UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação Modular - 2017/1

PROJETO FINAL

ADLER'S ADVENTURE:

THE SEARCH FOR THE LOST CHINELO

Adler Melgaço Ferreira
Hiago de Souza Cruz
Marina Monteiro Moreira
Ronald Davi Rodrigues

1 - INTRODUÇÃO

Para o projeto final, o tema escolhido foi um jogo de plataforma 2D. A temática do jogo gira em torno de um membro do grupo, Adler, que é o personagem principal, sendo que vários elementos do jogo são baseados em características de sua vida que o tornam uma pessoa icônica, como os chinelos (que atuam como as moedas) e seu desgosto por vegetais (que são os inimigos).

Para o desenvolvimento do projeto, foi usada a framework libgdx, uma ferramenta não familiar à nenhum integrante, e o grupo tentou aplicar os princípios e utilizar os padrões de projetos orientados a objetos ao máximo.

2 - COMO O JOGO FUNCIONA

Como todo jogo de plataforma, Adler's Adventure é bem direto: percorra as fases superando seus obstáculos e chegue ao final antes de suas vidas acabarem.

Quando o personagem colide com um oponente, ele perde um “coração”, quando todos os corações são perdidos, o personagem perde uma vida, e volta ao início da fase. O jogo possui 3 curtas fases, cada uma com um mapa distinto.

2.1 - O que Adler faz

As únicas habilidades do personagem principal são correr e pular. O jogador deve, com apenas isso, superar todos os desafios dos mapas.

2.2 - Inimigos

 ***Cenourito***: Apenas anda pelo mapa. Um encontro com ele causa um coração de dano.

2.3 - Elementos das fases

 ***Gelo:*** Superfície escorregadia, caso não tenha cuidado, acabará caindo em algum abismo ou coisa pior.

 ***Espinhos:*** Terrenos pontiagudos que podem existir nos quatro sentidos. Um encontro com eles será fatal.

 ***Cogumelo:*** Os cogumelos empurram o personagem na direção contrária ao seu movimento, podem existir no mapa em todos os quatro sentidos

 ***Chinelos:*** São as moedas do jogo, aumentam o score do jogador.

 ***Bandeira da vitória*****:** Quando o personagem a alcança, o jogador passa de nível.

3 - EDITOR DE MAPAS

O editor de mapas foi criado para facilitar a criação de níveis. Inicialmente criado para auxiliar o desenvolvimento, ao longo do projeto ele foi estendido para que o usuário também pudesse utilizá-lo, oferecendo maior liberdade caso o jogador tenha o interesse de criar seus próprios níveis.

Os comandos que auxiliam a sua utilização são os seguintes:

- Botão esquerdo do mouse: Cria o elemento selecionado.
- Botão direito do mouse: Apaga o elemento.
- Ctrl + mouse: Ao segurar *ctrl* e mover o mouse, é possível mover a câmera até os limites do mapa.
- Tecla I: Abre ou fecha um mini-menu que permite a visualização e a seleção dos elementos possíveis de se colocar no mapa.
- Tecla U: Abre outro mini-menu que permite alternar entre os mapas já criados.
- Tecla ESC: Sai do jogo e também do editor de mapas.

Um maior detalhamento sobre o funcionamento do editor será feito mais a frente.

4 - IMPLEMENTAÇÃO

O jogo foi todo implementado na linguagem Java, auxiliada pela framework libgdx, como já mencionado. A seguir, uma descrição de cada pacote (como o código ficou muito extenso, a descrição por classes será feita com as mais fundamentais).

4.1 - collidable

É o principal pacote de tratamento de colisões. O tratamento de colisões é uma parte extremamente importante da implementação (pode-se dizer que é a base do funcionamento do jogo), e é feito em diversos pacotes, todos os quais herdam desse pacote.

Collidable é composto de uma interface e três classes abstratas, *CollidableObject*, *GameObject* e *DynamicCollider. CollidableObject* implementa a interface, definindo seus métodos de forma recursiva: as funções *handleCollision()* chamam a si mesmas, sendo que o primeiro parâmetro é

sempre convertido para o seu supertipo. Isso é feito para manter o princípio de substituição de Liskov.

DynamicCollider e *GameObject* são herdeiros de *CollidableObject*. *Game Object* define todos os objetos que tem um “corpo” no mapa. Elementos como imagem, *Bounding Box* e outras coisas essenciais à todos os outros objetos são implementados nessa classe. *Dynamic Collider* herda de *GameObject* e implementa funcionalidades necessárias à objetos que podem se mover.

4.2 - tile

Tiles são elementos objetos que colidem, mas não se movem. Uma fase, ou um mapa, é composto de vários tipos de tiles, que são blocos de tamanho único (32x32 px) sobre os quais colocamos os sprites, animados ou não. Dito isso, o
pacote Tile cuida da criação e diferenciação desses blocos.

Contém 8 classes, sendo uma abstrata (*AbstractTile*), da qual todos os Tiles herdam. São eles:

- *Coin* (os chinelos) → estende de *Pickup*, que por sua vez representa todos os tiles que interagem com o personagem principal para aumentar algum de seus parâmetros, como vida, velocidade, dinheiro (no momento apenas *Coin* está implementada).
- *Repulsor* (o cogumelo) → manda o personagem na direção contrária com velocidade fixa.
- *Tile* → representa um tile genéricos, que pode ser usado para representar chão, paredes, tiles decorativos, etc. Possui campos de atrito e velocidade máxima, que são usado quando o personagem colide com esses tiles.
- *Hazard* → é um tile com uma segunda *Bounding Box*, a *danger Area*, que causa dano no jogador quando ele entra em contato com esse tile.
- *LevelEndingTile* → um tile que marca o final das fases.
- *EmptyTile* → que representa um espaço vazio no mapa;

É importante notar que *EmptyTile* implementa a interface da framework *Serializable*, para salvar esses tiles no arquivo que contém o mapa. Isso acontece apenas em *EmptyTile* pois as únicas informações relevantes para a serialização desses tiles são a sua posição nos eixos x e y, então uma

serialização customizada permite que muita memória seja salva no processo de armazenar o mapa.

4.3 - enemy

Este é o pacote responsável pelos inimigos do jogo. Possui duas classes, a primeira é classe abstrata *AbstractEnemy*, que extende a classe *DynamicCollider* e implementa a interface IDamageable. Note que, dessa maneira, inimigos são objetos que podem ser danificados e destruídos.

O fato de ser uma classe abstrata permite que a criação de diferentes tipos de inimigos seja facilitada, já que basta estendê-la e especificar como será o tratamento de algumas propriedades na classe do novo inimigo criado.

A segunda é a classe *WalkerEnemy,* que representa os inimigos que apenas andam de um lado para o outro, como o Cenourito. O método *updateMovement()* da classe *DynamicCollider* é sobrescrito para implementar essa funcionalidade.

4.4 - gameScreens

O pacote *gameScreens* guarda as classes responsáveis pelas telas do jogo, como por exemplo a tela de menu inicial, o editor de mapas e a tela do nível em si.

A classe *InitialScreen* é a tela de menu inicial, responsável por inicializar o jogo, possibilitando ao jogador iniciar uma nova aventura, ir para o editor de mapas ou sair do programa, já as classe *BuilderScreen* e *StageScreen* inicializam o criador de mapas e a fase, respetivamente.

StageScreen é responsável por iniciar a fase, ativando propriedades como a música do nível, HUD(Heads-Up Display, são as informações que ficam na tela, como número de vidas, corações, etc) e o próprio mapa. Quando o jogador passa de nível, o método *nextLevel()* é chamado e um novo mapa é carregado para a tela.

Caso o jogador queira sair do jogo, a classe de UI *BackToMenuDialog* fica responsável por isso, criando a janela de pergunta e se certificando que o jogador realmente quer isso.

4.5 - level

O pacote *level* é utilizado para armazenar, carregar e salvar os níveis do jogo. Possui três classes: *Level, LevelNode* e *LevelSerializer*.

Level é a classe responsável por armazenar as fases do jogo. Uma propriedade importante de cada fase é que ela é composta de diversos mapas, os quais são armazenados em *LevelNodes.*

A a classe *LevelSerializer* possui a função de salvar e carregar dados dos mapas para os nós, e isso é feito através dos métodos *storeStage()* e *loadStage().* Através do caminho para localizar o arquivo e da classe padrão *Json*, é possível serializar as informações mais essenciais presentes no mapa, evitando um gasto enorme de memória.

4.6 - map

Esse pacote é responsável por gerenciar o mapa e as interações que ocorrem dentro dele. Possui 4 classes: *MapSerializer*, que faz a serialização do mapa em Json, *TileMap* e *CollisionMap*, que juntas criam e removem elementos do mapa e verificam se colisões vão ocorrer, e *Quadtree,* que auxilia na busca de blocos a colidir.

TileMap guarda os tiles e define diversas operações auxiliares à colisão. *CollisionMap* guarda um *TileMap* e insere os objetos dinâmicos na quadtree e em uma lista própria. Essa classe é responsável também por encontrar colisões entre objetos dinâmicos/tiles.

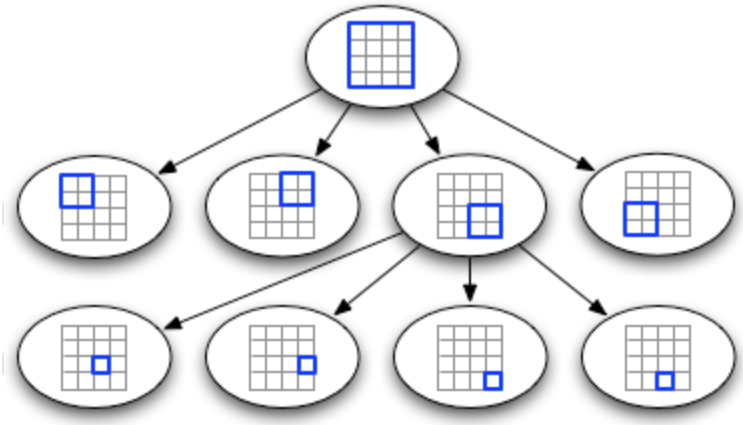
A checagem de colisões é feita em dois processos: primeiro são verificadas colisões com objetos dinâmicos, depois, com tiles estáticas. A cada frame de jogo, é percorrida a distância que o personagem vai se mover em teoria, a partir do seu vetor velocidade. Então, é verificado, com auxílio da quadtree se existe um objeto cuja distância até o jogador é menor do que a distância que ele quer percorrer.

Verificamos então colisões com tiles estáticos. Caso tenhamos encontrado algum objeto dinâmico na rota do personagem, procuramos por tiles que estejam ainda mais próximos. Isso é feito de modo mais simples, apenas olhando no mapa as coordenadas que o bloco do personagem ocuparia caso fizesse seu movimento completo.

Na classe *CollisionMap*, os métodos *closestCollisionX()* e *closestCollisionY()* são encarregados de encontrar as colisões mais próximas

que bloquearão a passagem do personagem, enquanto os métodos *horizontalCollision()* e *verticalCollision()* executam as colisões que não afetam o caminho do jogador.

Como dito anteriormente**,** uma quadtree é utilizada para auxiliar o tratamento de colisões envolvendo objetos dinâmicos, como inimigos e o próprio personagem.



Ela funciona da seguinte maneira: a região envolvida pela quadtree corresponde a todo o mapa, sendo que inicialmente possui quatro regiões, como em um plano cartesiano. A cada vez que um nó-folha estoura a sua capacidade de objetos armazenados, ele cria quatro filhos e redistribui os objetos que pertenciam ao antigo nó-folha, através dos métodos *split()* e *insert(),* respectivamente.

Cada nó-folha possui um *ArrayList* que armazena objetos da classe *DynamicCollider*, para mostrar em qual região cada um deles está. Dessa maneira, é possível realizar a detecção de possíveis colisões para um determinado objeto mais facilmente, já que, através dos métodos *getNearby()* e *retrieve(),* é possível verificar apenas uma região específica(a região que o objeto se encontra), e não todo o mapa.

4.7 - mapBuilder

O pacote *mapBuilder* contém as classes que lidam com o editor de mapas, possuindo seis delas: *MapBuilder, FolderSelector, TileSelector, ConfirmableTextField, LevelSelector* e *NewLevelField*.

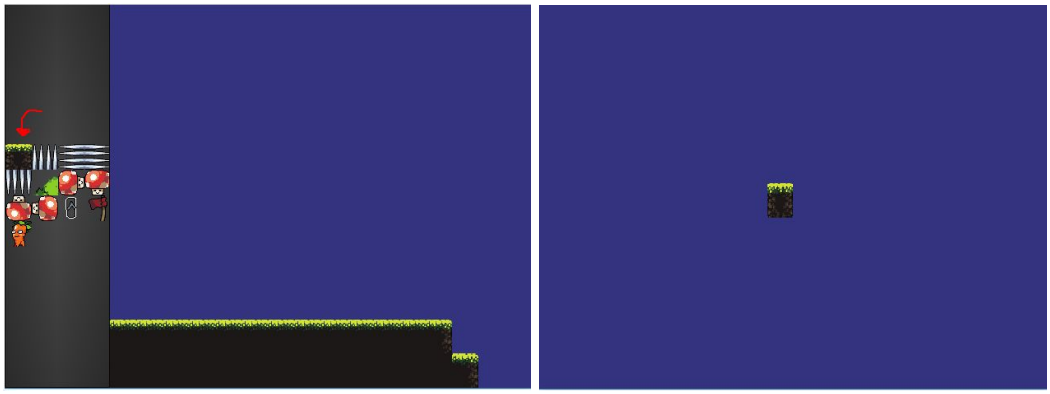
MapBuilder é a classe principal deste pacote, sendo responsável por construir a interface do editor, como o mini-menu que mostra os possíveis

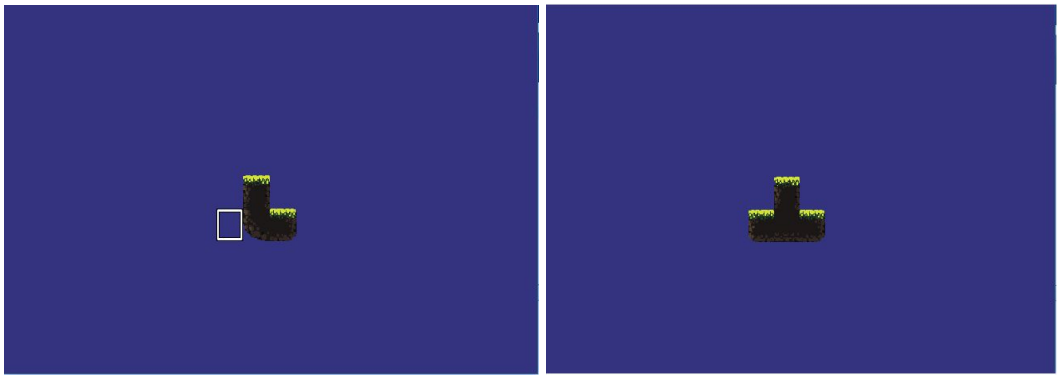
elementos a serem utilizados. O método *handleInput()* é encarregado de lidar com a entrada oriunda do teclado, como movimentar a câmera.

Há também uma classe interna, com o nome de *Processor*, que é incubida de lidar com a entrada proveniente do mouse, possuindo métodos como *touchDown()* e *touchDragged()*, que são chamados quando um dos botões estão sendo apertados ou arrastados, respectivamente.

Além disso, uma importante feature da classe *MapBuilder* é a presença do método *applyTileSet()*, que aplica um algoritmo chamado autotiling, facilitando a junção de tiles com seus vizinhos, deixando-a mais natural e evitando que o usuário tenha que selecionar um tile a cada momento que quiser um novo modelo. Para facilitar o entendimento, segue a imagem:

Percebe-se que existem diferentes tipos de blocos, pois dependendo da forma como estão conectados, o seu design deve ser alterado para fazer algum sentido. Dessa maneira, caso o método não fosse utilizado, o mini-menu teria que conter cada um desses 16 exemplares para permitir que o usuário os selecionasse, causando um custo maior no sentido de código mas fazendo também com que a interface ficasse desagradável. Com o método *applyTileSet(),* o que ocorre é seguinte:





Apenas com um exemplar é possível criar todos os possíveis blocos, pois como é possível notar, no momento de sua criação, tanto os blocos vizinhos como o que será criado se adaptam às condições já estabelecidas do level.

Quando se deseja criar um novo mapa, é criado um novo arquivo para guardá-lo, para que o usuário possa acessá-lo mais tarde. São as classes *FolderSelector* e *LevelSelector* as responsáveis por isso, permitindo a criação de um novo mapa na pasta desejada, enquanto a classe *ConfirmableTextField* fica encarregada de criar as janelas de confirmação.

Já a classe *TileSelector* é a que cria o seletor de tiles, funcionando como um mini-menu que mostra as escolhas de tile que o usuário pode realizar.

4.8 - player

Esse pacote possui apenas duas classes e está encarregado de tudo diz respeito ao personagem principal e ao jogador.

A classe *ControllableCharacter* implementa todas as colisões que podem ocorrer ao personagem principal. Através dos métodos *handleCollision()*, que são sobrescritos da interface *ICollidable*, a classe implementa os códigos de colisão com cada um dos tipos de tile. Além disso, recebe o input do jogador.

A classe *HUD* tem a função de mostrar as informações importantes ao jogador, como seu score atual, seu número de vidas, número de corações e pode servir no futuro para mostrar algum outro tipo de informação, como um possível power-up ou algo parecido.

4.9 - state

Ao longo do mapa, pode-se dizer que o personagem Adler está sempre em um determinado estado, seja ele se movimentando, pulando ou simplesmente parado, sendo assim, uma máquina de estados pode ser utilizada para essa abstração. Foi com essa intuição que as classes no pacote *State* foram desenvolvidas, com o auxílio também do padrão de projeto State.

Dessa maneira, cada uma das classes nesse pacote representam um estado do personagem, sendo que todas as classes que estendem *AnimationState* são estados que simbolizam uma animação, como pular ou se mover.

4.10 - game

No pacote *game* estão as classes que envolvem aspectos mais técnicos do jogo e não se encaixam em nenhum outro pacote, como o hitbox, câmera e as interfaces relacionadas a dano e morte. Seus detalhes serão dados a seguir:

Classe BoundingBox:

É a classe que define a hitbox de um objeto, possuindo propriedades como largura e altura. Com isso, a classe acaba por auxiliar também na detecção de colisões, já que uma colisão é detectada quando a bounding box de dois objetos se intercedem. Quando um objeto chama os métodos *getHorizontalDistance()* ou *getVerticalDistance(),* a distância entre seu eixo x ou o eixo y com os eixos do parâmetro é calculada e retornada.

Classe CollisionInfo:

Esta classe guarda informações sobre a colisão, mais precisamente sobre em qual eixo ela está ocorrendo, a maneira pela qual foi feita permite que ela seja facilmente estendida no futuro, possibilitando o armazenamento de outros tipos de informações.

Classe FollowerCamera:

Esta classe é herdeira da classe *OrthographicCamera*, a qual vem da biblioteca libgdx. Sua função é fazer com que a câmera do jogo acompanhe o movimento do personagem ao longo do mapa.

Ela também é utilizada no editor de mapas, para o usuário poder mover a câmera e interagir com outras partes do mapa que não estão nos limites da tela.

É o método *update()* que permite esse cálculo, traduzindo as coordenadas da câmera e verificando se ela ainda está nos limites do mapa e da tela.

Classe TextureSheet:

Esta classe existe para facilitar a criação de animações e o algoritmo do autotiling. Através de uma matriz, texturas são armazenadas, sendo que cada linha representa um “spritesheet” diferente, mas antes de continuar, o que é um “spritesheet”?

Na imagem acima, essa sequência de sprites é o que é chamado de “spritesheet”, e cada “Adler” representa um frame da animação. É com o spritesheet que as animações se tornam possíveis.

Desta maneira, cada linha da matriz *images* nesta classe representa uma animação diferente, a partir do método *toAnimation()* e da classe padrão *Animation*, que vem da libgdx, é possível reunir todas elas em um *ArrayList*, tornando o tratamento delas mais eficiente.

Classe Adler: É a classe que inicializa o programa.

Classe Axis: É apenas um *enum* para os eixos cartesianos.

Interface IDestroyable: Interface que possui o método *die(),* que determina como acontece a morte de um objeto, como Cenourito ou o próprio Adler.

Interface IDamageable: Interface que possui o método *takeDamage()*, que calcula o dano sofrido do objeto.

Interface IGenerator: Interface que possui o método generateFrom(), que permite gerar um objeto a partir dos dados de um outro objeto.

Interface IBuildableObject: Interface dos objetos que podem ser criados no mapa através do map builder.

5 - COMPILAÇÃO

Para exibir as opções disponíveis para a construção do projeto, digite ‘make’. Caso não tenha instalado o programa ‘gradle’, basta dar ‘make install’ (isso requerirá acesso de superuser para ser instalado). Após a instalação do gradle, digite ‘make build’ para compilar e linkar os arquivos gerados. Apoś isso, basta dar ‘make run’ para que a execução do jogo aconteça.

Para remover os arquivos gerados pela compilação, o ‘gradle’ e suas dependências, basta digitar ‘make clean’ (isso também requerirá acesso de superuser).

6 - FERRAMENTAS

Como desenvolver um jogo é um projeto bastante extenso e com muitos elementos que requerem atenção além do código puro, utilizamos várias ferramentas para diversos fins:

NetBeans: IDE onde o jogo foi programado, compilado, e testado.

Gradle: Ferramenta para construção e execução dos arquivos do jogo em ambiente Linux.

Libgdx: biblioteca auxiliar da qual usamos algumas implementações muito básicas, como a câmera, texturas, etc.

LMMS: Software usado para fazer as músicas de cada fase.

***GraphicsGale*:** Software usado para animar os sprites dos personagens dinâmicos.

Paint: Software usado para fazer os sprites (e toda a arte do jogo).

7 - CONCLUSÃO

Ao longo do projeto foi possível adquirir um grande arcabouço de conhecimento a respeito de diversos aspectos de POO e programação em geral, já que fazer um jogo é uma tarefa surpreendente, tanto por sua dificuldade quanto pelo interesse que tínhamos em realizá-lo.

Os aspectos que mais dificultaram o projeto foram o tratamento da colisão, devido a enorme quantidade de variáveis envolvidas, e também o desenvolvimento do editor de mapas, cuja serialização tomou grande parte do tempo.

Infelizmente, não foi possível implementar todas as mecânicas planejadas, como mais inimigos e um possível boss, mas, apesar do tempo curto para tal tarefa, o grupo se sentiu satisfeito com o resultado alcançado e apreciou bastante a oportunidade.

8 - REFERÊNCIAS

https://github.com/libgdx/libgdx/wiki

http://higherorderfun.com/blog/2012/05/20/the-guide-to-implementing-2d-platformers/

http://www.saltgames.com/article/awareTiles/

https://en.wikipedia.org/wiki/Double_dispatch

https://gamedev.stackexchange.com/questions/72387/how-do-i-handle-collision-response-between-many-different-types-of-game-objects