
3 CONCEITOS GERAIS	7
3.1 Endless Running (Conceito e história)	7
3.2 Regras e funcionamento do jogo	9
4 PLANO DE DESENVOLVIMENTO DO JOGO	11
4.1 Pixel art	11
4.2 Processo criativo	12
4.3 Tecnologias utilizadas.....	15
4.3.1 Programação Orientada a Objetos – Java	16
4.3.2 Swing	17
5 PROJETO DO PROGRAMA.....	18
5.1 Classe “Personagemrincipal”	18
5.2 Métodos Comuns entre as Classes do Domínio	19
5.3 Métodos.....	21
5.4 Classe: GerenciadorInimigos	23
6 RELATÓRIO COM AS LINHAS DE CÓDIGO	29
7 BIBLIOGRAFIA	36

1. OBJETIVO E MOTIVAÇÃO DO TRABALHO

Criar um jogo com interface gráfica na linguagem Java, consistido em um sistema de plataforma e rolagem lateral baseado em comandos básicos e objetivos e mostrar também toda a etapa de desenvolvimento, soluções criativas e apresentar as tecnologias utilizadas. Motivado pelo interesse proveniente da proposta do trabalho, em mostrar que a experiência de desenvolvimento e aplicação de um projeto com conceito essencialmente lúdico pode ser muito didático mesmo que aliado a uma temática ecológica/sustentável, em particular, educação ambiental, tendo como base a vida em uma grande metrópole. Tema aparentemente pedante, porém, se bem abordado e interpretado por quem o faz, bastante reflexivo, esteticamente interessante e socialmente utilitário.

2. INTRODUÇÃO

No cenário da social atual, existe uma evidente construção de imagem que se tornou impregnada ao senso comum e a concepção das pessoas de modo geral, este, que se fere ao modo que o mundo enxerga o desenvolvimento, o universo da chamada programação de softwares e aplicações. Que tudo o que se refere a tecnologia, não passa de um punhado de coisas que integram um ecossistema baseado em experiências penosas e sem estímulo criativo, preenchido por pessoas exageradamente pragmáticas e metódicas motivadas, ainda que de forma artificial, por mero planejamento ou consecutivas desesperadas tentativas de se manterem atualizadas e competitivas dentro desse mercado que além de predatório, não é de fato justo com absolutamente ninguém. Os programadores, são peças de uma linha de produção onde podem ser substituídos como uma mera bobina ou pequena engrenagem, onde a produtividade cega e incessante é primordial e sumário, os projetos: sempre tem como objetivo final dar corpo e funcionalidade a uma aplicação, na grande maioria baseado em PDV, ERP ou em algum sistema de gestão de dados e informação, cujo objetivo final é apenas possuir serventia corporativa e/ou comercial, onde tudo, desde o processo de planejamento até a implementação e desenvolvimento do mesmo, se baseia na execução de uma série de procedimentos engessados e, em certos termos, maçante.

Entretanto, em uma realidade onde o auxílio de *chats* de texto inteligentes, IDEs cada vez mais coloridas, intuitivas e variadas com seus *plugins* que visam uma produtividade amistosa e possibilidades fantásticas com a utilização de linguagens orientadas a objeto são reais e palpáveis, observa-se ali uma possibilidade criativa muito evidente, onde o manuseio dessas tecnologias, pode ser direcionado para o entretenimento. Onde conceitos estéticos e artísticos compõem uma excepcional amalgama com as implementações advindas da utilização de linguagens de programação, especialmente, as orientadas a objeto: onde o mundo passa a ser enxergado de forma abstrata, como uma enorme coleção de objetos. Sendo assim, as possibilidades são inúmeras, podendo nesse contexto, até mesmo unir o útil ao agradável, como por exemplo: passar uma mensagem sobre conscientização ecológica e sustentável em um jogo simples, divertido e objetivo.

Neste artigo, será mostrado que essa possibilidade é, além de um certo sucesso em termos de implementação, como também impressiona no que se refere a relação entre: quantidade de recursos utilizados e tempo investido e os retornos obtidos através deles. Onde, por meio de uma equipe homogênea e sinérgica, a criação de um jogo no estilo Endless Running com temática sustentável, visando abordar o assunto “educação ambiental” em meio à uma realidade estritamente urbanizada e constituída em metrópoles, por parte de alunos entusiastas e amadores no manuseio de linguagens orientadas a objeto, não só é, como foi possível.

3. CONCEITOS GERAIS

O jogo consiste em basicamente reproduzir mais um exemplo do estilo de jogos *Endless Running*, gênero muito popular em meados de 2010 principalmente em dispositivos mobile que traz um conceito simples e objetivo para uma experiência lúdica e de incisivo fator *replay*. Com o personagem jogável em um cenário, em certos termos, “infinito”, saltar e agachar enquanto diversos obstáculos impedem o pequeno corredor em tela de prosseguir e realizar a coleta seletiva de pequenos itens ecológicos, serão as principais ações responsáveis por garantir que o jogador usufrua de forma positiva e integral a experiência de jogar o *game*.

3.1. Endless Running (Conceito e história)

O estilo de jogo se trata de um subgênero dos jogos de plataforma no qual o personagem corre em um único sentido de forma linear em uma rolagem unidirecional onde o seu principal objetivo é acumular a maior quantidade de pontos o possível, seguindo a linha dos jogos tipificados como *arcade*, onde não há objetivo fundamentado em uma linha do tempo com começo, meio e fim, há apenas o foco em compor um “placar” executando determinadas ações ou apenas “correndo”. Como exemplos claros, temos os sucessos *Temple Run* e *Jetpack Joyride* (figura 1 e 2 respectivamente), ambos muito populares até hoje.



Figura 1: Temple Run



Figura 2: Jetpack Joyride (2011)

O gênero, possivelmente foi fecundado no seu potencial predecessor, o jogo no estilo de rolagem horizontal projeto inicialmente para as plataformas *Spectrum*, *MSX* e *Apple II* chamado *BC's Quest for Tires* de 1983 (Figura 3).



Figura 3: *BC's Quest for Tires* (1983)

No *game* desenvolvido por Johnny Hart, o jogador assume o papel de um homem das cavernas chamado Thor, cuja missão é resgatar sua namorada que fora sequestrada por um dinossauro. Para fazê-lo, ele deve viajar em seu monociclo de pedra até seu destino movendo-se para direita e para a esquerda evitando perigos e obstáculos. Nessa época, o conceito de uma “viagem sem fim” do personagem jogável, ainda não existia, porém, a composição de um placar de pontos enquanto movia-se bilateralmente, já dava seus primeiros passos na indústria dos jogos.

Entretanto, embora um sucesso comercial na época, o jogo não teve seu modelo de *level design* copiado. Os *endless run game* só se firmaram como um subgênero dos estilos plataforma no ano de 2009 com o lançamento do jogo independente *Canabalt* (Figura 4), desenvolvido por Adam Saltsman para as plataformas portáteis iOS e PSP.

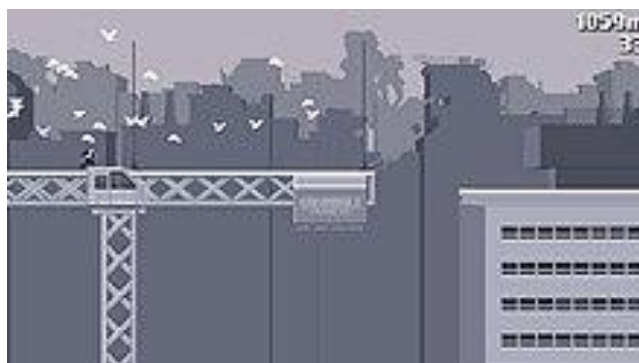


Figura 4: *Canabalt* (2009)

3.2. Regras e funcionamento do jogo

Falando agora sobre o projeto e sua versão final, jogável e vistosa. Não há nada de novo ou singular no que se refere a jogabilidade do game *EcoRunner* implementado neste trabalho em questão: iniciando sua jornada pelo *game* com o simples pressionar da barra de espaço do seu teclado (Figura 5), o jogador controla



Figura 5: Tela inicial do jogo

O pequeno Peter (Figura 6), que deve percorrer uma corrida onde acumula pontos desviando de pequenos dejetos de lixo à medida que os salta, utilizando a tecla



Figura 6: O pequeno Peter

espaço (Figura 7), e de drones importunantes se abaixando ao se aproximarem, pressionando a seta para baixo (Figura 8) e realizando a coleta de pequenos vasos de planta bem afeiçoados pelo caminho.

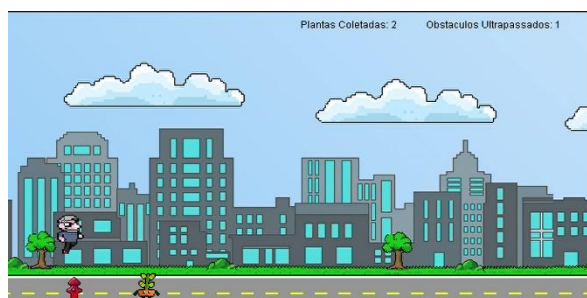


Figura 7: O pequeno Peter saltando um hidrante

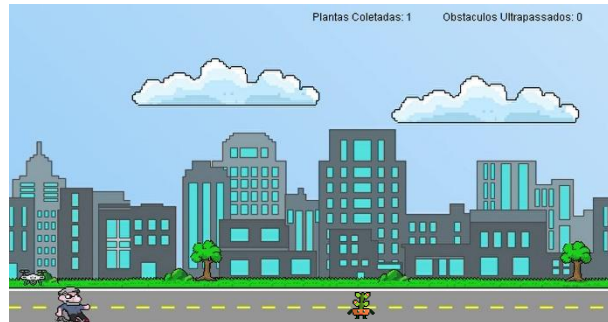


Figura 8: O pequeno Peter deslizando por debaixo de um drone

O objetivo, é realizar a maior pontuação de desvio de obstáculos e coleta de plantas o possível. Se O pequeno Peter colidir com um obstáculo, uma tela de *game over* irá aparecer sinalizando o final do jogo (Figura 9).



Figura 9: game over

4. PLANO DE DESENVOLVIMENTO DO JOGO

Nesta seção, será explanado todo o processo de criação do jogo, desde sua concepção até seu desenvolvimento e implementação de fato. Serão citadas as tecnologias utilizadas, estilos de arte e criação e a serão apresentados alguns conceitos que possivelmente irão auxiliar na melhor compreensão sobre cada elemento dos bastidores desse *game* que inicialmente sequer seria implementado no estilo plataforma.

4.1. Pixel art

Falando sobre a estrutura gráfica e visual do jogo, adotamos este estilo de arte já que atendia com louvor os preceitos estéticos que o grupo estava procurando. O objetivo era simples, utilizar uma interface intuitiva que auxiliasse na experiência lúdica do *gameplay*, organizar todos os elementos em tela de forma que a visualização fosse clara e aplicar uma gama de cores diversa e atraente. Após uma breve síntese das opiniões propostas, o consenso rapidamente chegou: o jogo deveria ter uma estrutura audiovisual “pixelada”. Desde os elementos gráficos até os elementos sonoros, todos remetem a uma época em que o Super Nintendo ou Mega Drive, eram figuras carimbadas nas casas dos jovens dos anos 90.



Figura 1: Sonic the Hedgehog 2 (Mega Drive - 1992)

Para fixação do conceito, podemos definir a *pixel art*, como um modelo estético onde os elementos e objetivos são construídos utilizando como partículas fundamentais os pixels, termo definido pela primeira vez por Adele Goldberg e Robert Flegal em 1982.

Este modelo de criação de cenários foi desenvolvido para atender as demandas relacionadas a *level design* das primeiras gerações de console, onde estes, eram de fato muito limitadas em termo de hardware. Entretanto, a estética de se firmou e não se esvaiu com o passar de todos esses anos, pelo contrário, tornou-se um conceito que remete a elementos nostálgicos e uma forma de tornar *design* de jogos mais inclusivo devido sua curva de aprendizado alta e fácil implementação. Motivados por todo esse contexto histórico, optamos por adotá-la como um dos pilares do desenvolvimento do jogo.

4.2. Processo criativo

Como dito anteriormente, o jogo inicialmente não seria um Endless Run. Foram propostas muitas ideias, inclusive fora fortemente cogitado que o *game* fosse mais um do estilo *Hack and Slash*, porém bidimensional, onde personagem jogável corre por uma área combatendo inimigos e resolvendo breves quebra-cabeças pelo caminho, como *God of War* (Figura 2) e *Devil May Cry* (Figura 3) que são exemplos muito populares de *Hack and Slashes* tridimensionais.



Figura 2: *God of War* (2005)



Figura 3: *Devil May Cry* (2001)

Foi pensado, até mesmo, em um estilo de combate alternativo para o jogo, já que ideia de criar um sistema de *gameplay* bidimensional já era basilar, e em determinado momento, foi voltada nossa visão para alguns jogos análogos ao que se pensava, como por exemplo o clássico *The Legend of Zelda* (Figura 4).



Figura 4: *The Legend of Zelda* (1986)

Onde Link, o personagem jogável viaja pelo mapa combatendo criaturas e antigos rivais para chegar até Zelda, princesa do reino de Hyrule que fora sequestrada. Barra de vida, inventário de itens, coleta de recursos, combate intuitivo e quadrilateral, história concisa, tudo parecia simples e objetivo, tínhamos uma base perfeita e aparentemente, de fácil implementação. Entretanto, logo na fase de coleta de dados e bagagem de aprendizado sobre as tecnologias a serem usadas, foram percebidas complexidades que eliminavam a ideia como viável.

Então, após uma série de pesquisas, foi identificado uma possibilidade muito interessante que logo de cara, foi adotada como a alternativa e principal fundamento do desenvolvimento do projeto como um *Endless Running Game*. Tudo isso, graças ao *Chrome Dino*.



Figura 5: *Chrome Dino* (2014)

Após definição do método de criação de artes e estilo de *gameplay* estarem definidos, bastou apenas aplicar os conceitos da temática proposta “educação ambiental, tendo como base a vida em uma grande metrópole” e as roupas adequadas como identidade dos personagens, bastou apenas passar para a fase de implementação.

Já era desejado há muito tempo realizarmos a homenagem ainda que nominal ou apenas velada a nossos professores orientadores do curso, e no projeto em questão, percebemos o momento adequado.



Figura 6: O pequeno Peter

O pequeno Peter, nome próprio, extenso e prolixo, tanto quanto a linguagem Java utilizada nesse projeto, tecnologia essa que automaticamente remete a um dos professores do currículo deste curso, Peter Jandl, brilhante professor e autor dos livros mencionados como base bibliográfica deste trabalho e merecidamente homenageado neste nosso personagem jogável do *game*.

Para a construção de cenário, optamos por utilizar a inserção de uma diversidade de grandes prédios com uma variação de cores pobre, garantindo a imersão e sensação de estar em uma metrópole onde o cinza e a poluição visual são muito presentes.



Figura 7: Cenário do jogo

Para representação dos obstáculos, usamos variações simples e definidas de forma pseudorrandômica pelo algoritmo do jogo. Todas elas pensadas para dar impressão de estar em uma metrópole agitada e movimentada, com drones voando pelos céus (Figura 8), hidrantes inconvenientes no meio do caminho (Figura 9), parquímetros desnecessários (Figura 10) e latas de lixo mal posicionadas (Figura 11).

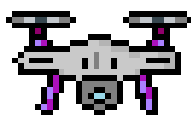


Figura 8: Drone



Figura 9: Hidrante



Figura 10: Parquímetro



Figura 11: Lixeira

Como objetivo secundária, há também a possibilidade de realizar a coleta de pequenos vasos de planta bem sorridentes (Figura 12), criados para serem carismáticos e atrativos para o jogador o bastante para serem coletados.

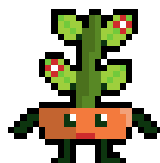


Figura 12: Vaso de planta

É interessante pontuar também que em determinado momento, houve a séria cogitação de criar a aplicação para que fosse nativamente *mobile* ou multiplataforma, para rodar em smartphones, entretanto, a ideia fora descartada no processo de planejamento.

4.3. Tecnologias utilizadas

Neste tópico, será abordado o assunto “tecnologias utilizadas para desenvolvimento do projeto” que em seus primórdios, passou por diversas potenciais tecnologias a serem implementadas, como LibGDX e Unit, mas que se firmou como um projeto desenvolvido em Swing, modesta ferramenta da linguagem Java, que será apresentada nesta seção.

4.3.1. Programação Orientada a Objetos – Java

As populares LPOO (Linguagens de Programação Orientada a Objetos), são definidas como um padrão que se refere a orientação a objetos abstratos, onde, neste paradigma, o mundo passa a ser interpretado como uma galeria de objetos frutos de um processo de abstração da realidade, onde possuem atributos e são baseados em classes. Como populares exemplos dessas linguagens de programação, podemos citar o C#, o C++ e Pascal.

Neste projeto, foi utilizada a linguagem de programação Java (Figuras 1 e 2), esta que se destaca por sua facilidade em implementar soluções que utilizam orientação a objeto (OO) e por suas particularidades em sua sintaxe e arquitetura de linguagem.

Desenvolvida por James Gosling e sua equipe na *Sun Microsystems* (adquirida posteriormente pela Oracle Corporation), a “linguagem da caneca de café” foi lançado em 1995 e desde então se tornou uma das linguagens mais populares do mundo. Uma das características mais marcantes do Java é sua portabilidade. Os programas Java são compilados em *bytecode*, que é uma representação intermediária entre o código fonte e o código de máquina. Esse *bytecode* é executado pela Máquina Virtual Java (Java Virtual Machine - JVM), que está disponível para várias plataformas, como Windows, macOS e Linux. Isso significa que um jogo apresentado nesse projeto, pode ser executado em diferentes sistemas operacionais sem a necessidade de modificação do código-fonte.

E é claro, devemos citar que o Java possui uma biblioteca padrão abrangente, conhecida como Java Standard Library, que oferece uma ampla gama de classes e métodos para realizar tarefas comuns, como manipulação de strings, entrada e saída, redes, acesso a banco de dados, entre outras. Essa biblioteca facilita o desenvolvimento de aplicativos em Java, pois fornece recursos prontos para uso, como por exemplo, o *Swing*.

4.3.2. Swing

O *Swing*, introduzido a primeira vez ao Java na versão 1.2 da plataforma, é um recurso nativo do próprio Java, e se define como um *framework* utilizado para a criação de interfaces gráficas. Ele faz parte do pacote `javax.swing` (é incorporado ao código dessa forma) e é uma das principais tecnologias utilizadas para desenvolver interfaces de usuário em Java, como na implementação desse projeto, onde todo o ambiente gráfico foi implementado utilizando esse recurso.

Ele oferece uma ampla variedade de componentes visuais, como: botões, campos de texto, caixas de seleção, tabelas, barras de rolagem e muitos outros, que podem ser combinados e personalizados para criar interfaces interativas e visualmente atraentes. O Swing utiliza componentes leves (*lightweight*) em vez de componentes pesados (*heavyweight*) fornecidos pelo sistema operacional. Isso significa que os componentes Swing são desenhados e renderizados pela própria biblioteca, o que permite uma aparência e comportamento consistentes em diferentes plataformas.

O Swing é baseado no modelo de eventos, onde as interações do usuário, como cliques de botões ou digitação em campos de texto, geram eventos que podem ser capturados e tratados por meio de *listeners*. Eles são interfaces ou classes que definem métodos específicos para lidar com os eventos, e tudo isso, seguindo o padrão de projeto MVC, que separa a lógica de negócio (*Model*), a apresentação dos dados (*View*) e o controle das interações do usuário (*Controller*). Isso facilita a organização e manutenção do código, tornando-o mais modular e flexível.

5. PROJETO DO PROGRAMA

Neste momento, será apresentado todo o endo esqueleto da aplicação. Como foi pensada e implementada em termos de tecnologia e desenvolvimento penando, agora de maneira prática, nos conceitos apresentados anteriormente.

5.1. Classe: “Personagemprincipal”

```
1      public static final int posicao_terreno = 312;
2      public static final float gravidade = 0.4f;
3
4      private static final int corrida_normal = 0;
5      private static final int pulo = 1;
6      private static final int corrida_abaixado = 2;
7      private static final int morte = 3;
8
9      private float posY;
10     private float posX;
11     private float speedX;
12     private float speedY;
13     private Rectangle rectBound;
14
15     public int pontuacao = 0;
16     public int coletaveis = 0;
17
18     private int estado = corrida_normal;
19
20     private Animation animacaocorridanormal;
21     private BufferedImage pulando;
22     private Animation animacaocorridaabaixada;
23     private BufferedImage imagemmorte;
24
25     private AudioClip sompulo;
26     private AudioClip morteson;
27     private AudioClip barulhobonus;
28
```

Atributos e Variáveis

Nas primeiras linhas da classe do personagem, é onde iremos definir todas as características que o protagonista (ou jogador) irá possuir, como a posição que ele irá ocupar na tela, a velocidade de locomoção e as pontuações.

A mesma estratégia foi aplicada para todos os objetos dentro do jogo, onde definimos as suas características, estados e atributos, para a sua eventual manipulação.

5.2. Métodos Comuns entre as Classes do Domínio: Aumento de Velocidade e Desenhar

```
1  public float getSpeedX() {
2      return speedX;
3  }
4
5  public void setSpeedX(int speedX) {
6      this.speedX = speedX;
7  }
8
9  public void draw(Graphics g) {
10     switch(estado) {
11         case corrida_normal:
12             g.drawImage(animacaocorridanormal.getFrame(), (int) posX, (int) posY, null);
13             break;
14         case pulo:
15             g.drawImage(pulando, (int) posX, (int) posY, null);
16             break;
17         case corrida_abaixado:
18             g.drawImage(animacaocorrídaabaixada.getFrame(), (int) posX, (int) (posY + 5), null);
19             break;
20         case morte:
21             g.drawImage(imagemmorte, (int) posX, (int) posY, null);
22             break;
23     }
24 }
25
26
27 public void atualiza() {
28     animacaocorridanormal.updateFrame();
29     animacaocorrídaabaixada.updateFrame();
30     if(posY >= posicao_terreno) {
31         posY = posicao_terreno;
32         if(estado != corrida_abaixado) {
33             estado = corrida_normal;
34         }
35     } else {
36         speedY += gravidade;
37         posY += speedY;
38     }
39 }
```

Para definir a velocidade dos objetos no jogo, criamos os habituais getters e setters para a sua manipulação, tornando-se uma estratégia padrão entre todos os

objetos. A mesma padronização de métodos foi aplicada para a função “draw” e “atualizar”, que respectivamente realizam a inserção dos elementos gráficos na janela do jogo e atualiza-os conforme a progressão do game-loop.

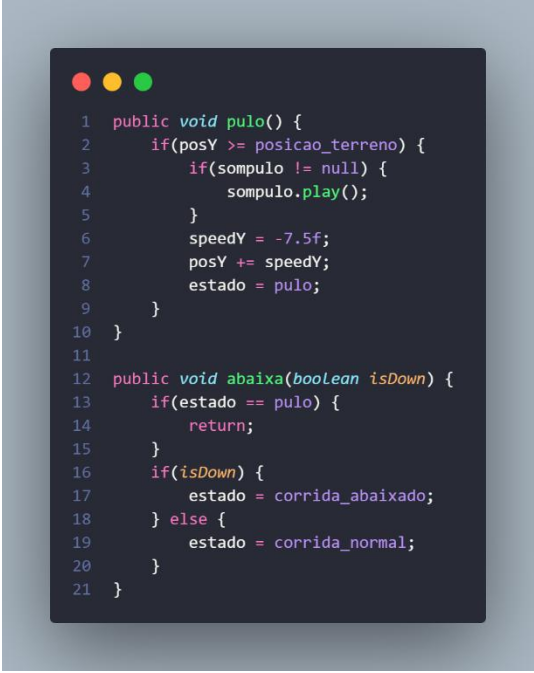
Hit-Box (getBound)

```
1 public Rectangle getBound() {
2     rectBound = new Rectangle();
3     if(estado == corrida_abaixado) {
4         rectBound.x = (int) posX + 5;
5         rectBound.y = (int) posY + 20;
6         rectBound.width = animacaocorridaabaixada.getFrame().getWidth() - 10;
7         rectBound.height = animacaocorridaabaixada.getFrame().getHeight();
8     } else {
9         rectBound.x = (int) posX + 5;
10        rectBound.y = (int) posY;
11        rectBound.width = animacaocorridanormal.getFrame().getWidth() - 10;
12        rectBound.height = animacaocorridanormal.getFrame().getHeight();
13    }
14    return rectBound;
15 }
```

O método “getBound” é o que define o tamanho da hit-box dos nossos objetos. Novamente será um método encontrado em todas as classes do domínio, e funciona definindo um retângulo “invisível” em volta do objeto, através de dimensões pré-definidas que variam de acordo com o estado do personagem (correndo, ou abaixado), e caso este retângulo colida com outro, o jogo se encerra.

5.3. Métodos:

Pular e Abaixar



```
1 public void pulo() {
2     if(posY >= posicao_terreno) {
3         if(sompulo != null) {
4             sompulo.play();
5         }
6         speedY = -7.5f;
7         posY += speedY;
8         estado = pulo;
9     }
10 }
11
12 public void abaixa(boolean isDown) {
13     if(estado == pulo) {
14         return;
15     }
16     if(isDown) {
17         estado = corrida_abaixado;
18     } else {
19         estado = corrida_normal;
20     }
21 }
```

As ações dos personagens são manipuladas através “pulo” e “abaixa”, manipulando os **estados** de cada objeto, de acordo com suas posições na janela do jogo. No método pulo, nós acionamos o som correspondente a ação, diminuimos levemente a posição y do personagem causando o efeito de salto dele dentro da tela.

Para o efeito de agachamento, utilizamos o estado “corridaabaixado”, e ao estabelecer este parâmetro, a altura do personagem é configurada para 5 pixels a mais, para que os objetos acima disso não “batam” na sua hit-box e ocasionem o fim do jogo.

Morte

```
1 public void morte(boolean estamorto) {  
2     if(estamorto) {  
3         coletaveis = 0;  
4         pontuacao = 0;  
5         estado = morte;  
6     } else {  
7         estado = corrida_normal;  
8     }  
9 }
```

Este método verifica se o objeto colidiu com outro (através da variável “estamorto”), e caso o retorno seja verdadeiro, ele zera os placares e coloca o estado como “morte”, no que desencadeia o fim do jogo.

Efeitos Sonoros

```
1 public void tocasommorte() {  
2     mortesom.play();  
3 }  
4  
5  
6 public void tocasommoeda(){  
7     barulhobonus.play();  
8 }
```

Utilizando a biblioteca Applet e AudioClip, podemos realizar a inserção de sons e invocá-los através de métodos simples como os que são demonstrados acima. Após a criação, basta chamá-los dentro de um bloco, que no nosso caso é através das ações dos personagens e na coleta de colecionáveis.

Aumento de Pontos e Coletáveis

```
1 public void aumentapontos() {
2     pontuacao += 1;
3     if(pontuacao % 100 == 0) {
4         barulhobonus.play();
5     }
6 }
7
8 public void aumentacoletaveis() {
9     coletaveis += 1;
10    if(coletaveis % 100 == 0) {
11        barulhobonus.play();
12    }
13 }
```

Por fim, os métodos acima aumentam e tocam os respectivos sons ao coletar e aumentar os pontos do personagem. (Obstáculos superados e Plantas coletadas).

5.4. Classe: GerenciadorInimigos

Para organizar e gerar os inimigos dentro do jogo, foi criado a classe “GerenciadorInimigos”, no qual nós instanciamos e criamos todos os obstáculos que são gerados no jogo. Todos os objetos são gerenciados dentro de um Array List, e então acessados de forma aleatória no decorrer do jogo.

```

1 private BufferedImage obstaculo1;
2 private BufferedImage obstaculo2;
3 private BufferedImage obstaculo3;
4 private BufferedImage drone;
5 private Random aleatorio;
6 private Drone Drone;
7
8 private List<Inimigos> inimigos;
9 private Personagemprincipal personagemprincipal;
10
11 public GerenciadorInimigos(Personagemprincipal personagemprincipal) {
12     aleatorio = new Random();
13     obstaculo1 = Resource.getResourceImage("data/obstaculo1.png");
14     obstaculo2 = Resource.getResourceImage("data/obstaculo2.png");
15     obstaculo3 = Resource.getResourceImage("data/obstaculo3.png");
16     drone = Resource.getResourceImage("data/drone1.png");
17     inimigos = new ArrayList<Inimigos>();
18     this.personagemprincipal = personagemprincipal;
19     inimigos.add(criarInimigo());
20 }

```

Variáveis utilizadas e construtor

Além dos métodos *atualiza*, *draw* apresentados anteriormente, possuímos o método *criarInimigo* no qual passa a de fato criar o obstáculo na tela, de maneira aleatória.

```

1 private Inimigos criarInimigo() {
2     // if (enemyType = getRandom)
3     int type = aleatorio.nextInt(4);
4
5     if(type == 0) {
6         return Drone = new Drone(personagemprincipal, 900, drone.getWidth() - 10, drone.getHeight() - 10, drone);
7     }
8     else if(type == 1) {
9         return new Obstaculos(personagemprincipal, 905, obstaculo1.getWidth() - 10, obstaculo1.getHeight() - 10, obstaculo1);
10    }
11    else if(type == 2){
12        return new Obstaculos(personagemprincipal, 920, obstaculo2.getWidth() - 10, obstaculo2.getHeight() - 10, obstaculo2);
13    }
14    else {
15        return new Obstaculos(personagemprincipal, 935, obstaculo3.getWidth() - 10, obstaculo3.getHeight() - 10, obstaculo3);
16    }
17 }
18 }

```

Por último, temos os métodos *colidiu* e *resetar*, que servem para nos dizer quando que um inimigo colidiu com o personagem principal, e para de fato resetar o array de inimigos e cria-los novamente.


```

1 public boolean colidiu() {
2     for(Inimigos e : inimigos) {
3         if (personagemprincipal.getBounds().intersects(e.getBounds())) {
4             return true;
5         }
6     }
7     return false;
8 }
9
10 public void resetar() {
11     inimigos.clear();
12     inimigos.add(criarInimigo());
13 }

```

```

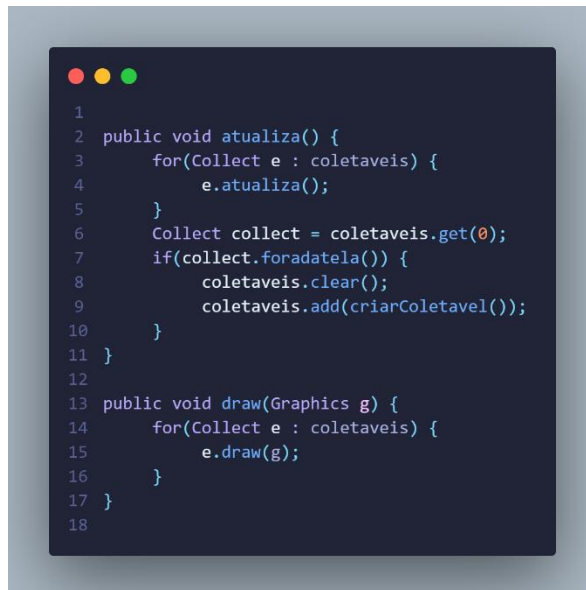
1
2 public GerenciadorColetaveis(Personagemprincipal personagemprincipal) {
3     aleatorio = new Random();
4     coletavel1 = Resource.getResourceImage("data/planta1.png");
5     coletavel2 = Resource.getResourceImage("data/planta2.png");
6     coletaveis = new ArrayList<Collect>();
7     this.personagemprincipal = personagemprincipal;
8     coletaveis.add(criarColetavel());
9 }

```

Função GerenciadorColetaveis

Essa função define uma variável com valor aleatório, define também duas imagens dos coletáveis, e cria um novo array, também chama a personagem principal

e adiciona no array recém criado uma lista de obstáculos usando o método `criarColetaveis`.



```
1
2 public void atualiza() {
3     for(Collect e : coletaveis) {
4         e.atualiza();
5     }
6     Collect collect = coletaveis.get(0);
7     if(collect.foradatela()) {
8         coletaveis.clear();
9         coletaveis.add(criarColetavel());
10    }
11 }
12
13 public void draw(Graphics g) {
14     for(Collect e : coletaveis) {
15         e.draw(g);
16     }
17 }
18
```

Função Atualiza e draw

A função `atualiza` usa um `for` para atualizar os coletáveis e abaixo dela tem uma condicional para caso o coletável esteja fora da tela ele limpar o coletável e adicionar ele novamente na lista de coletáveis. A função `draw` é a responsável por desenhar os coletáveis na tela.

```

1 private Collect criarColetavel() {
2     // if (enemyType = getRandom)
3     int type = aleatorio.nextInt(2);
4     if(type == 0) {
5         return new Planta(personagemprincipal, 950, coletavel1.getWidth() - 10, coletavel1.getHeight() - 10, coletavel1);
6     } else {
7         return new Planta(personagemprincipal, 923, coletavel2.getWidth() - 10, coletavel2.getHeight() - 10, coletavel2);
8     }
9 }

```

Função CriarColetavel()

A função "criarColetavel" cria um objeto do tipo "Collect" . A função usa um gerador de números aleatórios para selecionar um tipo de coletável a ser criado. Se o número aleatório for igual a 0, a função cria um objeto do tipo "Planta" em uma certa posição, caso contrário, se o número aleatório for diferente de 0, a função cria um objeto do tipo "Planta" em outra posição, o objeto de tipo planta é o nosso coletável.

```

1 public boolean colidiu() {
2     for(Collect e : coletaveis) {
3         if (personagemprincipal.getBounds().intersects(e.getBounds())) {
4             return true;
5         }
6     }
7     return false;
8 }
9
10 public void resetar() {
11     coletaveis.clear();
12     coletaveis.add(criarColetavel());
13 }

```

Função Colidiu e resetar

A Função colidiu é responsável por verificar se o personagem principal está colidindo com os coletáveis assim para quando for chamada em outra parte do código consiga tratar uma condicional como a que aumenta os pontos

Já a função resetar é responsável por limpar os coletáveis e adicionar novamente na lista de coletáveis.

6. RELATÓRIO COM AS LINHAS DE CÓDIGO

```
1 public class Janelajogo extends JFrame {
2
3     public static final int SCREEN_WIDTH = 720;
4     private Telajogo gameScreen;
5
6     public Janelajogo() {
7         super("Eco Runner");
8         setSize(SCREEN_WIDTH, 410);
9         setLocation(400, 200);
10        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        setResizable(false);
12
13        gameScreen = new Telajogo();
14        addKeyListener(gameScreen);
15        add(gameScreen);
16    }
17
18    public void iniciaJogo() {
19        setVisible(true);
20        gameScreen.iniciaJogo();
21    }
22
23    public static void main(String args[]) {
24        (new Janelajogo()).iniciaJogo();
25    }
26 }
```

Classe: JanelaJogo

Ao utilizar a biblioteca JFrame, é costumeiro utilizarmos uma classe apenas para a configuração da “moldura” do jogo, ou seja, a janela em que os componentes principais do jogo irão estar interagindo entre si, e por sua vez desenhados a cada interação do loop principal.

Os métodos principais desta classe, são:

- JanelaJogo() – Construtor da classe, onde todas as especificações da janela são de fato aplicadas quando instanciamos um objeto da classe.
- iniciaJogo() – Neste método, é quando estabelecemos a janela do jogo como visível.
- main () – Este é o método que irá de fato iniciar o jogo. Ou seja, nosso **ponto de partida** para o jogo de fato, iniciar.

```

1 private static final int estado_inicioJogo = 0;
2 private static final int estado_jogando = 1;
3 private static final int estado_fimjogo = 2;
4 private static final int estado_menu = 3;
5
6 private Ceu ceu;
7 private Terreno land;
8 private Personagemprincipal personagemprincipal;
9 private GerenciadorInimigos gerenciadorInimigos;
10 private Nuvens nuvens;
11 private Thread thread;
12 private GerenciadorColeteveis GerenciadorColeteveis;
13
14 private AudioClip musica;
15
16 private boolean teclapressionada;
17
18 private int estadodejogo = estado_menu;

```

Classe TelaJogo - Declarações de variáveis e classes

Nessa parte do código da classe é criada e inicializadas algumas variáveis como por exemplo as variáveis de estado de jogo que são utilizadas futuramente para controlar as ações do jogo, temos também as declarações de classe que futuramente serão inicializadas, a variável “teclapressionada” que utilizamos para os controles do jogo, e definimos a variável “estadodejogo” como “estado_menu” para já iniciarmos o jogo no menu

```

1 public TelaJogo() {
2     ceu = new Ceu();
3     personagemprincipal = new Personagemprincipal();
4     land = new Terreno(Janelajogo.SCREEN_WIDTH, personagemprincipal);
5     personagemprincipal.setSpeedX(4);
6     imagenfimdejogo = Resource.getResourceImage("data/fimdejogo.png");
7     menu = Resource.getResourceImage("data/menu_principal.png");
8     gerenciadorInimigos = new GerenciadorInimigos(personagemprincipal);
9     nuvens = new Nuvens(Janelajogo.SCREEN_WIDTH, personagemprincipal);
10    GerenciadorColeteveis = new GerenciadorColeteveis(personagemprincipal);
11
12    try {
13        musica = Applet.newAudioClip(new URL("file","", "data/aumentaponto.wav"));
14    } catch (MalformedURLException e) {
15        e.printStackTrace();
16    }
17 }

```

Função TelaJogo

A função tela jogo é responsável pelas inicializações de classe como por exemplo a classe “ceu”, “Personagemprincipal”, “gerenciadorInimigos” etc, define também a velocidade do personagem principal, e inicializa a variável música com um clip de áudio que é utilizado futuramente quando o jogo é aberto.

```

1
2 public void atualizajogo() {
3     if (estadodejogo == estado_jogando) {
4         land.atualiza();
5         nuvens.atualiza();
6         personagemprincipal.atualiza();
7         gerenciadorInimigos.atualiza();
8         GerenciadorColeteveis.atualiza();
9         if (gerenciadorInimigos.colidiu()) {
10             personagemprincipal.tocasomorte();
11             estadodejogo = estado_fimjogo;
12             personagemprincipal.morte(true);
13         }
14         if (GerenciadorColeteveis.colidiu()) {
15             personagemprincipal.tocasommoeda();
16             estadodejogo = estado_jogando;
17             personagemprincipal.aumentacoletaveis();
18             GerenciadorColeteveis.resetar();
19         }
20     }
21 }
22

```

Função atualizajogo

A função atualiza jogo, verifica se a variável “estadodejogo” é igual a “estado_jogando” se ela essa condição for verdadeira ela chama o método atualiza das classes land, nuvens, personagemprincipal, gerenciadorinimigos, gerenciadorcoletaveis, a função tem mais uma condicional internamente ela verifica se o método colidiu das classes personagem principal e gerenciador coletáveis foi ativada, se a gerenciadorinimigos colidiu for verdadeira o código toca o som de morte e atualiza o estado de jogo para fim de jogo e define o personagem como morto através de “personagemprincipal.morte(true)” e se for a gerenciadorColetaveis for verdadeira ele toca o som de quando uma moeda é pega, e os estado de jogo continua como estado jogando a condicional também aumenta o número de coletáveis que o jogador pegou, e reseta a classe gerenciador coletáveis para eles serem reconstruídos novamente.

```

1 public void paint(Graphics g) {
2     g.setColor(Color.decode("#f7f7f7"));
3     g.fillRect(0, 0, getWidth(), getHeight());
4
5     switch (estadoJogo) {
6     case estado_inicioJogo:
7         ceu.draw(g);
8         land.draw(g);
9         personagemPrincipal.draw(g);
10        break;
11    case estado_jogando:
12    case estado_fimJogo:
13        ceu.draw(g);
14        nuvens.draw(g);
15        land.draw(g);
16        GerenciadorColetaveis.draw(g);
17        gerenciadorInimigos.draw(g);
18        personagemPrincipal.draw(g);
19        g.setColor(Color.BLACK);
20        g.drawString("Obstaculos Ultrapassados: " + personagemPrincipal.pontuacao, 500, 20);
21        g.drawString("Plantas Coletadas: " + personagemPrincipal.coletaveis, 350, 20);
22        if (estadoJogo == estado_fimJogo) {
23            g.drawImage(imagemfimJogo, 200, 150, null);
24        }
25        break;
26    case estado_menu:
27        ceu.draw(g);
28        nuvens.draw(g);
29        land.draw(g);
30        g.setColor(Color.BLACK);
31        //g.drawString("Ecorunner, um jogo sobre sustentabilidade", 200, 20);
32        //g.drawString("APORTE ESPACO PARA COMECAR ", 200, 50);
33        //g.drawString("APORTE ESC PARA SAIR ", 200, 100);
34        g.drawImage(menu, 10, 0, null);
35        break;
36    }
37 }
38

```

Função paint

A função paint é a responsável por “desenhar” todos os objetos na tela conforme o estado de jogo e realiza mudanças na mesma como por exemplo no “estado_menu” ele “desenha” o céu do jogo as nuvens e o terreno.


```

1  @Override
2      public void run() {
3
4          int fps = 100;
5          long msPerFrame = 1000 * 1000000 / fps;
6          long lastTime = 0;
7          long tempopassado;
8
9          int microsegundo;
10         int nanosegundo;
11
12         long fimprocesso;
13         long lag = 0;
14
15         tocamusica();
16         while (true) {
17             atualizajogo();
18             repaint();
19             fimprocesso = System.nanoTime();
20             tempopassado = (lastTime + msPerFrame - System.nanoTime());
21             microsegundo = (int) (tempopassado / 1000000);
22             nanosegundo = (int) (tempopassado % 1000000);
23             if (microsegundo <= 0) {
24                 lastTime = System.nanoTime();
25                 continue;
26             }
27             try {
28                 Thread.sleep(microsegundo, nanosegundo);
29             } catch (InterruptedException e) {
30                 e.printStackTrace();
31             }
32             lastTime = System.nanoTime();
33         }
34     }
35

```

função run

É a responsável pelas definições de frames por segundo e a classe que possibilita as animações do jogo funcionarem.

```

1  @Override
2  public void keyPressed(KeyEvent e) {
3      if (!teclapressionada) {
4          teclapressionada = true;
5          switch (estadodejogo) {
6              case estado_iniciojogo:
7                  if (e.getKeyCode() == KeyEvent.VK_SPACE) {
8                      estadodejogo = estado_jogando;
9                  }
10                 break;
11             case estado_jogando:
12                 if (e.getKeyCode() == KeyEvent.VK_SPACE) {
13                     personagemprincipal.pulo();
14                 } else if (e.getKeyCode() == KeyEvent.VK_DOWN) {
15                     personagemprincipal.abaixa(true);
16                 }
17                 break;
18             case estado_fimjogo:
19                 if (e.getKeyCode() == KeyEvent.VK_SPACE) {
20                     estadodejogo = estado_jogando;
21                     resetajogo();
22                 }
23                 break;
24             case estado_menu:
25                 if (e.getKeyCode() == KeyEvent.VK_SPACE) {
26                     estadodejogo = estado_jogando;
27                 }
28                 break;
29             }
30         }
31     }
32 }

```

KeyPressed

```

1
2  @Override
3  public void keyReleased(KeyEvent e) {
4      teclapressionada = false;
5      if (estadodejogo == estado_jogando) {
6          if (e.getKeyCode() == KeyEvent.VK_DOWN) {
7              personagemprincipal.abaixa(false);
8          }
9      }
10 }

```

Key Released

A função `keypressed` é responsável pelos controles do jogo e pelo início do jogo quando o estado de jogo é igual “estado_iniciojogo”, ou por exemplo quando o estado de jogo é igual “estado_jogando” se a tecla espaço for pressionada ele chama o método de pulo do personagem, e caso seja igual a tecla para baixo ele chamado o método que abaixa o personagem, também temos a condicional caso a variável fim de jogo seja a atual, ele espera a teclado espaço e se foi pressionada ela muda o estado para jogando novamente e reseta o jogo. Também temos outra função chamada `keyReleased` que é utilizada para quando as teclas são soltas e utiliza isso para voltar o personagem para cima quando a tecla de seta para baixo é solta.



Resetajogo() e tocamusica()

A função resetajogo é utilizada para reiniciar o personagem principal, define o estado de morte dele como false, e os gerenciadores de coletáveis e de inimigos são reiniciados, a função toca música é utilizada para tocar o som que inicializamos no início do jogo e chamamos no método RUN.

7. BIBLIOGRAFIA

JANDL, Peter. **JAVA: guia do programador**. 4ª Edição. Jundiaí: Novatec, 2021.