Simulação do Problema dos Três Corpos

Nome: Yuri de Melo Zorzoli Nunes

Matricula: 19103489

Introdução

Este projeto é uma simulação do problema dos três corpos, um clássico problema da mecânica celestial que estuda o movimento de três corpos sob a influência da gravitação mútua. A simulação utiliza C++ para a lógica de cálculo gravitacional e Python com a biblioteca Pygame para a visualização gráfica.

Arquitetura do projeto

Linguagens

1. **C++:** O código feito em C++ calcula as forças gravitacionais entre os corpos e atualiza suas posições e velocidades com base nas interações gravitacionais. Toda a lógica é transportada para um código Python através de uma biblioteca compartilhada.
2. **Python:** O código Python utiliza a biblioteca Pygame para criar uma janela que exibe a simulação em tempo real. Ele carrega a biblioteca C++, chama as funções para inserir objetos e simular as interações, e em seguida exibe o resultado.

Comunicação entre C++ e Python

Para permitir a comunicação entre as duas linguagens, foi utilizado uma biblioteca compartilhada, feita da seguinte maneira:

Biblioteca compartilhada

Primeiramente definimos quais funções e estruturas do código C++ serão utilizadas no Python, para isso usamos `extern "C" {}` e colocamos dentro das chaves as estruturas e funções desejadas.

```
extern "C" {
    void addObject(float x, float y, float vX, float vY, float strength);
    Vector2D iteratePhysic(int index);
}
```

Em seguida, precisamos compilar o nosso código C++ com o seguinte comando:

```
g++ -std=c++11 -fPIC -shared -o libthreebodyproblem.so gravity.cpp
```

Integração

No Python, a biblioteca criada é carregada e são declaradas as funções exportadas usando a biblioteca `ctypes`.

```
import ctypes

dll = ctypes.CDLL('./libthreebodyproblem.so')
dll.iteratePhysic.argtypes = [ctypes.c_int]
dll.iteratePhysic.restype = Vector2D
```

```
dll.addObject.argtypes = [ctypes.c_float, ctypes.c_float, ctypes.c_float, ctypes.c_float
dll.addObject.restype = None
```

- `ctypes.CDLL`: Define a biblioteca compartilhada usada.
- `dll.<functionName>.argtypes`: Define os argumentos passados para a função exportada.
- `dll.<functionName>.restype`: Define o tipo de retorno da função exportada.
- `dll.<functionName>(args)`: Chama a função.

Métodos e definições

C++

Objetos

Os objetos são definidos a partir da classe `GravityObject` que é definida por

```
class GravityObject
{
    Vector2D pos;
    Vector2D vel;
    float strength;

public:
    GravityObject(float posX, float posY, double vx, double vy, float strength)
    {
        pos.x = posX;
        pos.y = posY;
        vel.x = vx;
        vel.y = vy;
        this->strength = strength;
    }

    Vector2D getPos()
    {
        return pos;
    }

    float getStrength()
    {
        return strength;
    }

    void updatePhysics(std::vector<GravityObject>& objects) {
        // Método com calculo da gravidade e
        // atualização das posições
    }

};
```

Calculo da gravidade

No código, o calculo da gravidade é feito a cada iteração, e em cada iteração é feito o calculo da gravidade entre todas as duplas de objetos possíveis.

Para fazer o calculo então, primeiramente precisamos calcular a distancia entre os dois objetos da iteração utilizando a seguinte formula: $\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

No código ficou da seguinte maneira:

```
Vector2D delta = {
  objs.getPos().x - this->getPos().x,
  objs.getPos().y - this->getPos().y
};
float distance = sqrt(pow(delta.x, 2) + pow(delta.y, 2));
```

Nessa etapa foi adicionado um limite de distancia entre os objetos para que evitasse com que uma colisão impedisse a simulação de continuar.

```
if (distance < 100) {
  distance = 100;
  delta.x = delta.x * (100 / (distance + 0.01));
  delta.y = delta.y * (100 / (distance + 0.01));
}
```

Apos isso temos o necessário para o calculo da atração gravitacional entre os objetos com a seguinte formula:

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

onde

F = Força Gravitacional

G = Constante Gravitacional

m1 = Massa do objeto 1

m2 = Massa do objeto 2

r = distancia entre os objetos 1 e 2

A massa no código está definida como `strength`

```
float forceMagnitude =
    G * (this->getStrength() * objs.getStrength()) / pow(distance, 2);
```

Com isso podemos calcular a força total aplicada em x e y e somar ela a velocidade do objeto, com a velocidade atualizamos a posição nova:

$$\text{force}_x = \Delta x \cdot \frac{\text{forceMagnitude}}{\text{distance}}$$

$$\text{force}_y = \Delta y \cdot \frac{\text{forceMagnitude}}{\text{distance}}$$

No Código ficou assim

```
    Vector2D force = {
      delta.x * (forceMagnitude / distance),
      delta.y * (forceMagnitude / distance)
    };

    totalForce.x += force.x;
    totalForce.y += force.y;
}
```

```
vel.x += totalForce.x * 0.1;
vel.y += totalForce.y * 0.1;

pos.x += vel.x * 0.1;
pos.y += vel.y * 0.1;
```

Funções exportadas

As funções que serão exportadas para o Python são:

`addObject` ira receber do código Python a posição, velocidade e massa do objeto e vai inserir este novo objeto na lista de objetos da simulação

```
void addObject(float x, float y, float vX, float vY, float strength)
    objects.push_back(GravityObject(x, y, vX, vY, strength));
}
```

`iteratePhysic` ira receber o index do objeto da iteração e irá chamar o método `updatePhysics` para calcular a sua nova posição, após os cálculos ela retornará na forma de uma struct a posição nova do objeto.

```
Vector2D iteratePhysic(int index)
    objects[index].updatePhysics(
    return objects[index].getPos();
}
```

Python

Primeiramente iniciamos e definimos a biblioteca compartilhada usando o `ctypes`

```
dll = ctypes.CDLL('./libthreebodyproblem.so')

class Vector2D(ctypes.Structure):
 _fields_ = [("x", ctypes.c_float),
               ("y", ctypes.c_float)]

dll.iteratePhysic.argtypes = [ctypes.c_int]
dll.iteratePhysic.restype = Vector2D

dll.addObject.argtypes = [ctypes.c_float, ctypes.c_float, ctypes.c_float, ctypes.c_float
dll.addObject.restype = None
```

Apos isso configuramos o Pygame

```
pygame.init()

width, height = 1500, 1000
screen = pygame.display.set_mode((width, height))
pygame.display.set_caption("Problema dos 3 corpos")
clock = pygame.time.Clock()

circles = [
    {'color': (255, 255, 255), 'pos': [475.0, 700.0], 'vel': [6, 0], 'strength': 1000, '
    {'color': (255, 255, 255), 'pos': [750.0, 300.0], 'vel': [-6, 6], 'strength': 1000,
    {'color': (255, 255, 255), 'pos': [1025.0, 700.0], 'vel': [0, -6], 'strength': 1000,
]
```

```
simulationPositions = []
```

Funções

`initObjects` aciona a função `addObjects` para cada circulo adicionado anteriormente para adicioná-los na simulação.

```
def initObjects():
    for circle in circles:
        dll.addObject(circle['pos'][0], circle['pos'][1], circle['vel'][0], circle['vel'
```

`runSimulation` aciona para cada objeto em cada iteração a função `iteratePhysic` , que retorna a nova posição do objeto e adiciona na lista de posições

```
def runSimulation(j):
    position = dll.iteratePhysic(j)
    simulationPositions.append([position.x, position.y])
```

para cada circulo, atualiza a nova posição com base na lista de posições calculada e adiciona a posição em ‘history’ para criar o rastro.

```
def updateCirclesPosition():
    for circle in circles:
        if simulationPositions:
            position = simulationPositions.pop(0)
            circle['history'].append(position)
            if len(circle['history']) > 1000:
                circle['history'].pop(0)
            circle['pos'] = position
```

apaga os círculos e as linhas e os desenha com a nova posição.

```
def draw():
    screen.fill((0, 0, 0))
    for circle in circles:
        if len(circle['history']) > 1:
            pygame.draw.lines(screen, circle['lineColor'], False, circle['history'], 2)
        pygame.draw.circle(screen, circle['color'], (int(circle['pos'][0]), int(circle['
    pygame.display.flip()
```

Conclusão

Como C++ e Python possuem vocações distintas, isso torna a implementação conjunta dessas linguagens bem útil. Utilizar ambas permite combinar o desempenho e a eficiência do C++ com a facilidade e flexibilidade do uso do Python.
No contexto desse trabalho, usando C++ para os cálculos nós conseguimos os resultados em uma velocidade altíssima, e o que seria um desafio criar uma interface em C++ para demonstrar visualmente a simulação, se tornou mais simples e prático integrar essa parte visual com Python.