

UNIVERSIDADE FEDERAL DE MINAS GERAIS
Engenharia de Software – Professor Eduardo Figueiredo

Guilherme Teres Nunes – 2016077187

JOGO ARCADE

Belo Horizonte

2019

Sobre

Esse trabalho foi desenvolvido por **Guilherme Teres Nunes** e é referente ao projeto prático da disciplina de **Engenharia de Software** ministrada pelo professor **Eduardo Figueiredo** na **Universidade Federal de Minas Gerais – UFMG**. O projeto foi desenvolvido com o intuito de criar e passar por toda a estrutura, engenharia e projeto de software necessário para desenvolver e escalar um jogo digital a partir do zero.

Nesse documento há a explicação completa de tudo o que foi desenvolvido, incluindo, mas não se limitando a, requisitos, modelo de processo, estruturação e diagramação do software, funcionamento e lógica por trás do Motor de Jogo desenvolvido para o Game e suas relações com a Engenharia de Software.

O **código fonte** completo do projeto pode ser encontrado através do link <<https://github.com/UnidayStudio/JavaArcadeGame1>> .

Descrição

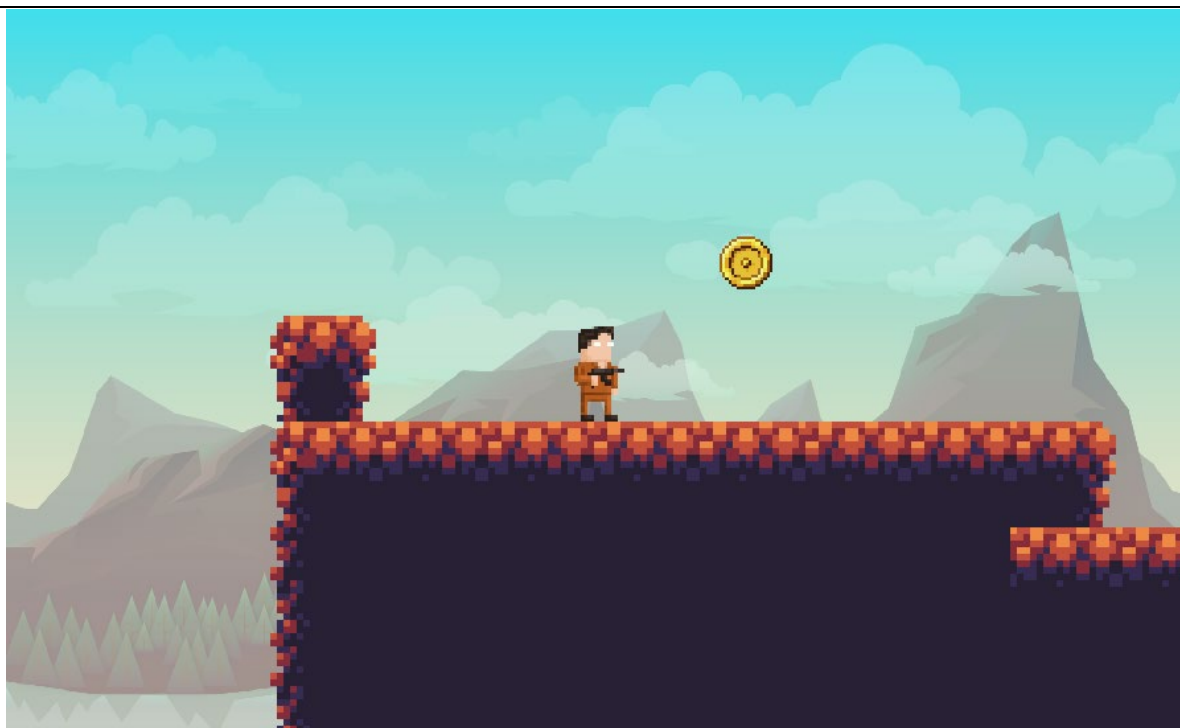
Se trata de um Jogo de plataforma 2D, com personagens animados via *sprites* (sequencia de imagens 2D), onde o usuário (jogador) poderá andar, pular e coletar moedas pelo cenário. Também há a presença de NPC'S (*non playable character*) pelo mapa com comportamento próprio. O motor (*game engine*) por trás do jogo deverá ter suporte a cenas para construir as fases e uma interface base para um objeto 2D que poderá ser herdada para criar o visual e a lógica dos diferentes tipos de entidades do jogo (blocos, personagem, inimigos). Além disso, o motor também deverá tratar as entradas do usuário (teclado e mouse) e fornecer o cálculo de física básico entre os diferentes tipos de objetos na cena.

Como executar o Projeto

O projeto foi inteiramente desenvolvido utilizando a linguagem e programação **Java** e através da IDE **IntelliJ IDEA**, que pode ser obtida gratuitamente através do site [jetbrains.com/idea](https://www.jetbrains.com/idea). Para executar o projeto basta abrir a pasta **JavaArcadeGame1** através dessa IDE e executar o arquivo *Main.java*, localizado em *src/main/com.game/app/Main.java*. Para controlar o personagem, use as teclas A e D e a tecla W para pular.

Demonstração

Um vídeo de demonstração do jogo pode ser encontrado no **YouTub**e através do link:
< <https://youtu.be/yhut3Z9Qxkk> > . Abaixo algumas imagens do jogo em execução.



Requisitos Funcionais

Jogo	[R01] O software deve permitir que o jogador inicie um jogo. [R02] O software deverá detectar colisão entre dois objetos. [R03] O software deverá mostrar os objetos da cena na tela do usuário.
Fase	[R04] A fase deve permitir a existência de blocos (plataformas) onde o jogador pode pisar. [R05] A fase deve permitir a existência de inimigos. [R06] A fase deve permitir a existência de moedas coletáveis.
Personagem	[R07] A personagem poderá pular de acordo com a entrada do jogador. [R08] A personagem poderá andar de acordo com as entradas do jogador.
Inimigos	[R09] Os inimigos deverão se movimentar aleatoriamente para os lados.

Requisitos Não Funcionais

- O sistema deve funcionar a 60 frames por segundo.
- O sistema deverá reconhecer entradas de teclado, até três simultâneas.
- O tempo de resposta entre a entrada do teclado e a ação deverá ser inferior a 0.5 segundo.
- O sistema deverá ser executado no sistema operacional Windows.
- O sistema deverá ser capaz de renderizar *sprites* em pixel art.

Modelo de Processo

O modelo de processo para o desenvolvimento de jogos deve permitir testes a qualquer momento, criação de protótipos e deve lidar bem com mudanças, uma vez que o escopo de um jogo digital pode mudar drasticamente de acordo com o resultado de *playtests* (testes com jogadores). Um dos modelos que mais se adapta as especificações apresentadas é o **Modelo Espiral**, que também é frequentemente utilizado e adaptado para a indústria de games. As maiores vantagens de utilizar o Modelo Espiral no desenvolvimento de jogos são o risco baixo, a possibilidade de começar pelas *Features* mais importantes, rápida prototipação e a viabilidade de mudanças a todo o momento. Abaixo você verá um esquema de seu processo seguido de uma adaptação feita para a indústria.

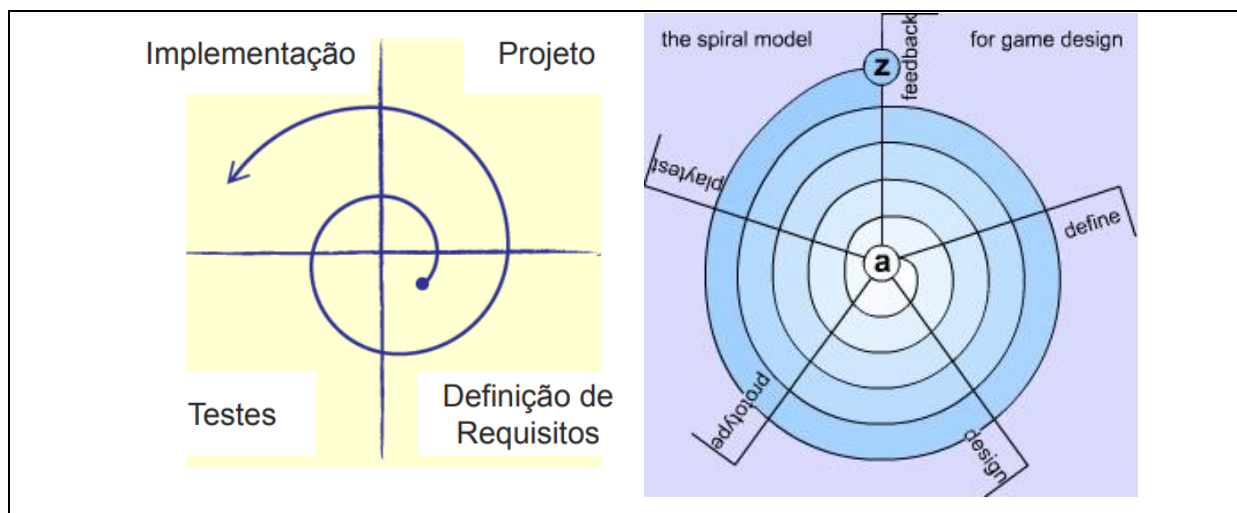
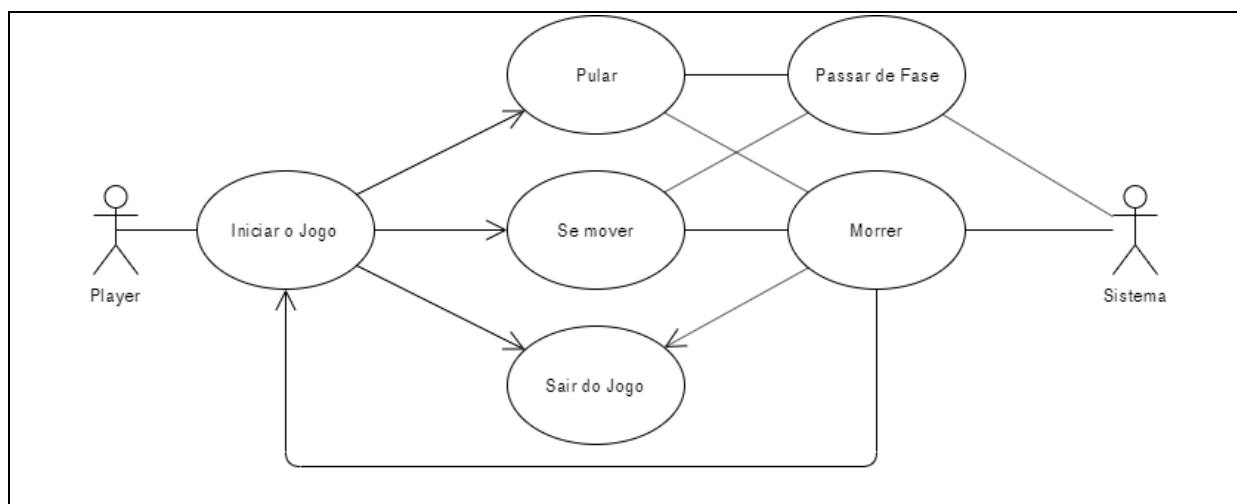


Diagrama de Uso



Exemplo de Cenários

Nome do Cenário: **Pular**

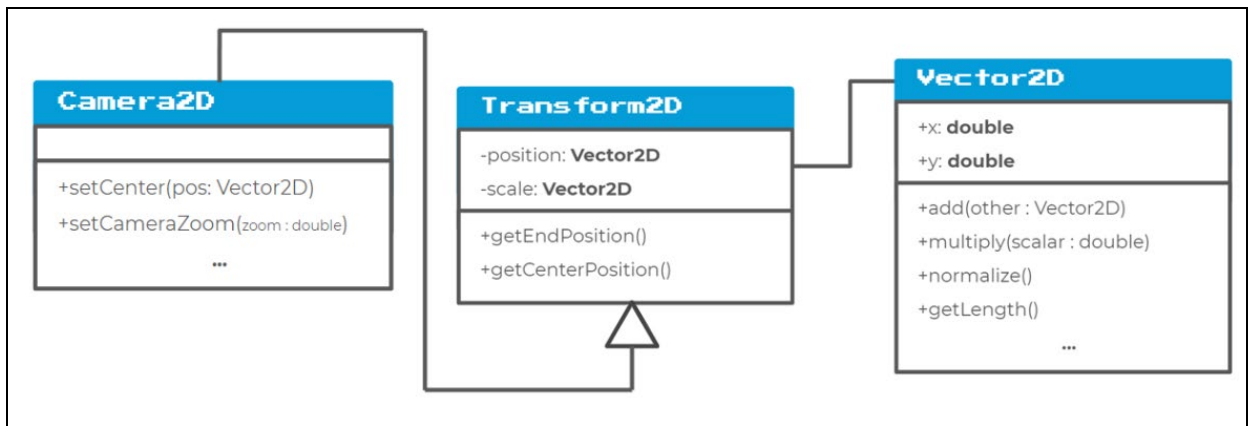
- Ator: Jogador
- Pré-condição: Iniciar o Jogo
- Fluxo normal
 1. Iniciar o Jogo
 2. Apertar a tecla de pulo
 3. Pular
- Fluxo alternativo: Nenhum
- Pós-condição: Nenhuma

Nome do Cenário: **Passar de Fase**

- Atores: Jogador, Sistema
- Pré-condição: Iniciar o Jogo
- Fluxo normal
 1. Iniciar o Jogo
 2. Se mover até o fim da fase
 3. Chegar ao final da fase
 4. Sistema carrega a próxima fase
- Fluxo alternativo: Nenhum
- Pós-condição: Nenhuma

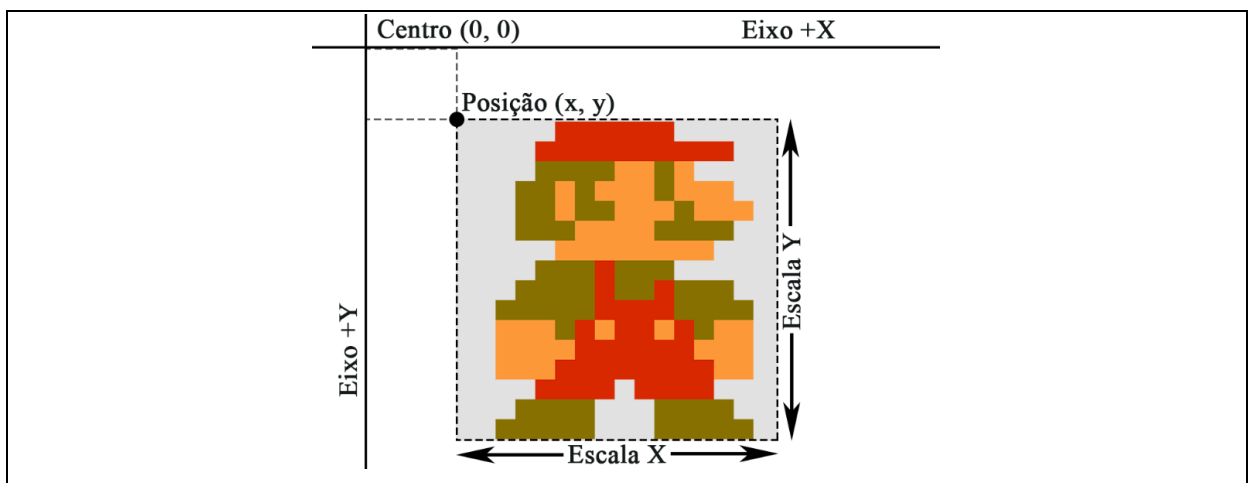
Classes Matemáticas

O primeiro conjunto de classes apresentado é referente, principalmente, a **operações matemáticas** Vetoriais, essenciais para o funcionamento de qualquer jogo digital com imagens na tela. Segue um breve diagrama de classes (UML) seguido de uma explicação detalhada:



A classe **Vector2D** armazena dois números *double* e possui algumas das operações mais básicas para operar vetores, tais como soma, subtração, multiplicação e divisão por escalar, normalização do vetor, obter tamanho. A classe **Transform2D** armazena dois vetores: um para a posição e outro para a escala. Para entender o funcionamento do **Transform2D**, é necessário entender como os objetos (e/ou *sprites*) são desenhados na tela e quais os parâmetros necessários.

É importante ressaltar que o **Graphic** da linguagem Java (utilizado para desenhar as primitivas na tela) considera o eixo X da esquerda para a direita e o eixo Y de cima para baixo. Ou seja: somar um valor positivo a posição Y de um objeto faz com que ele desça pela tela.



Na imagem apresentada acima é possível identificar exatamente o que os vetores de *posição* e *escala* representam do objeto. A classe ***Transform2D*** também possui métodos para obter a posição final (posição + escala) e a posição central (posição + escala/2) da entidade.

A classe da câmera (***Camera2D***) reaproveita as funcionalidades do ***Transform2D***, já que uma câmera, por padrão também pode ser representada através dos mesmos parâmetros de posição e escala, mas acrescenta métodos específicos para definir seu centro (posição + escala/2) diretamente e manipular a escala de acordo com o *Aspect Ratio* da janela onde o jogo está acontecendo.

NOTA: A câmera tenta abstrair do usuário o conceito de escala, já que essa deve ser calculada baseada no *Aspect Ratio* da tela, caso contrário, a imagem exibida na tela ficará esticada. Ainda sim é possível acessar e alterar tal valor diretamente pois pode ser de interesse do usuário esticar propositalmente a imagem final para criar algum tipo de efeito especial dentro do jogo.

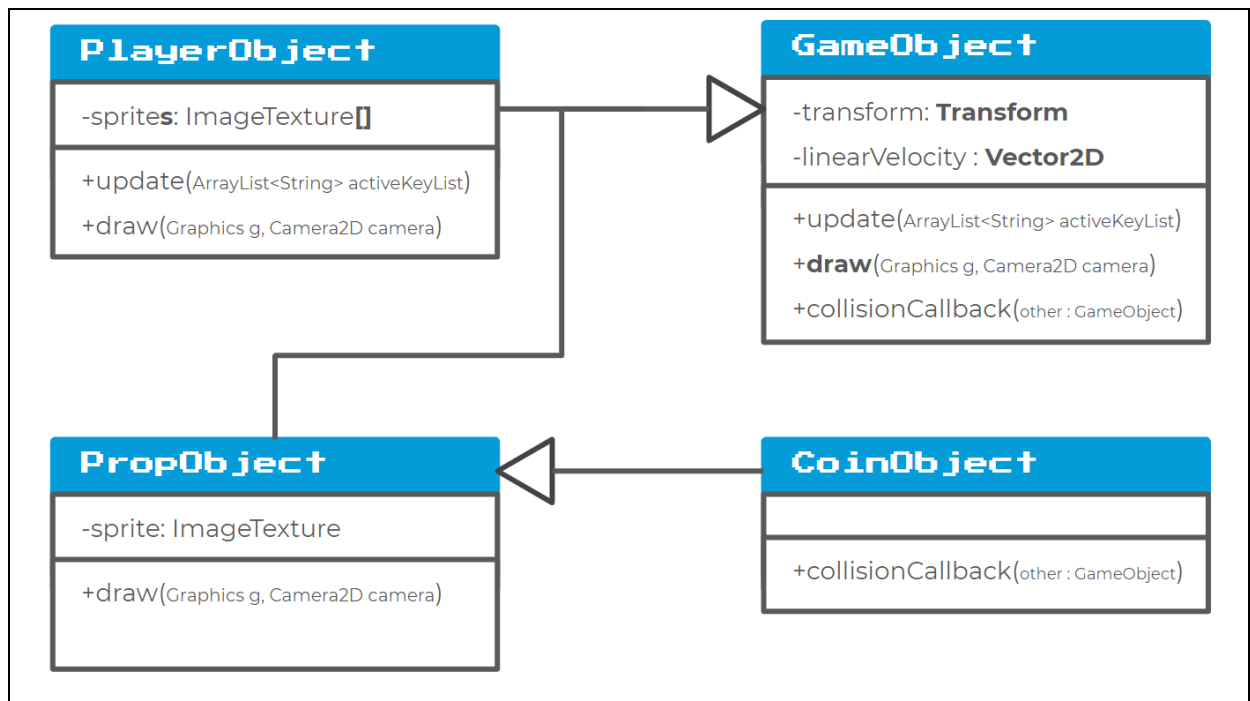
Classes principais:

Um jogo digital pode ser modelado através de uma lista de objetos polimórficos (chamados de ***GameObjects***, ou ***Entities***) com métodos (virtuais) para tratar três situações principais:

- **Executar lógica** e/ou tratar os inputs (entradas) do usuário (jogador). Se o jogador apertar uma tecla, por exemplo, e a personagem do jogo tiver que pular, esse evento será tratado nesse método. Outro exemplo é o cálculo de inteligência artificial para inimigos.
- Responder á **colisões de Física entre objetos**. Para facilitar o desenvolvimento e reduzir as chances de erros, a detecção de colisão bem como o tratamento das mesmas (mover os objetos de forma com que duas entidades com física não ocupem o mesmo espaço – uma dentro da outra) são abstraídas do usuário e calculadas automaticamente pelo ***GameApp***. Ainda sim, para cada colisão detectada entre dois objetos, ambos são notificados através de um método (***collisionCallback***) que recebe o outro objeto envolvido. Isso é necessário, por exemplo, para fazer uma moeda ser excluída quando colidir com o jogador.
- **Renderizar o objeto na tela** (draw). Qualquer ***GameObject*** tem autonomia para se desenhar na tela (ou não ser desenhado) como quiser. Para isso cada entidade do jogo

pode sobrescrever seu método de *draw* e conta com dois parâmetros essenciais: o *Graphic*, classe do *Java* necessária para desenhar qualquer elemento na tela, e a câmera (*Camera2D*). Pode ser de interesse do usuário não utilizar a câmera (ou todos os seus parâmetros) para desenhar o objeto como, por exemplo, para criar uma imagem estática de fundo (*background*) que não se altera em relação a câmera ou elementos de interface que tem uma posição fixa.

Conforme mencionado, o *GameObject* é uma classe polimórfica que pode ser herdada para a criação de diferentes tipos de entidades no jogo: Moedas, paredes, personagens, inimigos, tiros, etc. Abaixo um diagrama de classes UML exemplificando seu uso:



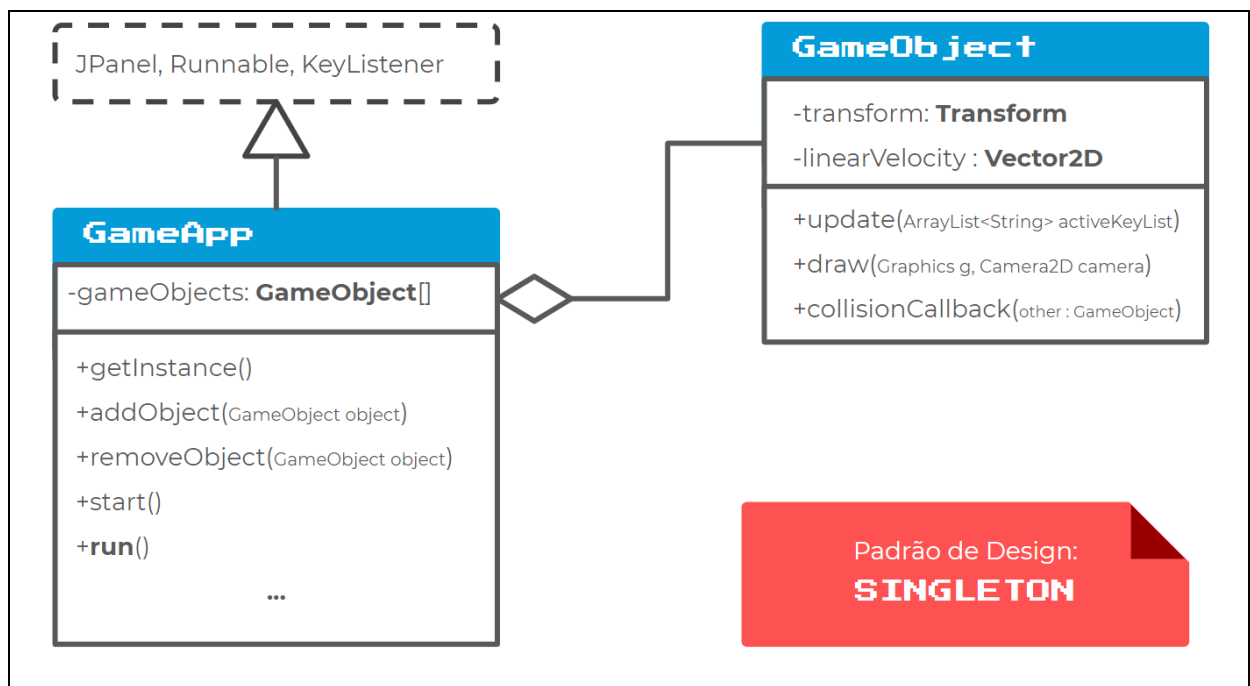
Para gerenciar e executar todo esse conjunto de objetos polimórficos e também ser responsável pela criação de uma janela do sistema operacional e detectar as entradas de teclado corretamente, uma classe *GameApp* é utilizada. Ela tem como função principal executar um loop (conhecido como *Game Loop*) que gerencia todos os *GameObjects* do jogo. Veremos detalhes sobre o game loop a frente.

A classe *GameApp* também possui métodos para adicionar e/ou remover *GameObjects*. Tais métodos devem ser tratados de forma especial, através de *queues* (listas de esperas) para serem resolvidos sempre ao final de cada gameloop. Isso é necessário, pois um objeto, durante a execução de seu método de *update* ou de *collisionCallback*, pode solicitar a adição ou

remoção de outro objeto, o que resultaria na modificação da *ArrayList* de *GameObject* durante sua iteração, causando erros.

Para garantir que tais funcionalidades como o gerenciamento de Janelas do sistema operacional e o reconhecimento de inputs (entradas) do usuário, o *GameApp* estende a classe padrão Java *JPanel* e implementa as interfaces padrões *Runnable* e *KeyListener*.

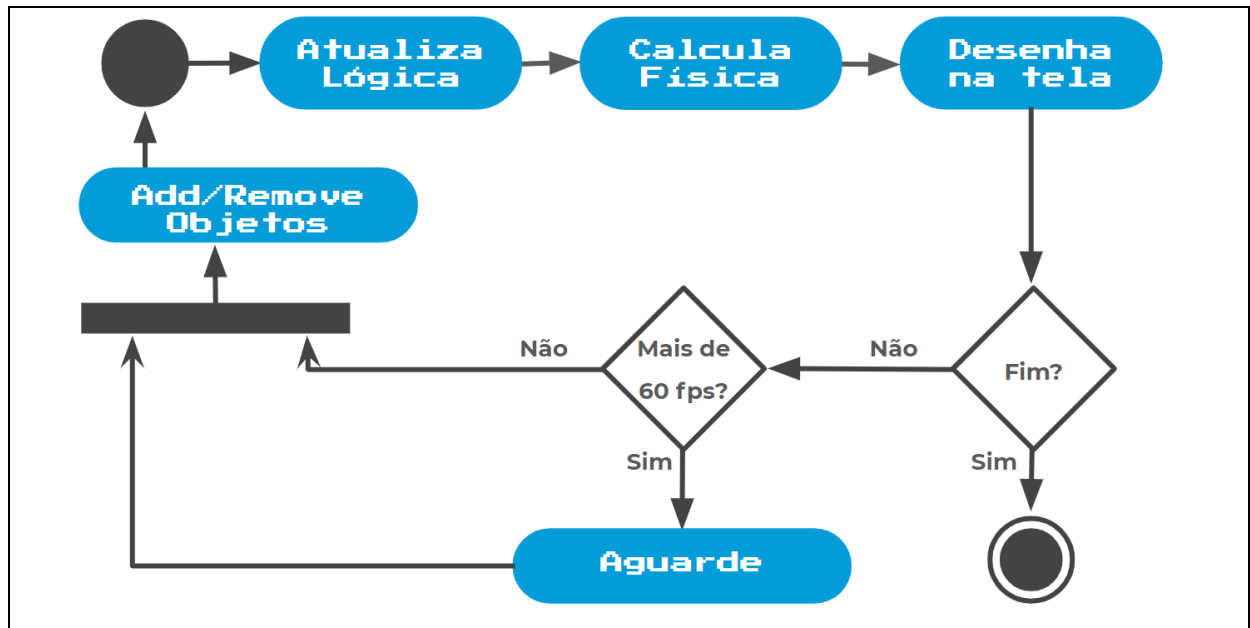
Abaixo um diagrama de classes UML demonstrando a estrutura básica do *GameApp*:



É essencial que a classe *GameApp* seja acessível globalmente, para permitir que um *GameObject* adicione/remova outro *GameObject* ou acesse a câmera do jogo, por exemplo. Também é importante garantir que apenas uma instância dessa classe seja criada, pois ela é a classe principal que controla todo o estado do jogo. Outra instância significaria outro jogo (software) completamente independente sendo executado no sistema. Para atender a esse requisito o padrão de design *Singleton* foi utilizado.

O método *run*, responsável por executar o game loop, tenta manter uma taxa de execução próxima de 60 vezes por segundos. Caso o computador seja mais potente e demore menos do que isso para processar os dados, o thread do jogo é colocado para dormir por alguns milissegundos até que a taxa seja respeitada. Isso é necessário para evitar que o game tenha velocidades diferentes em diferentes computadores. Nesse loop todos os três casos citados anteriormente sobre os *GameObjects* (atualização da lógica, tratamento de física e

renderização) são computados, na ordem apresentada. No diagrama de atividades abaixo é possível verificar a lógica executada em cada iteração do loop.



O guarda “*Fim?*” verifica se houve alguma interrupção do loop, como o jogador ter fechado a janela do jogo. Perceba que há um passo extra, logo no final do loop, para tratar a adição e/ou remoção de objetos.

Cálculo de Física

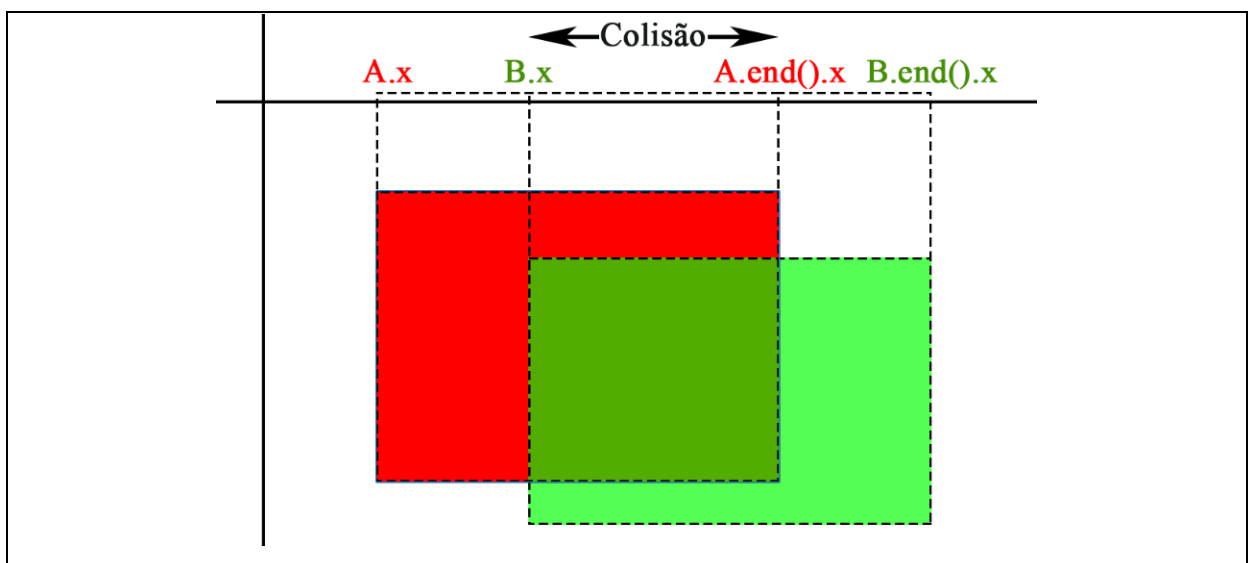
Durante o game loop, logo após a atualização da lógica dos objetos (método *update* dos *GameObjects*) e antes de desenhar na tela, há o cálculo e física entre os objetos. Essa etapa consiste em, para cada objeto na cena, percorrer todos os objetos (complexidade assintótica $O(n^2)$) e realizar três passos:

- Verificar se dois objetos estão se sobrepondo. Se sim, então houve uma colisão e os dois passos seguintes podem ser executados.
- Tratar a colisão quando necessário.
- Chamar o método de *CollisionCallback* dos dois objetos envolvidos.

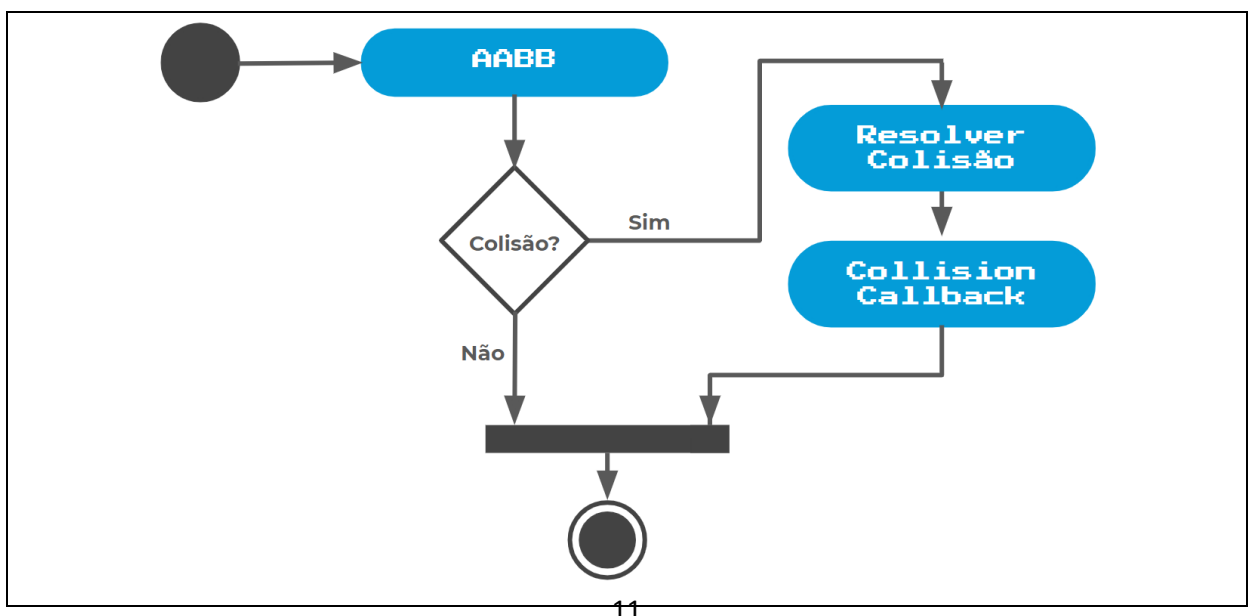
Nota: Há métodos e podas que reduzem a complexidade assintótica do cálculo de física, mas vai além do escopo desse jogo e, em casos de projetos mais simples como esse tal

complexidade não afeta o desempenho de forma significativa, já que a quantidade de objetos na cena é pequena.

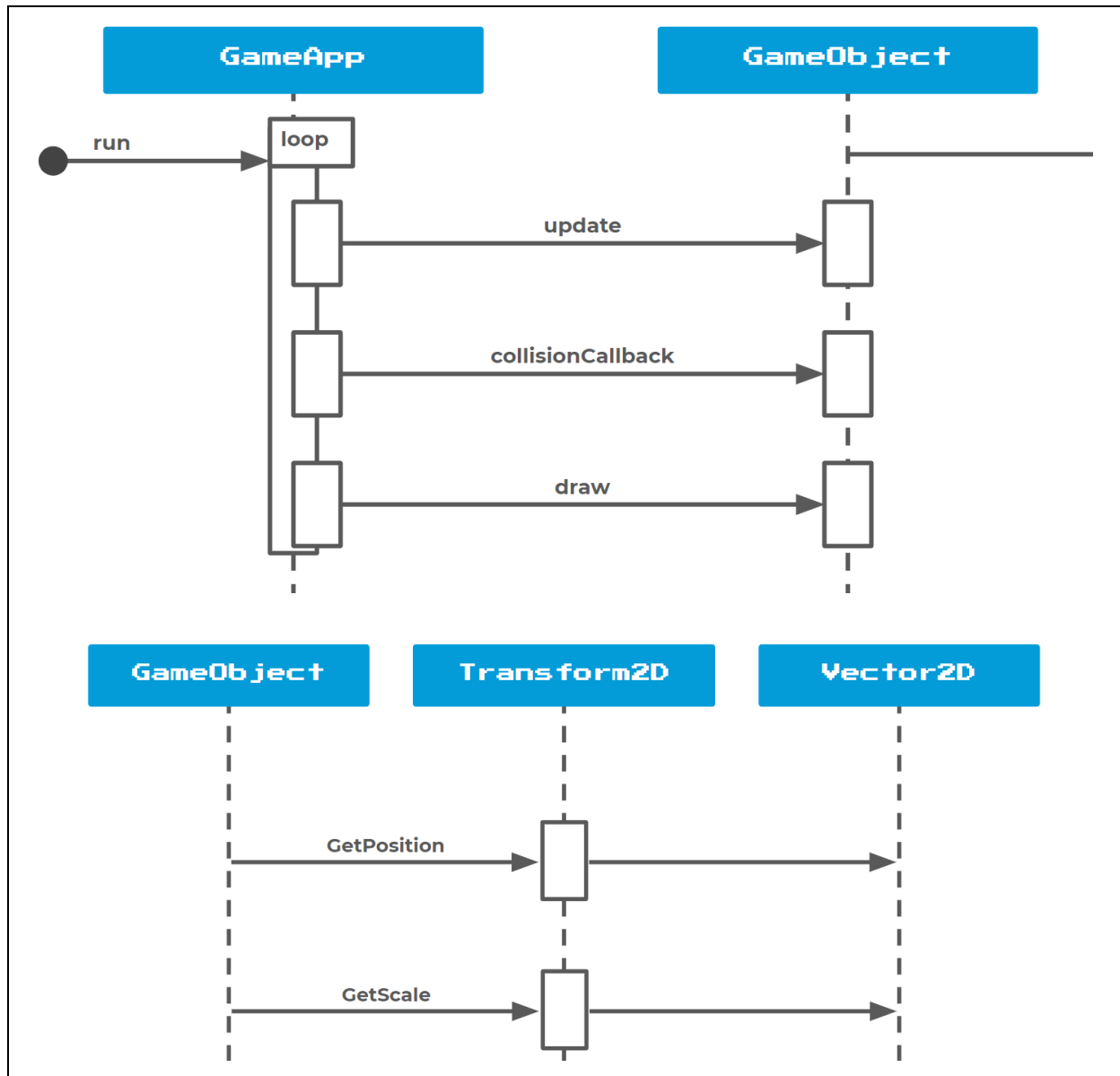
A colisão entre dois objetos, considerando que os *GameObjects* tem sempre uma forma geométrica retangular que não pode ser rotacionada (definida pela posição e escala do *Transform2D*), o algoritmo de detecção de colisão *Axis-Aligned Bounding Box*, ou **AABB** é o suficiente. Ele funciona verificando se há uma interseção entre cada eixo individualmente, conforme a imagem abaixo. Se houve colisão em todos os eixos (no caso, eixo X e eixo Y), então houve colisão entre os dois objetos.



Em outras palavras, para cada eixo (x e y), se **B.x** for menor que **A.end().x** e **A.x** for menor que **B.end().x**, houve uma colisão no eixo. No diagrama de atividades abaixo é possível identificar todas as etapas de colisão realizadas para cada par de objetos no jogo.



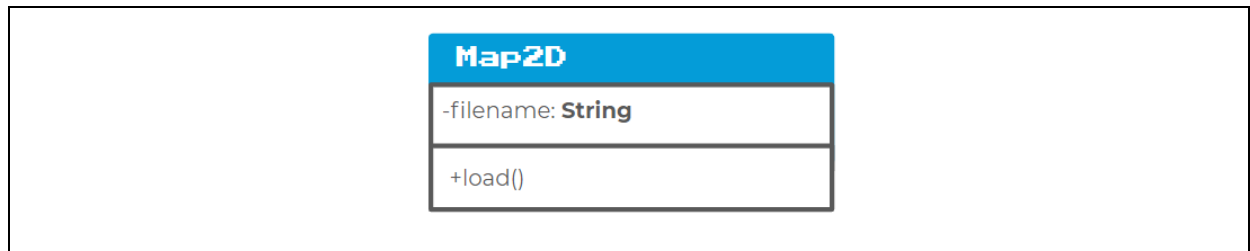
Abaixo o diagramas de sequência UML referentes ao `GameApp`, `GameObject`, `Transform2D` e afins.



Editor de Mapa

Um dos desafios reais enfrentados durante o desenvolvimento de jogos é a criação dos mapas do jogo. Por se tratar de algo extremamente visual e que geralmente cria e manipula uma quantidade alta de objetos (*GameObjects*), frequentemente um editor específico precisa ser desenvolvido. Para esse projeto, uma classe *Map2D* simples foi desenvolvida que tem como único objetivo ler um arquivo com informações do mapa do jogo em disco e carregar

todos os **GameObjects** em suas devidas posições, seguindo uma grade. Através do diagrama de classe UML abaixo se pode perceber a simplicidade do **Map2D**:



Quando o método ***load()*** é executado, o código lê o arquivo da string ***fileName***, que se assemelha a essa sequência de caracteres:

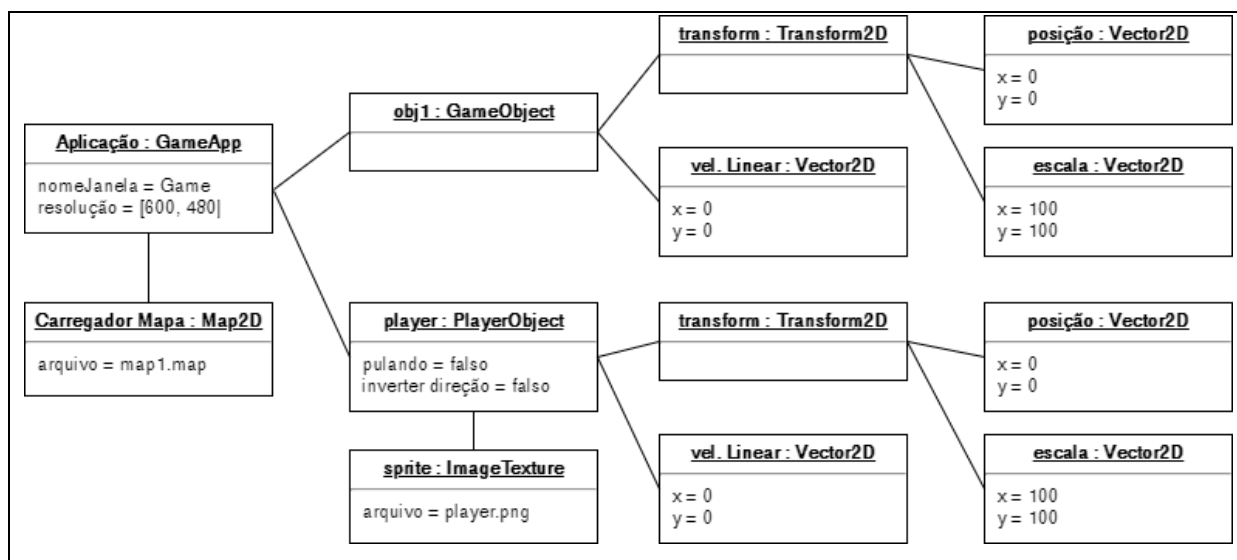
[illegible]

E adiciona ***GameObjects*** específicos de acordo com o caractere apresentado. O ponto final, por exemplo, é ignorado, sendo considerado como um espaço vazio. Onde há uma arroba (@), uma moeda é adicionada e onde há uma cerquilha (#), um inimigo. Os números são associados à lista (***ArrayList***) de sprites carregada no construtor do ***Map2D***, onde um número N lido no arquivo corresponde a posição N-1 na array.

Através dessa classe, inclusive, um mesmo arquivo de mapa pode ser lido e apresentado com aparências diferentes, alterando os *sprites* carregados, ou pode ser lido e adicionado ao jogo mais de uma vez (útil para reiniciar uma fase, por exemplo).

Visão Geral da Estrutura do Jogo

Para entender ainda melhor como as classes de objetos se relacionam dentro do jogo, o diagrama de objetos UML abaixo foi desenvolvido. Nele é possível identificar com clareza, por exemplo, que a classe **Map2D** apenas carrega os **GameObjects** dentro do **GameApp**, mas não os armazena em seu escopo.



Também é possível perceber que a distribuição de responsabilidades dentro das classes do jogo ficou uniforme, sem “*classes God*” – com responsabilidades demais.

Testes Unitários

Apesar de ser um desafio conhecido e discutido criar testes unitários para jogos digitais (principalmente funcionalidades gráficas), há classes em uma **Game Engine** que necessitam com prioridade máxima a realização de testes de unidade. Tais classes se referem principalmente a operações matemáticas e vetoriais. Conforme já abordado, algumas classes matemáticas foram implementadas para o desenvolvimento do projeto, tais como a **Vector2D** e **Transform2D**.

Foram criados testes unitários para todos os métodos referentes a essas classes utilizando a biblioteca **JUnit** do Java. Tais testes podem ser encontrados no diretório **src.test.com.game.math**. Nota: Caso algum dos testes esteja comentado, basta remover o comentário e executa-los normalmente.

Conclusão

Lançar mão dos recursos disponíveis através da Engenharia de Software, tais como os modelos de processo, decisões de projeto, diagramação, planejamento e arquitetura de software no âmbito do desenvolvimento de Jogos se mostraram extremamente prático e eficaz, permitindo a implementação em si do código completo desse trabalho em um período de tempo extremamente menor do que o esperado, mesmo sem vasto conhecimento sobre a linguagem Java. Isso mostra que o conhecimento e técnicas desenvolvidas e aplicadas durante a criação do Jogo serviram como fundamento essencial para a criação e escalabilidade do software em questão.

Também foi observado durante o desenvolvimento do trabalho que várias das empresas de médio e grande porte na indústria internacional de jogos fazem o uso das técnicas e métodos aplicados aqui para obter sucesso na criação de seus produtos.

A linguagem Java se mostrou eficiente e totalmente capaz de lidar com o projeto e um jogo digital, permitindo, inclusive, a criação de todo o trabalho sem a necessidade de instalar nenhuma biblioteca externa (exceto JUnit, para testes). Tal feito não é muito comum em outras linguagens populares na indústria de jogos como C++ e Python, resultando em uma vantagem de utilizar Java. Foi observado, também, que grandes jogos comerciais atuais foram desenvolvidos através dessa linguagem, como *Minecraft* (inicialmente escrito utilizando a biblioteca Java lwjgl.org) e *Equilinox* (página do jogo à venda na plataforma Steam: <<https://store.steampowered.com/app/853550/Equilinox/>>).

Referências

- Material de estudo sobre **Engenharia de Software**: <
<https://homepages.dcc.ufmg.br/~figueiredo/>>.
- Livro: **Game Programming Patterns**, autor Robert Nystrom. Cópia digital gratuita (oficial) disponível em <
<https://gameprogrammingpatterns.com/>>.
- Singleton Pattern (extraído do livro do autor Robert Nystrom citado anteriormente) <
<https://gameprogrammingpatterns.com/singleton.html>>.
- Livro: **Clean Code**, autor Robert Cecil Martin.
- Ian Sommerville. Engenharia de Software, 9a. Edição. 2011
- G. Booch, J. Rumbaugh, I. Jacobson. UML, Guia do Usuário, 2a Edição. Editora Campus, 2005.
- H. M. Deitel, P. J. Deitel. Java: Como Programar, 8a. Edição. Pearson, 2010.
- □ Koscianski, A.; Soares, M. S. Qualidade de Software, 2a Edição. Novatec, 2007.
- Os *sprites* do cenário utilizados na criação do jogo foram obtidos com licença CC0 (domínio público) através do link: <
<https://opengameart.org/content/a-platformer-in-the-forest>>.
- O *sprite* usado para exemplificar o funcionamento do **Transform2D** nesse documento foi extraído do jogo **Super Mario Bros**, propriedade intelectual da **Nintendo**.
- A Spiral Model of Software Development and Enhancement:
<https://csse.usc.edu/TECHRPTS/1988/usccse88-500/usccse88-500.pdf>.
- Spiral Model For Game Development: Techniques To Develop Games:
<http://gamedevelopertips.com/spiral-model-for-game-development/>.
- Requisitos não funcionais para Jogos Digitais:
<http://www.sbgames.org/sbgames2016/downloads/anais/157887.pdf>.
- Functional Requirements:
https://app.assembla.com/wiki/show/tank_wars/Functional_Requirements.