Relatório do Trabalho Final

Bacharelado em Ciência da Computação - Padrões de Projeto
Discentes: Matheus Marchi Moro e Celio Slomp

Este documento relata sobre um projeto de simulação de colisões que foi refeito utilizando os padrões de projeto *Strategy* e *Iterator*.

1 Introdução

O projeto original [3] simula movimentação, aceleração e colisão entre corpos retangulares e circulares com algoritmos específicos para colidir círculo com círculo e círculo com retângulo [2] [5]. Como a lógica de colisão é limitada devido ao fato de não utilizar um procedimento geral que se aplica a qualquer polígono, não foi possível otimizar esta parte específica com padrões de projeto. Contudo, pôde-se desacoplar tanto os diferentes comportamentos dos objetos que representam polígonos quanto a combinação de pares de polígonos para a verificação de colisão com os padrões *Strategy* e *Iterator* numa aplicação nova [4].

2 Características do simulador

O simulador inicial é um motor de colisões em duas dimensões escrito em *C++* que utiliza a biblioteca *Simple DirectMedia Layer 2* para visualização dos objetos e vetores matemáticos para o cálculo de física.

Sua estrutura é composta por uma classe *App* que possui o laço principal do programa e duas listas de corpos que são submetidos à simulação. Esta classe utiliza as classes estáticas *DisplayLogic* para desenhar as figuras na janela, *MovementLogic* para mover os corpos e *CollisionLogic* para verificar e tratar colisões. As listas de corpos, por usa vez, armazenam ponteiros para objetos RectBody ou CircleBody, respectivamente corpos retangulares e circulares. Desta forma, App itera sobre estas listas de corpos e os move, colide e mostra na tela, entre outros controles. O diagrama de classes está na figura 1.

OBS.: O diagrama da figura 1 foi simplificado para fins de clareza. Desta forma, as variáveis de controle, o regulador de tempo, a classe que representa vetores bidimensionais e diversos métodos foram desprezados. Além disso, as classes estáticas *Exhibition* e *Mechanics* foram destrinchadas para *DisplayLogic*, *MovementLogic* e *CollisionLogic* para se adequarem à compreensão do leitor.

3 Problemas

3.1 Detecções de colisão

Sejam n corpos, App realiza $n^2 + n$ iterações para cada frame de simulação. São n vezes para a movimentação e exibição dos corpos e n^2 vezes para as verificações de colisão. Como não houve a implementação de um algoritmo mais eficaz para estas verificações, foi necessário o uso de um algoritmo simples porém de ordem de complexidade $O(n^2)$ que escala mal. O número n^2 pode expressar uma matriz de iterações A, onde cada elemento a_{ij} da matriz é um par (b_i, b_j) de corpos.

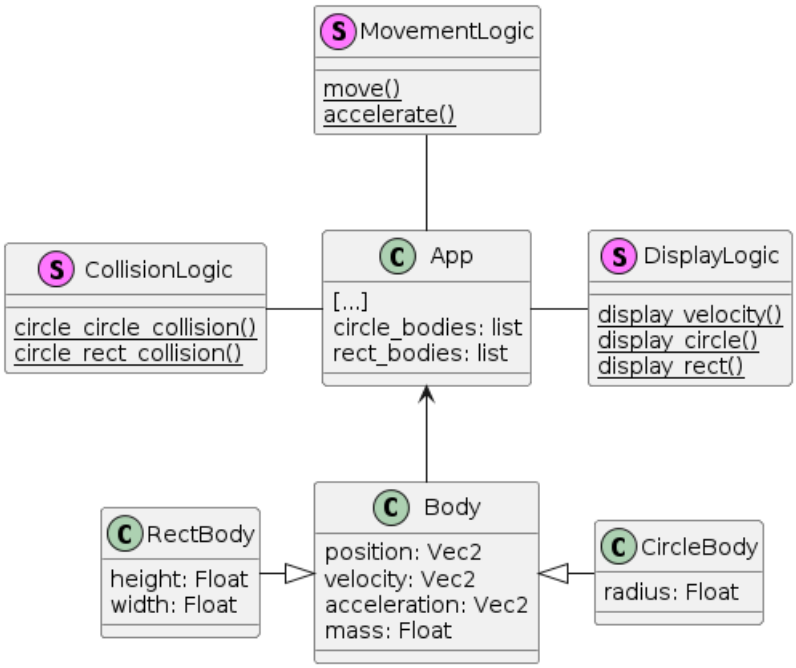

Figura 1: Diagrama de classes simplificado do simulador

$$A = \begin{bmatrix} (b_1, b_1) & (b_1, b_2) & \cdots & (b_1, b_j) \\ (b_2, b_1) & (b_2, b_2) & \cdots & (b_2, b_j) \\ \vdots & \vdots & \ddots & \vdots \\ (b_i, b_1) & (b_i, b_2) & \cdots & (b_i, b_j) \end{bmatrix}$$

Ao iterar sobre os elementos da matriz, *App* efetua as seguintes operações:

- Se $b_i = b_j$, continua para a próxima iteração pois não é possível colidir um corpo com ele mesmo. Neste caso, é toda a diagonal da matriz, ou seja, são n iterações dispensadas;
- Se $b_i \neq b_j$, verifica se os corpos estão se aproximando. Caso estejam, verifica se um está sobrepondo o outro, isto é, se está ocorrendo uma colisão. Nesta condição, modifica os vetores de velocidade dos objetos. Aqui, é importante ressaltar que mesmo se os corpos estiverem se sobrepondo, não colidem se a distância entre eles tender ao aumento.

Entretanto, há um problema evidente neste método. Caso, por exemplo, a colisão (b_1, b_2) seja apurada, posteriormente não vai ser necessário verificar a colisão (b_2, b_1). O evento acontece também para (b_1, b_3) e (b_3, b_1), (b_2, b_3) e (b_3, b_2), e assim por diante. Em suma, para todo par (b_m, b_n), a problemática acontece para (b_n, b_m). Deste modo, todas as iterações inferiores à diagonal da matriz também são dispensadas, visto que não se precisa analisá-las. Portanto, a matriz de iterações efetiva A' é a seguinte matriz, que será utilizada na refatoração:

$$A' = \begin{bmatrix} 0 & (b_1, b_2) & (b_1, b_3) & \cdots & (b_1, b_j) \\ 0 & 0 & (b_2, b_3) & \cdots & (b_2, b_j) \\ 0 & 0 & 0 & \cdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & (b_{i-1}, b_j) \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}$$

3.2 Acoplamento forte de representações de corpos

Originalmente, decidiu-se representar somente retângulos e círculos nas simulações. Foi implementado o *CircleBody* de maneira a ser visível, movimentar-se, acelerar e exibir vetor de velocidade. O *RectBody*, por sua vez, não se move e nem exibe seu vetor de velocidade, mesmo o possuindo. Se o programador quisesse, neste caso, implementar um círculo que não se move, ele teria que criar uma classe nova e modificar a lógica de iterações dos corpos na *App* criando uma lista específica para objetos desta classe. Caso deseje retângulos invisíveis que se movem, também deverá conceber outra classe diferente das demais e adicionar mais uma lista na *App*.

Percebe-se então que os diferentes comportamentos dos polígonos estão fortemente acoplados no projeto, o que gera manutenção desnecessária. Para somente duas representações, a implementação ainda não é tão trabalhosa. Entretanto, como é altamente provável que se queira simular diferentes polígonos e adicionar comportamentos num desenvolvimento futuro do simulador, viu-se a oportunidade de implementar o padrão de projeto Estratégia para resolução desta problemática.

4 Refatoração

4.1 Principais mudanças

Por questões de usabilidade e legibilidade, o projeto refatorado foi escrito em *Python* e utilizou-se a biblioteca *Pygame* com o mesmo intuito da *Simple DirectMedia Layer 2*. Ademais, conforme o diagrama da figura 2, adicionou-se ao projeto:

- a interface *Iterator* e sua classe concreta *BodyIterator* que se relacionam ao padrão Iterador;
- as interfaces responsáveis pelo funcionamento do padrão Estratégia, que são *MovementBehav*, *VelDisplayBehav* e *BodyDisplayBehav*;
- um ponteiro `body_iter` na *App* para uma instância de *BodyIterator*;
- ponteiros em *Body* para diferentes comportamentos de corpos, que são os atributos com final `_behav`.

Além disso, por conta do Estratégia, foi possível eliminar as listas específicas de corpos em *App* (`circle_bodies` e `rect_bodies`) e manter somente uma que armazena objetos *Body* generalizados (`bodies`).

A lógica da classe *MovementLogic* foi incorporada na classe concreta de comportamento *MovementBehav*. De maneira análoga, *DisplayLogic* foi para as classes *VelDisplayBehav* e *BodyDisplayBehav*.

4.2 Aplicação do padrão *Iterator*

Para o programa iterar sobre a matriz A', foi utilizado o padrão Iterador para isto [6]. O Iterador Concreto, que é a classe *BodyIterator*, foi implementada de modo a se especializar em oferecer um serviço de iteração mais efetivo para a verificação de colisões dos corpos. Esta classe recebe a lista de corpos de *App*, representada pela matriz A, monta internamente uma lista dos pares especificados na matriz A', armazena um índice atual *index* e oferece métodos de iteração sobre esta lista. A lista dos pares tem o nome `pairs` e os métodos são `has_next()`, `next()` e `first()`. Não foi utilizada uma

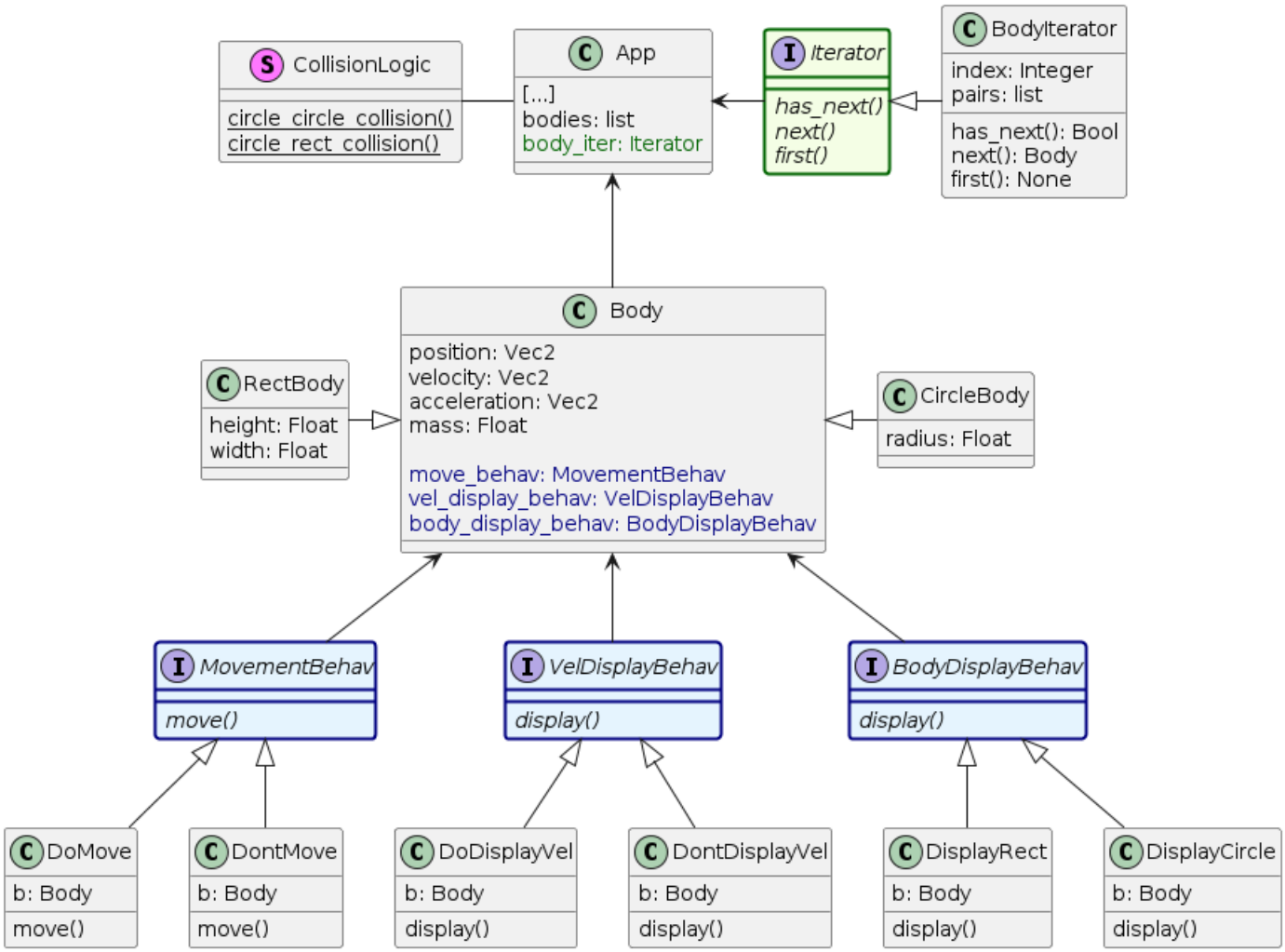

Figura 2: Diagrama de classes refatorado do simulador

classe *ConcreteCollection* pois a estrutura `pairs` é simples demais para necessitar gestão própria.

O tipo de lista utilizado em todo o projeto foi o tipo primitivo `list` do Python. Este tipo de lista é armazenado como um array na memória e possui um acesso de tempo constante [8], o que não contradiz a próxima constatação: desprezando o tempo de acesso à lista `pairs`, já que é constante, o algoritmo que a itera tem ordem de complexidade $O(\frac{n^2-n}{2})$, que é aproximadamente 100% melhor que o algoritmo anterior. Percebe-se que, nesta ordem de complexidade, n^2 representa toda a matriz A, enquanto que n é sua diagonal e a fração $1/2$, ao multiplicar n^2 e n, corresponde à sua matriz triangular inferior. O número

$$\frac{n^2 - n}{2}$$

oferece a quantidade de elementos da lista que é utilizada pelo método `has_next()`. Curiosamente, também é possível obter esta quantidade ao calcular

$$\frac{n!}{2!(n-2)!}$$

que é a combinação simples de todos os corpos tomados de 2 em 2.

Os métodos desta classe são caracterizados a seguir.

- `has_next()`: Retorna um valor booleano *True* se a lista ainda possui elementos a serem iterados. Caso esteja no último elemento, retorna *False*;
- `next()`: Retorna o atual par de corpos (bi, bj) na forma de tupla e incrementa `index` em 1;
- `first()`: Faz com que `index` receba o valor 0.

Com esses três métodos e a lógica de concepção da lista `pairs` embutida no construtor da classe, é possível utilizar a mesma estrutura várias vezes sem precisar recriá-la. *App* então usa os pares retornados pelo método `next()` e os passa como parâmetros diretamente aos métodos da classe *CollisionLogic* após uma verificação de tipos, sem precisar de uma implementação *hard code* para gerir as iterações.

4.3 Aplicação do padrão *Strategy*

Usufruiu-se do padrão de projeto Estratégia [7] para desacoplar as diferentes particularidades dos corpos e oferecer uma gestão dinâmica de suas características. Ao criar os atributos com fim `_behav` na classe *Body* que apontam para comportamentos concretos, possibilitou-se também a escalabilidade do projeto, visto que para alterar os procederes dos corpos basta mudar as instâncias de comportamentos antes do laço principal. E, para adicionar, é suficiente criar novos comportamentos concretos e/ou acrescentar ponteiros na *Body*. Tudo isto sem precisar de modificações na *App*.

Não obstante, o projeto original contém uma classe chamada *Persistance* que carrega as listas de corpos e armazena os resultados da simulação. Por conta do *Strategy*, tornou-se possível a leitura de diferentes peculiaridades dos corpos a partir de um *input*, ou arquivo, o que estabelece uma simulação mais personalizável.

Os métodos das classes concretas são detalhados a seguir.

- *MovementBehav*, método `move()`
  - *DoMove*: Adiciona a velocidade à posição do corpo e depois adiciona a aceleração à velocidade do corpo. Ou seja, faz basicamente

    ```
    b.position = b.position + b.velocity
    b.velocity = b.velocity + b.acceleration
    ```

    onde `b` é o ponteiro do corpo, `position` é o vetor de posição, `velocity` é o vetor de velocidade e `acceleration` é o vetor de aceleração. A classe *Vec2*, responsável pela administração dos vetores matemáticos, já faz a sobrecarga do operador de soma;
  - *DontMove*: Executa `pass` [1]. Este comando faz nada;
- *VelDisplayBehav*, método `display()`
  - *DoDisplayVel*: Exibe o vetor de velocidade do objeto. Pinta uma reta na tela a partir da posição do corpo até a posição + velocidade e depois a completa com retas menores para formar a ponta da flecha;
  - *DontDisplayVel*: Faz nada. Não mostra o vetor;
- *BodyDisplayBehav*, método `display()`
  - *DisplayRect*: Desenha quatro retas na tela de modo a formarem o retângulo. Utiliza a posição do objeto, sua altura e sua largura;

– *DisplayCircle*: Divide uma circuferência em uma quantidade x de fatias para deduzir o ângulo α entre cada secção em radianos. Ou seja, uma circunferência desenhada com este algoritmo tem x lados. O número escolhido no projeto foi 18 pois é uma quantidade confortável de lados e não muito custosa. Após isso, o algoritmo cria dois vetores, rotaciona um deles em α radianos e depois desenha uma reta de um vetor a outro e rotaciona ambos x vezes, até completar a circunferência;
– (IMPLEMENTADO EM AMBIENTE DE TESTES) *DisplayNothing*: Faz nada (corpo invisível).

Referências

[1] Simple statements - Python 3.12.2 documentation. *pass keyword*. URL: `https://docs.python.org/3/reference/simple_stmts.html#pass`.

[2] Chad Berchek. *Dimensional Elastic Collisions without Trigonometry*. 2009. URL: `https://www.vobarian.com/collisions/2dcollisions2.pdf`.

[3] Matheus M. Moro e Celio Slomp. *Motor simples de colisões refatorado*. 2024. URL: `https://github.com/mathsmm/bcc-padroes-trabalho`.

[4] Matheus M. Moro e Celio Slomp e Eduardo E. Poli. *Simulador de fuga de multidões*. 2023. URL: `https://github.com/mathsmm/bcc-poo2-simulador`.

[5] CodeTheBoat. *Circle rectangle collision detection (Clamp function)*. 2019. URL: `https://youtu.be/_xj8FyG-aac?si=mqoiYGAHvI9QcvhF`.

[6] Refactoring Guru. *Iterator*. URL: `https://refactoring.guru/design-patterns/iterator`.

[7] Refactoring Guru. *Strategy*. URL: `https://refactoring.guru/design-patterns/strategy`.

[8] The Python Wiki. *TimeComplexity*. 2023. URL: `https://wiki.python.org/moin/TimeComplexity`.