Objetivos Atingidos

Aline Marcelo Garlet Millani

14 de junho de 2013

Resumo

Este trabalho implementa um simulador de partículas elétricas em OCaml.

1 Descrição do problema

O problema consiste em, dado um conjunto de partículas, similar as forças de atração elétrica entre elas, desenhando-as conforme se movem.

Para desenhar numa janela, usamos as bibliotecas lablgl e lablglut, que são, respectivamente, implementações de OpenGl e GLUT para OCaml.

2 OCaml

OCaml é uma linguagem multi-paradigma que apresenta características imperativas, funcionais e de orientação a objetos. Atualmente, é a principal implementação de Caml.

Possui características de ML, como tipagem estática, que permite uma maior confiabilidade no código por sabermos em tempo de compilação que não haverão operações não definidas, e inferência de tipos, permitindo que o programador não se preocupe com o tipo de cada variável e melhorando a redigibilidade e diminuindo o acoplamento por tipagem.

Outra propriedade muito interessante de OCaml é pattern matching, pois aumenta consideravelmente a expressividade da linguagem ao descrever construções if-else encadeadas através de uma sintaxe muito mais clara, além de permitir uma decomposição estrutural de forma mais natural.

Por ser uma linguagem compilada, com tipagem estática e uso de referências a memória, programas em OCaml são bastante eficientes, fazendo com que a linguagem possa ser usada em diversos casos práticos, diferentemente de algumas linguagens como Python e Java, que apresentam um custo em processamento ou memória proibitivos para diversas aplicações.

A orientação a objetos de OCaml permite o encapsulamento de informações ao fornecer para o programador meios de definir métodos e atributos privados ou públicos. Python não possui mecanismos para isso, de forma que convenções devem ser adotados. Por um lado, se tudo for público há maior flexibilidade, mas o uso de uma classe pode se tornar mais obscuros por não sabermos quais alterações manterão a consistência dos dados.

3 Implementação

3.1 Definição e uso de classes

Foram criadas 3 classes: Body, Particle e Electric, sendo Body abstrata.

```
class virtual body:
  float ->
  float ->
  object
    val mutable position : float * float
    val mutable velocity : float * float
    val mutable acceleration : float * float
    method getPosition : float*float
    method setPosition : float*float -> unit
    method getAcceleration : float*float
    method loopBound : float*float -> float*float -> unit
    method setAcceleration : float*float-> unit
    method applyAcceleration : float*float -> unit
    method virtual draw : unit
    method move : unit
  end;;
```

Figura 1: Interface da classe body

3.2 Encapsulamento e proteção dos atributos

Todos os atributos das classes são privados. O encapsulamento de dados pode ser visto através do método move, por exemplo, que esconde o fato de se estar usando três vetores (posição, velocidade e aceleração) para se calcular o movimento da partícula.

```
method move =
  let (px,py) = position in
  let (vx,vy) = velocity in
  let (ax,ay) = acceleration in
  position <- (px +. vx,py +. vy);
  velocity <- (vx +. ax,vy +. ay);
  acceleration <- (0.0,0.0)
```

Figura 2: Implementação do método move da classe body

3.3 Organização do código em espaços de nome diferenciados

Em OCaml, cada arquivo conta como um espaço diferente, de forma que este deve ser incluído com open ou prefixado com seu nome antes de ser possível usar suas funções.

3.4 Mecanismo de Herança

Foram usados três níveis de hierarquia, sendo que body é a superclasse de particle, que por sua vez é superclasse de electric.

```
class particle x y =
  object (self)
    inherit body x y
    method draw =
      GlDraw.begins `points;
      GlDraw.vertex2 position;
      GlDraw.ends ()
  end;;
```

Figura 3: Implementação da classe particle, mostrando a herança

3.5 Especificação de uma classe abstrata

A classe body é abstrata por possuir o método draw como virtual.

3.6 Polimorfismo por inclusão

3.7 Polimorfismo paramétrico

3.7.1 Especificação de algoritmo utilizando o recurso

3.7.2 Especificação de estrutura de dados genérica

3.8 Polimorfismo por sobrecarga

OCaml não suporta polimorfismo por sobrecarga. Isso está relacionado com o mecanismo de inferência de tipos, que seria muito mais complicado de se implementar (talvez até impossível para qualquer caso). No entanto, a linguagem permite a criação de funções que recebem parâmetros de tipos arbitrários (como, por exemplo, map), mas a implementação é a mesma para todos os tipos.

3.9 Especificação e uso de funções como elementos de primeira ordem

Para carregar as partículas de um arquivo, foi feita uma função para ler as linhas de um canal, que foi passada como argumento para a função parseChannel.

```
let readLine chan =
  let rec readLineAux acc =
    let c =
      try
        input_char chan
      with
        End_of_file -> if String.length acc = 0 then raise End_of_file else '\n'
    in
    match c with
      '\n' -> acc
    |'\r' -> acc
    | _ -> (readLineAux (acc ^ (String.make 1 c)) )
  in
    readLineAux ""

let loadConfig filename =
  let chan = open_in_bin filename in
  let lines = parseChannel chan readLine in
```

Figura 4: Função que lê uma linha de um arquivo e trecho de uma função que carrega o arquivo de configuração

3.10 Especificação e uso de funções de ordem maior

Foi feita uma função que lê o conteúdo de um arquivo, sendo que um de seus parâmetros é uma função que recebe um canal e retorna um elemento.

```
let rec parseChannel chan readElement =
  try
    let el = readElement chan in
    el::(parseChannel chan readElement)
  with
    End_of_file -> []
```

Figura 5: Função que lê o conteúdo de um arquivo

3.10.1 Currying

Foi usado currying para a função de callback de display da glut. Como essa deve ser do tipo unit, mas queríamos uma que recebesse uma lista de pontos, simplesmente aplicamos display a dots, produzindo, assim, uma função do tipo unit, mesmo que display recebesse um argumento.

```
let display dots ()=
  GlClear.clear [ `color ];
  drawDots !dots;
  Glut.swapBuffers ()

let _ =
  ignore (Glut.init Sys.argv);
  Glut.initDisplayMode ~double_buffer:true ();
  ignore (Glut.createWindow ~title:"Simparticle");

  Glut.displayFunc ~cb:(display dots);
  Glut.reshapeFunc ~cb:reshape;
  Glut.mouseFunc ~cb:mouseHandler;
  Glut.timerFunc ~ms:mili ~cb:timerF ~value:1;
  Glut.mainLoop()
```

Figura 6: Funções display e main do programa

3.11 Pattern Matching

Diversas funções feitas usaram pattern matching. A versão sem pattern matching das mesmas seria mais longa e menos legível. No exemplo abaixo, são feitas duas operações no pattern matching: primeiro compara-se a cabeça da lista com "eletric"e atribui-se o resto da lista para rest.

```
let readParticleCategory lst = match lst with
    "electric"::rest -> readElectricCategory rest
  | _ -> raise Invalid_format
```

Figura 7: Função que usa pattern matching

3.12 Recursão como mecanismo de iteração

```
let readParticleCategory lst = match lst with
    "electric"::rest -> readElectricCategory rest
  | _ -> raise Invalid_format
```

Figura 8: Função que usa pattern matching

3.13 Delegates

Como OCaml suporta funções como elementos de primeira ordem, delegates se tornam completamente desnecessários, uma vez que são apenas uma forma de se passar métodos como argumento para alguma classe em paradigmas orientados a objeto. Como é possível passar a função diretamente, OCaml não possui sintaxe para delegates.

- Uso de lista para manipulação de estruturas em funções de ordem maior (as funções devem ser puras)

- Uso de funções lambda
    1. Poderia ser usado em uma função que aplica atrito (Physics.ml)