UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

ABEL ROCHA ESPINOSA - 9277513 - Turma 94
JOÃO MARCELO ROSSETTO FERNANDES DA SILVA - 9277833 - Turma 04
LUCAS PIPA CERVERA - 8094403 - Turma 94
THYAGO RIBEIRO DOS SANTOS - 9377491 - Turma 94
YANN RIGHAS ABRAHÃO - 9277030 - Turma 94

RELATÓRIO DO EP DE COMPUTAÇÃO ORIENTADA A OBJETOS
JOGO SHOOT ‘EM UP

SÃO PAULO
2016

SUMÁRIO

CRÍTICAS AO CÓDIGO ORIGINAL

Sobre o código original, apesar de rodar sem problemas aparentes, se abordado com um olhar mais técnico, são ressaltados pontos onde más práticas de programação são usadas. Começando a crítica, deve-se ressaltar o fato do uso de arrays para armazenar as entidades do jogo, que foi citado na própria especificação do projeto. O array é uma estrutura estática, o que traz problemas quanto ao armazenamento (tanto excedente quando insuficiente) e manipulação de seus conteúdos, que podem facilmente levar à redundância.

A classe Main.java do código é onde está toda a lógica do código, sem contar as inicializações e atualizações de entidades (presente em GameLib.java), sem nenhum tipo de estruturação orientada a objetos. Todas as variáveis e configurações das entidades do jogo são definidas e manipuladas dentro da classe Main.java. O fato do uso de arrays para cada uma das entidades faz com que hajam muitas manipulações semelhantes, mas que são feitas de maneira repetida no código. Esse problema gerado pelo uso dos arrays poderia facilmente ser solucionado com a implementação de APIs.

Quando se trata da classe GameLib.java, as APIs usadas ainda assim são de caráter gráfico ou de eventos, e há alguns casos de redundância entre os métodos implementados.

Ao longo do código, fica claro que ele não foi pensando de uma maneira que a POO fosse visada. Não existem muitas classes, objetos, a herança é muito pouco usada (somente em GameLib.java, que deveria ser tratada como de terceiros), não há nenhuma API que não seja relacionada à interface gráfica e eventos.

Quanto ao jogo, em uma visão de usuário, a simplicidade da interface (sem um contador de vida, por exemplo) e a dificuldade do jogo se tornam quase como bloqueios para o jogo não ser jogado.

NOVA ESTRUTURA DE CLASSES E INTERFACES

DESCRIÇÃO

A nova estrutura é baseada em pacotes. São divididas, em maioria, em entidades. Cada uma representa algum artifício do jogo. Tanto os “personagens” do jogo - Inimigos, chefes, jogador, powerups e projéteis - , quanto eventos - colisão, spawn de personagens, modificador de estado e modificador de armas - e design - constantes, background, estado de tela, fase, utilidades e HUD - são divididos em cada pacote de sua entidade. Há ainda as classes Main.java, Vector2D.java, Gamelib.java, GameCore.java e um pacote de controle de entidades.

Cada entidade tem, de uma forma geral, uma disposição de classes de acordo com seu tipo:

As entidades de personagem, em geral, têm:

- Renderizador;
- As definições do personagem;
- Verificador de colisão.

Cada entidade “irmã”, como os dois inimigos e os dois chefes, ou então os dois power-ups, estendem uma classe mãe - inimigo ou power-up- que traz atributos em comum entre eles.

As entidades Powerup e inimigos/chefes tem cada um uma classe mãe que é estendida.

As entidades de eventos têm um comportamento mais específico:

- Colisão :
    - Interface “com colisão”, para ser estendida pelos personagens;
    - Um verificador de colisão;
    - Um estado de colisão

- Spawner:
    - Um spawner para cada personagem;
    - Um spawner que invoca o gerenciador;
    - Um gerenciador de spawn.
- Estado:
    - Um atualizador de estado;
    - Um estado de explosão.
- Armas:
    - Definição da arma (cada arma tem um projétil diferente, e é definido pelos powerups ou personagem à qual pertence);
    - Um “criador” de armas;

Os pacotes de design não necessáriamente são entidades, e têm, assim como as entidades de evento, um comportamento mais específico:

- Constantes
    - Enumerador de armas;
    - Enumerador de entidades.
- Background:
    - Uma definição das estrelas do plano de fundo.
- Estado de Tela:
    - Verificador de estado de tela;
    - Contexto de tela;
    - Controle e execução de tela.
- HUD:
    - Renderizador da barra de vida;
    - Definição da barra de vida;
    - Renderizador de informações (numero de inimigos e fase);
    - Definição das informações;

Fases e Utilidades não são entidades:

- Utilidades:
    - Duas classes que lidam com o tempo de jogo.
- Fases:
    - Um spawner para chefes;
    - Um spawner para inimigos;
    - Um spawner para power-ups;
    - Controle e execução dos spawners e fase em geral.

O gerenciador/controlador de entidades é um pacote que tem como função executar, controlar e verificar as entidades, como player, tiros e inimigos.
Há também a classe Main.java, que lê os arquivos de configuração, Vector2D.java, que faz os cálculos gráficos, Gamelib.java, que tem a maioria dos inicializadores e desenha os gráficos, verifica controle do jogador, e atualiza o display, e GameCore.java, que serve para tratar as mecânicas do jogo em geral.

JUSTIFICATIVA

A estrutura do código refatorado tem como função facilitar a manutenção do programa. Tanto para inserção de novas funcionalidades, como os chefes e power-ups, quanto para correção de erros que podem surgir ao longo da implementação. Alterar algum detalhe no código se torna algo muito mais simples, já que há bastante reutilização de métodos e classes, não sendo necessárias várias alterações pontuais em cada parte do código, e sim algumas poucas alterações que ajam de uma maneira geral no contexto.

USO DE APIs EM SUBSTITUIÇÃO AOS ARRAYS

A parte da API usada para substituir os vetores que gerenciavam o conjunto de informações da aplicação foi a classe ArrayList. Devido ao tipo de uso a que os vetores estavam submetidos, precisando atualizar constantemente seu conteúdo, a substituição por ArrayList foi vantajosa, uma vez que pode-se aproveitar o dinamismo oferecido por esse tipo de estrutura, já implementado na biblioteca do Java.

A maneira como foi implementada se manteve bastante semelhante, porém, contando com a facilidade que a classe traz.

PADRÕES DE PROJETO

DESCRIÇÃO

Os padrões de projetos utilizados no código são:

- Facade:
    - CollisionChecker.java
- Factory:
    - WeaponsFactory.java
- Singleton:
    - Time.java
- State:
    - Boss1CollisionState.java;
    - Boss1State.java;
    - Boss2CollisionState.java:
    - Boss2State.java;
    - CollisionState.java;
    - Enemy1CollisionState.java;
    - Enemy1State.java;
    - Enemy2CollisionState.java:
    - Enemy2State.java;
    - InfosState.java;
    - LifeBarState.java;
    - PlayerCollisionState.java;
    - PlayerState.java;
    - PowerUp1State.java;
    - PowerUp1CollisionState.java;
    - PowerUp2State.java;
    - PowerUp2CollisionState.java;
    - ProjectileState.java.

JUSTIFICATIVA

Os padrões foram usados da seguinte maneira:

- Facade foi usado para facilitar a complexidade do código e simplificar o uso, pois esconde o detalhamento bastante extenso do tratamento de colisão;
- Factory foi usado para a criação de diferentes armas (chefe, inimigo e jogador), pois foi uma maneira simples de implementar diferentes características em armas que são semelhantes;
- Singleton foi usado no controle do tempo do jogo, e ajudou a centralizar e unificar os comandos, já que funciona para permitir apenas uma instancia;
- State foi usado para auxiliar na tomada de decisão no jogo, como movimentação das entidades e colisões, pois se torna simples obter os estados que as entidades se encontram.

NOVAS FUNCIONALIDADES

Com a nova estrutura de código, fica muito mais simples implementar qualquer característica, visto que há lógicas aplicadas que podem ser reproduzidos sem muitas diferenças. As novas funcionalidades são:

- Barra de vida
    - Uma classe que estende entidade e tem os métodos de redução e atualização da vida e seu retorno para a barra;
    - Renderizador da barra de vida, implementa“estado de entidade”.

- Fases;
    - Uma classe que gerencia o spawn de powerups, inimigos e chefes;
    - Uma classe de spawn para cada um dos spawnables, com definições e métodos que são usadas no gerenciador.

- 2 chefes de fase:
    - Renderizador do chefe, que estende “estado de entidade” ;
    - As definições do chefe, que estende inimigo, que define movimentação e armas;
    - Verificador de colisão, que implementa “estado de colisão”.

- Power ups:
    - Verificador de colisão, que implementa “estado de colisão” e a sobrescreve para tratar especificamente o powerup.
    - Renderizador do powerup, que estende “estado de entidade” ;
    - As definições do powerup, que estende powerup que define sua movimentação;

Ainda foi implementado um HUD (heads-up display) que contém o número de inimigos da fase e o número da fase em que o jogador se encontra, além das barras de vida do jogador e do chefe de fase quando esse aparece.

A classe Main.java foi modificado para que pudesse receber os arquivos de entrada como parâmetro das fases. Contudo, ainda é possível editar os arquivos que serão recebidos no código fonte. Se não for inserido nenhum argumento, o jogo rodará duas fases padrão, que podem ser modificadas na raiz do projeto.

Os dois chefes implementados apresentam uma barra de vida e um sinalizador de acerto (flash), assim como o jogador. Os dois ainda possuem uma movimentação diferente, tanto entre eles, quanto ao resto dos inimigos.

Além do power up do escudo que orbita o jogador, ainda foi implementado um power up de “mega-tiro”, que atravessa os inimigos comuns (piercing shot), detonando-os e só é destruído quando atinge o chefe de fase, tirando dois pontos de vida dele.

O fato de existir muito uso de herança no código é um benefício causado pela refatoração do código. Se não houvesse qualquer tipo de organização do código, como era o original, seria muito mais complexo a criação de todas as funcionalidades extras, já que seria necessário muita repetição de código e redundância.

Contudo, com a implementação das funcionalidades extras, surgiu um problema (bug): quando dois inimigos são atingidos pelo mesmo projétil o programa trava sem lançar excessões e o usuário precisa fechar a janela. Por esse motivo, não é recomendado criar uma fase com muitos inimigos simultâneos.