Controllore fuzzy per giocatore automatico di "Space Invaders"

Riccardo Cantoni

September 20, 2017

1 IL GIOCO ORIGINALE

Space Invaders (1978) è un videogioco arcade in cui il giocatore controlla un cannone mobile che può essere mosso orizzontalmente, con cui deve distruggere un gruppo di alieni prima che questi raggiungano la Terra.
Il giocatore può utilizzare come copertura dai proiettili lanciati dagli alieni alcuni "bunker" fissi. Queste strutture sono in grado di resistere ad alcuni colpi prima di essere distrutte.
Il giocatore è sconfitto se il cannone viene colpito per tre volte, oppure se gli alieni raggiungono la Terra.
Una volta sconfitto un gruppo di nemici, altri seguono, all'infinito, mentre la difficoltà aumenta.
Il giocatore ottiene punti per ogni nemico che distrugge. Inoltre, di tanto in tanto, appaiono in cima allo schermo nemici "bonus" che possono essere distrutti per ottenere punti aggiuntivi.

2 IL GIOCO IMPLEMENTATO

E' stata implementata una versione leggermente diversa del gioco. Alcuni aspetti sono stati leggermente semplificati, per ridurre le variabili in gioco ma anche per mantenere un buon livello di difficoltà: i bunker non vengono più distrutti pixel per pixel da un gran numero di colpi, ma mantengono la forma (e quindi il livello di copertura fornito) originaria, pur potendo assorbire solo tre colpi prima di essere distrutti.

La partita termina con una vittoria del giocatore qualora tutti i nemici di una ondata siano distrutti. Allo stesso tempo la partita termina con una sconfitta non appena il giocatore viene colpito anche una volta sola.
I nemici hanno una velocità di movimento massima minore di quella del gioco originale. Allo stesso tempo i proiettili lanciati dal giocatore sono molto più lenti che nel gioco originale. Questo premia la capacità del giocatore di "mirare" correttamente il proprio bersaglio.
I nemici "bonus" sono più frequenti e veloci. Questo per evidenziare le capacità dell'IA di cambiare bersaglio in modo opportuno. In questo modo, infatti, l'impatto sul punteggio finale della distruzione di questi bersagli è molto maggiore che nel gioco originale, e questo premia giocatori capaci di riconoscere le opportunità di colpirli e sfruttarle efficacemente.
Il gioco è stato implementato usando Unity, e C# come linguaggio di programmazione.

3 I BEHAVIOURS

Il controllore fuzzy implementato sfrutta un'architettura a due layers.
Il layer inferiore è composto da 5 diversi controllori fuzzy che implementano diversi comportamenti, implementati come istanze della classe" FuzzyBehaviour".

- Avoid: il giocatore si allontana dal proiettile nemico più vicino.
- Shelter: il giocatore si ripara dai proiettili vicini sotto un bunker non ancora distrutto.
- AttackClosest: il giocatore cerca di distruggere il nemico più vicino.
- AttackLowest: il giocatore cerca di distruggere il nemico più vicino alla Terra.
- AttackRedShip: il giocatore cerca di distruggere il nemico bonus (Red Ship)

Questi oggetti sono in grado di influenzare direttamente le azioni del cannone nel gioco, agendo sui suoi input: l'input di movimento è descritto da un valore float compreso tra -1 e +1, dove -1 corrisponde al movimento più veloce possibile verso sinistra, e +1 corrisponde al movimento più veloce possibile verso destra.
Un valore booleano è invece sufficiente per controllare quando il cannone spara.

Del layer superiore fa invece parte un solo FuzzyBehaviour, chiamato BehaviourSelector, che decide ad ogni frame il comportamento da seguire tra quelli del livello inferiore.

Tutti i FuzzyBehaviour in gioco rappresentano controllori fuzzy, e necessitano quindi di valori di input su cui operare i propri calcoli. Tutti gli input provengono dall'ambiente di gioco e sono misurati da un oggetto di classe Sensor.
Il "sensore" analizza la situazione di gioco e calcola, durante ogni frame, una serie di valori che la descrivono. Valori che vengono utilizzati dai behaviours di entrambi i layer.
Il sensore è unico e condiviso tra tutti i behaviour. La sua unicità è garantita dall'uso di un singleton pattern nella sua implementazione.

4 IL SENSORE

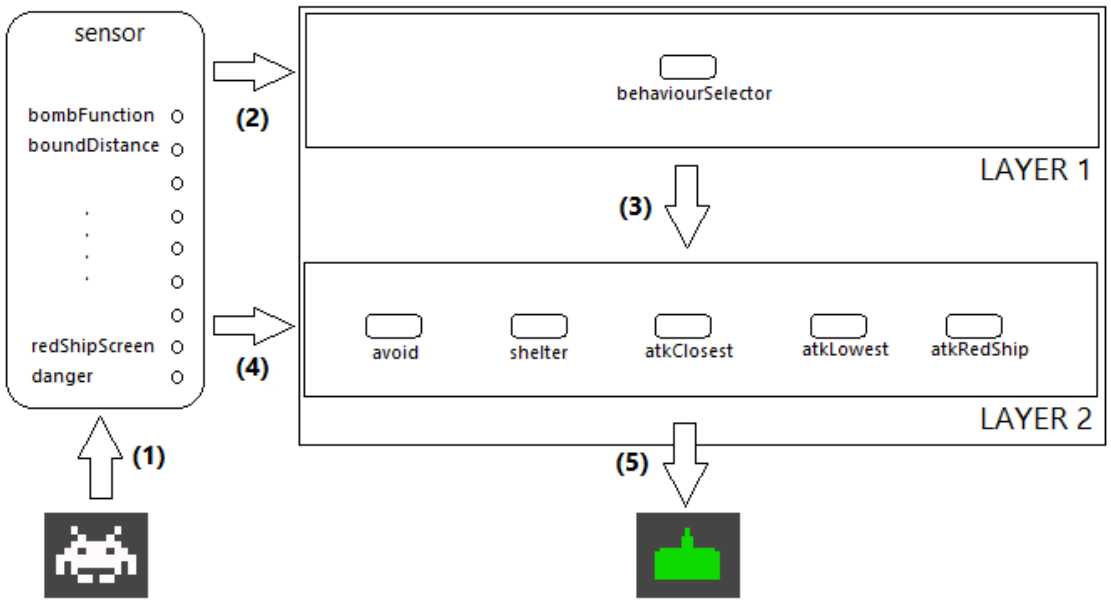
Il sensore analizza lo spazio di gioco ed per ogni frame aggiorna il valore di una serie di variabili pubbliche che possono poi essere utilizzare dai vari behaviour.
queste variabili sono:

- **bombFunction**: componente orizzontale della distanza tra il giocatore e il proiettile nemico più vicino.
- **boundDistance**: distanza orizzontale dal "bordo" invalicabile dell'area di gioco.
- **shieldDistance**: distanza orizzontale dal bunker ancora attivo più vicino.
- **alienVelocity**: componente orizzontale del vettore velocità dei nemici.
- **lowestAlienDistance**: componente orizzontale della distanza tra il giocatore e il più vicino tra i nemici che si trovano nella "fila" più bassa della formazione.
- **lowestAlienHeight**: distanza dalla Terra dell'alieno più vicino ad essa.
- **lowestAlienLead**: quantità dipendente da velocità e alteza da terra del più vicino tra i nemici che si trovano nella "fila" più bassa della formazione. Stima quanto il giocatore debba anticipare il movimento del bersaglio per colpirlo.
- **closestAlienDistance**: analogo a **lowestAlienDistance**, ma relativo al nemico più vicino al giocatore.
- **closestAlienHeight**: analogo a **lowestAlienHeight**, ma relativo al nemico più vicino al giocatore.
- **closestAlienLead**: analogo a **lowestAlienLead**, ma relativo al nemico più vicino al giocatore.
- **redShipPresent**: uguale a 1 se è presente una navicella rossa in gioco, zero altrimenti.
- **redShipDistance**: analogo a **lowestAlienDistance**, ma relativo alla navicella rossa, qualora presente.
- **redShipLead**: analogo a **lowestAlienLead**, ma relativo alla navicella rossa, qualora presente .
- **redShipScreen**: numero di nell'area sottostante la navicella rossa.
- **danger**: stima del "pericolo" che il giocatore sia colpito da un proiettile. E' calcolata come il numero dei proiettili N_p in una certa area intorno al giocatore, pesati per l'inverso della loro distanza, di modo che proiettili lontani pesino poco, e proiettili vicini pesino tanto.
  $danger = \sum_{k=1}^{N_p} \frac{1}{dist(p_k)}$.

- **alienDistributionIndex**: funzione della "compattezza" o "orizzontalità" della formazione dei nemici. Si ottiene calcolando il numero di "file" di alieni presenti N_r, e risolta dal rapporto tra il numero di alieni che riempirebbero completamente (11 per fila) e il numero di alieni che effettivmente sono ancora in gioco N_a:
  $index = \frac{N_r * 11}{N_a}$. Il valore di questa variabile è maggiore (è nei fatti limitato a 11, dove $N_r = 1$ e $N_a = 1$) quando la formazione degli alieni è "bucata" a seguito della distruzione di alcuni di essi.
  E' importante notare che qualora il giocatore si concentrasse esclusivamente sulla distruzione dei nemici nelle file più basse, la loro formazione rimarrebbe piuttosto compatta.

5 Flusso di esecuzione

Ad ogni frame, i diversi elementi concorrono nell'elaborazione dell'input finale che va a muovere il cannone:



1. Il sensore analizza lo spazio di gioco e aggiorna i propri attributi pubblici.
2. Il behaviour (unico) del primo strato legge alcune delle variabili pubblicate dal sensore e calcola il proprio output di conseguenza.
3. L'output di behaviourSelector identifica quale behaviour calcolerà il comportamento del giocatore virtuale per il frame corrente.
4. Il behaviour "attivato" legge a sua volta le variabili pubblicate dal sensore a cui è interessato e calcola il proprio output di conseguenza.
5. l'output del behaviour del secondo strato viene usato per muovere il cannone.

6 FuzzyBehaviour e ciclo di update

FuzzyBehaviour è superclasse di tutti i diversi behaviour implementati. fattorizza quindi tutti quegli aspetti che sono comuni ad ognuno di essi:

```
5  public class FuzzyBehaviour {
6
7      public bool spacebarOutput;
8      public float movementOutput;
9      protected Sensor sensor;
10     protected GameObject displayObject;
11
12     public virtual void updateOutputs (){
13     }
14
15     public void init(Sensor s, GameObject display){
16         this.sensor = s;
17         this.displayObject = display;
18     }
19
20     protected void setDisplayMessage(List<string> msg){
21         displayObject.GetComponent<Display> ().stringList = msg;
22     }
23
24 }
25
```

I due attributi pubblici **spacebarOutput** e **movementOutput** sono poi quelli che verranno utlizzati per muovere il giocatore.
I campi protected conterranno invece i riferimenti al sensore, e ad un oggetto che si occupa del display delle informazioni di debug.
Il metodo **init()** inizializza gli attributi, mentre **setDisplay()** si occupa del passaggio dei messaggi da mostrare all'oggetto che si occupa della loro visualizzazione.
updateOutputs() è un metodo virtuale la cui implementazione è delegata alle sottoclassi. Sarà la definizione di questo metodo a determinare il comportamento specifico dei singoli behaviour.
Ogni singolo behaviour implementa un controllore fuzzy. la sua funzione di aggiornamento seguirà quindi una sequenza di operazioni che prevede la definizione delle classi fuzzy di input, la "fuzzyficazione" dei valori di entrata in valori di membership relativi alle classi, il calcolo delle membership di uscita secondo le regole logiche della FAM, e la defuzzyficazione di queste membership nei valori di uscita effettivi.
Chiaramente non è necessario ridefinire le classi sulle variabili di input ad ogni iterazione. di conseguenza questa operazione non entrerà a far parte del ciclo di aggiornamento implementato dal metodo **updateOutputs()**.

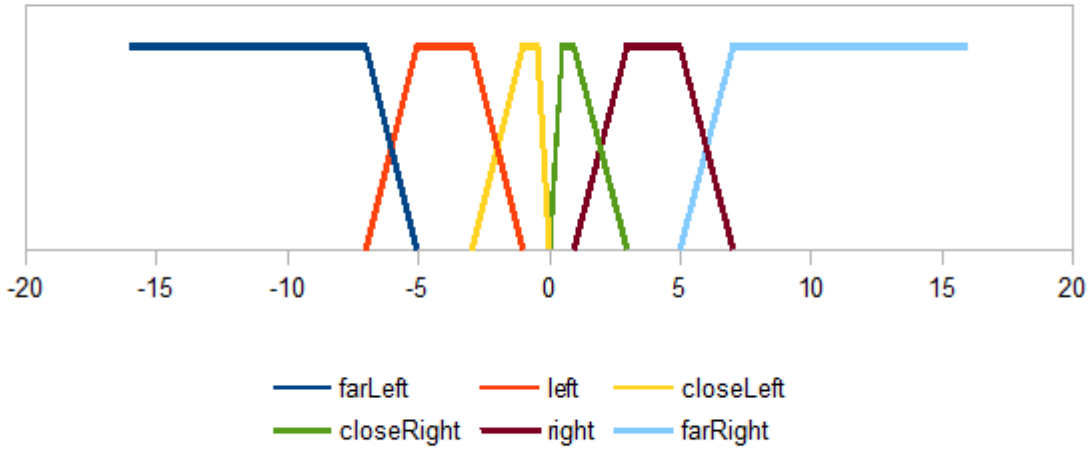
6.1 DEFINIZIONE DELLE CLASSI

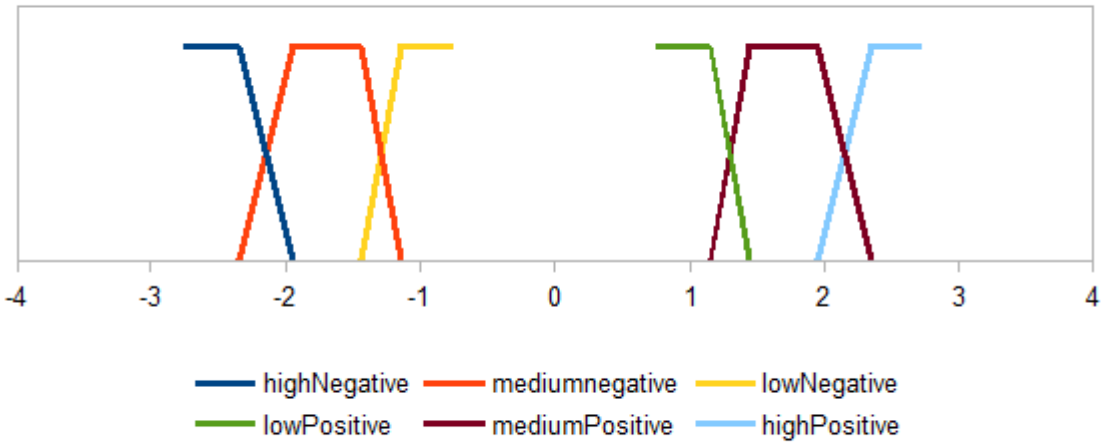
La fuzzyficazione dei valori di entrata è ottenuta per mezzo di classi apposite, che espongono il metodo **calculateMembership()**. Questo metodo riceve in input un valore float (la variabile di ingresso) e restituisce un valore float (il valore di membership).
Le classi sono:

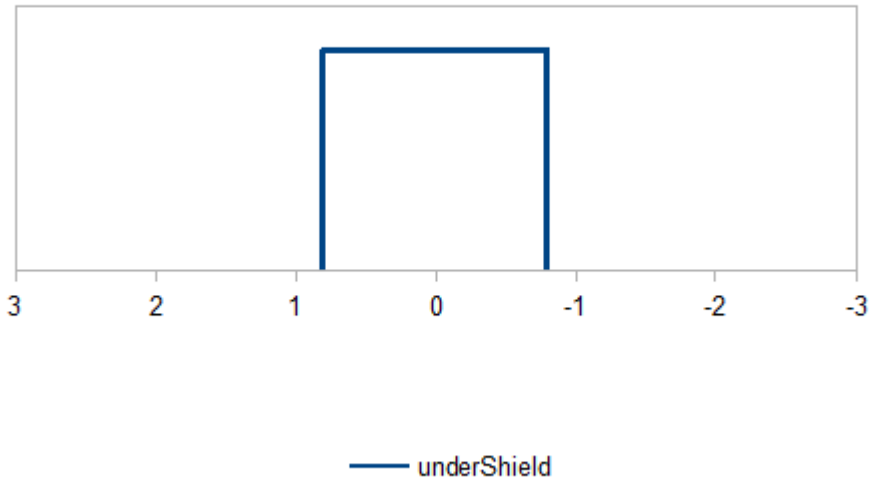
- **FuzzyClass**: definita da 4 valori float, descrive una classe fuzzy di forma triangolare (nel caso in cui due punti coincidono) o trapezoidale (nel caso in cui i punti sono distinti).
- **CrispClass**: definita da 2 valori float, descrive una classe "crisp" affine a una variabile logica classica. è utilizzata per classi che sono inerentemente totalmente vere o totalmente false.

Ad esempio, nel behaviour **attackClosest**, una delle variabili di input utilizzate è **alienVelocity**, che misura la componente orizzontale del vettore velocità di un nemico.
Il range della variabile risulta essere due intervalli separati: $[-2.75, 0.75], [0.75, 2.75]$, poi chè la velocità dei nemici ad inizio partita è, in modulo, $0.75 m/s$ e cresce linearmente fino a $2.75 m/s$. Non è mai uguale a 0 perchè il pattern di movimento dei nemici li fa invertire direzione bruscamente senza rallentare.
su questa variabile sono state definite 6 classi. 3 per quando i nemici si muovno da sinistra verso destra (velocità "positiva") e tre per quando si muovono in senso opposto (velocità negativa). Le tre classi "partizionano" l'intervallo in tre porzioni uguali: nel caso dell'intervallo positivo si ha "lowPositive" per velocità bassa, "mediumPositive" per velocità media, e "highPositive" per velocità alta.
Il behaviour utilizza anche un'altra variabile: **targetDistance**, che identifica la distanza orizzontale del "bersaglio" corrente, ottenuta sommando la distanza orizzontale del nemico più vicino al giocatore e la distanza orizzontale a cui il giocatore si ddeve porre da esso per colpirlo compensandone il movimento.
In questo caso il range è un singolo intervallo continuo $[-16, 16]$ in quanto l'area di gioco è larga 16 e il valore della variabile mantiene il segno.
anche per questa variabile sono state identificate 6 classi. Ma invece di partizionare l'intervallo in modo uniforme si è scelto di avere classi più strette vicino allo 0, per ottenere un livello di precisione maggiore nella definizione del comportamento quando il bersaglio si trova vicino al giocatore.
in particolare le classi più "esterne": **farLeft** e **farRight**, sono molto più grandi delle altre perchè identificano tutte le situazioni in cui il bersaglio è "lontano". Non è quindi necessario in questi casi produrre un comportamento particolarmente raffinato: il giocatore che si stesse comportando secondo questo behaviour dovrà semplicemente avvicinarsi al bersaglio.
Una classe "crisp" è invece usata per fare in modo che il cannone non spari quando si trova sotto un bunker, perchè questo lo danneggerebbe. Chiaramente non è necessario alcun livello di fuzzyness per descrivere l'occorrenza di questo evento: o il cannone è sotto un bunker, o non lo è. La variabile utilizzata è **shieldDistance**, che misura la distanza del cannone dal punto centrale del bunker più vicino.
La variabile ha un range simile a $[-16, 16]$, ma la classe **underShield** produrrà una membership di 0 per qualsiasi distanza maggiore in modulo di 0.8, in quanto 0.8 è metà della

larghezza dell'area che si trova sotto un bunker.
Di conseguenza la classe crisp è unica.







Le classi fuzzy sono definite separatamente rispetto alle variabili che useranno: i costruttori delle classi **FuzzyClass** e **CrispClass** lavorano semplicemente su float per definire la "curva" della membership.
La variabile entrerà a far parte del calcolo in un momento seguente, all'interno del ciclo di update del controllore.

```
    #region targetPosition

    FuzzyClass targetCloseRight = new FuzzyClass (0, 0.5f, 1, 3);
    FuzzyClass targetRight = new FuzzyClass (1, 3, 5, 7);
    FuzzyClass targetFarRight = new FuzzyClass (5, 7, 16, 17);
    FuzzyClass targetCloseLeft = new FuzzyClass (-3, -1, -0.5f, 0);
    FuzzyClass targetLeft = new FuzzyClass (-7, -5, -3, -1);
    FuzzyClass targetFarLeft = new FuzzyClass (-17, -16, -7, -5);

    #endregion

    #region shield

    CrispClass underShield = new CrispClass (-0.8f, 0.8f);

    #endregion

    #region alienSpeed

    FuzzyClass speedLowPositive = new FuzzyClass (0, 0.75f, 1.15f, 1.45f);
    FuzzyClass speedMediumPositive = new FuzzyClass (1.15f, 1.45f, 1.95f, 2.35f);
    FuzzyClass speedHighPositive = new FuzzyClass (1.95f, 2.35f, 2.75f, 3);
    FuzzyClass speedLowNegative = new FuzzyClass (-1.45f, -1.15f, -0.75f, 0);
    FuzzyClass speedMediumNegative = new FuzzyClass (-2.35f, -1.95f, -1.45f, -1.15f);
    FuzzyClass speedHighNegative = new FuzzyClass (-3, -2.75f, -2.35f, -1.95f);

    #endregion
```

6.2 FUZZYFICAZIONE

Il calcolo dei valori di membership a partire da valori di input delle variabili e classi fuzzy è gestito dal metodo **calculateMembership()** delle classi **FuzzyClass** e **CrispClass**. Il metodo accetta un valore flot che rappresenta il valore puntuale della variabile su cui si vuoe calcolare la membership, e restituisce un valore float nell'intervallo $[0,1]$ che rappresenta la membership calcolata.
I valori di membership vengono assegnati a variabili temporanee che saranno poi usate nelal definizione delle regole della FAM.

```
37          //variabile composta
38          float targetPosition = sensor.closestAlienDistance + sensor.closestAlienLead;
39
40          //fuzzyficazione
41          float tgp, tgpp, tgppp, tgn, tgnn, tgnnn;
42          tgp = targetCloseRight.calculateMembership (targetPosition);
43          tgpp = targetRight.calculateMembership (targetPosition);
44          tgppp = targetFarRight.calculateMembership (targetPosition);
45          tgn = targetCloseLeft.calculateMembership (targetPosition);
46          tgnn = targetLeft.calculateMembership (targetPosition);
47          tgnnn = targetFarLeft.calculateMembership (targetPosition);
48
49          float shield;
50          shield = underShield.calculateMembership (sensor.shieldDistance);
51
52          float sp, spp, sppp, sn, snn, snnn;
53          sp = speedLowPositive.calculateMembership (sensor.alienVelocity);
54          spp = speedMediumPositive.calculateMembership (sensor.alienVelocity);
55          sppp = speedHighPositive.calculateMembership (sensor.alienVelocity);
56          sn = speedLowNegative.calculateMembership (sensor.alienVelocity);
57          snn = speedMediumNegative.calculateMembership (sensor.alienVelocity);
58          snnn = speedHighNegative.calculateMembership (sensor.alienVelocity);
59
```

6.3 FUZZY ASSOCIATIVE MEMORY (FAM)

Le classi di output definite per il movimento sono uguali per tutti i behaviour e sono 4:

- **hardLeft**: con valore associato -0.8.
- **left**: con valore associato -0.4.
- **right**: con valore associato 0.4.
- **hardRight**: con valore associato 0.8.

Per la gestione dello sparo è sufficiente invece un booleano.

nel caso del behaviour in esame, le regole della FAM possono essere così definite:

- "se il bersaglio è lontano a destra, oppure a distanza media a destra e si sta muovendo velocemente verso destra o lentamente verso sinistra, oppure è vicino a destra e il cannone è sotto un bunker, allora muovi il cannone a destra velocemente."
- "se il bersaglio è vicino a destra, oppure a distanza media a destra e si sta muovendo velocemente verso sinistra, allora muovi il cannone lentamente verso destra"
- la regola per il movimento rapido verso sinistra è speculare a quella per il movimento rapido a destra.
- la regola per il movimento lento a sinistra è speculare a quella per il moveimento lento a destra.

Queste regole fanno in modo che il cannone si avvicini al bersaglio il più velocemente possibile se questo si trova lontano. segua la sua posizione lentamente (per non allontanarsene) quando questo si trova vicino, e anticipi il suo movimento quendo questo è a distanza media, dipendentemente dalla velocità a cui il bersaglio si sta muovendo.
inoltre, se il cannone si trova sotto un bunker, cercherà di anticipare leggermente il movimento del bersaglio spostandosi al di fuori dell'area di copertura nella direzione in cui il bersaglio si sta muovendo.
Siccome questo behaviour tende a far posizionare il cannone nella posizione ottimale per colpire il bersaglio più vicino, è sufficiente che il giocatore virtuale tenga sempre premuto il tasto di sparo a meno che non sia sotto un bunker.

```
60          //FAM
61          float hardRight, right, hardLeft, left;
62          float tgMovingRight, tgMovingLeft;
63          float tgIsRight, tgIsLeft;
64
65          // tgppp
66          //OR
67          // tgpp & (mvright | sn)
68          //OR
69          // shield & tgRight
70          tgMovingRight = FuzzyLib.fuzzyOr(sp, spp, sppp);
71          tgIsRight = FuzzyLib.fuzzyOr (tgp, tgpp, tgppp);
72          hardRight = tgppp;
73          hardRight = FuzzyLib.fuzzyOr (hardRight, FuzzyLib.fuzzyAnd
74              (tgpp, FuzzyLib.fuzzyOr (tgMovingRight, sn)));
75          hardRight = FuzzyLib.fuzzyOr(hardRight, FuzzyLib.fuzzyAnd (shield, tgIsRight));
76
77          // tgp
78          //OR
79          // (tgpp & (tgnn | tgnnn))
80          tgMovingLeft = FuzzyLib.fuzzyOr (tgnn, tgnnn);
81          right = FuzzyLib.fuzzyOr (tgp, FuzzyLib.fuzzyAnd (tgpp, tgMovingLeft));
82          right = FuzzyLib.fuzzyAnd (right, FuzzyLib.fuzzyNot (shield));
83
84          // tgn
85          //OR
86          // (tgnn & (tgpp | tgppp))
87          tgMovingRight = FuzzyLib.fuzzyOr (tgpp, tgppp);
88          left = FuzzyLib.fuzzyOr (tgn, FuzzyLib.fuzzyAnd (tgnn, tgMovingRight));
89          left = FuzzyLib.fuzzyAnd (left, FuzzyLib.fuzzyNot (shield));
90
91          // tgnnn
92          //OR
93          // (tgnn & (sn | snn | snnn | sp)
94          //OR
95          // shield & tgleft
96          tgMovingLeft = FuzzyLib.fuzzyOr (sn, snn, snnn);
97          tgIsLeft = FuzzyLib.fuzzyOr (tgn, tgnn, tgnnn);
98          hardLeft = tgnnn;
99          hardLeft = FuzzyLib.fuzzyOr (hardLeft, FuzzyLib.fuzzyAnd
100             (tgnn, FuzzyLib.fuzzyOr (tgMovingLeft, sp)));
101         hardLeft = FuzzyLib.fuzzyOr(hardLeft, FuzzyLib.fuzzyAnd(shield, tgIsLeft));
102
```

6.4 DEFUZZYFICAZIONE

una volta calcolati i valori di uscita della FAM, è necessario trasformali in valori delle variabili di output.
il processo utilizzat è piuttosto semplice: tutte le classi di tutti i controllori sono state proget-

tate in modo che la sommatoria delle classi attivate per una certa variabile in ogni momento non ecceda mai 1.
Possono chiaramente esserci valori delle variabili di input per cui la sommatoria delle membership delle classi attivate è inferiore a uno o addirittura zero. Ad esempio, le classi che modellano la distanza del bersaglio dal cannone sono fatte in modo che in corrispondenza del valore 0 sulla variabile nessuna di esse sia attivata.
Questo perchè semplicemente non si vuole che il giocatore si sposti dalla posizione ottimale raggiunta.
La defuzzyficazione dell'output di movimento è calcolata dalla formula $\sum_{k=1}^{n} m(c_k) * v_k$, dove $c_1 ... c_n$ sono le classi di output della FAM, e $v_1 ... v_n$ sono i valori di membership associati.
Il metodo **toBool()** mappa il valore float 1 a $True$, e ogni altro valore a $False$.

```
103         //defuzzyficazione
104         movementOutput = (-0.8f) * hardLeft + (-0.4f) * left
105                          + 0.4f * right + 0.8f * hardRight;
106
107         spacebarOutput = FuzzyLib.toBool (FuzzyLib.fuzzyNot (shield));
```

7 I COMPORTAMENTI

Ogni behaviour implementa un comportamento. Nella sezione precedente si è mostrato il comportamento **attackClosest**, secondo cui il giocatore cerca di distruggere il nemico a lui più vicino.
I comportamenti **attackLowest** e **attackRedShip** sono implementati in modo molto simile. Ciò che cambia è solo la variabile di ingresso, che andrà ad esprimere le distanze di un altro bersaglio. In realtà **attackRedship** è leggermente più semplice, in quanto non necessita di analizzare la velocità del bersaglio, che è costante (le navi rosse bonus si muovono con una velocità costante in modulo).
Il comportamento **shelter** è ancora più semplice: il cannone si dirige alla massima velocità possibile in direzione del bunker più vicino e lì si ferma. anche questo meccanismo è implementato lasciando un'intervallo sulla variabile distanza senza classi associate. In questo modo le membership delle classi che definiscono il movimento vanno tutte a zero quando la distanza dal bunker è 0, e il giocatore si ferma.
Avoid, invece, fa in modo che il giocatore tenti di evitare i proiettili nemici. per fare questo utilizza come variabili di input la componente orizzontale della posizione relativa al cannone del proiettile più vicino, ma anche la distanza tra il cannone e il limite dell'area di gioco.
Questa seconda variabile serve per fare in modo che questo comportamento non spinga troppo spesso il giocatore a rifugiarsi ai margini dell'area di gioco, dove è molto probabile che i proiettili nemici lo spingano. Posizionarsi vicino al bordo è pericoloso, in quanto è relativamente facile che si verifichino situazioni senza via d'uscita, in cui ilcannone rimane schiacciato tra i proiettili nemici e il bordo dell'area di gioco.
La FAM di **avoid** può essere così descritta:

- "Se ci sono proiettili vicini a destra, oppure il bordo è vicino a destra, allora muoviti velocemente a sinistra".
- "Se ci sono proiettili lontani a destra, allora muoviti lentamente a sinistra".
- specularmente sono definiti i movimenti nell'altra direzione.

La scelta di un movimento "lento" può sembrare controintuitiva ma ha una ragione "strategica": se il proiettile da cui si vuole fuggire è lontano, è infatti meglio spostarsi di poco, in quanto questo modificherà solo in maniera minore la posizione ottimale prodotta dagli altri comportamenti.
Perchè muoversi del tutto quando il pericolo è lontano? la distanza dai proiettili non è solo sulla componente orizzontale. Quindi un proiettile che si trova esattamente sulla verticale del giocatore ma piuttosto in alto risulterà "lontano", ma si avvicinerà molto in fretta, ed è quindi necessario allontanarsene.
In realtà questo comportamento non è in grado di garantire che il cannone eviterà tutt i proiettili. Il giocatore infatti gioca in maniera "greedy", scegliendo di istante in istante la posizione che massimizza la distanza dal pericolo, senza poter pianificare in avanti una strategia che permetta di evitare tutti i proiettili.
Questo è in realtà un limite generale dell'approccio "statico" dei controllori fuzzy: non facendo

pianificazione il giocatore è esposto a sequenze di eventi che possono portare inevitabilmente alla sconfitta. In questo caso, queste sequenze di eventi sono leghate al pattern con cui i proiettili vengono lanciati dai nemici, e questo pattern è casuale.
Una attenta definizione (raffinata con numerosi cicli di testing) delle classi fuzzy fa in modo che in questo caso i pattern di questo tipo siano relativamente pochi e quindi piuttosto rari.

7.1 IL SELETTORE

Un ruolo molto importante è ovviamente rivestito dal behaviour che si occupa di selezionare il comportamento da seguire istante per istante.
behaviourSelector usa, come variabili di input:

- **danger**, su cui son definite tre classi (low, medium, high), dove **dangerLow** è molto stretta e **dangerHigh** compre invece molti valori. questo è dovuto al fatto che la funzione pericolo è una sommatoria di elementi di tipo $\frac{1}{d}$, dove d è una distanza sempre positiva. Queste quantità tendono a ∞ per d vicine a zero, ma il loro valore cala molto bruscamente con d vicino a 1, per poi calare molto lentamente per $d > 1$. Di conseguenza si viene ad avere un range molto grande di valori che identificano una situazione di pericolo, mentre la funzione produce valori molto più ravvicinati in situazioni di bassa pericolosità.

- **shieldDistance**, su cui sono definite due classi, **shieldClose** e **shieldFar**.

- **redshipPresent**, su cui si ha una sola classe crisp che indica se c'è una navetta rossa in gioco o meno.

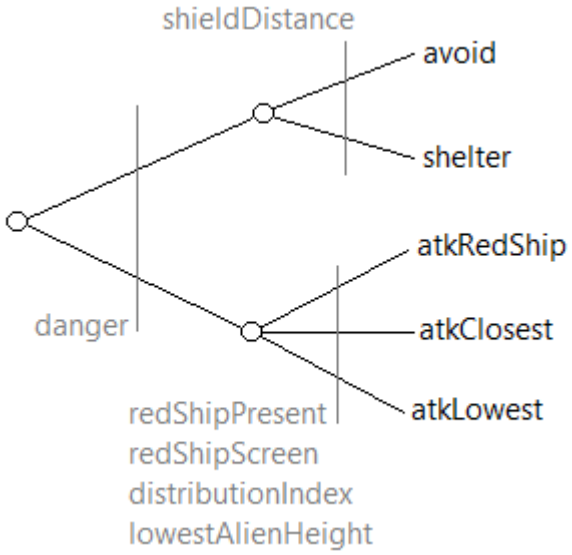
- **redShipScreen**, su cui sono definite tre classi (weak, medium, strong). La classe **screenShipStrong** è molto più "larga" delle altre: diventa infatti sensato analizzare con precisione quanti nemici sono in posizione tale da coprire la linea di tiro verso la navetta bonus solo quando ce ne siano relativamente pochi.

- **alienDistributionIndex**, su cui si hanno 3 classi uniformi: **distributionVertical** (la formazione nemica è irregolare, e presenta varchi verticali sfruttabili), **distributionUndefined**, e **distributionHorizontal** (la formazione nemica è compatta).

- **lowestAlienHeight**, su cui sono definite due classi uniformi: **lowestAlienLow** (i nemici sono ormai nela parte bassa dell'area di gioco e minacciano di raggiungere la Terra) e **lowestAlienHigh** (i nemici sono confinati alla parte alta dell'area di gioco).

Il design della FAM del selettore ricorda in qualche modo la struttura di un albero di decisione: invece di suddividere direttamente lo spazio dell'input in numerosi sottospazi a cui associare classi di output, una struttura ad albero provoca una serie di suddivisioni successive, la cui granularità non è necessariamente la minima possibile.
Infatti, la variabile **danger** permette di discriminare immediatamente tra due "rami" che portano a comportamenti "difensivi" (**avoid**, **shelter**, alto/medio livello di pericolo) e comportamenti "offensivi" (**attackClosest**, **attackLowest**, **attackRedShip**, basso livello di pericolo).

Allo stesso modo, il sottoramo dei comportamenti difensivi è suddivisibile secondo la variabile **shieldDistance**: se ho è presente un bunker vicino sotto cui ripararsi verrà selezionato **shelter**, altrimenti **avoid**. Non tutto l'albero però può essere costruito con decisioni che si basino su una sola variabile. Ad esempio, per decidere quale comportamento offensivo adottare è necessario utilizzare una combinazione di tutte le variabili rimanenti.



Gli output della FAM sono 5 classi corrispondenti ai 5 behaviours possibili. La FAM finale risulta contenere le seguenti regole:

- "se il pericolo è basso, allora gioca in attacco".
- "se il pericolo è medio o alto, allora gioca in difesa" (la mappatura "pericolo medio - difesa" è stata scelta sperimentalmente).
- "se in difesa e c'è un bunker vicino, allora **shelter**".
- "se in difesa e non c'è un bunker vicino, allora **avoid**".
- "se in attacco e nemici vicini alla terra, allora **attackLowest**".
- "se in attacco e la navetta bonus è presente e si ha che
  - ci sono pochi nemici sulla linea di tiro verso di essa, oppure
  - la formazione nemica è irregolare, oppure
  - la formazione nemica è indefinita, e la copertura della navetta bonus è media, e la formazione nemica è lontana dalla terra

  allora **atkRedShip**".
- "se in attacco e i nemici sono lontani dalla terra e
  - la formazione nemica è compatta, oppure

– la navetta rossa non è presente, e la formazione nemica è irregolare o indefinita

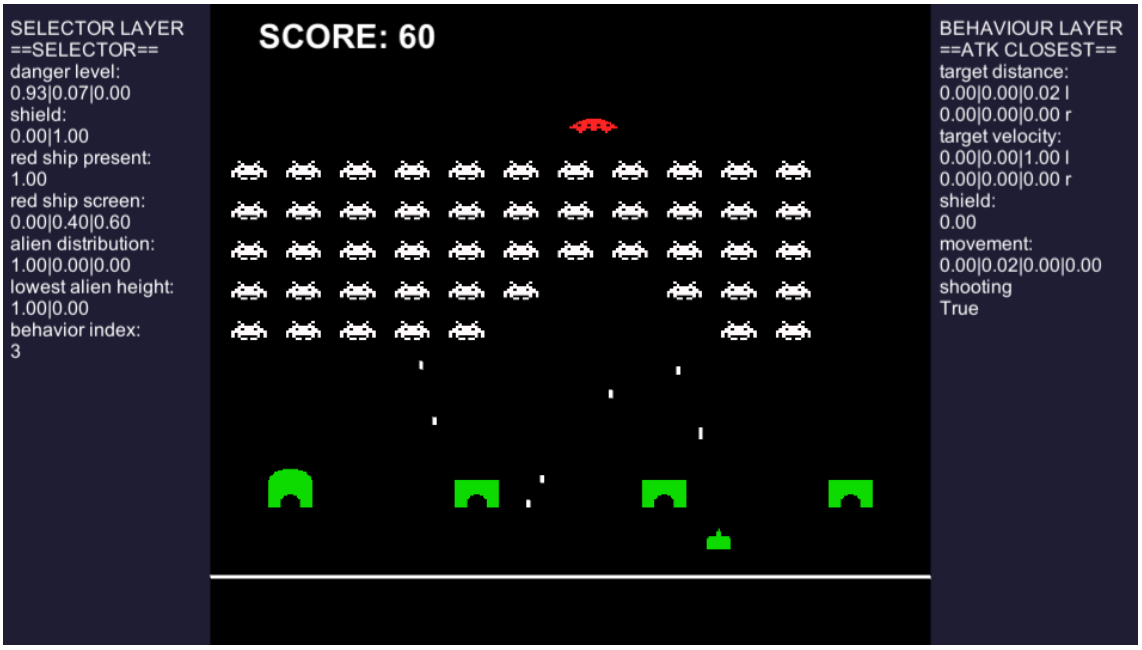
allora **atkClosest**".

Le regole che determinano attacco/difesa servono solo a facilitare la scrittura e la leggibilità, potrebbero facilmente essere scritte per esteso all'interno delle altre.

```
87          //FAM
88          float defend = FuzzyLib.fuzzyOr (dngm, dngh);
89          float attack = dngl;
90
91          //avoid = defend & !shc
92          behaviours[avoid] = FuzzyLib.fuzzyAnd(defend, shf);
93          
94          //shelter = defend & shc
95          behaviours [shelter] = FuzzyLib.fuzzyAnd (defend, shc);
96
97          //atkred = attack & pres & (sw | dv | (sm & du & lwh))
98          behaviours[atkRedship] = FuzzyLib.fuzzyAnd(
99              attack,
100             rs,
101             FuzzyLib.fuzzyOr(sw, dv, FuzzyLib.fuzzyAnd(sm, du, lwh)));
102
103         //atkClosest = attack & lwh & (dh | (!rs & (du | dv)))
104         behaviours[atkClosest] = FuzzyLib.fuzzyAnd(
105             attack,
106             lwh,
107             FuzzyLib.fuzzyOr(
108                 dh,
109                 FuzzyLib.fuzzyAnd(
110                     FuzzyLib.fuzzyNot(rs),
111                     FuzzyLib.fuzzyOr(du, dv))));
112
113         //atkLowest = attack & lwl
114         behaviours[atkLowest] = FuzzyLib.fuzzyAnd(attack, lwl);
115
```

I livelli di attivazioni delle classi di output sono memorizzati in un array. Ad ogni posizione di questo array è associato un comportamento.
La defuzzyficazione prevede semplicemente di trovare l'indice del comportamento con il massimo livello di attivazione. L'identificatore di questo comportamento ne determinerà la selezione per il frame corrente.

```
116         //defuzzyficazione
117         behaviourIndex = MyUtils.argmax (behaviours);
```



La schermata è divisa in tre parti: al centro c'è l'area di gioco (sfondo nero), con il cannone comandato dal giocatore e i bunker nella parte bassa (in verde), la formazione nemica in bianco, e una navetta bonus rossa nella parte superiore.
Nella parte sinistra della schermata sono visualizzate informazioni relative allo stato del layer superiore: le classi fuzzy di input e di output, nonchè l'output finale defuzzyficato ("behaviour index").
Nella parte destra sono invece mostrate le stesse informazioni relativamente però al layer inferiore classi fuzzy relative alla stessa variabile sono state raggruppate su di una stessa riga.
Nell'immagine si vede che sulla variabile che indica la distanza del bersaglio è attivata solo la classe "vicino-sinistra" **targetCloseLeft** con livello di attivazione 0.02, poichè il bersaglio è molto vicino al cannone (in termini di componente orizzontale delle posizioni).
per quanto riguarda invece la velocità del bersaglio è attivata con membership 1 la classe "lento-sinistra" **slowPositive**, e per quanto riguarda i bunker la classe **underShield** non è attiva.
In uscita dalla FAM si ha al classe dassociata a un movimento lento verso sinitra attivata con valore 0.02, e l'output booleano che indica lo sparo a $True$.

9 VALUTAZIONE DELLE PRESTAZIONI

9.1 GIOCATORE AUTOMATICO

Per valutare le prestazioni del giocatore automatco rispetto ad un giocatore umano medio sono state scelte alcune metriche semplici e intuitive: la percentuale di partite vinte, il punteggio ottenuto, e la durata delle partite.
Le statistiche vengono raccolte tramite un sistema di logging, che al termine di una partita scrive su un file i risultati ottenuti.
I dati sulle prestazioni del giocatore virtuale sono stati raccolti lanciando tre cicli da 100 partite l'uno e raccogliendo i risultati:

- percentuale di vittorie: 93%,
- punteggio medio: 761,
- durata media: 49s,
- durata media in caso di sconfitta: 21s.

Quello che si nota immediatamente è che il giocatore virtuale vince quasi sempre. Le sconfitte che sono state osservate direttamente sono state causate da pattern casuali nella caduta dei proiettili che hanno spinto il cannone verso un estermo dell'area di gioco senza lasciare spazio sufficiente perchè venisse mosso nuovamente verso il centro.
Così bloccato, il cannone è stato poi colpito da un successivo proiettile prima di potersi liberare dalla posizione di pericolo.
L'incidenza di questo tuipo di pattern sembra però essere relativamente bassa. In realtà non è stato possibile osservare direttamente tutte le partite in cui il giocatore virtuale è stato sconfitto, quindi è possibile che ci siano altri pattern a cui non riesce a rispondere adeguatamente.
Analizzando il punteggio medio si nota che 760 punti corrispondono a 550 punti derivanti dalla distruzione di tutti i nemici (che valgono 10 punti ciascuno), e 210 punti derivanti dalla distruzione delle navette bonus (100 punti ciascuna, per una media di 2,1). Questo calcolo sarà utile per confrontare la capacità di cambiare bersaglio del giocatore virtuale con quella del giocatore umano.
La durata media delle partite è di 49 secondi, consideranto che il giocatore deve distruggere 55 nemici più eventuali nemici bonus per vincere una partita, questo risulta in un'efficienza nell'eliminare nemici di almeno 1.122 nemici distrutti al secondo.
La durata media delle partite in cui il giocatore ha perso è abbstanza regolare: 21 secondi in media, con un massimo di 25.5 e un minimo di 18.1. Questo indica che le sconfitte sono sempre avvenute nella fase "centrale" della partita, in cui i nemici sono ancora numerosi e piuttosto distribuiti sull'area di gioco. Questo porta a pensare che il giocatore virtuale sia in grado di evitare di essere colpito anche quando i proiettili sono distribuiti in modo piuttosto denso (cosa che accade a fine partita), ma può fallire invece quando sono distribuiti orizzontalmente lungo tutta l'area di gioco.

9.2 GIOCATORI UMANI E CONFRONTO DELLE PERFORMANCE

Sono state raccolte statistiche sulle prestazioni di giocatori umani: 8 diverse persone hanno potuto provare il gioco due volte senza che i dati relativi venissero registrati, e poi hanno giocato circa dieci partite a testa, per un totale di 93 partite:

- percentuale di vittorie: 51%,
- punteggio medio: 390,
- durata media: 35s,
- durata media in caso di sconfitta: 18s.

Questi risultati suggeriscono che il campione di giocatori umani ha performance nettamente inferiori rispetto al giocatore virtuale. In particolare i giocatori umani sono molto meno in grado di evitare i proiettili e tendono quindi ad essere sconfitti nelle parti iniziale e centrale della partita, quando i nemici sono distribuiti.
Oltre alla raccolta di dati satistici è stato possibile osservare direttamente giocatori umani mentre giocano, è quindi possibile aggiungere un'analisi empirica della loro performance basata sull'osservazione diretta.
I giocatori umani hanno grosse difficoltà a gestire tanti eventi contemporaneamente. Per questo motivo numerose partite sono terminate con una sconfitta nelle primissime fasi di gioco, entro i primi 10 secondi.
Le difficoltà continuano nella parte centrale della partita: i nemici sono meno numerosi ma ancora distribuiti su tutta l'area di gioco.
Una volta ridotto sufficientemente il numero di nemici da cui difendersi, i giocatori umani riescono a pianificare strategie complesse in risposta a un numero di eventi ridotto. di conseguenza performano molto bene nelle fasi finali della partita.
Un altro aspetto da considerare è che il campione presenta una grande varianza in termini di abitudine e dimestichezza in attività di questo tipo: alcuni giocatori sono abituati ai videogiochi e quindi capaci di coordinare molto bene un'analisi dellop spazio di gioco e una risposta imemdiata. Altri invece non hanno questa dimestichezza e faticano ad analizzare l'area di gioco nella sua interezza o a tradurre il movimento che pianificano in un movimento preciso sullo schermo.
Per questa ragione i risultati raccolti presentano anch'essi una grande variabilità: la partita "peggiore" è terminata dopo soli 4 secondi con un punteggio di 20, mentre i giocatori più abili hanno raggiunto performance che si avvicinano a quelle del giocatore virtuale con un rateo di vittorie vicino all'80%.
Ciò che nessun giocatore umano riesce a fare tanto bene quanto il giocatore virtuale è ottenere punteggi alti: anche i giocatori migliori faticano durante la parte centrale della partita (in cui quindi ci sono ancora tanti nemici) ad individuare le opportunità di colpire i nemici bonus. Questof a sì che essi riescano a guadagnare punti bonus solo nella fase finale.
Inoltre il giocatore virtuale è più efficiente nel distruggere i nemici, a parità di tempo: la media dei giocatori umani è inferiore 1 nemico distrutto al secondo in caso di vittoria, questo è dovuto al fatto che, rispetto al giocatore virtuale, "sprecano" più tempo concentrandosi

nell'evitare i proiettili e lo fanno in maniera meno efficiente. Infatti, per evitare i proiettili, i giocatori umani tendono a muoversi direttamente nell'area dello schermo più libera, dovendo poi spendere più tempo per rimettersi in posizione.
Un'altra differenza netta è la difficoltà, riscontrata per tutti i giocatori umani, di mirare e colpire correttamente nemici nella parte alta dello schermo, come ad esempio le navette bonus.
Questo introduce un'ulteriore inefficienza nell'uso del tempo e un'ulteriore inefficienza nell'ottenimento di punteggi alti. Il giocatore virtuale invece è in grado di posizionarsi in maniera quasi perfetta in modo da colpire il proprio bersaglio molto rapidamente.
In definitiva, il giocatore artificiale ha performance nettamente migliori di quelle espresse dai giocatori umani del campione. Presenta però un punto debole derivante dalla propria incapacità di pianificare in anticipo strategie complesse, il che lo rende vulnerabile ad avvenimenti casuali che restringono progressivamente le sue possibilità di movimento fino a bloccarlo in una situazione senza uscita.
Sarebbe quasi sempre possibile districarsi da queste situazioni, un giocatore umano potrebbe infatti riconoscere un pattern di questo genere e tentare di liberarsene finchè ne ha la possibilità. Il giocatore virtuale implementato, però, non è capace di un'analisi così complessa di quello che sta avvenendo e di conseguenza non elabora contromisure sufficienti.

10 SVILUPPI E MIGLIORAMENTI

il sistema può essere sicuramente migliorati in diverse direzioni:

- migliore processo di raccolta statistiche: per quanto riguarda le statistiche relative ai giocatori umani, sarebbe sensato allargare il campione. Il campione usato è ristretto sia nel numero di giocatori coinvolti sia nel numero di partite per giocatore.
  Per quanto riguarda invece il giocatore automatico sarebbe interessante migliorare l'analisi dei casi in cui la partita viene persa, in modo da poterli progressivamente studiare ed eventualmente ridurli. Un'idea portebbe essere quella di registrare quelo che avviene sullo schermo e tenere sempre memoria degli ultimi secondi, imm nodo da poter osservare a posteriori le catene di eventi che hanno portato ad una sconfitta.

- Una grande semplificazione che si è fatta nell'implementazione dell'ambiente di gioco è la decisione di tenere la diffioltà costante. La relativa staticità di un controllore fuzzy rende particolarmente complessa la gestione di un problema che cambia continuamente, ma un controllore capace di gestire un problema del genere sarebbe un risultato particolarmente interessante.

- Una analisi piàù avanzata dell'ambiente di gioco potrebbe permettere al giocatore virtuale di gestire situazioni più difficili. Invece che analizzare la posizione dei singoli proiettili nemici potrebbe essere utile clusterizzarli per posizione in gruppi in modo da ottenere una sorta di mappatura della densità di pericolo. Sarebbe quindi possibile identificare in qualche modo settori dell'area di gioco dove muoversi in modo da minimizzare le possibilità di sconfitta.