MAC 216
Relatório da segunda parte do projeto

Cesar Gasparini Fernandes (10297630)
Artur M. R. dos Santos (10297734)
Daniel Silva Nunes (10297612)

22/10/2017

1 Visão geral

Na segunda fase do projeto, implementamos o código em arena.c, juntamente com o arena.h, que representam a arena em que as batalhas entre os exércitos de robôs ocorrerão no jogo. Ela possui as funções de sistema, criação e destruição de um robô, atualização, coleção dos cristais, ataque, encerramento do jogo, movimentação e, claro, a de inicialização da arena. Portanto, na nossa implementação, a arena que gerencia todo o jogo, executando as principais funções para que o jogo possa ocorrer.

A movimentação foi feita usando o grid hexagonal, de forma que valores determinam para qual local o robô se movimenta (descrito no código de arena.c).

A função de ataque possui somente uma opção de ataque, mas futuramente iremos implementar ataques diferenciados para os robôs.

Para a segunda fase do projeto, o sistema está elaborado de modo a administrar as ações de uma máquina no jogo, possuindo apenas a estrutura geral de um escalonador, mas que ainda não lida com múltiplos robôs e exércitos.

A função sistema, a principal do programa, gerencia todo o cenário do jogo, sendo a única que manipula a pilha de forma direta. Ademais, é necessário uma requisição ao sistema para que os robôs possam executar seus pedidos, assim, evitamos possíveis ações indesejadas e trapaças (cheats).

Os arquivos da 1 a fase necessitaram algumas alterações:

- O montador foi adaptado de modo a compor o array **prog** no código gerado com o formato correto do instrução, que agora conta com um **OpCode** e um **Operando**. Operandos possuem um **tipo** e um valor. (Ver exemplo do montador abaixo)

- O maq.c, que possui as instruções, recebeu cases adicionais para tratar as interações com a arena. Por consequência, a struct **Máquina**, em maq.h, recebeu alterações em sua struct. Os valores que estão dentro das structs 'Célula' e 'Robô' foram acessados de maneira diferente da implementação anterior, pois agora acessamos valores destas.

- As instruções em **instr.h** foram ampliadas, como dito anteriormente, e sua estrutura foi levemente alterada. Foram especificados novos tipos, direções para movimentação na arena, e o operando (OPERANDO) recebeu novos valores que foram implementados por meio de uma struct e uma union. Uma observação: a union foi criada porque temos o intuito de criar um 'box' na próxima fase do projeto (na parte gráfica), e portanto, poderemos acessar os valores que desejamos apenas 'chamando' essa union.

Ao implementar a arena e o sistema, decidimos implementar ambos em um código só, arena.c, pois achamos mais fácil manipular a arena no mesmo arquivo do sistema e para diminuir o fluxo de dados entre os arquivos, entretanto separamos, no codigo, nitidamente o que é do sistema e o que é da arena.

Segue agora alguma de nossas implementações na arena:

- Decidimos implementar, ao invéz da intrução "SIS", as funções "MOV", "ATK", "CLT", "DEP", e "INF" (que serão explicadas mais adiante), pois achamos que ficaria mais fácil para o jogador digitar o seu código.

- O usuário não necessita empilhar as ações na pilha de dados para chamar o sistema, achamos melhor ele apenas usar um dos comandos acima, tornando assim o codigo mais curto, prático e compreensível. Internamente, o comando chama direto o sistema, passando como argumento a ação e a direção que se deseja executar (também passa o ponteiro de sua máquina).

- Sempre que o usuário requisitar uma ação ao sistema, ele empilhará "true" ou "false" na pilha de dados do robo indicando se foi possível realizar a ação, ou se não foi possível, respectivamente.

- Quando o usuário pede informação da célula e quer saber o índice de um ítem com o ATR, a ordem é, em função do argumento:

  1 "tipo de terreno";

  2 "está vazia";

  3 "número de cristais";

  4 "é base".

- Quando o usuário quer saber se a célula está vazia, ele pode receber "0" que indica "não está", ou "1" que indica "está sendo ocupada por um robo do time 1", ou "2" que indica "está sendo ocupada por um robo do time 2", e assim por diante.

- O mapa da arena é gerado, em grande parte aleatoriamente.

- Acabamos nos antecipando e, nos comentários, estão implementações preparadas eventualmente para a terceira parte, as quais são irrelevantes ou não totalmente corretas no momento.

- Quando um robô "morre", isto é, o atributo vida chega a zero, ele acaba "derrubando" seus cristais ao seu redor (de forma aleatória).

- Para um time ganhar um ponto, ele deve depositar um cristal em sua base, caso ele deposite um cristal na base inimiga, o inimigo ganhará um ponto.

Foi apresentada apenas uma visão geral da estrutura do sistema nessa segunda fase. Para mais detalhes, está tudo comentado no código.

2 Testes

2.1 Montador

No momento, para que os códigos gerados funcionem bem com o Makefile feito, o arquivo gerado pelo montador deverá obrigatoriamente se chamar **tprog.c**. Assim, a execução do montador no terminal deve ser feita da seguinte forma:

```
python montador < (arquivo com opcodes) > tprog.c
```

O seguinte código em "linguagem de máquina" no arquivo **tprog** foi dado na entrada do montador pela entrada padrão e gerou o seguinte código em linguagem C no arquivo **tprog.c** como está mostrado a seguir:

Código em tprog:

```
PUSH 1
DUP
ADD
PRN          #imprimir 2
PUSH 5
PUSH 2
SUB
PRN         #imprimir 3
PUSH 4
DUP
MUL
PRN         #imprimir 16
PUSH 10
DUP
DIV
PRN        #imprimir 1
END
```

Código em tprog.c:

```
#include <stdio.h>
#include "maq.h"
INSTR prog[] = {
{PUSH, NUM, {1}},
{DUP, NONE, {0}},
{ADD, NONE, {0}},
{PRN, NONE, {0}},
{PUSH, NUM, {5}},
{PUSH, NUM, {2}},
{SUB, NONE, {0}},
{PRN, NONE, {0}},
{PUSH, NUM, {4}},
{DUP, NONE, {0}},
{MUL, NONE, {0}},
{PRN, NONE, {0}},
{PUSH, NUM, {10}},
{DUP, NONE, {0}},
{DIV, NONE, {0}},
{PRN, NONE, {0}},
{END, NONE, {0}},
};

int main(int ac, char **av) {
CriaArena(20, 2, 10, 5);
        Maquina *maq = cria_maquina(prog, 0, 0, 100, 0, 1);
        exec_maquina(maq, 1000);
        destroi_maquina(maq);
        return 0;
}
```

Nas fases seguintes adicionaremos maior flexibilidade ao montador, se for necessário. O código em tprog foi feito a fim de testar o funcionamento das novas funções que realizam operações aritméticas. O resultado obtido a partir da execução do arquivo "arena", executável do tprog.c, foi:

```
2
3
16
1
```

2.2 Chamadas ao sistema

Primeiramente, para executar o programa gerado pelo montador na máquina virtual e verificar o funcionamento geral do sistema, deve-se executar o arquivo **arena** gerado pelo makefile da seguinte forma:

```
./arena
```

O jogador faz chamadas ao sistema usando principalmente cinco das funções implementadas em maq.c. São elas:

- MOV *Dir*: Solicitação para mover-se na arena para uma nova célula, indica pelo argumento Dir, que pode assumir os valores NORTH, SOUTH, NORTHEAST, SOUTHEAST, NORTHWEST, SOUTHWEST ou CURRENT.
- ATK *Dir*: Ataca a célula vizinha ou a própria célula indicada pela direção do argumento Dir, com comportamento similar ao da função MOV.
- CLT *Dir*: Coleta cristais, se existirem, na célula vizinha indicada pelo argumento Dir, com comportamento similar ao das funções anteriores.
- INF: Solicita ao sistema informações sobre a célula em que ele está. O sistema então empilha um objeto do tipo **Celula** na pilha de dados da máquina.
- DEP *Dir*: Deposita cristais na célula indicada por Dir.

  Executando o código a seguir:

```
MOV NORTH
PRN
ATK NORTH
PRN
INF
ATR 1
PRN
CLT SOUTH
PRN
DEP SOUTHEAST
PRN
END
```

  Obteve-se a seguinte saída:

```
true
false
0
false
false
```

  Que significa, de acordo com cada linha, que:

– O robô conseguiu se mover para a célula norte

– Não conseguiu atacar a célula ao norte de sua posição (pois ela está vazia, só há um robô na arena por enquanto)

– A informação pedida sobre célula atual, o atributo 1 (que é se está vazia ou não), retornou 0 (falso, mas **vazia** é int na structure Celula)

– Tentou coletar cristais na célula sul mas não conseguiu, pois não havia cristais lá, então retornou falso.

– Tentou depositar seu cristais na célula a sudeste de sua posição, mas não conseguiu porque não possui cristais.

2.3 Makefile

O makefile feito compõe um executável chamado provisoriamente de **arena** a partir dos outros códigos feitos e/ou modificados ao longo do projeto. O makefile necessita que o arquivo gerado pelo montador tenha nome **tprog.c**. Nas fases futuras, adicionaremos maior flexibilidade nesse aspecto. Utilizamos variáveis para compor suas instruções:

```
CC = gcc
CFLAGS = -g -w
default: arena

arena: maq.o pilha.o arena.o tprog.o
	$(CC) $(CFLAGS) maq.o pilha.o arena.o tprog.o -o arena

maq.o: maq.c maq.h pilha.h instr.h
	$(CC) $(CFLAGS) -c maq.c

tprog.o: tprog.c maq.h pilha.h instr.h
	$(CC) $(CFLAGS) -c tprog.c

pilha.o: pilha.c pilha.h instr.h util.h
	$(CC) $(CFLAGS) -c pilha.c

arena.o: arena.c arena.h
	$(CC) $(CFLAGS) -c arena.c

clean:
	rm -f arena *.o
```

- Execução do makefile:

```
make clean

rm -f arena *.o

make

gcc -g -w -c maq.c
gcc -g -w -c pilha.c
gcc -g -w -c arena.c
gcc -g -w -c tprog.c
gcc -g -w maq.o pilha.o arena.o tprog.o -o arena
```