

Objetivos Atingidos

Aline Marcelo Garlet Millani

15 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 obscuro 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.

```
1 class virtual body :
2   float ->
3   float ->
4   object
5     val mutable position : float * float
6     val mutable velocity : float * float
7     val mutable acceleration : float * float
8     method getPosition : float*float
9     method setPosition : float*float -> unit
10    method getAcceleration : float*float
11    method loopBound : float*float -> float*float -> unit
12    method setAcceleration : float*float -> unit
13    method applyAcceleration : float*float -> unit
14    method virtual draw : unit
15    method move : unit
16 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.

```
1 method move =
2   let (px,py) = position in
3   let (vx,vy) = velocity in
4   let (ax,ay) = acceleration in
5   position <- (px +. vx, py +. vy);
6   velocity <- (vx +. ax, vy +. ay);
7   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`.

```
1 class particle x y =  
2   object (self)  
3     inherit body x y  
4   method draw =  
5     GLDraw.begins 'points;  
6     GLDraw.vertex2 position;  
7     GLDraw.ends ()  
8 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

Foi usada uma quadtree para a simulação das interações entre as partículas. A definição da estrutura é válida para qualquer tipo de dado, mas apenas um de cada vez, isto é, podemos ter uma árvore de inteiros, de partículas, etc... mas não podemos misturar floats e ints na mesma árvore.

Como OCaml usa inferência de tipos, o tipo da árvore não precisa estar explícito no código.

```
1 type 'a quadtree =  
2 | Empty  
3 | Leaf of 'a  
4 | Node of 'a quadtree*'a quadtree*'a quadtree*'a quadtree;;
```

Figura 4: Descrição da estrutura de uma quadtree. `????` indica o tipo parametrizado

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`.

```
1 let readLine chan =
2   let rec readLineAux acc =
3     let c =
4       try
5         input_char chan
6       with
7         End_of_file ->
8           if String.length acc = 0 then
9             raise End_of_file
10            else '\n'
11     in
12     match c with
13     | '\n' -> acc
14     | '\r' -> acc
15     | ' ' ->
16       if String.length acc = 0 then readLineAux acc else acc
17     | '\t' ->
18       if String.length acc = 0 then readLineAux acc else acc
19     | _ -> (readLineAux (acc ^ (String.make 1 c)) )
20   in
21   readLineAux ""
22
23 let loadConfig filename categoryFunc =
24   let chan = open_in_bin filename in
25   let lines = parseChannel chan readLine in
26   parseList categoryFunc lines
```

Figura 5: Função que lê uma linha de um arquivo e outra 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.

```
1 let rec parseChannel chan readElement =
2   try
3     let el = readElement chan in
4     el :: (parseChannel chan readElement)
5   with
6     End_of_file -> []
```

Figura 6: 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.

```
1 let display dots () =
2   GlutClear.clear [ 'color ];
3   drawDots !dots;
4   Glut.swapBuffers ()
5
6 let _ =
7   ignore (Glut.init Sys.argv);
8   Glut.initDisplayMode ~double_buffer:true ();
9   ignore (Glut.createWindow ~title:"Simparticle");
10
11   Glut.displayFunc ~cb:(display dots);
12   Glut.reshapeFunc ~cb:reshape;
13   Glut.mouseFunc ~cb:mouseHandler;
14   Glut.timerFunc ~ms:mili ~cb:timerF ~value:1;
15   Glut.mainLoop()
```

Figura 7: 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, seria necessário fazer ifs possivelmente longos ou criar uma máquina de estados para se obter o mesmo comportamento.

```
1 let readElectricParticle lst =
2   let rec readAux x y charge lst = match lst with
3     | "x" :: "=" :: value :: rest ->
4       readAux (fun k -> float_of_string value) y charge rest
5     | "y" :: "=" :: value :: rest ->
6       readAux x (fun k -> float_of_string value) charge rest
7     | "charge" :: "=" :: value :: rest ->
8       readAux x y (fun k -> float_of_string value) rest
9     | h::r -> (new electric (x 0) (y 0) (charge 0) , r)
10    | _ -> (new electric (x 0) (y 0) (charge 0) , lst)
11   in
12   let none = (fun x -> raise Invalid_format) in
13   readAux none none none lst
```

Figura 8: Função que usa pattern matching

3.12 Recursão como mecanismo de iteração

```
1 let rec drawDots points = match points with
2   | h::r -> h#draw;
3             drawDots r;
4   | [] -> ()
```

Figura 9: Função para desenhar uma lista de pontos na tela

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)

Referências

- [1] http://en.wikipedia.org/wiki/Barnes-Hut_simulation
- [2] <http://caml.inria.fr/pub/docs/manual-ocaml/libref/>