Relatório do Trabalho de Física

Igor Patrício Michels
Isaque Vieira Machado Pim

Novembro de 2019

1 Introdução

O trabalho proposto pelo professor constitui na elaboração de um modelo com o objetivo de simular um fenômeno físico. Para tanto, nós escolhemos simular uma mesa de Sinuca, no qual o usuário pode interagir com a simulação, como se estivesse jogando uma partida de Sinuca mesmo.

2 Objetivo

O objetivo do trabalho é modelar, de maneira simples, uma mesa de Sinuca, mais especificamente simular as colisões entre as bolas, utilizando como base as medidas em escala com as dimensões reais, além de apresentar o resultado de maneira interativa com o usuário podendo interagir com a física das tacadas.

3 Material

Para o desenvolvimento do trabalho, utilizamos a plataforma Processing com o modo Python para realizar a parte da computação gráfica. Todos os métodos de iteração são baseados no conteúdo apresentado em sala de aula.

4 Metodologia

4.1 Movimentação

A movimentação foi implementada pelo Método de Euler. Com dados da velocidade e da posição calculamos a próxima posição baseada no intervalo de tempo entre os frames do programa.

4.2 Colisões

Para as colisões não foi feito nenhum trabalho sobre conservação de energia. Todas as colisões são perfeitamente elásticas, ou seja, há conservação de energia total. A detecção de colisão é feita iterando-se sobre uma lista de lados e bolas, onde os resultados de operações com o produto escalar que ligam o centro da bola às extremidades dos lados verificam a colisão. Para a colisão entre duas bolas A e B, cria-se um vetor ligando os centros das duas bolas, tomando o mesmo como "eixo x" para a colisão. Projeta-se as velocidades de cada bola sobre esse vetor, obtendo, portando, a componente v_x da velocidade de cada bola. Fazendo $v_A - v_{xA}$ obtêm-se v_{yA}, o mesmo para as velocidades de B. Note que a componente y não sofre alteração alguma na colisão, portando, podemos utilizar os métodos vistos em sala para calcularmos a nova componente x de cada velocidade.

4.3 Atrito

A modelagem do atrito foi similar ao do trabalho realizado para a A1, ou seja, buscamos modelar o atrito de forma que o mesmo se adequasse as velocidades da bola, visando conforto ao usuário.

5 Desenvolvimento do Código

Aqui será feito apenas o detalhamento do código que gera a física do jogo. As partes de detalhamento visual não serão comentadas.
A bola foi implementada como uma classe. Essa classe contém as características essenciais da bola como a posição, a velocidade e o raio. Além disso, contém os métodos de movimentação da bola: movimentos simples e colisões, as quais foram implementadas como funções que recebem como argumento dois objetos da classe bola.

5.1 Classe bola

Definição da classe boa, nela define-se que a bola recebe um vetor posição (posição x e posição y), o raio, que, seguindo as proporções utilizadas no trabalho resultou em $8,5$, a massa, a qual foi definida como 1 para todas as bolas, uma vez que a massa de duas bolas é a mesma, e um vetor com a cor da bola.
Nessa classe, define-se a movimentação da bola, adicionando o produto da velocidade com o intervalo de tempo na posição, além de considerar uma pequena diminuição de velocidade em virtude do atrito dado.

```
class Bola:
    def __init__(self, pos, raio, massa, cor):
        self.pos = pos
        self.r = raio
        self.m = massa
        self.v = PVector(0, 0)
        self.a = PVector(0, 0)
        self.cor = cor
```

5.2 Funções da classe Bola

As principais funções da classe Bolas são funções de movimento da bola, desenho e as funções que verificam se a bola colidiu com alguma paredee/ou outra bola.

```
    def move(self, dt):
        if self.v.x**2 + self.v.y**2 < 0.000016:
            self.a = PVector(0, 0)
            self.v = PVector(0, 0)
        else:
            self.a = PVector(self.v.x, self.v.y)
            self.v = self.v - 0.0004 * self.a * dt
        self.pos = self.pos + self.v * dt

    def desenha(self):
        stroke(0)
        fill(self.cor[0], self.cor[1], self.cor[2])
        ellipse(self.pos.x, self.pos.y, 2*self.r, 2*self.r)

    def verifica_colisao(self, a):
        if (self.pos.x - a.pos.x)**2 + (self.pos.y - a.pos.y)**2 < (self.r + a.r)**2:
            self.colide(a)

    def verifica_colisao_parede(self, p):
        pa = p[0] - p[1]
        v1 = pa.dot(self.pos - p[1])
        v2 = pa.dot(self.pos - p[0])

        if v1 > 0 and v2 < 0:
            a_ = (self.pos-p[1]) - (v1/(v1-v2))*pa
            if a_.x**2 + a_.y**2 <= self.r**2:
                self.colide_parede(a_)
```

5.3 Funções para colisão

Nessa parte, estamos definindo as funções que calculam as colisões da bola junto aos lados da pista e/ou outras bolas. Lembramos que a colisão é perfeitamente elástica, ou seja, quando ocorre uma colisão, o único efeito é a troca da direção da velocidade. Vale ressaltar que tais funções também pertencem a classe Bola.

```
    def colide(self, a):
        dir_x = a.pos - self.pos
        dir_x.normalize()
        va_x = dir_x * (self.v.dot(dir_x))
        va_y = self.v - va_x

        vb_x = dir_x * (a.v.dot(dir_x))
        vb_y = a.v - vb_x

        if va_x.dot(vb_x) > 0.0:
            if va_x.dot(dir_x) < 0:
                new_ax = - va_x.mag() * ((self.m - a.m)/(self.m + a.m)) - vb_x.mag() *
                    ((2*a.m)/(self.m + a.m))
                new_bx = - va_x.mag() * ((2*self.m)/(self.m + a.m)) - vb_x.mag() * ((a.m
                    - self.m)/(self.m + a.m))
                self.pos += 2 * self.v
                a.pos += 2 * a.v

            else:
                new_ax =  va_x.mag() * ((self.m - a.m)/(self.m + a.m)) + vb_x.mag() *
                    ((2*a.m)/(self.m + a.m))
                new_bx =  va_x.mag() * ((2*self.m)/(self.m + a.m)) + vb_x.mag() * ((a.m -
                     self.m)/(self.m + a.m))
                self.pos += 2 * self.v
                a.pos += 2 * a.v
        else:
            new_ax =  va_x.mag() * ((self.m - a.m)/(self.m + a.m)) - vb_x.mag() * ((2*a.m
                )/(self.m + a.m))
            new_bx =  va_x.mag() * ((2*self.m)/(self.m + a.m)) - vb_x.mag() * ((a.m -
                self.m)/(self.m + a.m))
            self.pos += 2 * self.v
            a.pos += 2 * a.v

        self.v = new_ax * dir_x + va_y
        a.v = new_bx * dir_x + vb_y

    def colide_parede(self, n):
        n.normalize()
        vy = n*(self.v.dot(n))
        vx = self.v - vy

        vy = vy* (-1)
        self.pos.add(2*vy)
        self.v = vx + vy
```

5.4 Definições finais

Por fim, seguimos a ideia do projeto da A1, mantendo a resolução em 800x600, além de criarmos uma animação de um taco de sinuca para bater na bola branca.