UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO E ENGENHARIA DE COMPUTAÇÃO

GABRIEL CARVALHO TAVARES
GUILHERME D'ÁVILA PINHEIRO
MAXIMUS BORGES DA ROSA

Simulador de Sistema Solar: Implementação em Python e Haskell

Relatório apresentado como requisito parcial para a obtenção de conceito na Disciplina de Modelos de Linguagens de Programação

Prof. Dr. Leandro Krug Wives
Orientador

Porto Alegre
2025

SUMÁRIO

1 INTRODUÇÃO

Este trabalho apresenta a implementação de um simulador de sistema solar, desenvolvido como projeto para a disciplina de Modelos de Linguagens de Programação (MLP). O simulador foi implementado em duas linguagens distintas: Python, seguindo o paradigma orientado a objetos, e Haskell, seguindo o paradigma funcional.

O objetivo principal deste projeto é demonstrar como diferentes paradigmas de programação podem ser aplicados para resolver um mesmo problema computacional, explorando as vantagens e limitações de cada abordagem. Além disso, o simulador permite visualizar o comportamento dos corpos celestes sob a influência da gravidade, reproduzindo fenômenos como órbitas elípticas e colisões.

A simulação gravitacional é baseada na Lei da Gravitação Universal de Newton, que define que a força entre dois corpos é proporcional ao produto de suas massas e inversamente proporcional ao quadrado da distância entre eles. Esta lei é aplicada para calcular as forças entre todos os pares de corpos celestes, e a partir dessas forças são calculadas as acelerações, velocidades e posições dos corpos ao longo do tempo.

1.1 Sobre o Projeto

O simulador implementa um sistema de N-corpos que interagem gravitacionalmente, permitindo a simulação de sistemas como o nosso sistema solar. Os principais componentes do projeto incluem:

- **Representação de corpos celestes**: planetas, estrelas e outros objetos espaciais com propriedades como massa, posição, velocidade e raio.
- **Cálculo de forças gravitacionais**: implementação da Lei da Gravitação Universal para calcular as interações entre os corpos.
- **Simulação temporal**: atualização do estado do sistema ao longo do tempo usando integração numérica.
- **Detecção de colisões**: identificação e tratamento de colisões entre corpos celestes.

O sistema foi implementado em duas linguagens diferentes para permitir a comparação entre paradigmas de programação:

- **Python**: Implementação orientada a objetos, utilizando classes para representar corpos celestes, vetores e o simulador.
- **Haskell**: Implementação funcional, usando tipos de dados algebraicos e funções puras para modelar o comportamento do sistema.

1.2 Organização do Relatório

Este relatório está organizado da seguinte maneira:

O Capítulo 2 apresenta a arquitetura e design do sistema, detalhando as estruturas de dados e algoritmos utilizados em ambas as implementações.

O Capítulo 3 descreve a implementação do simulador, com foco nas técnicas utilizadas para calcular as forças gravitacionais, atualizar o estado do sistema e detectar colisões.

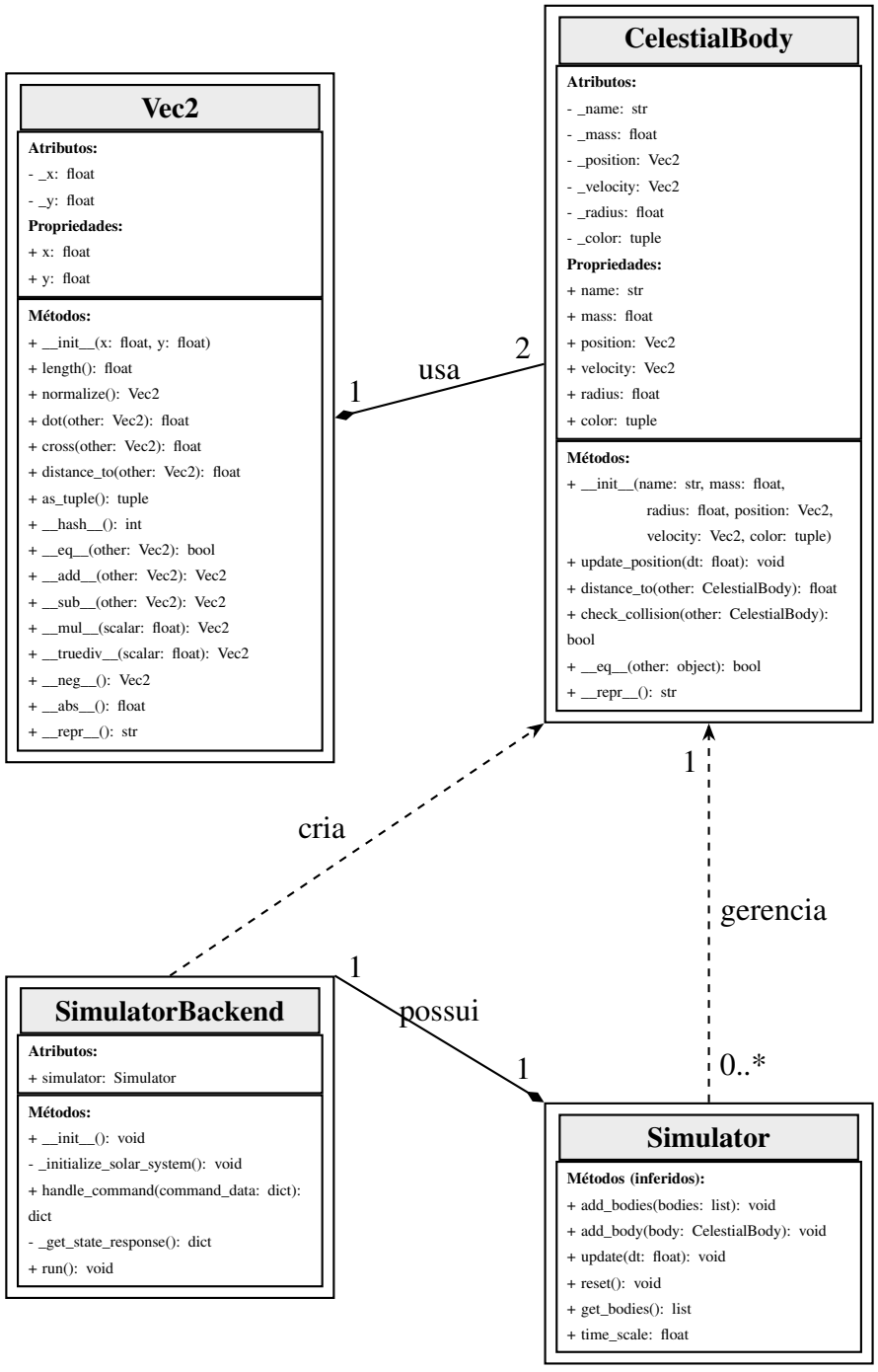
A conclusão resume os principais resultados obtidos, discute as diferenças entre as implementações e sugere possíveis melhorias e extensões para o projeto.

2 ARQUITETURA E DESIGN DO SISTEMA

Este capítulo apresenta a arquitetura e design do simulador de sistema solar, descrevendo as estruturas de dados e algoritmos utilizados em ambas as implementações.

2.1 Implementação em Python

A implementação em Python segue uma abordagem orientada a objetos, com as seguintes classes:



2.1.1 Estrutura de Classes

• **Vec2**: Representa um vetor bidimensional com operações como adição, subtração, multiplicação por escalar, normalização e cálculo de distância.

• **CelestialBody**: Representa um corpo celeste com propriedades como nome, massa, posição, velocidade, raio e cor. Inclui métodos para atualizar a posição e verificar colisões.

• **Simulator**: Gerencia a simulação, mantendo uma coleção de corpos celestes e atualizando seus estados ao longo do tempo. Implementa a lógica para cálculo de forças gravitacionais e detecção de colisões.

Além disso, o módulo `physical_constants` define constantes físicas utilizadas na simulação, como a constante gravitacional.

2.1.2 Encapsulamento e Validação

A implementação em Python faz uso extensivo de encapsulamento e validação de dados:

• Properties são utilizadas para controlar o acesso aos atributos das classes, permitindo validação de valores e garantindo a integridade dos dados.

• Validações rigorosas são aplicadas para garantir que valores como massa e raio sejam positivos, e que vetores sejam instâncias da classe Vec2.

• Exceções são lançadas quando valores inválidos são fornecidos, facilitando a identificação de erros.

Tabela 2.1: Principais métodos da classe Simulator em Python

Método	*Descrição*
add_body	Adiciona um corpo celeste ao sistema
remove_body	Remove um corpo celeste do sistema
update	Atualiza o estado do sistema para o próximo passo de tempo
_update_physics	Calcula e aplica as forças gravitacionais entre os corpos
_manage_collisions	Detecta e gerencia colisões entre corpos

Fonte: autor

2.2 Implementação em Haskell

A implementação em Haskell segue uma abordagem funcional, utilizando tipos de dados algebraicos e funções puras para modelar o comportamento do sistema.

2.2.1 Estrutura de Dados

A implementação em Haskell é composta pelos seguintes tipos de dados principais:

• **CelestialBody**: Tipo de dado que representa um corpo celeste, contendo campos como nome, massa, posição, velocidade, raio e cor.

• **Simulator**: Tipo de dado que representa o simulador, contendo uma lista de corpos celestes, a escala de tempo e o número de colisões detectadas.

2.2.2 Funções Puras

A implementação em Haskell utiliza funções puras para modelar o comportamento do sistema:

• **updatePosition**: Atualiza a posição de um corpo celeste com base em sua velocidade e no intervalo de tempo.

• **computeGravitationalForce**: Calcula a força gravitacional entre dois corpos celestes.

• **updateSimulator**: Atualiza o estado do simulador para o próximo passo de tempo, calculando as forças gravitacionais, atualizando posições e velocidades, e detectando colisões.

Tabela 2.2: Principais funções do módulo Simulator em Haskell

Função	*Descrição*
initializeSolarSystem	Inicializa o simulador com os planetas do sistema solar
updateSimulator	Atualiza o estado do simulador para o próximo passo de tempo
updatePhysics	Calcula e aplica as forças gravitacionais entre os corpos
manageCollisions	Detecta e gerencia colisões entre corpos
computeAccelerations	Calcula as acelerações dos corpos a partir das forças

Fonte: autor

2.3 Comparação entre as Implementações

As implementações em Python e Haskell diferem em vários aspectos importantes:

• **Mutabilidade**: A implementação em Python utiliza objetos mutáveis, permitindo a modificação direta do estado dos objetos. Em contraste, a implementação em Haskell utiliza estruturas de dados imutáveis, criando novas versões dos objetos a cada modificação.

• **Efeitos colaterais**: A implementação em Python faz uso de efeitos colaterais, como modificar o estado interno de objetos. A implementação em Haskell evita efeitos colaterais, utilizando funções puras que não modificam o estado do sistema.

• **Tipagem**: Python utiliza tipagem dinâmica com anotações de tipo opcionais, enquanto Haskell utiliza tipagem estática forte, permitindo a detecção de erros de tipo em tempo de compilação.

• **Organização do código**: A implementação em Python organiza o código em classes com métodos, enquanto a implementação em Haskell organiza o código em tipos de dados e funções separadas.

3 IMPLEMENTAÇÃO DO SIMULADOR

Este capítulo descreve os detalhes da implementação do simulador, com foco nas técnicas utilizadas para calcular as forças gravitacionais, atualizar o estado do sistema e detectar colisões.

3.1 Modelagem Física

A simulação é baseada na Lei da Gravitação Universal de Newton, que define a força gravitacional entre dois corpos de massas m_1 e m_2 separados por uma distância r como:

$$F = G\frac{m_1 m_2}{r^2}$$

Onde G é a constante gravitacional, cujo valor é aproximadamente $6.67430 \times 10^{-11} m^3 kg^{-1} s^{-2}$.

Para simular o movimento dos corpos celestes, a aceleração de cada corpo é calculada a partir da força resultante e da segunda lei de Newton ($F = ma$):

$$a = \frac{F}{m}$$

A velocidade e posição de cada corpo são então atualizadas utilizando integração numérica. No simulador, utilizamos o método de Euler:

$$v(t + \Delta t) = v(t) + a(t)\Delta t$$
$$p(t + \Delta t) = p(t) + v(t)\Delta t$$

Onde $v(t)$ é a velocidade no tempo t, $a(t)$ é a aceleração no tempo t, $p(t)$ é a posição no tempo t e Δt é o intervalo de tempo.

3.2 Cálculo de Forças Gravitacionais

3.2.1 Implementação em Python

Na implementação em Python, o cálculo das forças gravitacionais é realizado pelo método `_compute_gravit` da classe `Simulator`. Este método calcula a força gravitacional entre dois corpos celestes e retorna a força aplicada no primeiro corpo devido ao segundo:

```
# Calcular vetor distancia
dx = body2.position.x - body1.position.x
dy = body2.position.y - body1.position.y
distance_squared = dx**2 + dy**2
distance = math.sqrt(distance_squared)

# Evitar divisao por zero ou distancias muito pequenas
min_distance = (body1.radius + body2.radius) * 1e-9
if distance < min_distance:
    distance = min_distance
    distance_squared = distance**2

# Calcular magnitude da forca gravitacional
force_magnitude = G * body1.mass * body2.mass / distance_squared

# Componentes da forca (unitario * magnitude)
if distance > 0:
    force_x = force_magnitude * dx / distance
    force_y = force_magnitude * dy / distance
else:
    force_x = force_y = 0
```

Este método é chamado pelo método `_update_physics` para calcular as forças entre todos os pares de corpos celestes, aplicando a terceira lei de Newton (ação e reação) para otimizar os cálculos e reduzir a complexidade computacional.

3.2.2 Implementação em Haskell

Na implementação em Haskell, o cálculo das forças gravitacionais é realizado pela função `computeGravitat` do módulo `CelestialBody`:

```
computeGravitationalForce :: CelestialBody -> CelestialBody -> (Double, Double)
computeGravitationalForce b1 b2 = (fx, fy)
  where
    g = 6.67430e-11
    (x1, y1) = position b1
    (x2, y2) = position b2
    dx = x2 - x1
    dy = y2 - y1
    dist = sqrt (dx*dx + dy*dy)
    force = g * mass b1 * mass b2 / (dist * dist)
```

```
    fx = force * dx / dist
    fy = force * dy / dist
```

As forças resultantes para cada corpo são calculadas pela função `computeGravitationalForces`, que soma as forças de todos os outros corpos:

```
computeGravitationalForces :: [CelestialBody] -> [(Double, Double)]
computeGravitationalForces bodies =
    [ sumForces body | body <- bodies ]
  where
    sumForces b1 = foldl addForce (0, 0)
                   [ computeGravitationalForce b1 b2 | b2 <- bodies, b1 /= b2 ]
    addForce (fx1, fy1) (fx2, fy2) = (fx1 + fx2, fy1 + fy2)
```

3.3 Detecção de Colisões

3.3.1 Implementação em Python

Na implementação em Python, a detecção de colisões é realizada pelo método `check_collision` da classe `CelestialBody`, que verifica se a distância entre dois corpos é menor que a soma de seus raios:

```
def check_collision(self, other: 'CelestialBody') -> bool:
    """Verifica se este corpo colide com outro baseado na distancia e raios"""
    if not isinstance(other, CelestialBody):
        raise TypeError("Other must be a CelestialBody instance")
    return self.distance_to(other) <= (self.radius * 1e9 + other.radius * 1e9)
```

O gerenciamento de colisões é realizado pelo método `_manage_collisions` da classe `Simulator`, que identifica pares de corpos que colidiram e remove o corpo de menor massa:

```
def _manage_collisions(self):
    """Gerencia colisoes do sistema"""
    destroyed_bodies = set()

    bodies = self.get_bodies()
    if len(bodies) < 2:
        return

    for i, body1 in enumerate(bodies):
        for body2 in bodies[i+1:]:
            if body1.mass == 0 or body2.mass == 0:
                continue
            if not body1.check_collision(body2):
```

```
                continue
            # Colisao detectada: Determinar qual corpo remover
            if body1.mass < body2.mass:
                destroyed_bodies.add(body1.name)
            elif body1.mass > body2.mass:
                destroyed_bodies.add(body2.name)
            else:
                # Se as massas forem iguais, remover ambos
                destroyed_bodies.add(body1.name)
                destroyed_bodies.add(body2.name)

    destroyed_bodies = list(destroyed_bodies)
    self.num_collisions += len(destroyed_bodies)
    self.remove_bodies(destroyed_bodies)
```

3.3.2 Implementação em Haskell

Na implementação em Haskell, a detecção de colisões é realizada pela função `checkCollision` do módulo `CelestialBody`:

```
checkCollision :: CelestialBody -> CelestialBody -> Bool
checkCollision b1 b2 = distance b1 b2 <= (radius b1 + radius b2)
```

O gerenciamento de colisões é realizado pela função `manageCollisions` do módulo `Simulator`, que identifica pares de corpos que colidiram e retorna uma lista de nomes de corpos a serem removidos:

```
manageCollisions :: Simulator -> [String]
manageCollisions sim = map (name . weakest) collisions
  where
    pairs = [ (b1, b2) | (b1:rest) <- tails (bodies sim), b2 <- rest ]
    collisions = filter (uncurry checkCollision) pairs
    weakest (a, b) = if mass a <= mass b then a else b
```

3.4 Inicialização do Sistema Solar

Para facilitar a utilização do simulador, foi implementada uma função que inicializa o sistema com os planetas do Sistema Solar:

```
initializeSolarSystem :: Simulator
initializeSolarSystem =
    let timeScale = 24 * 3600 * 10 -- 10 days in seconds
```

```
    sun     = CelestialBody "Sol"      1.989e30 (0, 0)         (0, 0)
       40 (255, 220, 0)
    mercury = CelestialBody "Mercurio" 3.285e23 (5.791e10, 0) (0, 47.87e3)  6
       (169, 169, 169)
    venus   = CelestialBody "Venus"    4.867e24 (1.082e11, 0) (0, 35.02e3)  9
       (255, 198, 73)
    earth   = CelestialBody "Terra"    5.972e24 (1.496e11, 0) (0, 29.78e3)
       10 (100, 149, 237)
    mars    = CelestialBody "Marte"    6.39e23  (2.279e11, 0) (0, 24.077e3) 8
       (205, 92, 92)
    jupiter = CelestialBody "Jupiter"  1.898e27 (7.785e11, 0) (0, 13.07e3)
       25 (255, 165, 0)
    saturn  = CelestialBody "Saturno"  5.683e26 (1.429e12, 0) (0, 9.69e3)
       22 (238, 232, 205)
    uranus  = CelestialBody "Urano"    8.681e25 (2.871e12, 0) (0, 6.81e3)
       16 (173, 216, 230)
    neptune = CelestialBody "Netuno"   1.024e26 (4.495e12, 0) (0, 5.43e3)
       15 (0, 0, 128)
in Simulator [sun, mercury, venus, earth, mars, jupiter, saturn, uranus,
   neptune] timeScale 0
```

Os planetas são inicializados com valores aproximados de suas massas, posições, velocidades, raios e cores. As posições e velocidades são simplificadas, considerando órbitas circulares no plano XY.

3.5 Implementação do Frontend

Além do backend, foi desenvolvido um frontend em Python usando a biblioteca Pygame para visualização do sistema solar. O frontend foi projetado para ser independente da implementação do backend, permitindo a comunicação tanto com o backend Python quanto com o backend Haskell.

3.5.1 Arquitetura do Frontend

O frontend foi implementado seguindo uma arquitetura modular, com os seguintes componentes principais:

- **MainWindow**: Gerencia a janela principal da aplicação, processa eventos de entrada do usuário e coordena a atualização e renderização do sistema.

- **Renderer**: Responsável pela renderização dos corpos celestes, rastros de órbitas e elementos da interface de usuário.

• **Camera**: Implementa uma câmera virtual que permite ao usuário navegar pelo sistema solar, com recursos de zoom e movimentação.

• **CelestialBodyCreator**: Permite a criação interativa de novos corpos celestes através de cliques do mouse.

• **BackendProxy**: Fornece uma camada de abstração para a comunicação com o backend, permitindo a troca entre diferentes implementações.

3.5.2 Comunicação entre Frontend e Backend

A comunicação entre o frontend e o backend é realizada através de um protocolo baseado em JSON, onde o frontend envia comandos e o backend responde com o estado atualizado do sistema. Esta abordagem permite que o mesmo frontend seja utilizado com diferentes implementações de backend.

```
def update(self, dt, paused=False):
    """Atualiza a simulacao"""
    command = {
        'command': 'update',
        'dt': dt,
        'paused': paused
    }
    if self._send_command(command):
        response = self._wait_for_response()
        if response and 'bodies' in response:
            self.current_state = response
            return True
    return False
```

O `BackendProxy` implementa uma comunicação assíncrona com o backend através de um processo separado, utilizando subprocessos e threads para evitar o bloqueio da interface do usuário:

```
def start(self):
    """Inicia o processo backend"""
    try:
        self.process = subprocess.Popen(
            self.backend_cmd,
            stdin=subprocess.PIPE,
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True,
            bufsize=1  # Line buffered
```

```
            )
            self.running = True

            # Thread para ler respostas do backend
            self.reader_thread = threading.Thread(target=self._read_responses)
            self.reader_thread.daemon = True
            self.reader_thread.start()

            # Aguardar estado inicial
            initial_response = self._wait_for_response(timeout=5.0)
            if initial_response:
                self.current_state = initial_response
            return True

        except Exception as e:
            print(f"Erro ao iniciar backend: {e}")
            return False
```

3.5.3 Visualização e Interface de Usuário

O `Renderer` é responsável pela visualização do sistema solar, implementando técnicas para renderizar corpos celestes, rastros de órbitas e estrelas de fundo:

- **Corpos Celestes**: Renderizados como círculos com tamanho proporcional ao raio do corpo, com cores correspondentes aos planetas reais.
- **Rastros de Órbitas**: Implementados como uma série de pontos que representam as posições anteriores dos corpos, permitindo visualizar as trajetórias orbitais.
- **Estrelas de Fundo**: Geradas aleatoriamente para criar um ambiente espacial, com efeito de paralaxe ao mover a câmera.
- **Interface de Usuário**: Inclui informações sobre o estado da simulação, controles disponíveis e um slider para ajustar a massa de novos corpos celestes.

```
def draw_bodies_with_camera(self, screen, bodies_data, camera, simulation_speed=1)
    :
    """Renderiza corpos celestes usando dados JSON do backend"""
    # Desenhar estrelas de fundo primeiro
    self._draw_stars(screen, camera)
```

```
        # Atualizar e desenhar rastros
        self._update_trails(bodies_data, simulation_speed)
        self._draw_trails(screen, camera)

        # Desenhar corpos celestes
        for body_data in bodies_data:
            # Calcular posicao na tela considerando a camera
            screen_x = (body_data['position'][0] - camera.x) / (self.scale / camera.
                zoom) + self.width / 2
            screen_y = (body_data['position'][1] - camera.y) / (self.scale / camera.
                zoom) + self.height / 2

            # Calcular raio na tela
            radius_on_screen = max(1.0, body_data['radius'] * camera.zoom)

            # Desenhar apenas se estiver visivel na tela
            margin = radius_on_screen + 5  # Margem maior para suavizacao
            if (-margin <= screen_x <= self.width + margin and
                -margin <= screen_y <= self.height + margin):

                # Converter cor de lista para tupla se necessario
                color = tuple(body_data['color']) if isinstance(body_data['color'],
                    list) else body_data['color']

                # Usar coordenadas float para melhor precisao
                center_x = int(round(screen_x))
                center_y = int(round(screen_y))
                radius_int = int(round(radius_on_screen))

                # Desenhar corpo principal
                if radius_int >= 1:
                    # Circulo preenchido anti-aliased
                    pygame.gfxdraw.filled_circle(screen, center_x, center_y,
                        radius_int, color)
                    pygame.gfxdraw.aacircle(screen, center_x, center_y, radius_int,
                        color)
```

3.5.4 Controle da Câmera

A classe `Camera` implementa uma câmera virtual que permite ao usuário navegar pelo sistema solar:

- Suporte para movimentação nos eixos X e Y usando as teclas de seta ou WASD.
- Funcionalidade de zoom usando a roda do mouse, permitindo aproximar ou afastar a visualização.
- Aplicação de limites para evitar que o usuário se perca no espaço.
- Ajuste automático da velocidade de movimento baseado no nível de zoom, para uma navegação mais intuitiva.

```
def update(self, dt, keys_pressed):
    """Atualiza a posicao da camera baseada nas teclas pressionadas"""
    # A velocidade eh inversamente proporcional ao zoom para manter movimento
       consistente
    current_speed = self.base_speed / self.zoom

    # Movimento horizontal
    if keys_pressed.get('left', False):
        self.x -= current_speed * dt
    if keys_pressed.get('right', False):
        self.x += current_speed * dt

    # Movimento vertical
    if keys_pressed.get('up', False):
        self.y -= current_speed * dt
    if keys_pressed.get('down', False):
        self.y += current_speed * dt

    # Aplicar limites
    self._apply_limits()
```

3.5.5 Criação Interativa de Corpos Celestes

O `CelestialBodyCreator` permite ao usuário criar novos corpos celestes interativamente através de cliques do mouse:

- O usuário pode ajustar a massa do novo corpo através de um slider, com referências visuais a corpos conhecidos (como Terra, Júpiter, Sol).
- A posição do novo corpo é determinada pelo clique do mouse, convertendo coordenadas de tela para coordenadas do mundo.
- O raio do corpo é calculado automaticamente com base na massa, usando uma escala logarítmica.

- Cores são geradas aleatoriamente para facilitar a identificação visual.

```
def _calculate_radius_from_mass(self, mass):
    """Calcula raio baseado na massa usando uma escala logaritmica suave"""
    # Definir limites de massa e raio
    min_mass = 3.285e23  # Mercurio
    max_mass = 1.989e30  # Sol
    min_radius = 6       # Raio minimo (Mercurio)
    max_radius = 40      # Raio maximo (Sol)

    # Usar escala logaritmica para uma distribuicao mais natural
    mass_log = math.log10(mass)
    min_mass_log = math.log10(min_mass)
    max_mass_log = math.log10(max_mass)

    # Normalizar a massa na escala logaritmica (0 a 1)
    normalized_mass = (mass_log - min_mass_log) / (max_mass_log - min_mass_log)

    # Calcular raio baseado na massa normalizada
    radius = min_radius + (max_radius - min_radius) * normalized_mass

    # Garantir que o raio fique dentro dos limites
    return max(min_radius, min(max_radius, round(radius)))
```

Esta abordagem de design do frontend permite uma separação clara entre a visualização e a lógica de simulação, seguindo o padrão MVC (Model-View-Controller), onde o backend implementa o modelo e o frontend implementa a visualização e o controle.

4 CONCLUSÃO

Este trabalho apresentou a implementação de um simulador de sistema solar em duas linguagens diferentes: Python, seguindo o paradigma orientado a objetos, e Haskell, seguindo o paradigma funcional. As principais contribuições deste trabalho são:

- Implementação de um modelo físico de simulação gravitacional, baseado na Lei da Gravitação Universal de Newton.
- Comparação entre as abordagens orientada a objetos e funcional para a resolução de um mesmo problema computacional.
- Demonstração das diferenças e similaridades entre as implementações em termos de estrutura de código, gerenciamento de estado e tratamento de efeitos colaterais.

A comparação entre as implementações revelou diferentes abordagens para o gerenciamento de estado e efeitos colaterais. A implementação em Python utiliza objetos mutáveis e permite efeitos colaterais, facilitando a modelagem de sistemas com estado que muda ao longo do tempo. A implementação em Haskell, por outro lado, utiliza estruturas de dados imutáveis e funções puras, resultando em um código mais declarativo e mais fácil de raciocinar, mas exigindo uma abordagem diferente para modelar sistemas com estado.

Uma limitação importante do simulador é a precisão da integração numérica. O método de Euler utilizado é simples mas pode acumular erros ao longo do tempo, especialmente para simulações de longa duração. Métodos mais avançados, como Runge-Kutta ou Verlet, poderiam ser implementados para melhorar a precisão da simulação.

Possíveis extensões para este trabalho incluem:

- Implementação de métodos de integração numérica mais avançados.
- Adição de suporte para órbitas em três dimensões.
- Implementação de uma interface gráfica para visualizar a simulação.
- Adição de suporte para efeitos relativísticos, como a precessão do periélio de Mercúrio.

Em conclusão, este trabalho demonstrou como diferentes paradigmas de programação podem ser aplicados para resolver um mesmo problema computacional, destacando as vantagens e limitações de cada abordagem. A comparação entre as implementações em Python e Haskell oferece insights valiosos sobre as diferenças entre programação orientada a objetos e programação funcional, contribuindo para uma compreensão mais profunda dos paradigmas de programação.

REFERÊNCIAS

Referências úteis para o desenvolvimento do trabalho:

Livros sobre Paradigmas de Programação

SEBESTA, Robert W. **Concepts of Programming Languages**. 12th ed. Boston: Pearson, 2019.
VAN ROY, Peter; HARIDI, Seif. **Concepts, Techniques, and Models of Computer Programming**. Cambridge: MIT Press, 2004.
HUTTON, Graham. **Programming in Haskell**. 2nd ed. Cambridge: Cambridge University Press, 2016.
LUTZ, Mark. **Learning Python**. 5th ed. Sebastopol: O'Reilly Media, 2013.

Física Computacional e Simulação

LANDAU, Rubin H.; PÁEZ, Manuel J.; BORDEIANU, Cristian C. **Computational Physics**: Problem Solving with Python. 3rd ed. Weinheim: Wiley-VCH, 2015.
PANG, Tao. **An Introduction to Computational Physics**. 2nd ed. Cambridge: Cambridge University Press, 2006.

Artigos sobre Simulação de N-Corpos

AARSETH, Sverre J. Gravitational N-Body Simulations: Tools and Algorithms. **Cambridge Monographs on Mathematical Physics**, Cambridge: Cambridge University Press, 2003.
HERNQUIST, Lars; KATZ, Neal. TREESPH: A unification of SPH with the hierarchical tree method. **Astrophysical Journal Supplement Series**, v.70, p.419-446, June 1989.

Documentação e Tutoriais

PYTHON SOFTWARE FOUNDATION. **Python 3 Documentation**. Disponível em: <https://docs.python.org/3/>. Acesso em: junho 2023.
HASKELL.ORG. **Haskell Programming Language**. Disponível em: <https://www.haskell.org/>. Acesso em: junho 2023.
PYGAME DEVELOPMENT TEAM. **Pygame Documentation**. Disponível em: <https://www.pygame.org/docs/>. Acesso em: junho 2023.

Trabalhos Relacionados

SILVA, João A. **Comparação entre Paradigmas de Programação na Implementação de Algoritmos Científi-**

cos. 2020. 89 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Matemática e Estatística, USP, São Paulo.

SANTOS, Maria B. **Simulação Computacional de Sistemas Físicos: Uma Abordagem Funcional**. 2019. 67 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.