2018_2D_beta_OD_MV_1.cpp (PRIMA soluzione per la fase 2D_beta) - parte1

```
float point[3], score, target[3], rot[3];
float myState[12], otherState[12], debris[11][4], wall[6][3];
float dist[4], pmed[4];
int passa;
void init() {
    wall[0][0] = -0.64;
    wall[0][1] = 0.1;
    wall[0][2] = 0.0;
    wall[1][0] = 0.64;
    wall[1][1] = 0.1;
    wall[1][2] = 0.0;
    rot[0] = -otherState[6];
    rot[1] = otherState[7];
    rot[2] = otherState[8];

    game.getDebris(debris);
    for (int i=0; i<11; i++)
        DEBUG(("i= %.2d radius= %5.3f posDebris= %+5.3f %+5.3f %+5.3f ",
        i, debris[i][3], debris[i][0], debris[i][1], debris[i][2]));

    dist[0] = dista(wall[0],    debris [8]);
    dist[1] = dista(debris [8], debris [9]);
    dist[2] = dista(debris [9], debris[10]);
    dist[3] = dista(debris[10], wall[1]);

    pmed[0] = debris [8][0] - dist[0]/2.0;
    pmed[1] = debris [9][0] - dist[1]/2.0;
    pmed[2] = debris[10][0] - dist[2]/2.0;
    pmed[3] = debris[10][0] + dist[3]/2.0;

    passa = distanzaMag(dist);

    for (int i=0; i<4; i++)
        DEBUG(("\n i:%d: d:%5.3f m:%5.3f", i, dist[i], pmed[i]));
    DEBUG(("\nPassaggio: %d", passa));
    point[0] = pmed [passa];
    point[1] = 0.1;
    point[2] = 0.0;
    //  float score = game.getScore();
}
```

Variabili GLOBALI (riconosciute in tutte le funzioni)

`rot[3]` array utilizzato con `setAttitudeTarget`

dist array delle 4 distanze relative agli **Orange**

pmed array delle 4 ascisse dei punti medi

wall[0][3] array coordinate del bordo sin (X-)

wall[1][3] array coordinate del bordo des (X+)

NOTA:

debris e **wall** sono matrici - array bidimensionali)

point è il punto del semipiano **Y+** (su Z=0) cui **BLUE** si dirigerà (in **loop**) per evitare gli **Orange** ;

point viene determinato (nella **init**):

- calcolando le distanze (**dist**) degli Orange esterni da bordo campo (**wall**) e le distanze tra gli Orange,
- calcolando l'ascissa del punto medio dei 4 varchi,
- individuando il varco più largo con la funzione **distanzaMag** che restituisce lo spazio in cui passare (posizione negli array **dist** e **pmed**):
    - 0 per varco tra **wall[0]** e `debris[8]`
    - 1 per varco tra `debris[8]` e `debris[9]`
    - 2 per varco tra `debris[9]` e `debris[10]`
    - 3 per varco tra `debris[10]` e **wall[1]**
- utilizzando nella valorizzazione di **point** l'intero restituito da **distanzaMag** e salvato nella variabile **passa** come indice per l'array **pmed**

```
void loop() {
    api.getMyZRState(myState);
    api.getOtherZRState(otherState);

    for(int i = 0; i < 4; i++)
        target[i] = otherState[i];

    if (myState[1] > 0)
        api.setPositionTarget(point);

    else {

        api.setPositionTarget(target);
        api.setAttitudeTarget(rot);


        if (game.checkRendezvous())
            game.completeRendezvous();
        }
}
float dista(float point1[], float point2[]) {

 float distanza = fabsf(point1[0] - point2[0]) ;

    return distanza;

}
int distanzaMag(float d[ ]) {
    float max = 0;
    int  imax = -1;

    for(int i = 0; i < 4; i++) {
        if (d[i] > max){
            max  = d[i];
            imax = i;
        }
    }
    return imax;
}
```

2018_2D_beta_OD_MV_1.cpp - parte2

Il programma porta **BLUE** verso **point** fino a quando BLUE si trova nella zona dei detriti (semipiano **Y+** sul piano Z=0), cioè finché l'ordinata è positiva;

successivamente dirige **BLUE** verso **RED** (coordinate ricercate ad ogni secondo e impostate in **target**) e richiama la funzione `checkRendezvous()` (ogni secondo in cui **BLUE** si trova nel semipiano **Y-**)

NOTE:

- `completeRendezvous()` va richiamata solo quando `checkRendezvous()` restituisce **VERO**
- prima viene eseguita la funzione `checkRendezvous()` poi viene valutata la condizione nell' `if` ; se la funzione restituisce VERO l' `if` attiva la seconda funzione `completeRendezvous()`

fabsf funzione della libreria matematica per determinare il valore assoluto di un numero float (qui per la differenza tra 2 float)

distanzaMag

la funzione restituisce la posizione (**imax**) nell'array `dist` per la quale `dist[i]` è il valore maggiore.

Inizialmente si ipotizza che il valore massimo (`max`) sia 0 (nessuna distanza può essere zero!!!) e che NON ci sia posizione corrispondente a zero nell'array; poi si analizzano le 4 distanze con il ciclo `for` e appena si trova una distanza maggiore di `max` (`if (d[i] > max)`) , viene salvata in `max` e viene aggiornata la posizione `imax` restituita al chiamante (**loop**)

NOTA:

`dist[4]` è il parametro ***attuale*** passato alla funzione (riciamata da **loop**)

`d[ ]` è il parametro ***formale*** della funzione