Francesco, Valentina e Laura Sisini

Sfidare gli algoritmi

Cinque ricette per videogiochi in C su Linux

Edizioni: i Sisini Pazzi, 2019

Ringraziamenti

Il primo ringraziamento va ad Anna che ha ideato diverse illustrazioni e che ci ha sostenuto attivamente durante la preparazione di questo testo producendo i prototipi grafici e i modelli in cartone per le animazioni di Pacman.

Vogliamo poi ringraziare tutti i membri del gruppo facebook

Reti neurali in C

che, con la loro presenza, ci hanno dato la motivazione per impegnarci nella realizzazione di questo volume sui gioco-programmi della Scuola Sisini.

Avvertenze

Codice sorgente
Il libro nasce per essere un testo auto-consistente, quindi in esso è riportato tutto il codice di ognuno dei 5 giochi, comunque, al momento in cui viene scritta questa avvertenza, il codice è presente anche su GitHub all'indirizzo `https://github.com/francescosisini` Il codice riportato nel libro è stato in parte modificato per ragioni di impaginazione, inoltre sono stati eliminati alcuni commenti.

Commenti nel codice
Il codice e l'architettura dei giochi sono spiegati nel testo. I commenti nel codice sono stati ridotti al minimo. Piuttosto si è scelto di usare quasi sempre l'italiano per i nomi delle funzioni e delle variabili principali per rendere più chiaro il flusso del ragionamento.

Stile di editing
Che il codice debba essere chiaro e in ordine, siamo tutti d'accordo. Però dobbiamo puntualizzare che il codice che presentiamo qui non segue alcuni canoni stilistici particolarmente in auge in questi anni. Per esempio, per noi scrivere:

```
int a=4;
```

è lo stesso che scrivere

```
int a = 4;
```

inoltre dormiamo bene lo stesso anche se scriviamo questo:

```
for(int i=1;i<10;i++) {
...
}
```

invece di

```
for(int i=1;i<10;i++)
  {
  ...
  }
```

Informazioni sulla proprietà intellettuale e la licenza d'uso

INTRODUZIONE

Perché scrivere un videogioco? Perché scrivere un libro sui videogiochi a caratteri nel 2019?

Quando i videogiochi sono nati e si sono diffusi, dagli anni '70 in avanti, il terreno per il loro successo era molto fertile. La gente vedeva la tecnologia che da lontano si stava avvicinando e stava invadendo le case, le scuole e gli ambienti di lavoro. Cercare di capirla e prendervi confidenza era la reazione naturale della società.

I videogiochi costituirono l'espressione perfetta della didattica a distanza. Non vi fu bisogno di insegnare l'elettronica e l'informatica di base alla generazione degli anni '70 e '80, perché ne capirono le potenzialità già giocando in sala giochi per duecento lire a partita, oppure acquistando una console e le relative cartucce.

Nel 2019, anno in cui viene terminato questo libro, nel mondo benestante, tutti conoscono le potenzialità dell'elettronica digitale. L'idea che un'immagine, come quella dello schermo di un videogioco, possa trasformarsi in base alle azioni che vengono compiute attraverso un joystick, non è per nulla sconvolgente.

Allora, cosa hanno ancora da dirci i videogiochi?

Le grosse compagnie di videogame stanno provando a rilanciare il mercato con investimenti notevoli in realtà virtuale ed aumentata. Questa tendenza testimonia che i videogame si stanno trasformando da strumenti di gioco/apprendimento quali erano, a oggetti di intrattenimento come i film e le serie.

Eppure, sotto la cenere, la fiamma del gioco brucia ancora.

Inizialmente è stata la tecnologia a stupire tutti con le sue potenzialità. Nel 1983, nella pubblicità del Commodore 64, un monitor mostrava un paesaggio natalizio su cui scendeva la neve. Per l'epoca, un'animazione prodotta in tempo reale, cioè che non fosse stata precedentemente videoregistrata era impensabile, invece con un computer, la si poteva produrre scrivendo un programma. Nasceva la computer graphics, e i videogiochi ne erano i primi testimonial.

Oggi è l'intelligenza dei computer che ci sta stupendo, come la capacità

di un servizio digitale di interpretare l'espressione di un viso o l'intonazione della voce.

Se lo stupore è svanito, il suo posto non è stato preso dalla conoscenza.

La maggior parte tra coloro che ritengono, per esempio, il gioco del Pacman superato, non sanno su che tecnologia si basasse ne saprebbero ricrearlo. Sebbene non si possa essere tutti informatici, è chiaro che un tale atteggiamento verso la tecnologia, porta ad una continua *esaltazione* per il nuovo, che poi si trasforma rapidamente in disinteresse e nella ricerca della prossima novità capace di riaccendere le emozioni: in pratica il ciclo dell'assuefazione.

Quando la fame di nuove emozioni non è saziata dalla catena di produzione delle novità, allora si guarda al passato. A volte al passato prossimo, rivivendo in forma *vintage* i feticci che ci hanno lasciato, altre al passato remoto, caricando di misteri e valori imperscrutabili le opere degli antichi.

Come nel vero amore, l'emozione autentica viene dopo l'innamoramento, quando si guarda la propria metà e si capisce che non era infatuazione, ma un sentimento profondo radicato nella complementazione.

Così, chi ha amato i videogame classici sa che non era solo la tecnologia ad incollarlo al video, ma era la sfida di confrontarsi con l'*intelligenza* del computer usando logica, attenzione, riflessi e un joystick.

La stessa logica e la stessa attenzione sono l'elemento legante che valorizza il passato che continua nel presente. Anziché usando i riflessi e il joystick, le sfide possono essere accettate e combattute usando le stesse armi del computer, il suo linguaggio più primitivo.

Questo libro illustra cinque giochi in cui il giocatore deve codificare la propria strategia in linguaggio C, scrivendo una funzione che risponda al suo posto ogni volta che la macchina gli passa il turno.

PROGRAMMAZIONE DI UN VIDEOGIOCO

Ambiente, personaggi e regole

La creazione di un videogioco è simile alla realizzazione di altre forme di narrazione e intrattenimento come ad esempio una storia epica, un poema, un dramma teatrale, un romanzo o un film cinematografico: si tratta di un'opera creativa guidata dal metodo e da alcune regole.

Per quel che riguarda i cinque giochi di Tuki e Giuli abbiamo seguito uno schema molto semplice:

1. scelta del tema
2. scelta dei personaggi e dei ruoli
3. scelta del campo di gioco (o ambiente)
4. scelta dell'obiettivo

Piuttosto che affrontare questi argomenti in modo troppo astratto, vediamo come sono stati applicati al gioco Tuki 1, il primo di questa serie, o *Sfida del trifoglio*. È possibile che troviate dei riferimenti agli stessi giochi con dei titoli leggermente diversi, per esempio, sempre riferendosi a Tuki 1, potreste trovare *Sfida all'ultimo trifoglio* su qualche post di Facebook o Instagram. non importa. Ci siamo accorti da tempo che non siamo molto bravi nel processo di *branding*. Cambiare il nome ad un prodotto è uno sbaglio perché limita la diffusione del prodotto stesso, però questo fa parte di una nostra natura creativa che, fino ad ora, non siamo riusciti a contenere, speriamo in futuro di fare meglio.

Tornando al gioco, vediamo come si applicano i punti visti sopra a Tuki 1:

1. tema: algoritmi per la ricerca di un cammino in un territorio inesplorato e privo di mappa
2. personaggi: Tuki e Giuli, antagonisti paritetici
3. ambiente: campo di trifogli con presenza di ostacoli
4. obiettivo: mangiare più trifogli dell'antagonista

Il gioco Tuki 1 è stato ispirato dai nuovi robot aspirapolvere. Molti di questi robot devono completare le pulizie di un pavimento senza disporre di una mappa, infatti, in ambiente domestico la disposizione degli oggetti sul pavimento può cambiare sia in numero che in posizione ed è impensabile che l'utenza media (cioè tutti noi) si debba

preoccupare di fornire una mappa al robot prima di chiedergli di eseguire le pulizie.
Si potrebbe anche fare, giusto per sperimentare una tecnologia, una, due volte, ma, se si intende venderlo come un elettrodomestico, deve essere semplice da usare e non richiedere delle operazioni complesse.

Per questo motivo i robot aspirapolvere hanno un compito complicato: 1) muoversi senza una mappa e 2) percorrere (pulire) l'intera area. L'idea è quindi quella di richiedere al giocatore del gioco-programma di porre l'attenzione su questo problema e provare a scrivere un algoritmo che risponda alle due esigenze evidenziate sopra.

Non entriamo già adesso nei dettagli di questo gioco, questi verranno discussi approfonditamente nei capitoli successivi, vediamo invece come procedere con l'analisi per completare la realizzazione di un gioco.

Scelta della tecnologia

Dopo la scintilla iniziale, che porta a definire l'idea generale del gioco, bisogna stabilire la tecnologia con cui realizzarlo, che è molto importante perché determina: 1) il pubblico del gioco, 2) i sensi coinvolti e 3)le emozioni suscitate.

Il pubblico

Un videogioco è un prodotto e come tale la sua distribuzione implica anche la diffusione di un messaggio. La percentuale di successo di un prodotto, cioè la percentuale con cui *succede* che il prodotto venga acquisito (o acquistato) e gradito, dipende in buona parte dal messaggio che porta. Ci sono tantissimi casi di prodotti che hanno avuto un successo inimmaginabile sulla carta, ovvero su cui in pochi avrebbero, anzi hanno, scommesso. Oggi questi pochi, o i loro discendenti, godono del frutto delle loro intuizioni. Un esempio tra tutti è quello dell'automobile. Senza fare ideologia, è chiaro a chiunque che l'utilizzo dell'automobile in ambiente urbano è un'assurdità. Se ci mettessimo d'accordo e utilizzassimo tutti dei mezzi alternativi, avremmo un vantaggio inimmaginabile in termini di qualità della vita e risparmio economico.

All'alba del XX secolo, nessuno, o quasi, avrebbe scommesso che nel XXI secolo ogni cittadino in età da patente avrebbe posseduto

un'automobile. I vantaggi di disporre di un'auto di proprietà non sono poi così tanti come si crederebbe. A conti fatti è una spesa molto importante, un rischio continuo di subire o causare incidenti, uno stress per l'attenzione e un inquinamento costante, questo lo capivano anche cento anni fa: allora com'è stato che oggi tutti abbiamo un'auto?

Prima di rispondere, consideriamo che i vantaggi di veicoli a motore si avrebbero anche senza le auto di proprietà, infatti, questi potrebbero essere usati dalle forze dell'ordine, dai mezzi di emergenza, dai trasporti pubblici e anche essere a disposizione in forme simili al noleggio o al car-sharing per viaggi in cui è richiesto di essere indipendenti dai mezzi, come, ad esempio, per andare in vacanza in un luogo non metropolitano. Inoltre, ricordo che, se per le strade metropolitane non dovessero circolare continuamente automobili, non sarebbe necessario asfaltarle, ma ci si potrebbe limitare alla terra battuta o all'acciottolato, riducendo sensibilmente i costi di manutenzione cittadina e anche le temperature estive.

Se, a conti fatti, tutti siamo capaci di capire che l'automobile privata non è un bene né per sé stessi né per la società, allora perché l'abbiamo comprata tutti? Perché abbiamo permesso a questo oggetto di invadere le nostre città? La risposta è ovviamente **il messaggio**. Agli inizi del secolo XX, quando ci si muoveva a piedi o in carrozza, l'auto era un messaggio, un messaggio di progresso, un messaggio di ciò che il progresso avrebbe portato.
I motori a scoppio e i motori elettrici promettevano all'uomo la libertà da fatiche a cui si era sottoposto da sempre per provvedere alle proprie esigenze. L'automobile era uno degli emblemi di questa promessa. Guidare un'auto non era solo esibire uno status, era anche diffondere un'idea e portarla là fino a dove c'erano ancora degli scettici.

I sensi
Certo, in fisiologia i sensi sono cinque e su questo non vogliamo discutere, spesso però nel linguaggio parlato ci riferiamo a concetti come *il senso dell'orientamento*, *il senso del ritmo*, *il senso del traffico* ecc., ed è con questa accezione più generale che vogliamo coinvolgere i sensi nei videogiochi, quindi non solo i sensi fisiologici, ma anche i sensi intesi in senso lato.

Suono La prima distinzione è tra giochi che presentano una base sonora e giochi che non la presentano. Anticipiamo che i giochi

presentati in questo testo dalla Scuola Sisini non presentano una base sonora, il motivo è che la gestione del suono è complessa e richiede delle librerie specifiche il cui uso esula dallo scopo di questo libro.

Il sonoro comunque è molto importante, così come è importante il senso dell'udito.
Il sonoro in un gioco può essere usato sia per chiarire le dinamiche del gioco che per accompagnare il giocatore con un sottofondo orecchiabile. La seconda delle due può essere facilmente rimpiazzata dall'ascolto della musica preferita durante la sessione di gioco, mentre le prima ha un valore intrinseco, perché i suoni o i *motivetti* sono sincronizzati all'azione che avviene nel videogioco permettendo al giocatore di immergersi maggiormente nel gioco e quindi di utilizzare al meglio i propri sensi per giocare.
L'utilizzo del suono è quindi un valore aggiunto al gioco, purché la dinamica del gioco non si basi specificatamente sul silenzio.

Video La componente video in un videogioco è scontata, altrimenti perché chiamarlo videogioco? Nonostante questo, è possibile realizzare dei giochi basati solo sull'audio e, in futuro, esisteranno probabilmente giochi che si baseranno soltanto sullo stimolo diretto dei neuroni, senza la mediazione della vista.

In un videogioco classico la componente video è molto importante proprio perché la vista è il senso che raccoglie le informazioni necessarie per giocare. La componente video non ha solo il compito di mostrare una grafica gradevole e coinvolgente, ma quello fondamentale di fornire al giocatore il contesto del gioco e la percezione degli eventi ai quali egli deve rispondere. Per questo motivo anche la tecnologia con la quale viene realizzato gioca un ruolo determinante.

Nella progettazione di un gioco è importante chiedersi cosa si vuole comunicare con le immagini, quali sono gli eventi significativi nel gioco e come questi vengono presentati al giocatore. Per fare un esempio pensiamo al famoso gioco del Pacman di Toru Iwatani.

Il gioco si svolge in un labirinto definito da muri sia perimetrali che interni. Il protagonista, Pacman, una pizza gialla a cui manca una fetta, deve mangiare tutte le pastiglie che sono disseminate lungo il labirinto. Quattro fantasimini: Blinky, Pinky, Inky e Clyde gli rendono il compito

difficile inseguendolo per mangiarlo.
Il compito del giocatore è quello di guidare Pacman all'interno del labirinto, evitando l'incontro con i fantasmini e cercando di mangiare tutte le pastiglie. Il giocatore è sempre libero di invertire il verso del moto di Pacman e, in corrispondenza degli incroci dei muri, può scegliere che direzione prendere.
Per compiere le proprie scelte, il giocatore ha bisogno di sapere dove si trovi il Pacman rispetto al labirinto e dove si trovino i fantasmini per evitare di incontrarli. Il video è quindi necessario per fornire queste informazioni in tempo reale al giocatore.
Dopo queste considerazioni si potrebbe dedurre affrettatamente che la comunicazione video possa essere limitata al trasferire le informazioni necessarie: non è proprio così. Attraverso il video viene mostrata non solo l'astrazione del gioco, ma anche l'idea originale che l'autore del gioco ha in mente.

Per esempio la figura qui sotto è una tavola nata durante la progettazione del gioco Tuki 1. Stavamo provando ad immaginare come potevano essere i protagonisti immersi nell'ambientazione, come poteva essere il labirinto e come si poteva immaginare che andassero realmente le cose. Laura allora, insieme ad Anna, ha iniziato a trasferire le idee sulla carta realizzando i disegni a matita, poi Vale ha completato la tavola con l'editing grafico.
La realizzazione di una grafica così sofisticata avrebbe richiesto l'uso di tecnologie più avanzate rispetto quelle che avevamo deciso di usare per questo gioco, ma la tavola ci diede l'obiettivo, l'idea base a cui ispirarsi.

Le emozioni
Rivolgersi alle emozioni può anche sembrare facile. Sui social-media spopolano post che cercano di commuovere, indignare, inorgolgire ecc., messaggi che si rivolgono direttamente al nostro cervello emotivo.
Per certi versi, suscitare emozioni `facili` può essere anche semplice. Non serve un genio per rattristare un essere umano empatico mostrandogli un immagine di un suo simile che soffre senza colpa.
Spesso questo tipo di comunicazione emotiva viene usata per estorcere denaro, o peggio consenso politico. Le emozioni così suscitate hanno però vita breve, e facilmente possono anche trasformasi in emozioni complementari o opposte. Se trenta anni fa vedere una scena di guerra al telegiornale poteva far passare l'appetito ed interrompere una cena,

oggi non è inconsueto che si cerchino scene analoghe su youtube per rompere la noia aspettando la cena.
Ben diverso è comunicare qualcosa usando delle emozioni. Le persone creative amano creare e comunicare le loro creazioni. Un musicista compone un tema per farlo ascoltare e per suscitare emozioni in chi lo ascolta. Il tema e l'emozione che esso suscita si fondono in una cosa sola. Se però il tema risulta banale o vuoto all'orecchio dell'ascoltatore, allora non viene comunicata nessuna emozione e il tema sarà presto dimenticato. Per fare un esempio quasi contemporaneo a questo testo, si provi ad ascoltare *One Punch Man Sadness* di Miyazaki Makoto. Il tema, scritto per il famoso anime, parla direttamente alle emozioni di chi lo ascolta, sfruttando la tecnica della composizione musicale, ma offrendo un prodotto genuino: le emozioni del compositore.
Le stesse emozioni possono essere offerte con la produzione di un videogioco. Un gioco può essere allegro, triste, romantico (si pensi a *Final fantasy*) o avventuroso. Se si è onesti e si padroneggia la tecnica, si possono comunicare emozioni autentiche che rimarranno nel cuore di chi ci gioca.

La scelta di usare solo il terminale come interfaccia di comunicazione con il giocatore limita molto le possibilità espressive del game designer. Il terminale non permette di trasformare il prodotto della propria fantasia direttamente nelle immagini che si vorrebbe, inoltre limita la fluidità del movimento riducendolo ad uno spostamento carattere per carattere.
Ciò nonostante, il game designer, posto di fronte a questi limiti, ha la possibilità di focalizzare al massimo la propria idea ispiratrice e di trovare il modo di concentrarsi solo su essa e sulla tecnologia per realizzarla che differentemente dalle modalità grafiche più evolute può essere afferrata e capita in profondità.

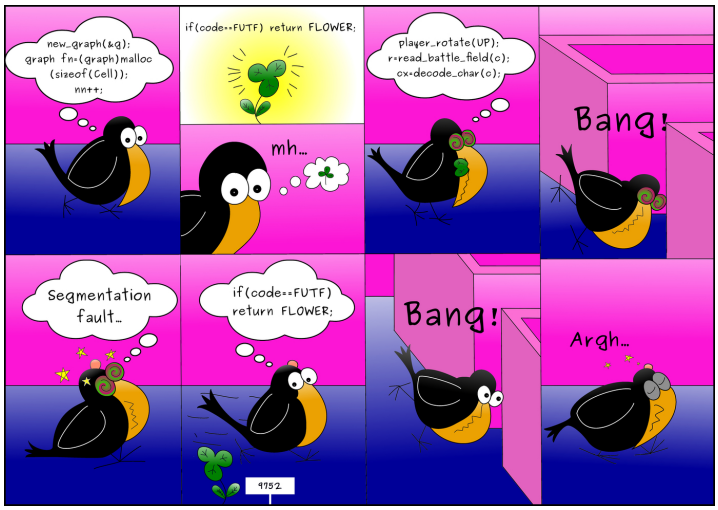


Fig. 1 - La figura mostra la tavola da cui fu tratta la prima grafica del gioco di Tuki 1.

Analisi e progettazione

Questo paragrafo è probabilmente uno dei più importanti dell'intero testo. I concetti chiave saranno presentati qui, poi verranno ripresi e spiegati a più riprese. Sebbene non si tratti di idee originali o particolarmente ostiche, i concetti e le idee che seguiranno sono la base della programmazione dei videogame classici e quindi vanno analizzate con cura.
Vediamo anzi tutto che l'idea alla base dei videogiochi classici è quella di rappresentare una situazione che sia compatibile con la fisica conosciuta o con una versione della fisica *alterata* ma comunque normata da leggi:

> ***Gli eventi che accadono in un videogioco sono governati dalle leggi della fisica***

Con questa premessa definiamo alcuni elementi che troveremo nei videogiochi:

1. campo da gioco
2. elementi mobili del campo
3. giocatori
4. regole del gioco

Il campo da gioco

Il campo da gioco è tipicamente una rappresentazione di una realtà di 2 o 3 dimensioni. Se la realtà rappresentata è bidimensionale, allora è possibile stabilire una relazione (o mappa) diretta tra la realtà da rappresentare e la sua rappresentazione video.

Per fare un esempio, consideriamo il gioco del Pacman. I personaggi e i muri sono chiaramente bidimensionali, in pratica i progettisti hanno pensato ad un gioco che potrebbe avvenire su di un piano e i cui personaggi hanno altezza o spessore trascurabile. Il gioco del Pacman ha quindi una mappa diretta tra il campo immaginario in cui avviene il gioco e l'immagine video che viene prodotta per giocare.

Quando invece un gioco deve rappresentare una realtà tridimensionale, cioè in cui i giocatori possono muoversi nelle tre direzioni *(x,y,z)*, è necessario creare una mappa dalla rappresentazione logica in 3 dimensioni alla rappresentazione video in 2. Un modo per creare detta mappa è l'uso delle regole della prospettiva o delle proiezioni geometriche. Non entreremo più a fondo nell'analisi di questo argomento perché i cinque giochi presentati in questo testo sono tutti di tipo 2D.

Normalmente con campo da gioco si intendono tutte quelle strutture fisiche che non mutano durante l'esecuzione del gioco. Tornando al gioco del Pacman, il campo da gioco è costituito dai muri e dalla stanza dei fantasmi, mentre le pastiglie sono elementi mobili.

La caratteristica principale del campo da gioco è che fornisce la *reazione vincolare* necessaria per il sostegno dei giocatori. Troppo difficile? No dai, si tratta solo di fare un po' di fisica elementare. Quando scendiamo le scale di casa ci stiamo muovendo lungo la terza dimensione *z*, giusto? Eppure non abbiamo le ali! Le scale, viste di profilo, possono essere viste come una struttura 2D. Quando appoggiamo il piede su uno scalino, il nostro corpo è attratto verso il suolo dalla forza di gravità, ma non precipita perché la forza di reazione vincolare, opposta dal gradino, lo sorregge. In questo modo, aggiungendo la forza muscolare a questo sistema di forze che si trova già in equilibrio, è possibile salire le scale e ottenere, visto di profilo, un movimento nel piano *(x,z)*.

Un esempio semplice di questa applicazione della fisica al campo di gioco si trova nel gioco Donkey Kong sviluppato da Shigeru Miyamoto

per la Namco. Il protagonista del gioco, Mario, si muove in un contesto 2D formato dal piano *(x,z)* costituito da delle travature di un'impalcatura sorrette da scale a pioli. Mario cammina sulle travature ed usa le scalette per salire da una travatura all'altra. Può inoltre saltare sul posto o saltare giù da una travatura verso quella più in basso.
In questo gioco il campo è costituito dall'impalcatura e dalle scale che forniscono la reazione vincolare per i movimenti di Mario.
Durante il gioco vale sempre la legge di gravità, per la quale sia Mario che gli altri elementi del gioco (barili e fiammelle) sono soggetti ad una forza proporzionale alla loro massa.

Elementi mobili del campo

Alcuni elementi del campo di gioco possono anche risultare mobili e, in questo caso, la distinzione tra giocatori ed elementi mobili può diventare sottile. Nel gioco Donkey Kong, al secondo livello, troviamo dei nastri trasportatori. Se Mario sale sui nastri viene trascinato dal loro moto, quindi i nastri rappresentano dei vincoli del sistema fisico che devono essere considerati per calcolare la traiettoria del moto di Mario, per questo motivo si dovrebbero considerare come elementi mobili del campo e non proprio come altri giocatori.

La distinzione comunque è sottile e soprattutto è solo formale: che si tratti di giocatori o di parti mobili, le traiettorie del moto devono sempre tenere presente tutte le componenti del sistema fisico, che nel nostro caso è costituito dal campo, dalle parti mobili e dai giocatori.

Giocatori

Nella maggior parte dei giochi arcade, il giocatore umano, ovvero chi sta giocando al videogioco, è rappresentato all'interno del gioco da un altro giocatore a cui ci riferiamo come *il protagonista*. Con riferimento ai giochi visti fin'ora, il giocatore è rappresentato da Pacman nel videogioco Pacman e da Mario nel videogioco Donkey Kong. Oltre al protagonista ci possono essere altri giocatori, per esempio nel Pacman ci sono i quattro fantasmini Blinky, Pinky, Inky e Clyde, mentre in Donkey Kong ci sono il gorilla, appunto Donkey Kong, e la fidanzata di Mario, Pauline.

Durante il gioco Donkey Kong lancia dei barili che rotolano verso Mario

e, una volta raggiunta la base dell'impalcatura, si incendiano, si animano di una vita propria, risalgono l'impalcatura e ostacolano il protagonista nella sua missione.

Anche in questo caso **la distinzione tra giocatori e parti mobili del gioco non è così diretta**. I barili scendono verso Mario animati dalla sola legge di gravità, senza una propria intelligenza, per questo motivo possiamo considerarli come parti mobili del campo.

Dopo essere caduti ed essersi incendiati, invece, i barili acquisiscono una propria intelligenza, sono infatti guidati da un algoritmo che non si limita ad applicare le leggi della meccanica: per questo motivo possiamo considerarli come dei giocatori.

Le regole del gioco

Come abbiamo visto il gioco deve essere governato da un insieme di leggi fisiche. Queste leggi possono anche non corrispondere alle leggi fisiche naturali. Un gioco potrebbe prevedere un salto nell'iperspazio oppure un sistema antigravitazionale, l'importante è che esista una *meccanica* che permetta al giocatore di comprendere le dinamiche del gioco. Nel seguito ci riferiremo sempre a queste leggi indicandole come la *fisica* del gioco, anche se appunto in alcuni casi potrebbero essere leggi non naturali.

La fisica del gioco è quindi l'ambiente base perché possa svolgersi la trama, ma per rendere il gioco unico e divertente è necessario dotarlo di regole proprie. Per fare un controesempio, proviamo ad immaginare il gioco degli scacchi dotato delle regole di movimento di ogni pezzo, ma privo delle regole fondamentali che determinano i turni di gioco e i meccanismi di cattura dei pezzi: non sarebbe affatto possibile giocare! Da questo esempio capiamo che la prima regola è il concetto di turno. Per ogni giocatore deve essere previsto un turno di gioco durante il quale egli può eseguire la sua mossa. L'equilibrio tra le durate dei turni di gioco è un elemento chiave della giocabilità di ogni gioco e videogioco.

Per rimanere nel tema degli scacchi, immaginiamo cosa succederebbe se per ogni mossa del giocatore bianco, il nero potesse eseguirne due. Questa situazione coincide con l'idea di un turno lungo il doppio. Ebbene, il nero vincerebbe in poche mosse, e non sarebbe possibile

avere delle partire equilibrate.
Allo stesso modo, proviamo ad immaginare come sarebbe il gioco del Pacman, se nell'arco di tempo concesso al protagonista per cambiare direzione, i fantasmini potessero cambiarla più volte.

La durata del turno è quindi molto importante. Mentre si gioca si può avere l'impressione che non esita un turno, ma che al contrario le istruzioni del giocatore umano, vengano recepite in tempo reale dal videogioco. Ovviamente non può essere così. Le azioni del giocatore sono inviate al videogioco per mezzo di periferiche di input, nei casi più semplici la testiera o un Joystick.

La lettura dei dati in ingresso dalle periferiche richiede del tempo macchina così come richiede del tempo macchina l'esecuzione degli algoritmi che implementano la logica degli altri giocatori. Il videogioco deve riservare un intervallo temporale per l'acquisizione e l'elaborazione dell'input proveniente dal giocatore e un intervallo temporale per l'elaborazione della logica degli altri giocatori.

Nel seguito vedremo come viene implementata la durata del turno nei giochi che vengono presentati qui.
Dopo il concetto di turno, che bene o male possiamo vedere come una regola universale, vengono tutte le regole che, insieme alla definizione del campo di gioco, rendono ogni videogioco unico.

Continuando ad analizzare il gioco del Pacman, troviamo come prima regola che il Pacman aumenta il proprio punteggio (**score**) passando sopra le pastiglie che sono disseminate lungo il labirinto, cioè mangiandole. Una seconda regola, è che Pacman perde una vita se viene in contatto con uno dei fantasmini. Questa regola ci porta di nuovo alla fisica del videogioco: la **collision detection** o verifica di collisione, è un elemento chiave e complesso di diversi videogiochi.

Nel caso del Pacman il problema si riduce a verificare se i due centri geometrici del fantasma e di Pacman sono all'interno della stessa cella: in tal caso il fantasma ha catturato Pacman. Come ci si può immaginare, il problema di verificare le collisioni può diventare più insidioso nei giochi 3D, infatti in quel caso non è sufficiente verificare che effettivamente due giocatori vengano in contatto, ma bisogna anche renderlo chiaro ed evidente nella rappresentazione in 2D del gioco se si vuole che il giocatore abbia una percezione corretta di come

evitare (o al contrario indurre) un contatto. Ancora più complessa è la verifica di collisione quando questa non si limita a due punti, come nel caso della verifica che i centri dei personaggi del Pacman non occupino la stessa cella, ma riguarda un corpo articolato come nei combattimenti di arti marziali che troviamo ad esempio in Street Fighters ideato e prodotto da Takashi Nishiyama e Hiroshi Mateumoto per la Capcom.

Architettura model, view e controller (MVC)

Lo sviluppo di qualsiasi software partendo da zero (ricordiamoci che questo è sempre il motto della Scuola Sisini) richiede sempre uno sforzo. Scrivere un gioco partendo da zero porta molte soddisfazioni, ma anche diverse difficoltà. In questo paragrafo proponiamo un approccio allo sviluppo che ne semplifica sia l'analisi che l'implementazione. Si tratta di un *design pattern* sviluppato negli anni 80' e utilizzato poi in diversi contesti. Il suo nome è appunto "**model, view, controller**".

l'idea è quella che per un software interattivo, cioè che abbia una relazione diretta con l'utenza finale, come appunto i videogame, si possano individuare tre macro aree: il **modello**, cioè la realtà che si vuole descrivere, la sua **visualizzazione**, cioè la soluzione grafica con cui si vuole mostrare il modello all'utenza umana ed infine il **controllo** cioè la procedura che assicura che vengano rispettate le regole del gioco tra le quali i turni dei giocatori.
Diversi programmatori e scuole di programmazione hanno implementato il MVC in modi diversi. In questo testo, Scuola Sisini propone l'implementazione che ha scelto per i propri videogiochi, ma non sosteniamo che sia migliore o peggiore delle implementazioni proposte da altri.

PROGRAMMAZIONE DI UN GIOCO
PROGRAMMA

Cosa cambia da un videogioco a un gioco programma? Entrambi sono giochi e entrambi servono per divertirsi, la differenza è che nel videogioco il giocatore controlla il protagonista con i propri muscoli, mentre nel gioco programma lo fa con il pensiero. Parliamo di telecinesi? No, non ancora. Se diciamo che nel gioco programma il giocatore controlla il protagonista con il pensiero è perché durante lo svolgimento della partita egli non può intervenire quindi i propri riflessi e le reazioni muscolari non prendono parte al gioco. Ciò che prende parte invece è la strategie pensata dal giocatore che deve essere abbastanza buona per permettergli di vincere senza il suo intervento durante la partita.

In guerra si dice che un bravo comandante deve sempre avere un piano, ma che deve saperlo modificare al momento opportuno al variare delle circostanze. In un gioco-programma bisogna essere più bravi di un bravo comandante perché **una volta iniziata la partita non è più possibile cambiare il proprio piano di azione**.

Il gioco programma consiste proprio in questo: nel definire un piano di azione e nel vederlo operativo. Qualcuno però deve preparare il gioco programma per permettere a qualcun altro di divertirsi. È come per i cruciverba, qualcuno li prepara perché qualcun altro si scervelli. In realtà, i gioco programmi che presentiamo in questo libro hanno una natura simmetrica, e possono essere usati anche per il confronto diretto tra due giocatori. Vedremo come sfruttare questa opportunità quando analizzeremo i giochi uno per uno.
Nel prossimo paragrafo vediamo come nasce l'idea di un gioco programma e come da questa si passi all'implementazione.

La sfida

La sfida è il sale del gioco. Che si tratti di scacchi, tennis, calcio, poker e via dicendo, quando si gioca si accetta una sfida. Nel gioco d'azzardo si accetta una sfida confidando in capacità soprannaturali di gestire il fato, nei giochi veri, si sfidano le proprie capacità reali, sia fisiche che mentali, per vincere una partita contro l'avversario.

Perché un gioco risulti divertente, la sfida deve essere chiara, e devono essere chiare le capacità che vengono messe sul piatto. Per scrivere un gioco programma, il primo passo è proprio decidere quale sia la sfida.

chiarezza):
<27><91><49><48><59><49><53><72><65>
che corrisponde alla stringa C:
"\x1b[10;15HA"

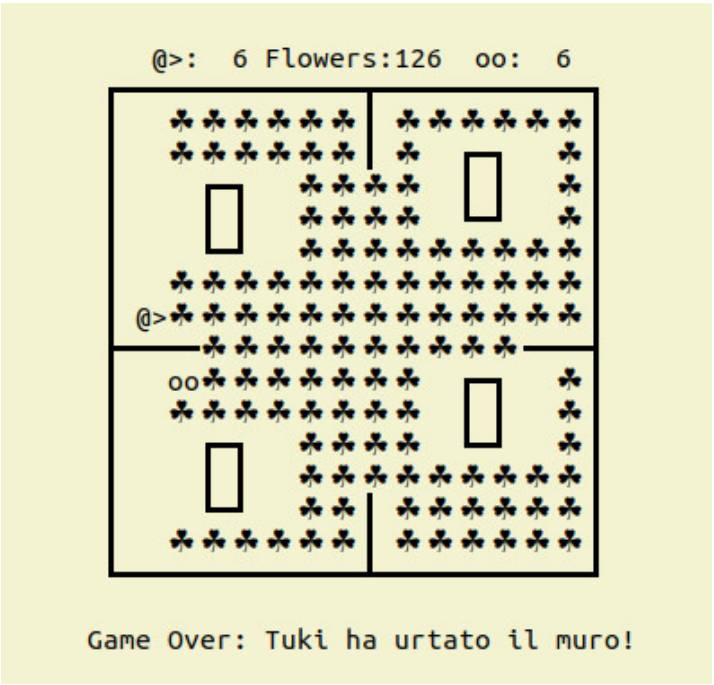


Fig. 3 - La figura mostra la griglia attuale di Tuki 1 resa monocromatica e con una diversa rappresentazione di Giuli.

Organizzazione dei file e distribuzione del pacchetto Tuki 1

L'organizzazione dei file cambia leggermente da gioco a gioco, comunque in tutti e cinque i gioco programmi è funzionale mantenere la funzione `turno_tuki` in un file separato. L'obiettivo principale dei giochi è infatti quello di fornire un prodotto che permetta di concentrarsi sullo sviluppo dell'algoritmo per muovere il Tuki. Un gioco programma scritto in C richiede però che l'implementazione utente della funzione `turno_tuki` venga compilata e linkata al file oggetto contenente il **model**, il **viewer** il **controller**, quindi al giocatore è richiesto di compilare il gioco con le sue modifiche. La strategia è quella di preparare i file oggetto di tutti i moduli (file) che non devono essere modificati dal giocatore lasciando in formato testo solo il file

`turno_tuki.c`.
Tutti i giochi della Scuola Sisini sono open source, la scelta di distribuire il gioco fornendo i file oggetto non è per impedire che il codice venga copiato, ma per permettere al giocatore di non essere tentato ne distratto da altro che non sia la **sfida**. Il giocatore potrà quindi concentrarsi solo sull'implementazione e poi compilare e linkare il proprio sorgente al resto del programma.
Questo aspetto è davvero interessante, infatti il giocatore non si limita a scrivere un po' di codice *script* perché venga eseguito in una sanbox, ma genera un vero eseguibile, raggiungendo così lo scopo auto-didattico del gioco.
I file sorgente di Tuki 1 sono i seguenti:

- tuki1_controller.c
- tuki1_model.c
- tuki1_view.c
- turno_giuli.c
- **turno_tuki.c**

dove l'ultimo è evidenziato in grassetto per sottolineare che è il file predisposto per essere modificato dall'utente. I file header invece sono:

- tuki1_model.h
- tuki1_view.h
- giocatore.h

Per generare la distribuzione del gioco, prima si compilano i file *immutabili* usando l'opzione -c che consente di compilare senza il linking:
`gcc -c tuki1_view.c tuki1_controller.c tuki1_model.c turno_giuli.c`
la compilazione genera i file:

- tuki1_controller.o
- tuki1_model.o
- tuki1_view.o
- turno_giuli.o

che vengono inseriti nel pacchetto del gioco. A questo punto si crea una cartella di distribuzione con i seguenti file:

- tuki1_controller.o

- tuki1_model.o
- tuki1_view.o
- turno_giuli.o
- **turno_tuki.c**
- **giocatore.h**

Il giocatore edita il file `turno_tuki.c`.
e procede alla compilazione con il seguente comando:
`gcc -o tuki1.game tuki1_model.o tuki1_view.o tuki1_controller.o turno_giuli.o turno_tuki.c`
Il risultato della compilazione è l'eseguile **tuki1.game** che vine lanciato per osservare il comportamento dell'algoritmo appena implementato:
`./tuki1.game`

Sfida alla pari!

La distribuzione del gioco appenda presentata è una sfida del giocatore finale verso la versione *ufficiale* di Giuli, cioè lo sfidante che implementa l'algoritmo da sfidare. Però è possibile anche pensare ad una distribuzione più paritetica del gioco, in cui si distribuiscono i sorgenti di entrambi turno_tuki.c e turno_giuli.c in modo che due utenti possano sfidarsi.

Fig. 4 - La figura mostra l'immagine originale con cui fu prodotto Tuki 1 nel 2017.

Il codice completo

tuki1_controller.c

```
/*_____________________________________
*|
*| tuki1_controller.c
*/

#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "tuki1_model.h"
#include "tuki1_view.h"

#define MAJOR    0
#define MINOR    1
#define RELEASE  1

/* VARIABILI GLOBALI  */
Giocatore giuli_player, tuki_player;
int t_or_g_1_or_2; //Giocatore attivo

/* PROTOTIPI  */
int turno_tuki();
int turno_giuli();

int main(int argc, char **argv)
{
  /* Disabilita la tasiera */
  vista_init();

  strcpy(tuki_player.name, "tuki");
  strcpy(giuli_player.name, "giuli");
  setlocale(LC_CTYPE,"");

  /* Prepara il campo */
  if(modello_prepara_campo() != 0) exit(1);
  modello_giocatore(1,1,TUTF,&tuki_player);
  modello_giocatore(1,14,GUTF,&giuli_player);

  /* Presenta gioco */
  vista_presentazione();

  /* Visulalizza campo da gioco */
  vista_stampa_campo(modello_campo());

  char c=0;
  int r=0;

  while(c != 'q')
    {
      direzione d;
      int xb,yb;
      read(STDIN_FILENO,&c,1);

      t_or_g_1_or_2 = 1;
      xb = tuki_player.posX;
      yb = tuki_player.posY;

      d = turno_tuki();

      r = modello_passo(&tuki_player,d);

      vista_stampa_giocatore(&tuki_player);

      if(r == 2)
        {
          vista_gameover("Game Over: Tuki ha urtato il muro!");
          break;
        }
```

```
      if(r == 1)
        {
          if(tuki_player.score>=giuli_player.score)
            vista_gameover("Vince Tuki!");
          else
            vista_gameover("Vince Giuli!");
          break;
        }

      vista_stampa_sfondo(yb,xb,modello_campo());

      t_or_g_1_or_2 = 2;
      xb = giuli_player.posX;
      yb = giuli_player.posY;
      d = turno_giuli();

      r = modello_passo(&giuli_player,d);

      vista_stampa_giocatore(&giuli_player);
      vista_stampa_sfondo(yb,xb,modello_campo());

      if(r == 2)
        {
          vista_gameover("Game Over: Giuli ha urtato il muro!");
          break;
        }

      if(r == 1)
        {
          if(tuki_player.score <= giuli_player.score)
            vista_gameover("Vince Giuli!");
          else
            vista_gameover("Vince Tuki!");
          break;
        }

      vista_punteggio
      (tuki_player.score, giuli_player.score,
       modello_numero_fiori());
    }

  modello_libera();
}

/*_____________________________________________________________
 *|
 *| Usata dal giocatore per leggere il campo da gioco
 */
elemento_campo controller_leggi_elemento_campo(direzione d)
{
  Giocatore *pl = t_or_g_1_or_2 == 1?&tuki_player:&giuli_player;
  return(modello_elemento_in_campo(d,pl));
}
```

tuki1_model.c

```
/*_________________________
*|
*| tuki1_model.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "tuki1_model.h"

elemento_campo ** bf;
```

```
Giocatore tuki_player;

Giocatore giuli_player;

int flowers_number;

void aggiungi_ostacolo_quadrato(int xc,int yc,int l)
{
  if(l<2) return;

  int xl = 0+xc-l/2;
  int xr = xl+l-1;
  int yt = 0+yc-l/2;
  int yb = yc+l-1;

  if(xl >= 0 && xr < CAMPO_LARGHEZZA
   && yt >= 0 && yb < CAMPO_LARGHEZZA)
    {
      bf[yt][xl] = TLC;
      bf[yb][xl] = BLC;
      bf[yb][xr] = BRC;
      bf[yt][xr] = TRC;

      for(int i=xl+1;i<xr;i++)bf[yt][i]=HLN;
      for(int i=xl+1;i<xr;i++)bf[yb][i]=HLN;
      for(int i=yt+1;i<yb;i++)bf[i][xl]=VLN;
      for(int i=yt+1;i<yb;i++)bf[i][xr]=VLN;

      /* Elimina i fiori sotto l'ostacolo */
      for(int i=xl+1;i<xr;i++)
        {
          for(int j=yt+1;j<yb;j++)
            {
              bf[j][i] = 0;
            }
        }

    for(int i=yt;i<=yb;i++)
      bf[i][xr+1] = 0;

    for(int i=yt;i<=yb;i++)
    bf[i][xl-1] = 0;
  }
}

void modello_libera()
{
  for(int in=0;in<CAMPO_ALTEZZA;in++)
    {
      free(*(bf+in));
    }
  free(bf);
}

int modello_prepara_campo()
{
  int level = 1;
  int l = 1, i;
  int h = CAMPO_ALTEZZA;
  int w = CAMPO_LARGHEZZA;

  /* Riserva la memoria per gli elementi del campo */
  bf = (elemento_campo**)malloc(h*sizeof(elemento_campo*));
  if(!bf) return 1;
  for(int in=0;in<h;in++)
    {
      *(bf+in) = malloc(w*sizeof(elemento_campo));
      if(!(*(bf+in))) return 1;
    }

  /*Riempie il campo di trifogli*/
```

```
for(int i=1;i<h-1;i+=1)
  {
    for(int j=1;j<w-1;j+=1)
      {
        bf[i][j] = FUTF;
      }
  }

/* Disegna il labirinto */

/* Angolo alto a sinistra*/
bf[0][0] = TLC;

/* Muro sopra*/
for(int j=1;j<CAMPO_LARGHEZZA-1;j++)
  {
    bf[0][j] = HLN;
  }

/* Angolo alto a destra*/
bf[0][CAMPO_LARGHEZZA-1] = TRC;

/* Muro destro*/
for(i=1;i<CAMPO_ALTEZZA-1;i++)
  {
    bf[i][CAMPO_LARGHEZZA-1] = VLN;

  }

/* Angolo basso destra*/
bf[CAMPO_ALTEZZA-1][CAMPO_LARGHEZZA-1]=BRC;

/*Muro basso*/
for(i=CAMPO_LARGHEZZA-2;i>0;i--)
  {
    bf[CAMPO_ALTEZZA-1][i]=HLN;
  }

/*Angolo basso sinistra*/
bf[CAMPO_ALTEZZA-1][0]=BLC;

/*Muro sinistra*/
for(i=CAMPO_ALTEZZA-2;i>0;i--)
  {
    bf[i][0]=VLN;
  }

/* Giunti e tramezze */
bf[CAMPO_ALTEZZA-1][CAMPO_LARGHEZZA/2]=TT;
bf[CAMPO_ALTEZZA-2][CAMPO_LARGHEZZA/2]=VLN;
bf[CAMPO_ALTEZZA-3][CAMPO_LARGHEZZA/2]=VLN;

bf[0][CAMPO_LARGHEZZA/2]=TB;
bf[1][CAMPO_LARGHEZZA/2]=VLN;
bf[2][CAMPO_LARGHEZZA/2]=VLN;

bf[CAMPO_ALTEZZA/2][0]=TL;
bf[CAMPO_ALTEZZA/2][1]=HLN;
bf[CAMPO_ALTEZZA/2][2]=HLN;

bf[CAMPO_ALTEZZA/2][CAMPO_LARGHEZZA-1]=TR;
bf[CAMPO_ALTEZZA/2][CAMPO_LARGHEZZA-2]=HLN;
bf[CAMPO_ALTEZZA/2][CAMPO_LARGHEZZA-3]=HLN;

switch (level){
case 0:
  break;

case 1:
  srand(time(NULL));
  int gg=rand()%3;
  int gh=rand()%2;
```

```
    aggiungi_ostacolo_quadrato
    (CAMPO_LARGHEZZA/(4)-1+gg,CAMPO_ALTEZZA/4+gh,2);

    gg=rand()%3;
    gh=rand()%2;

    aggiungi_ostacolo_quadrato
    (CAMPO_LARGHEZZA*(3.0/4.0)+1-gg,CAMPO_ALTEZZA/4-gh,2);

    gg=rand()%3;
    gh=rand()%2;

    aggiungi_ostacolo_quadrato
    (CAMPO_LARGHEZZA*(3.0/4.0)+1-gg,CAMPO_ALTEZZA*(3.0/4.0)-1-gh,2);

    gg=rand()%3;
    gh=rand()%2;

    aggiungi_ostacolo_quadrato
    (CAMPO_LARGHEZZA*(1./4.0)-1+gg,CAMPO_ALTEZZA*(3.0/4.0)-1+gh,2);

    break;
  }

  /* Conta i fiori dopo la preparazione del campo*/
  for(int i=0;i<h;i+=1)
    for(int j=0;j<w;j+=1){
      if(bf[i][j]==FUTF) flowers_number++;
    }

  return 0;
}

elemento_campo ** modello_campo()
{
  return bf;
}

elemento_campo modello_elemento_in_campo(direzione d,Giocatore *ap)
{
  int x,y;
  x = ap->posX;
  y = ap->posY;
  x=x+1*(d==DESTRA);
  x=x-1*(d==SINISTRA);
  y=y-1*(d==ALTO);
  y=y+1*(d==BASSO);

  if(x<0 || x>=CAMPO_LARGHEZZA || y<0 || y>=CAMPO_ALTEZZA)
  {
    return -1;
  }
  return bf[y][x];
}

void  modello_ruota_giocatore(direzione d, Giocatore* player)
{
  switch(d)
    {
    case ALTO:
      {
        player->dir=ALTO;
        break;
      }
    case DESTRA:
      {
        player->dir=DESTRA;
        break;
      }
    case BASSO:
      {
```

```
        player->dir=BASSO;
        break;
      }
    case SINISTRA:
      {
        player->dir=SINISTRA;
        break;
      }
    case IN:
      {
        player->dir=IN;
        break;
      }
    }
}

char_type tipo_carattere(elemento_campo code)
{
  if(code==TLC||code==TRC||code==BRC
  ||code==BLC||code==HLN||code==VLN||code==TT||code==TB)
    {
      return WALL;
    }

  if(code==0){
    return EMPTY;
  }

  if(code==FUTF){
    return FLOWER;
  }
  if(code==GUTF){
    return GIULI;
  }
  if(code==TUTF){
    return TUKI;
  }

  if(code==10210||code==9830){
    return ITEM;
  }
  return UNKNOWN;
}


/*_____________________________________________________
 *|
 *| Esegue un passo nella direzione indicata
 *| torna 2 se urta le pareti
 *| torna 1 se ha finito i trifogli
 *| toena 0 se non è successo niente
 */
int modello_passo(Giocatore *player, direzione d)
{
  int x_old=player->posX;
  int y_old=player->posY;

  char_type ct=UNKNOWN;

  switch(d)
    {
    case IN:
      return 0;
      break;

    case SINISTRA:
      ct=tipo_carattere(bf[y_old][x_old-1]);
      if(ct!=WALL&&x_old>0)  player->posX--;else return 2;
      break;
    case DESTRA:
      ct=tipo_carattere(bf[y_old][x_old+1]);
      if(ct!=WALL&&x_old<(CAMPO_LARGHEZZA-1))
```

```
        player->posX++;
      else return 2;
      break;
    case ALTO:
      ct=tipo_carattere(bf[y_old-1][x_old]);
      if(ct!=WALL&&y_old>0)
       player->posY--;
      else return 2;
    break;
    case BASSO:
      ct=tipo_carattere(bf[y_old+1][x_old]);
      if(ct!=WALL&&y_old<CAMPO_ALTEZZA-1)
       player->posY++;
      else return 2;
      break;
    }

  /* Verifica se mangia un fiore*/
  if(ct==FLOWER){
    player->score+=1;
    flowers_number--;
    bf[player->posY][player->posX]=0;
  }

  if(flowers_number==0)return 1;

  return 0;
}

/*_____________________________________________________________
 *|
 *| Permette al controller di controllare il giocatore
 */
void modello_giocatore
(int x, int y, elemento_campo e, Giocatore *p)
{
  p->posX = x;
  p->posY = y;
  p->UTF = e;

  if(bf[y][x] == FUTF)
    {
      flowers_number--;
      bf[y][x] = 0;
    }
}

int modello_numero_fiori()
{
  return flowers_number;
}
```

tuki1_view.c

```
/*_____________________________________________
 *|
 *| tuki1_view.c
 */

#include <ctype.h>
#include <locale.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <time.h>
```

```
#include <unistd.h>
#include "tuki1_model.h"

#define COFFSET(width) (cols-width)/2
#define ROFFSET(heigh) (rows-heigh)/2

struct termios orig_termios;
int rows,cols;

void die(const char *s) {
  write(STDOUT_FILENO, "\x1b[2J", 4);
  write(STDOUT_FILENO, "\x1b[H", 3);

  perror(s);
  exit(1);
}

int dimensioni_finestra(int *rows, int *cols)
{
  struct winsize ws;
  if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1 || ws.ws_col == 0)
    {
      return -1;
    } else
    {
      *cols = ws.ws_col;
      *rows = ws.ws_row;
      return 0;
    }
}

void terminale_cucinato()
{
  if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &orig_termios) == -1)
    die("tcsetattr");
}

void terminale_crudo()
{
  if (tcgetattr(STDIN_FILENO, &orig_termios) == -1)
    die("tcgetattr");

  atexit(terminale_cucinato);

  struct termios raw = orig_termios;
  raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
  raw.c_oflag &= ~(OPOST);
  raw.c_cflag |= (CS8);
  raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
  raw.c_cc[VMIN] = 0;
  raw.c_cc[VTIME] = 1;

  if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw) == -1)
    die("tcsetattr");
}

int vista_init()
{
  terminale_crudo();
  dimensioni_finestra(&rows, &cols);
  setlocale(LC_CTYPE, "");
}

char * get_str(elemento_campo e)
{
  if(e==TLC) return "\u250F\u2501";
  if(e==TRC) return "\u2513";
  if(e==BRC) return "\u251B";
  if(e==BLC) return "\u2517\u2501";
  if(e==HLN) return "\u2501\u2501";
  if(e==VLN) return "\u2503";
  if(e==TT) return "\u253b\u2501";
```

```
typedef enum  {SINISTRA,ALTO,DESTRA,BASSO,IN} direzione;
#define PLAYER
#endif

typedef enum {UL, UR, DR,DL,FC} angolo;

/*_____________________________________________________________________
 *|
 *| Il giocatore
 */
typedef struct  {
  int posX;
  int posY;
  direzione dir;
  int score;
  char name[10];
  int code;
  elemento_campo UTF;
} Giocatore;

enum chartype
{WALL,ITEM,FLOWER,UNKNOWN,GIULI,TUKI,EMPTY};

typedef enum chartype char_type;



/*_____________________________________________________________________
 *|
 *| Funzioni private
 */
void aggiungi_ostacolo_quadrato(int xc,int yc,int l);
char_type tipo_carattere(elemento_campo code);

/*_____________________________________________________________________
 *|
 *| Funzioni pubbliche
 */
void modello_giocatore(int x, int y, elemento_campo e, Giocatore *p);
void modello_libera();
elemento_campo modello_elemento_in_campo(direzione d,Giocatore *ap);
int modello_numero_fiori();
int modello_passo(Giocatore*,direzione);
elemento_campo ** modello_campo();
int modello_prepara_campo();
void modello_ruota_giocatore(direzione d, Giocatore * p);
```

tuki1_view.h

```
/*_________________________
*|
*| tuki1_view.h
*/

void vista_gameover(char * message);
int vista_init();
void vista_punteggio(int t_score, int g_score, int flowers);
void vista_presentazione();
int vista_stampa_campo(elemento_campo **maze);
int vista_stampa_giocatore(Giocatore * p);
int vista_stampa_sfondo(int r, int c,elemento_campo **maze);
```

giocatore.h

```
/*_____________________________________________________________________
 *|
 *| giocatore.h
 */

#ifndef PLAYER
typedef enum  {SINISTRA,ALTO
               ,DESTRA,BASSO,IN} direzione;
#define PLAYER
#endif

/*_____________________________________________________
 *|
 *| Elementi del campo di gioco
 */
enum cmp{
  GUTF =9787, //Giuli
  TUTF =0040, //Tuki
  TLC = 9487, //Angolo sinistro alto
  TRC = 9491, //Angolo destro alto
  BRC = 9499, //Angolo destro basso
  BLC = 9495, //Angolo sinistro basso
  HLN = 9473, //Muro orizzontale
  VLN = 9475, //Muro verticale
  TT =  9531, //Giunzione alto
  TB =  9523, //Giunzione basso
  TL =  9507, //Giunzione sisnistra
  TR =  9515, //Giunzione destra
  FUTF =9752  //Fiore (trifoglio)
};

typedef enum cmp elemento_campo;

/*_____________________________________________________________
 *|
 *| Legge il campo in corrispondenza della posizione/cella
 *| successiva a quella occupata dal giocarore, in direzione
 *|
 */
elemento_campo controller_leggi_elemento_campo( direzione d);

/*_____________________________________________________________
 *|
 *| Crea una pausa nell'esecuzione del gioco. Da usare per
 *| rallentare l'eseczione e osservarla meglio
 */
void delay(int milliseconds);
```

TUKI 2: SFIDA PER LADYB

La sfida

Tuki e Giuli sono su un prato dove al centro sorge un palo. LadyB, la loro amica si lascia sfuggire una mezza frase: «Che bello sarebbe poter salire fin' la sopra e vedere tutti i fiori del campo...» dice.
Tuki che fra i due è il meno romantico le ricorda che se vuole salire in cima al palo può farlo perché, come tutte le coccinelle, ha le ali per volare.
«Che sciocco che sei...» le risponde LadyB, «non sarebbe la stessa cosa volarci sopra. Io vorrei raggiungere la cima salendo una bella gradinata, come la principessa che sono» risponde altezzosa.
A quelle parole Giuli, che come Tuki è infatuato della bella coccinella, raccoglie dei tronchi di varia lunghezza per costruire una scalinata per LadyB. Tuki, sempre pronto sfottere l'amico, si sbellica dalle risate dicendo che lui non ha mai visto una scala fatta a quel modo. LadyB non sopporta quella sceneggiata troppo popolare, e gli dice che se si crede tanto superiore provi anche lui a costruirle una scala.
Tuki controvoglia raccoglie altrettanti tronchi e cerca di sistemarli in ordine, ma questi sono alti e lui dal basso non riesce a capire bene come li sta ordinando. Alla fine sul prato spicca il palo più alto e due schiere, una a destra e una a sinistra di pali: è qui che comincia la sfida. I due saltano sui pali e da lì finalmente riescono a percepirne correttamente l'altezza: non ne hanno sistemato giusto neanche uno! Che fare? Rimediare, provare a ordinarli e a preparare la scala per la bella coccinella prima che la prepari l'altro.

Questa volta, l'idea alla base del gioco è quella di sperimentare una procedura per ordinare un insieme di oggetti. Per gioco o per passione, fin' da bambini, abbiamo ordinato le matite nell'astuccio in base alla lunghezza (il rosso è sempre il colore più corto!), i libri nella scrivania in base ad una scala cromatica e tante altre collezioni di oggetti che la componente ossessiva che ognuno si porta dentro lo costringe ad ordinare secondo un certa relazione d'ordine.
Se si tratta di ordinare i chiodi nella cassetta degli attrezzi, le viti, e via dicendo, comunque oggetto presenti in numero molto limitato, normalmente si agisce ordinandoli senza ricorrere ad una procedura schematica. Spesso si prende in mano la prima cosa che capita e la si posiziona dove *sembra* che vada, poi eventualmente sé ne corregge la

posizione al prossimo giro. Altre volte si da un'occhiata di insieme poi si posiziona ogni oggetto nel giusto ordine. In tutti i casi quello che si fa è di eseguire una procedura di ordinamento.
Lo scopo di questa sfida è quello di indurre il giocatore a ragionare sul fatto che anche l'ordinamento ha bisogno di una procedura e questa è descrivibile per mezzo di un algoritmo.

Per i programmatori più esperti che già conoscono gli algoritmo di ordinamento può essere una prova per vedere se riescono ad applicarlo anche nel contesto del gioco, mentre per chi si sta affacciando all'informatica è l'occasione per farsi un'idea di come i problemi della vita ordinaria vengano analizzati e risolti in ambito informatico. Insomma: un classico.
Nella versione originale fornita dal codice di questo libro, Giuli implementa un algoritmo si **bubble sort**.

Analisi

Tuki 2 presenta l'architettura MVC come Tuki 1. Il **model** in questo caso è pero piuttosto diverso dal gioco visto precedentemente. Via via che analizzeremo i cinque giochi coglieremo l'occasione per analizzare aspetti diversi della programmazione sia di un normale videogioco che nella fattispecie di un gioco-programma. Ognuno di essi infatti ha delle caratteristiche peculiari che lo rendono adatto per intavolare delle specifiche sessioni di approfondimento. Questo per dire che se nel primo esempio vi è parso che non sia stato spiegato proprio tutto, è perché quel che manca è spiegato strada facendo.

Per esempio, con questo secondo gioco, vedremo come si gestisce il ripristino dello sfondo dopo il passaggio di un personaggio e come vengono gestite le parti mobili del campo. Il problema del ripristino dello sfondo avremmo potuto affrontarlo anche nel gioco precedente ma era meno interessante, perché in quel caso, a causa delle regole del gioco e dello specifico campo usato, la cella occupata dal giocatore (sia esso Tuki o Giuli) viene sempre ripristinata con un carattere vuoto (uno spazio) dopo che il giocatore si è spostato per occupare un'altra celle. In questo esempio invece, vedremo che il ripristino dello sfondo (o background) è leggermente più complesso.

Fig. 5 - La figura mostra il progetto grafico alla base di Tuki 2.

Modello di Tuki 2

L'idea grafica del gioco è riportata nella figura sopra. Si tratta ancora di un campo bidimensionale che giace sul piano *(x,z)*. I vincoli del campo sono costituiti dal suolo e dai tronchi, questi, potendo essere spostati dai giocatori, rappresentano, per i motivi analizzati nel primo capitolo, delle parti mobili del campo.
Tirando le somme il modello deve rappresentare:

- due giocatori
- il terreno
- un tronco fisso
- 10+10 tronchi mobili

e le **regole** del gioco. Queste sono piuttosto semplici, del resto questo è il nostro stile. Il giocatore nel suo turno di gioco può svolgere queste azioni:

1. misurare la lunghezza del palo su cui si trova
2. marcare il palo su cui si trova
3. eliminare la marcatura dal paolo su cui si trova
4. scambiare tra loro i due pali che risultano marcati

Le azioni 1), 2) e 3) hanno luogo esattamente nel momento in cui il

giocatore le esegue, mentre per la 4) il giocatore deve attendere il termine del proprio turno, quando comunicherà al controller la direzione in cui vuole saltare.
I giocatori sono rappresentati con una struttura dati simile a quella usata nel Tuki 1:

```
typedef struct  {

  /*_____________________________________
   *|
   *| Posizione del giocatore
   */
  int posX;
  int posY;

   /*_____________________________________
   *|
   *| Posizione passata del giocatore
   */
  int posXold;
  int posYold;

   /*_____________________________________
   *|
   *| Direzione di spostamento
   *| scelta dal giocatore
   */
  direzione dir;

  /*_____________________________________
   *|
   *| Indice dei due pali marcati
   *| marcato[0] indice primo palo
   *| marcato[1] indice secondo palo
   *| -1->non marcato
   */
  int  marcato[2];

   /*_____________________________________
   *|
   *| Logico 1,0
   *| 1-> il giocatore vuole scambiare
   *| i pali marcati
   */
  int scambia;

  char name[10];

  utf8code UTF;
 }
  Gioca;
```

in cui devono essere memorizzate:

- la posizione corrente del giocatore e quella appena abbandonata
- quali pali il giocatore ha marcato per essere scambiati di posto
- la volontà di scambiare i pali

I pali da ordinare sono dieci per ciascun giocatore e sono rappresentati dagli array `int pali_giuli [NPOLES]` e `int pali_tuki [NPOLES]`, dove `NPOLES` è una macro che rappresenta appunto il numero dei pali.

La configurazione iniziale è scelta in modo pseudo-casuale da una funzione del **modello**:

```
 /*_____________________________________________
  *|
  *| Assegna l'altezza iniziale
  *| di tutti i pali del gioco
  */
void model_prepara_campo()
{

  for (int i=0;i<NPOLES;i++)
  {
    pali_giuli[i]=i+1;
    pali_tuki[i]=i+1;
  }
  /*make a mess*/
  srand(time(NULL));
  for (int i=0;i<50;i++)
    {
      for (int j=0;j<NPOLES-1;j++)
        {
          if(rand()%2){
            int t=pali_giuli[j];
            pali_giuli[j]=pali_giuli[j+1];
            pali_giuli[j+1]=t;
          }
        }
      for (int j=0;j<NPOLES-1;j++)
        {
          if(rand()%2){
            int t=pali_tuki[j];
            pali_tuki[j]=pali_tuki[j+1];
            pali_tuki[j+1]=t;
          }
        }
    }
}
```

La **fisica** del gioco è forse la parte più complessa. Ormai quando si programmano dei videogiochi moderni si utilizzano degli ambienti in cui il motore fisico è completamente nascosto e in questo modo il programmatore/game-designer può concentrarsi più sulla comunicazione artistica del gioco che sugli aspetti implementativi. In questi giochi invece non usiamo nient'altro che le librerie del linguaggio C, anzi ne facciamo anche un uso molto ridotto. Per questo motivo la fisica del modello è un onere che spetta al programmatore.
Come accennato poco sopra il modello offre la reazione vincolare dovuta al suolo e quella dovuta alla presenza dei pali. Le regole del gioco vogliono che il giocatore possa saltare da un palo al palo che gli è prossimo, quindi è compito del modello calcolare la cinematica del salto. Per spostarsi da un palo all'altro il giocatore deve per forza saltare (queste sono le regole). Il giocatore non è chiamato a gestire la dinamica del salto, semplicemente esprime la direzione verso la quale vuole saltare. Il modello basandosi sul dislivello tra i due pali deve calcolare la velocità iniziale con la quale il giocatore deve spiccare il

balzo per poter raggiungere il palo opposto.

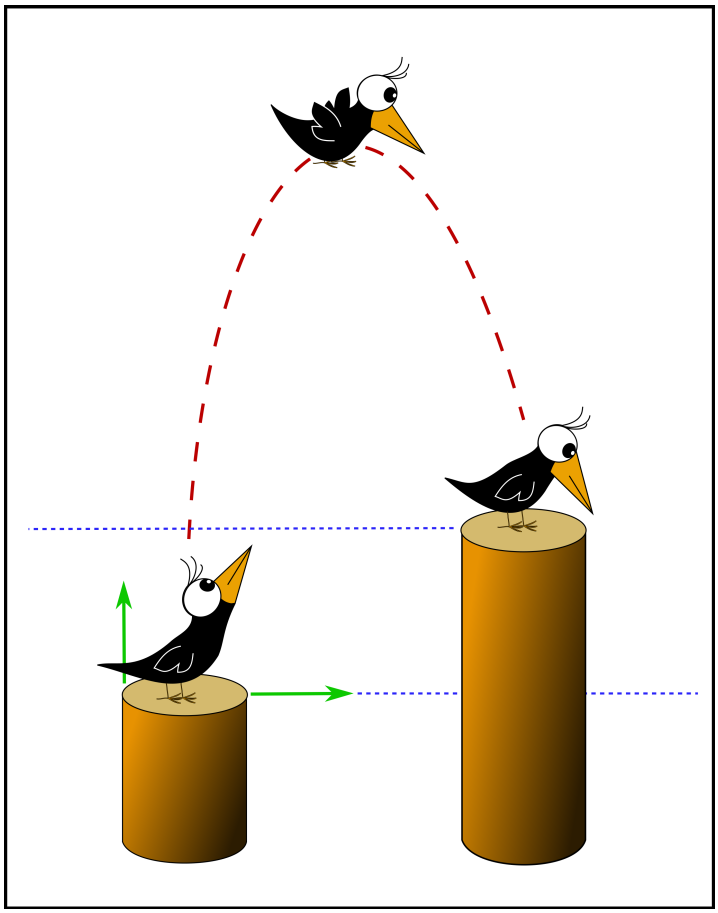


Fig. 6 - La figura mostra lo studio cinematico per il salto di Tuki nel gioco Tuki 2.

La cinematica ci insegna che, se trascuriamo l'attrito dell'aria, il moto che si compie lungo l'asse *z* durante un balzo è un **moto uniformemente vario** o uniformemente accelerato, mentre quello lungo l'asse *x* è un moto rettilineo uniforme. Da questi due moti nasce un traiettoria parabolica.
Per rendere il movimento credibile è importante che il moto sia realmente accelerato lungo la direzione *z*. Ovviamente i bit non sono soggetti alle leggi della dinamica, per dare la sensazione di un moto accelerato dobbiamo generare una sequenza di immagini in cui il centro del *corpo* soggetto alla forza, cioè Tuki o Giuli, cambi posizione secondo la corretta legge oraria. L'idea di base è piuttosto semplice: si identifica una variabile con il parametro **tempo** e si cicla su quella variabile. Ad

ogni iterazione si calcola la posizione del centro del corpo usando la variabile indice del ciclo come parametro temporale nella formula cinematica. Sempre all'interno dell'iterazione si provvede a visualizzare il corpo con la posizione calcolata, poi si conclude l'iterazione e si passa alla successiva.
Usando questo approccio si assume che le iterazioni avvengano ad intervalli costanti di tempo. Questo assunto potrebbe non essere sempre vero, specie se la CPU sta eseguendo qualche altro processo impegnativo. Comunque, nel contesto di questi gioco-programmi didattici, questo aspetto può essere calcolato, solo è importante rifletterci sopra. Se dovesse capitare di voler eseguire una rappresentazione fedele di un moto fisico, potrebbe essere meglio inserire il moto il un loop in cui il valore del tempo viene richiesto al sistema operativo anziché calcolato su una supposizione. In ogni modo, è bene conoscere i problemi, poi seguire la soluzione più adatta in base al contesto.
Nell'analizzare il frammento di codice qui sotto, bisogna tener presente che nei giochi che presentiamo, il piano di gioco è sempre indicato con le variabili X e Y, quindi nel caso in questione, rispetto al modello fisico, bisogna pensare alla variabile Y (o posY) come all'asse *z*.

```
/*_____________________________________________________________
 *|
 *| Calcola la traiettoria parabolica del salto e chiama
 *| la funzione di stampa per mostrare l'intero salto
 */
void salta(Gioca * p)
{
  /*Coordinate monitor*/
  int xo=p->posXold;
  int yo=p->posYold;
  int x=p->posX;
  int yf=p->posY;

  /*Coordinate fisiche*/
  float a=x;
  float ao=xo;
  float bf=BFHEIGHT-yf;
  float bo=BFHEIGHT-yo;
  float b=bo;

  float dt=0.005;/*secondo*/
  float t=0;
  float g=9.81;/*m/s^2*/
  float vy0;//m/s
  float vb;
  int ax=xo;
  int my,mb=-1;

  view_cancella_traccia_giocatore(p);

  /* Stabilisce la velocità iniziale del salto*/
  if((bf-bo)<0) vy0 = 3;
  else if((bf-bo) == 0)
    {
      vy0 = 5;
```

```
    }
  else
    {
      vy0 = sqrt(2*g*(bf-bo)*1.5);
    }
  while(!(ax==a&&(int)b==bf))
    {
      float _b = b;
      vb = vy0-g*(t);


      /* Moto uniformemente vario*/
      b = bo+vy0*t-0.5*g*(t*t);

      my = (int)(BFHEIGHT-b);
      t += dt;
      if(b>0 && b<BFHEIGHT)
        {

          if(vb>0)
            {
              fflush(stdout);
              ax = xo;
            }
          else
            {
              ax = a;
            }

          model_assegna_coordinate(ax, my, p);
          ritardo(1);
          mb = my;
        }
    }
  ritardo(1);
}
```

Un passaggio degno di nota è quello che segue il commento:

```
  /* Stabilisce la velocità iniziale del salto*/
```

Come si vede la velocità iniziale con cui compiere il salto è valutata in base al fatto che il palo obiettivo sia più in basso o più in alto del palo di partenza. Se il palo obiettivo è più in basso, allora, considerando che i due pali sono praticamente contigui, si potrebbe anche saltare con velocità iniziale nulla, semplicemente facendo un passo nel vuoto, e comunque si raggiungerebbe l'obiettivo. Se invece l'obiettivo è più in alto, allora bisogna che la velocità iniziale (quella con cui si lascia la terra) sia almeno sufficiente a raggiungere l'altezza del palo. Comunque, per rendere il movimento più gradevole, anche quando i giocatori saltano verso pali obiettivo più bassi del palo di partenza, si assegna una velocità iniziale non nulla.

Sia i pali che il suolo devono fornire la reazione vincolare per sostenere i giocatori. Nei giochi presentati qui, il modello è molto semplice: o c'è reazione o non c'è reazione. Con questa semplificazione, la reazione vincolare può essere modellata semplicemente impedendo al valore

della ordinata Y (o asse *z*) di assumere valori minori di una certa soglia che sono il livello del suolo o l'altezza del palo su cui si trova il giocatore.

Controllo Tuki 2

Rispetto a quanto visto in Tuki 1, in questo gioco il **controller** ha due compiti in più. Il primo è piuttosto semplice ed è di controllare l'argomento passato alla linea di comando, infatti linciando l'eseguibile e si deve esplicitare `-p` per iniziare una sessione di gioco. Il secondo invece è di gestire la scelta tra gioco interattivo e gioco programma. Tuki 2 infatti mostra come un gioco programma possa trasformarsi in un videogioco classico con estrema semplicità: vediamo come.
Analizzando il **controller** di Tuki 1 si è visto che la routine principale del gioco si occupa di chiamare alternativamente la funzione `turno_tuki()` e `turno_giuli()`. L'idea per questo gioco è stata quella di mostrare a pieno la modularità dell'architettura MVC e dell'idea del gioco programma in un colpo solo.
Abbiamo già detto che la funzione `turno_`*`giocatore`*`()` rappresenta la coppia giocatore/joystick e che il joystick è collegato al gioco per mezzo del **controller**. Per consentire al giocatore di giocare direttamente, quindi di trasformare il gioco programma in un videogioco, bisogna implementare nella funzione `turno_tuki()` la lettura del segnale del joystick, mentre la logica algoritmica sarà già in capo al giocatore e per quella non si deve fare nulla. Sempre per tenere bassa l'asticella e fare le cose inizialmente semplici, sostituiamo il jystick con la tastiera, in questo modo il compito della funzione si riduce alla lettura diretta della tastiera:

```
direzione turno_interattivo()
{
  direzione dir = IN;

  char c;

  read(STDIN_FILENO, &c, 1);

  switch(c)
    {
    case 'j':
      dir=SINISTRA;
       return dir;
      break;

    case 'l':
      dir=DESTRA;
       return dir;
      break;

    case 'i':
```

```
  return dist;
}


void scambia_pali()
{
  if(t_or_g_1_or_2 == 2)
    {
      giuli_gioca.scambia = 1;
    }
  else
    {
      tuki_gioca.scambia = 1;
    }
}

void _scambia_pali()
{
  fflush(stdout);
  Gioca* p;
  int *pole;
  char side;

  if(t_or_g_1_or_2 == 2)
    {
      giuli_gioca.scambia = 0;
      p = &giuli_gioca;
      side = 'R';
      pole = pali_giuli;
    }
  else
    {
      tuki_gioca.scambia = 0;
      p = &tuki_gioca;
      side = 'L';
      pole = pali_tuki;
    }

  /* _Non è possibile scambiare i pali se solo uno è marcato_ */
  if(p->marcato[0]==-1||p->marcato[1]==-1) return;

  view_evidenzia_palo(distanza_palo(p->marcato[1]),
                side,
                pole[p->marcato[1]],
                distanza_palo(p->marcato[0]),
                side,
                pole[p->marcato[0]]);

  int tmp=pole[p->marcato[0]];

  pole[p->marcato[0]] = pole[p->marcato[1]];

  pole[p->marcato[1]] = tmp;

  view_mostra_palo
  (distanza_palo(p->marcato[1]),side,pole[p->marcato[1]]);

  view_mostra_palo
  (distanza_palo(p->marcato[0]),side,pole[p->marcato[0]]);

  view_smarca_palo
  (distanza_palo(p->marcato[1]),side,pole[p->marcato[1]]);

  view_smarca_palo
  (distanza_palo(p->marcato[0]),side,pole[p->marcato[0]]);

  p->marcato[0] = -1;
  p->marcato[1] = -1;
}

int altezza_palo()
{
```

```
  Gioca* p = &tuki_gioca;
  int *pole = pali_tuki;
  char side = 'L';
  if(t_or_g_1_or_2 == 2)
    {
      p = &giuli_gioca;
      side = 'R';
      pole = pali_giuli;
    }
  int i = indice_palo_corrente(*p);
  if(i >= 0)
    {
      return pole[i];
    }
  return -1;
}

int cancella_marca()
{
  Gioca* p = &tuki_gioca;
  if(t_or_g_1_or_2 == 2) p = &giuli_gioca;

  p->marcato[0] =-1;
  p->marcato[1] =-1;

  return -1;
}

int marca()
{
  Gioca* p;
  int *pole;
  char side;

  if(t_or_g_1_or_2 == 2)
    {
      p=&giuli_gioca;
      side = 'R';
      pole = pali_giuli;
    }
  else
    {
      p = &tuki_gioca;
      side = 'L';
      pole = pali_tuki;
    }
  int i = indice_palo_corrente(*p);
  if(i >= 0)
    {
      if(p->marcato[1]>=0)
        {
        view_smarca_palo
        (distanza_palo(p->marcato[1]),side,pole[p->marcato[1]]);
      }
      p->marcato[1] = p->marcato[0];
      p->marcato[0] = i;
      view_marca_palo
      (distanza_palo(p->marcato[0]),side,pole[p->marcato[0]]);

      if(p->marcato[1]>=0){
      view_marca_palo
      (distanza_palo(p->marcato[1]),side,pole[p->marcato[1]]);
      fflush(stdout);
      }
    }
  return i;
}


int verifica_ordinamento(int array[])
{
  for(int i = 0;i<NPOLES-1;i++)
```

```
    {
      if(array[i]<array[i+1]) return 0;
    }
  return 1;
}

void cerca_vincitore()
{
  if(verifica_ordinamento(pali_tuki))
    {
      gioco = 0;
      view_tuki_win();
    }
  if(verifica_ordinamento(pali_giuli))
    {
      gioco = 0;
      view_giuli_win();
    }
}
```

Listato tuki2_view.c

```
 /*_____________________________________________
 *|
 *| tuki2_view.c
 */

#include <sys/ioctl.h>
#include <locale.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "giocatore.h"
#include "tuki2_view.h"
#include "tuki2_model.h"

extern Gioca tuki_gioca;
extern Gioca giuli_gioca;
extern int dl1,dl2,dl3;
extern int rows,cols;


char chat[BFHEIGHT] [BFWIDTH];
int clat[BFHEIGHT] [BFWIDTH];

void stampa_palo(int d, char side,int l,int bgc);



/*_____________________________________________
 *|
 *| Vittoria di Tuki
 */
void view_tuki_win()
{
  int rof = ROFFSET(BFHEIGHT*1);
  int cof = COFFSET(BFWIDTH*1);

  char* strw = "****Tuki ha vinto!!!****";
  int y = 5;
  int x = (BFWIDTH-strlen(strw))/2;
  printf("\x1b[%d;%dH%s",y+rof,x+cof,strw);
  printf("\x1b[%d;%dH",BFHEIGHT+2+rof,1+cof);
  ritardo(3000);
}
```

```
/*
 *|_____________________________________________
 *| Vittoria di Giuli
 */
void view_giuli_win()
{
  int rof = ROFFSET(BFHEIGHT*1);
  int cof = COFFSET(BFWIDTH*1);

  char* strw = "****Giuli ha vinto!!!****";
  int y = 5;
  int x = (BFWIDTH-strlen(strw))/2;
  printf("\x1b[%d;%dH%s",y+rof,x+cof,strw);
  printf("\x1b[%d;%dH",BFHEIGHT+2+rof,1+cof);
  ritardo(3000);
}

void view_cancella_messaggio()
{
  int rof = ROFFSET(BFHEIGHT*1);
  int cof = COFFSET(BFWIDTH*1);

  for(int i = 1;i<= 8;i++)
    for(int j = 1;j<BFWIDTH;j++)
      //printf("\x1b[%dm\x1b[%d;%dH ",BGC,i+rof,j+cof);
      printf("\x1b[%d;%dH ",i+rof,j+cof);
  fflush(stdout);

}

void fading( char * str,int r, int c)
{
  float gray = 255;
  float inc = 0.03;
  float vrs = 1;
  while(gray>232)
    {
      printf("\x1b[%d;%dH\x1b[38;5;%dm%s\x1b[m",r,c,(int)gray,str);
      if(gray>=255){
        vrs = -1;
      }
      if(gray<=232){
        vrs = 1;
      }
      ritardo(1);
      gray+=vrs*inc;
    }
  fflush(stdout);
}

void view_presentazione()
{


  char str2[50];
  char str[60];
  char * message[] = {"TUKI & GIULI",
  "Usa le tue abilità di","programmatore e",
  "vinci la sfida!","Coding-game prodotto e distribuito",
  "Produzioni i SisiniPazzi(R)","visita il sito pumar.it",
  "e scopri le news e i prodotti della tecnologia",
  "del futuro."};
  int l,rof,cof;

  printf("\x1b[%d;%dH",1,1);
  write(STDOUT_FILENO, "\x1b[2J", 4);
  for(int i = 0;i<9;i++)
    {
      l = strlen(message[i]);
      rof = ROFFSET(BFHEIGHT*1.5);
      cof = COFFSET(l);
```

```
      fading(message[i],rof+i,cof);
      ritardo(500);
    }
  printf("\x1b[%d;%dH",1,1);
  write(STDOUT_FILENO, "\x1b[2J", 4);
}

void view_opzioni()
{
  int rof = ROFFSET(BFHEIGHT);
  int cof = COFFSET(BFWIDTH);

  char wc[50];
  strcpy(wc,"Giuly e Tuki: Sfida per Lady Beetle");
  char opt1[35];
  strcpy(opt1,"Premi <p> per usare la tastiera");
  char opt2[35];
  strcpy(opt2,"Premi <a> per sfida tra Algoritmi");
  char opt3[30];
  strcpy(opt3,"<ctrl-q> uscire");
  printf("\x1b[?25l");
  printf("\x1b[%d;%dH\x1b[1m\x1b[;32m%s\x1b[0m",
  4+rof,(int)(MARGINI(wc))+cof,wc);
  printf("\x1b[%d;%dH\x1b[;30m%s\x1b[0m",
  5+rof,(int)(MARGINI(opt1))+cof,opt1);
  printf("\x1b[%d;%dH\x1b[;30m%s\x1b[0m",
  6+rof,(int)(MARGINI(opt2))+cof,opt2);
  printf("\x1b[%d;%dH\x1b[;30m%s\x1b[0m",
  7+rof,(int)(MARGINI(opt3))+cof,opt3);
  fflush(stdout);
}

void view_istruzioni()
{
  int rof = ROFFSET(BFHEIGHT*1);
  int cof = COFFSET(BFWIDTH*1);

  char i1[50];
  strcpy(i1,"Ordina i pali scambiadoli due a due");
  char i2[30];
  strcpy(i2,"<l> destra");
  char i3[30];
  strcpy(i3,"<j> sinistra");
  char i4[30];
  strcpy(i4,"<i> seleziona palo");
  char i5[30];
  strcpy(i5,"<space> scambia");
  printf("\x1b[%d;%dH\x1b[1m\x1b[;32m%s\x1b[0m",
  4+rof,(int)(MARGINI(i1))+cof,i1);
  printf("\x1b[%d;%dH\x1b[;30m%s\x1b[0m",
  5+rof,(int)(MARGINI(i2))+cof,i2);
  printf("\x1b[%d;%dH\x1b[;30m%s\x1b[0m",
  6+rof,(int)(MARGINI(i3))+cof,i3);
  printf("\x1b[%d;%dH\x1b[;30m%s\x1b[0m",
  7+rof,(int)(MARGINI(i4))+cof,i4);
  printf("\x1b[%d;%dH\x1b[;30m%s\x1b[0m",
  7+rof,(int)(MARGINI(i5))+cof,i5);
  fflush(stdout);
  ritardo(3000);
}

void view_mostra_erba()
{
  int rof  =  ROFFSET(BFHEIGHT);
  int cof  =  COFFSET(BFWIDTH);

  for(int i = 1;i<BFWIDTH;i++)
    for(int j = 1;j<BFHEIGHT;j++)
      {
        chat[j][i] = 32;
        clat[j][i] = BGC;
```

```
        printf("\x1b[%d;%dH \x1b[0m",j+rof,i+cof);
      }
  for(int i = 1;i<BFWIDTH;i++)
    {
      printf("\x1b[%d;%dH\x1b[0;%dm\x1b[1;36mv\x1b[0m",
      BASELINE+rof,i+cof,GGC);
      printf("\x1b[%d;%dH\x1b[0;%dm\x1b[1;36mv\x1b[0m",
      BASELINE+1+rof,i+cof,GGC);
      chat[BASELINE][i] = 'v';
      chat[BASELINE+1][i] = 'v';
      clat[BASELINE][i] = GGC;
      clat[BASELINE+1][i] = GGC;
    }
}

void view_marca_palo(int d, char side,int l)
{
  int bgc = 105;
  stampa_palo(d, side, l, bgc);
}

void view_smarca_palo(int d, char side,int l)
{
  int bgc = 43;
  stampa_palo(d, side, l, bgc);
}

void view_mostra_palo(int d, char side,int l)
{
  int bgc = 43;
  stampa_palo(d, side, l, bgc);
}

void view_evidenzia_palo(int d1, char side1,int l1,
int d2, char side2,int l2)
{
  int C1 = 101,C2 = 43;
  stampa_palo(d1, side1, l1, C1);
  stampa_palo(d2, side2, l2, C1);
  ritardo(dl1);
  stampa_palo(d1, side1, l1, C2);
  stampa_palo(d2, side2, l2, C2);
  ritardo(dl2);
  stampa_palo(d1, side1, l1, C1);
  stampa_palo(d2, side2, l2, C1);
  ritardo(dl2);
  stampa_palo(d1, side1, l1, C2);
  stampa_palo(d2, side2, l2, C2);
  ritardo(dl2);
  stampa_palo(d1, side1, l1, C1);
  stampa_palo(d2, side2, l2, C1);
  ritardo(dl3);
}

void stampa_palo(int d, char side,int l,int bgc)
{
  int rof = ROFFSET(BFHEIGHT);
  int cof = COFFSET(BFWIDTH);

  int bfwm = XCENTRALPOLE;
  int sgn = 1;
  if(side=='L') sgn = -1;
  int x = bfwm+sgn*d;
  /*clear the space for the pole*/
  for(int i = 0;i<NPOLES;i++)
    {
      chat[BASELINE-i][x] = 32;
      clat[BASELINE-i][x] = BGC;

      printf("\x1b[%d;%dH \x1b[0m",BASELINE-i+rof,x+cof);
    }
  for(int i = 0;i<l;i++)
```

```
    {
      char polec = ' ';

      printf("\x1b[%d;%dH\x1b[1;%dm%c\x1b[0m",
      BASELINE-i+rof,x+cof,bgc,polec);

      chat[BASELINE-i][x] = polec;
      clat[BASELINE-i][x] = bgc;
    }

  /*_modifica la y del giocatore se si trova
  su un palo in movimento_*/
  if(giuli_gioca.posX==x)
    {
      model_assegna_coordinate(x,BASELINE-l,&giuli_gioca);
    }
  if(tuki_gioca.posX==x)
    {
      model_assegna_coordinate(x,BASELINE-l,&tuki_gioca);
    }
  fflush(stdout);
}

void view_mostra_palo_centrale(int l)
{
  int rof = ROFFSET(BFHEIGHT);
  int cof = COFFSET(BFWIDTH);

  char polec = ' ';
  int x = XCENTRALPOLE;
  view_mostra_palo(0, 'R', l);

  printf
  ("\x1b[%d;%dH\x1b[1;32m\u2618\x1b[0m",BASELINE-l+1+rof,x+cof);

  chat[BASELINE-l+1][x] = polec;
  clat[BASELINE-l+1][x] = 43;
}

void view_cancella_traccia_giocatore(Gioca* p)
{
  int x = p->posX;
  int y = p->posYold-1;
}


void view_mostra_giocatore(Gioca p)
{
  int rof = ROFFSET(BFHEIGHT*1);
  int cof = COFFSET(BFWIDTH*1);

  int x = p.posX;
  int y = p.posY;
  int xo = p.posXold;
  int yo = p.posYold;

  /*_ripristina il fondo_*/
  printf("\x1b[%d;%dH\x1b[;%dm%c\x1b[0m",
  yo+rof,xo+cof,clat[yo][xo],chat[yo][xo]);
  printf("\x1b[%d;%dH\x1b[;%dm%c\x1b[0m",
  yo+rof,xo+1+cof,clat[yo][xo+1],chat[yo][xo+1]);
  if(xo>0)
    printf("\x1b[%d;%dH\x1b[;%dm%c\x1b[0m",
    yo+rof,xo-1+cof,clat[yo][xo-1],chat[yo][xo-1]);

  if(strcmp(p.name,"tuki")==0)
    {
      if(p.dir == DESTRA)
      {
        printf
        ("\x1b[%d;%dH\x1b[1;39m@\x1b[0m\x1b[1;91m>\x1b[0m\x1b[0m",
        y+rof,x+cof);
```

```
        }
      else
      {
        printf("\x1b[%d;%dH\x1b[1;91m<\x1b[1;39m@\x1b[0m\x1b[0m",
        y+rof,x-1+cof);

        if(xo == 0)
        {
         printf("\x1b[%d;%dH%c\x1b[0m",yo+rof,xo+cof,chat[yo][xo-1]);
        }
      else
        {
         printf("\x1b[%d;%dH%c\x1b[0m",yo+rof,xo-1+cof,chat[yo][xo-1]);
        }
      }
    }
  if(strcmp(p.name,"giuli")==0)
    {
      if(p.dir == DESTRA)
        printf("\x1b[%d;%dH\x1b[1;32moò\x1b[0m",y+rof,x+cof);
      else
        printf("\x1b[%d;%dH\x1b[1;32mòo\x1b[0m",y+rof,x+cof);
    }
  fflush(stdout);
}
```

Listato tuki2_controller.c

```
/*_________________________________________________
 |
 | tuki2_controller.c
 |
 */

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>

#include "giocatore.h"
#include "tuki2_model.h"
#include "tuki2_view.h"

int rows,cols;

/*_________________________________________________
 |
 |SEZIONE 1
 |Impostazioni del terminale di gioco
 |
 */
#define CTRL_KEY(k) ((k) & 0x1f)

struct termios orig_termios;

struct configurazione_terminale {
  int cx, cy;
  int righe;
  int colonne;
  struct termios orig_termios;
};
struct configurazione_terminale E;
```

```
void die(const char *s) {
  write(STDOUT_FILENO, "\x1b[2J", 4);
  write(STDOUT_FILENO, "\x1b[H", 3);

  perror(s);
  exit(1);
}

int posizione_cursore(int *rows, int *cols)
{
  char buf[32];
  unsigned int i = 0;
  if (write(STDOUT_FILENO, "\x1b[6n", 4) != 4) return -1;
  while (i < sizeof(buf) - 1) {
    if (read(STDIN_FILENO, &buf[i], 1) != 1) break;
    if (buf[i] == 'R') break;
    i++;
  }
  buf[i] = '\0';
  if (buf[0] != '\x1b' || buf[1] != '[') return -1;
  if (sscanf(&buf[2], "%d;%d", rows, cols) != 2) return -1;
  return 0;
}

int dimensioni_finestra(int *rows, int *cols)
{
  struct winsize ws;

  if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1 || ws.ws_col == 0)
   {
    if (write(STDOUT_FILENO, "\x1b[999C\x1b[999B", 12) != 12)
      return -1;

    return posizione_cursore(rows, cols);
  }
  else
  {
    *cols = ws.ws_col;
    *rows = ws.ws_row;
    return 0;
  }
}

char leggi_tasto()
{
  int nread;
  char c;

  while ((nread = read(STDIN_FILENO, &c, 1)) != 1)
    {
      if (nread == -1 && errno != EAGAIN) die("read");
    }
  return c;
}

void terminale_cucinato()
{
  if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &E.orig_termios) == -1)
    die("tcsetattr");
}

void terminale_crudo()
{
  if (tcgetattr(STDIN_FILENO, &E.orig_termios) == -1)
    die("tcgetattr");

  atexit(terminale_cucinato);

  struct termios raw = E.orig_termios;
  raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
  raw.c_oflag &= ~(OPOST);
```

```
  raw.c_cflag |= (CS8);
  raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
  raw.c_cc[VMIN] = 0;
  raw.c_cc[VTIME] = 5;
  if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw) == -1)
    die("tcsetattr");
}

int processa_tasto()
{
  char c = leggi_tasto();

  switch (c) {
    case CTRL_KEY('q'):
      write(STDOUT_FILENO, "\x1b[?25l", 6);
      write(STDOUT_FILENO, "\x1b[2J", 4);
      write(STDOUT_FILENO, "\x1b[H", 3);
      exit(0);
      break;
  case 'p':
    return 1;
  case 'a':
    return 2;
  }
}

void inizializza_terminale()
{
  if (dimensioni_finestra(&E.righe, &E.colonne) == -1)
    die("dimensioni_finestra");

  rows = E.righe;
  cols = E.colonne;

  if(E.righe<BFHEIGHT||E.colonne<BFWIDTH)
    {
      char str[200];
      sprintf(str,
      "Ridimensiona (%d,%d) la finestra del terminale e rilancia",
      BFHEIGHT,BFWIDTH);
      die(str);
    }
}

/*_____________________________________________________________
 |
 | SEZIONE 2
 | Controller del Gioco
 */
#define MAJOR    0
#define MINOR    1
#define RELEASE  1

extern int pali_giuli [NPOLES];
extern int pali_tuki [NPOLES];

/* I giocatori*/
Gioca giuli_gioca, tuki_gioca;

/*Stato del gioco*/
int t_or_g_1_or_2;
int gioco = 1;
int jritardo = 200;
int dl1 = 50,dl2 = 100,dl3 = 200;

/*_____________________________________________________________________
 |
 | Prototipi
 */
/*interni*/
static void linea_comando(int argc, char **argv);
```

```
/*esterni*/
direzione turno_giuli(); //turno_giuli.c
direzione turno_tuki(); //turno_tuki.c
direzione turno_interattivo(); //turno_interattivo.c
direzione (*challenger)(void);

int main(int argc, char **argv)
{
  strcpy(tuki_gioca.name, "tuki");
  strcpy(giuli_gioca.name, "giuli");
  tuki_gioca.marcato[0] = -1;
  giuli_gioca.marcato[0] = -1;
  linea_comando(argc, argv);

  terminale_crudo();
  inizializza_terminale();

  view_presentazione();

  while(1)
    {
      /*_____________________________________________________________
       |
       | Preparazione dei giocatori sul campo di gioco
       */
      model_assegna_coordinate
      (XCENTRALPOLE-NPOLES*SCALE,BASELINE+1,&tuki_gioca);

      tuki_gioca.UTF = TUTF;

      model_assegna_coordinate
      (XCENTRALPOLE+NPOLES*SCALE,BASELINE+1,&giuli_gioca);

      giuli_gioca.UTF = GUTF;

      model_prepara_campo();

      model_inizializza_campo();

      view_opzioni();

      while (1)
      {
        char chs;
        chs = processa_tasto();

        /*_____________________________________
         |
         | Gioca contro l'algoritmo utente
         */
        if(chs == 2)
        {
          jritardo = 200;
          challenger = &turno_tuki;
          break;
        }
        /*_____________________________________
         |
         | Gioca contro l'utente
         */
      if(chs == 1)
        {
          jritardo = 20; dl1 = 10; dl2 = 20; dl3 = 40;
          challenger = &turno_interattivo;
          view_cancella_messaggio();
          view_istruzioni();
          break;
        }
      }

      view_cancella_messaggio();
```

La sfida del cammino

La prima mossa intelligente per adattarsi all'ambiente è sempre quella di osservare, o più in generale, percepire l'ambiente prima di fare mosse azzardate. La sfida di Tuki 3 si basa proprio sulla capacità di osservare e trarre informazioni dall'osservazione, più nello specifico Tuki 3 è un problema di **pattern recognition**.
Tuki deve affrontare un cammino di una certa lunghezza. Camminando perde energia vitale e calorie e non potrebbe completare il cammino affidandosi solo alle scorte di energia che ha alla partenza. Per questo motivo Tuki ha bisogno di rifocillarsi per poter completare il suo *cammino*. Per fortuna durante il suo percorso incontra degli oggetti che può usare per recuperare le sue energie. Gli oggetti possono essere:

1. melanzane
2. gemme
3. pozioni velenose

Mangiando le melanzane Tuki recupera calorie, con le gemme recupera energia vitale, ma bevendo le pozioni si intossica. La sfida consiste nel riconoscere l'oggetto che Tuki incontra e scegliere l'azione giusta da intraprendere.

Analisi

Model del Tuki 3

Le regole di questo gioco sono più complesse di quelle incontrate fin'ora, ma per fortuna la **fisica** è molto semplice, infatti si limita a descrivere l'esistenza del suolo che sorregge Tuki. In effetti questo scrolling game è un gioco su un unica dimensione descritta dall'asse x. Sebbene il gioco sia mostrato come ovvio su uno display a due dimensioni, il modello del gioco è descritto da un campo ad un'unica dimensione. Per chi non è abituato a pensare in questi termini, basta considerare che gli eventi che capitano durante il gioco dipendono solo dall'istante di tempo corrente e dalla distanza a cui Tuki si trova rispetto al punto di partenza, e che tale distanza è misurata solo sull'asse orizzontale delle x. Prima di addentrarci nelle regole descriviamo bene il campo del gioco.

Campo di gioco

Il campo è una sequenza lineare di celle ognuna della quali ha una *precedente* ed una *successiva* tranne la prima cella che ha solo la successiva e l'ultima che ha solo la precedente. Ogni cella rappresenta una posizione occupabile da Tuki durante la sua camminata. Il campo è memorizzato nell'array `char cammino[LUNGHEZZA_CAMMINO]` ed è costituito solo da parti immobili. L'array è valorizzato come segue:

```
void model_inizializza_cammino(char cammino[]){
  for(int i = 0;i<LUNGHEZZA_CAMMINO;i++){
    if(i%6 == 0){
      cammino[i++] = '#';
      cammino[i] = '#';
    }
    else cammino[i] = ' ';
  }
}
```

L'inizializzazione ha come conseguenza la definizione di un pattern di caratteri costituito dall'alternarsi della struttura ## ogni sei celle. Lo scopo di questo è solo di rendere l'effetto dello scorrimento del campo durante il gioco, ma non serve per la modellazione della fisica del gioco. La reazione vincolare del suolo è ottenuta mantenendo costante la coordinata Y (cioè la riga del monitor) della cella da cui si inizia a stampare Tuki.
Nel campo sono presenti degli oggetti che possiamo definire come parti mobili, questi sono i già citati: melanzane, gemme e pozioni velenose. Classifichiamo questi oggetti come parti mobili perché svaniscono all'incontro con Tuki. Ogni categoria di oggetto ha una forma che lo contraddistingue e tre proprietà:

- valore nutrizionale
- la tossicità
- valore energetico

questi dati sono memorizzati nella struttura:

```
typedef struct{
    float nutrienti;
    float tossine;
    float energia;
    tipo_oggetto t;
    int pos_x;
    char pxl[DIM*DIM];
    char exists;
}oggetto;
```

Gli oggetti svaniscono all'incontro con Tuki purché lui non li eviti

saltandoli.
Come si può intuire, le tre proprietà possedute dagli oggetti del campo sono la chiave del gioco. Tuki deve usarli per garantirsi i livelli di energia e nutrienti e per fare questo deve riconoscerli. Come si vede dalla struttura dati qui sopra, ogni oggetto ha un array che ne definisce la forma. Questo array è passato alla funzione `turno_tuki` in modo che il giocatore possa analizzarne il contenuto e scegliere che azione intraprendere. L'azione può essere una tra le seguenti:

- mangia
- salta
- prendi

Oltre all'oggetto in arrivo, al giocatore è passata una struttura di tipo `stato` che permette di verificare i valori di energia, nutrizione e intossicazione di Tuki.

Livelli di gioco

Riconoscere la forma degli oggetti che Tuki si trova davanti è inizialmente facile perché è possibile vederli nell'introduzione del gioco, o come alcuni avranno già capito, è possibile addestrare Tuki a riconoscerli nelle prime fasi del cammino.
Il gioco però prevede diversi livelli di difficoltà che, come prevedibile, cresce insieme ai livelli. La difficoltà consiste nella variazione crescente dell'immagine della forma degli oggetti rispetto al loro aspetto originale. In pratica un algoritmo modifica, con probabilità proporzionale al livello raggiunto dal giocatore, i pixel dell'oggetto, rendendo la sua *recognition* via via più complessa. La logia dell'algoritmo è riportata qui sotto.

```
void model_aggiungi_rumore(oggetto *o,int livello)
{
  int r;
  for(int i = 0;i<DIM;i++){
    for(int j = 0;j<DIM;j++){
      r = (float)rand()/(float)RAND_MAX*10;
      if(r<livello)
        if(o->pxl[i*DIM+j])
          o->pxl[i*DIM+j] =! o->pxl[i*DIM+j];
    }
  }
}
```

Il termine **rumore** o *noise*, in inglese, si riferisce ad un segnale

presente nell'immagine che non dovrebbe esserci se il processo di produzione dell'immagine fosse ideale. Quindi, in questa nostra implementazione, Tuki si trova a dover riconoscere delle immagini *noisy* o rumorose. Una possibile alternativa potrebbe essere quella di deformare le immagini usando degli algoritmi che implementino una trasformazione fisica come una rotazione o una compressione invece che una deformazione dovuta ad un difetto di elaborazione del segnale. Comunque, una volta capita la logica, si è liberi di trasformare il gioco secondo le proprie esigenze o curiosità.

Le regole del gioco

Inizialmente Tuki ha una propria scorta energetica e nutrizionale e non presenta segni di intossicazione. La prima regola è che ad ogni passo Tuki perde energia e principi nutritivi che può recuperare agendo sugli oggetti che incontra. Se si vuole che il gioco risulti giocabile e divertente, quindi che non porti né a sconfitta tanto meno a vittoria certa il giocatore, bisogna calibrare bene queste quantità: di quanta energia dispone Tuki alla partenza? Quanti nutritivi? Quanto perde ogni passo e quanto guadagna agendo in modo opportuno sugli oggetti?
Le impostazioni date in questa implementazione, permettono do giocare con una certa soddisfazione. Non sono troppo dure, ed è possibile anche passare un livello per pura fortuna, ma di sicuro non è possibile finire i dieci livelli senza usare una buona strategia.
Anzitutto vediamo come vengono calcolati i valori limite che stabiliscono lo stato iniziale di Tuki e i *delta* di energia e nutrizione che vengono persi da Tuki ad ogni passo:

```
#define MAX_TOX (float)(GEMMA_TOX+POZIONE_TOX+MELANZANA_TOX)/3.*((float)NUMERO_
#define MIN_NTRL (MELANZANA_NTRL)*(float)NUMERO_OGGETTI/2.0
#define MIN_NRGY (GEMMA_NRGY)*(float)NUMERO_OGGETTI/2.0

#define T0_NTRL MIN_NTRL
#define T0_NRGY MIN_NRGY

#define D_NTRL T0_NTRL*1.5/LUNGHEZZA_CAMMINO
#define D_NRGY T0_NRGY*1.5/LUNGHEZZA_CAMMINO
```

questi vengono passati alla funzione di inizializzazione del giocatore:

```
model_inizializza(&gctr_tuki,X0_TUKI,T0_NTRL,0,T0_NRGY,"Tuki");
```

e usati durante il ciclo principale per modificarne lo stato:

```
/* Passo avanti*/
avanzamento_cammino += 1;
gctr_tuki.pos_xold = gctr_tuki.pos_x;
gctr_tuki.pos_x += 1;
gctr_tuki.stato_giocatore.nutrienti -= D_NTRL;
gctr_tuki.stato_giocatore.energia -= D_NRGY;
```

Costruire un buon gioco non è solo questione di grafica e suoni, ma soprattutto di studiarne le regole con cura per ottenere un equilibrio perfetto tra difficoltà e giocabilità. La vittoria deve essere alla portata del giocatore, ma non deve mai essere regalata.

Il viewer di Tuki 3

Una particolarità di Tuki 3 rispetto ai giochi precedenti è che mentre in Tuki 1 e 2 il campo è completamente visibile, in Tuki 3 ne viene mostrata solo una finestra di ampiezza definita dalla macro

```
#define LUNGHEZZA_CAMMINO_VISIBILE 140
```

Del resto, le tecniche usate per stampare i caratteri e i colori al terminale sono già state discusse quando abbiamo analizzato Tuki 1 e 2. Approfittiamo allora di questo capitolo per vedere un altro aspetto delle animazioni dei videogiochi.
Già durante la presentazione del gioco si vede Tuki camminare e raggiungere il centro dello schermo, lì Tuki descrive rapidamente il suo obiettivo, poi lascia la scena per mostrare gli oggetti del gioco. Questi, dopo essere apparsi per un breve attimo, svaniscono in una nuvoletta.
Entrambe le animazioni, cioè la camminata di Tuki e la dissolvenza degli oggetti, vengono prodotte usando la prima e più semplice tecnica di animazione grafica.
Per Tuki sono state preparate cinque diverse griglie di caratteri. Durante l'animazione iniziale ad ogni passo di Tuki, l'algoritmo di visualizzazione decide quale, tra le cinque, mostrare e stampare in corrispondenza della posizione corrente di Tuki. La logica dell'algoritmo è incapsulata in `view_stampa_giocatore`. La scelta della frazione di tempo da dedicare ad ogni fotogramma è un aspetto molto importante che influisce decisamente sul risultato finale. Per questo, se si è interessati a produrre una grafica e un'animazione di qualità, anche se realizzata con mezzi *primitivi* questo è uno degli aspetti a cui dedicare molta attenzione.

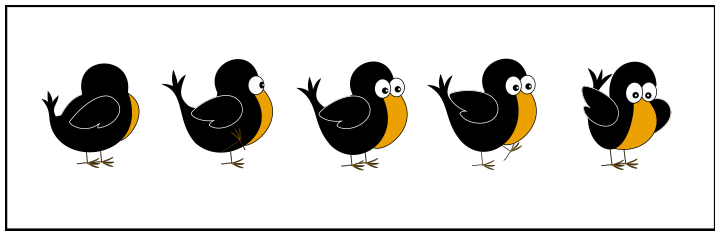


Fig. 8 - Lo studio originale per la camminata di Tuki nel gioco Tuki 3

Il Controller di Tuki 3

Tuki 1 e 2 sono potenzialmente giochi infiniti, nel senso che la durata di una partita non è stabilita dal né dal controller né dal model, ma dipende dalla specifica implementazione degli algoritmi di Tuki e di Giuli. Al contrario ogni partita a Tuki 3 ha una durata massima. Il **controller** ha il compito di far scorrere tutto il **campo** del model nella finestra del monitor. Quando il campo è stato completamente percorso aumenta il livello e arrivati al livello massimo, che è impostato a 10 nella versione originale, il gioco termina. Il controller ha quindi un compito supplementare rispetto a quanto visto fin'ora. Sempre al controller spetta la *collisione detection* con gli oggetti, che in questo caso è implementata verificando la distanza tra la posizione di Tuki e quella degli oggetti:

```
if(ob[oggetto_prossimo].pos_x == gctr_tuki.pos_x+TUKI_DIM-1)
```

Il passaggio di controllo al giocatore avviene solo quando questi può intervenire sul gioco, cioè quando la collisione risulta verificata. In tal caso il controller chiama la funzione `turno_tuki` passandogli i parametri:

```
(ob[oggetto_prossimo].pxl, gctr_tuki.stato_giocatore)
```

che rappresentano l'oggetto in arrivo e lo stao di Tuki.
Il calcolo della prossimità degli oggetti potrebbe anche essere demandato direttamente al **model**, ma essendo molto semplice può anche essere implementato nel **controller** senza rischi di confusione.
In questo gioco **model**, **view** e **controller** sono incapsulati nello stesso file, per renderne la gestion più agile. Per questo motivo la distinzione tra i tre è meno spiccata e rigorosa rispetto agli altri esempi visiti e quelli che vedremo.

Organizzazione dei file e distribuzione del pacchetto Tuki 3

L'organizzazione dei moduli di Tuki 3 è la seguente. Ci sono due soli file sorgente:

- tuki3_mvc.c
- **turno_tuki.c**

ed un unico file header:

- giocatore.h

La distribuzione del gioco è ottenuta compilando `tuki3_mvc.c` con l'opzione `-c`

```
gcc -c tuki3_mvc.c
```

che genera `tuki3_mvc.o` e creando un pacchetto (al solito vanno bene zip, tar ecc) con i file seguenti:

- tuki3_mvc.o
- giocatore.h
- **turno_tuki.c**

Il giocatore apporterà le proprie modifiche e compilerà come segue:

```
gcc -o tuki3.game tuki3_mvc.o turno_tuki.c
```

Il codice completo di Tuki 3

Listato tuki3_mvc.c

```
/*_____________________________________________
 *
 * tuki3_mvc.c
 */

#include<stdio.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <termios.h>
#include <string.h>
#include <time.h>
#include "giocatore.h"


#define CTRL_KEY(k) ((k) & 0x1f)
```

```
#define LARGHEZZA_T 140
#define ALTEZZA_T 30
#define LUNGHEZZA_CAMMINO 512
#define LUNGHEZZA_CAMMINO_VISIBILE 140
#define Y_CAMMINO 15
#define NUMERO_OGGETTI 10
#define GDELAY 40000
#define TUKI_DIM 8

#define GEMMA_TOX 0.1
#define POZIONE_TOX 0.7
#define MELANZANA_TOX 0.05
#define GEMMA_NTRL 0.2
#define POZIONE_NTRL 0.2
#define MELANZANA_NTRL 0.75
#define GEMMA_NRGY 0.7
#define POZIONE_NRGY 0.1
#define MELANZANA_NRGY 0.2
#define X0_TUKI 10

#define MAX_TOX (float)(GEMMA_TOX+POZIONE_TOX+MELANZANA_TOX)/3.*((float)NUMERO_
#define MIN_NTRL (MELANZANA_NTRL)*(float)NUMERO_OGGETTI/2.0
#define MIN_NRGY (GEMMA_NRGY)*(float)NUMERO_OGGETTI/2.0

#define T0_NTRL MIN_NTRL
#define T0_NRGY MIN_NRGY

#define D_NTRL T0_NTRL*1.5/LUNGHEZZA_CAMMINO
#define D_NRGY T0_NRGY*1.5/LUNGHEZZA_CAMMINO
#define MARGIN(str) (LARGHEZZA_T-strlen(str))/2.
#define MENU_ROW 1

#define LIVELLO 10

struct termios orig_termios;
struct configurazione_terminale {
  int cx, cy;
  int righe;
  int colonne;
  struct termios orig_termios;
};
struct configurazione_terminale E;

typedef struct{
  float nutrienti;
  float tossine;
  float energia;
  tipo_oggetto t;
  int pos_x;
  char pxl[DIM*DIM];
  char exists;
}oggetto;

typedef struct  {
  int id;
  int pos_x;
  int pos_xold;
  int pos_yold;
  int pos_y;
  int score;
  char name[10];
  char pxl[11][TUKI_DIM*TUKI_DIM];
  stato stato_giocatore;
 } Giocatore;

char ladyb[TUKI_DIM*TUKI_DIM]={
    0,0,1,0,0,1,0,0,
    0,0,0,1,1,0,0,0,
    0,0,2,2,2,2,0,0,
    0,2,1,2,2,1,2,0,
    2,1,2,2,2,2,1,2,
    0,2,1,2,2,1,2,0,
```

```
    0,0,2,2,2,2,0,0,
    0,0,1,0,0,1,0,0
  };
char goal[TUKI_DIM*TUKI_DIM]={
    3,4,0,0,0,0,0,0,
    4,3,4,3,0,0,0,0,
    5,0,3,4,3,4,0,0,
    5,0,0,0,4,3,4,3,
    5,0,0,0,0,0,3,4,
    0,0,0,0,0,0,0,5,
    0,0,0,0,0,0,0,5,
    0,0,0,0,0,0,0,5
  };

/*_____________________________________________
 *
 * Controllo modalità monitor e tastiera
 */
void cancella_schermo();
void initEditor();
void die(const char *s);
void  terminale_cucinato();
void terminale_crudo();
void inizializza_terminale();
int  posizione_cursore(int *rows, int *cols);
int dimensioni_finestra(int *rows, int *cols);
char leggi_tasto();
int processa_tasto();

/*_____________________________________
 *|
 *| Model
 */
void model_aggiungi_rumore(oggetto *o,int livello);
void model_distruggi(oggetto *objt,int x_left);
void model_inizializza(Giocatore *gctr,int x, float ntrl, float tox,float nrgy,
void model_oggetto(tipo_oggetto t,double nutrienti, double energia,double tossi
void model_inizializza_cammino(char cammino[]);
void fine_giocatore(char *message);
void model_imposta_pixels(char px1,char px2,char px3,char px4,char px5,char px6
void model_punteggio(Giocatore *gctr,oggetto *bjt);
stato model_azione(azione a);
char * model_disfatta(Giocatore *gctr);

/*_____________________________________
 *|
 *| View
 */
void view_punteggio(stato *gctr_stato,float percentage);
void view_stampa_giocatore(Giocatore *gctr,int x_left);
void view_oggetto(oggetto *objt,int x_left,int x_right);
void view_aumentata(int atx, int fromx,char *ch);
void view_nuvoletta(oggetto *ob,int x_left);

/*_____________________________________
 *|
 *| Controller
 */
void delay(int milliseconds);
void mostra_presentazione();
void mostra_livello(int livello, char completo);
void mostra_finale();



azione turno_tuki(char oggetto_inviato[DIM*DIM], stato stato_tuki);

/*** Globali***/
oggetto *g_oggetto;
Giocatore *g_gctr;

/*** Gioco ***/
```

```
int main()
{
  int livello_corrente=0;

  terminale_crudo();
  inizializza_terminale();

  Giocatore gctr_tuki;
  Giocatore gctr_giuli;

  char cammino[LUNGHEZZA_CAMMINO];
  char dsp_cammino[LUNGHEZZA_CAMMINO_VISIBILE+1];
  oggetto ob[NUMERO_OGGETTI]; //The object along the cammino
  oggetto cloud[5];//Clouds animation

  /* Inizio gioco */

  gctr_tuki.id=1;
  g_gctr=&gctr_tuki;
  model_inizializza(&gctr_tuki,X0_TUKI,T0_NTRL,0,T0_NRGY,"Tuki");

  mostra_presentazione();

  /* Cicla sui LIVELLI livelli di gioco */
  for(livello_corrente = 0;
  livello_corrente<LIVELLO;livello_corrente++)
    {
      int avanzamento_cammino=1;
      int oggetto_prossimo=0;

      cancella_schermo();

      mostra_livello(livello_corrente,0);

      cancella_schermo();


      model_inizializza(&gctr_tuki,X0_TUKI,T0_NTRL,0,T0_NRGY,"Tuki");
      model_inizializza_cammino(cammino);

      srand(time(0));
      int k;

      /* Prepara gli oggetti sul campo
       e li trasfigura in base alla difficoltà del livello */

      for(int i=0;i<NUMERO_OGGETTI;i++)
      {
        k=rand()%3;
        int x=(i+2)*((float)LUNGHEZZA_CAMMINO/(float)NUMERO_OGGETTI);
        switch (k)
        {
         case GEMMA:
          model_oggetto
          (GEMMA,GEMMA_NTRL,GEMMA_NRGY,GEMMA_TOX,x,1,ob+i);

          model_aggiungi_rumore(ob+i,livello_corrente);

          break;
         case MELANZANA:
          model_oggetto
          (MELANZANA,MELANZANA_NTRL,MELANZANA_NRGY,
          MELANZANA_TOX,x,1, ob+i);

          model_aggiungi_rumore(ob+i,livello_corrente);

          break;
         case POZIONE:
          model_oggetto
          (POZIONE,POZIONE_NTRL,POZIONE_NRGY,POZIONE_TOX,x,1,ob+i);

          model_aggiungi_rumore(ob+i,livello_corrente);
```

```
    break;
  }
}

cancella_schermo();

char *gtitle="  -Il duro cammino- (c)Scuola Sisini 2018-19 ";
char *tup=   " --------------- ";

printf("\x1b[%d;%dH\x1b[38;5;26m%s\x1b[0m Livello %d",
Y_CAMMINO+1,((int)(MARGIN(gtitle))),gtitle,livello_corrente+1);

for(int r=0;
r<LUNGHEZZA_CAMMINO-LUNGHEZZA_CAMMINO_VISIBILE;
r++)
{
  view_stampa_giocatore(&gctr_tuki,r);
  view_punteggio
  (&(gctr_tuki.stato_giocatore),
  (float)r/(float)        (LUNGHEZZA_CAMMINO-LUNGHEZZA_CAMMINO_VISIBILE))

  for(int i=0;i<NUMERO_OGGETTI;i++)
  {
    view_oggetto(ob+i,r,r+LUNGHEZZA_CAMMINO_VISIBILE);
  }
  if(!ob[oggetto_prossimo].exists)oggetto_prossimo+=1;

  g_oggetto=ob+oggetto_prossimo;

  memcpy(dsp_cammino,cammino+avanzamento_cammino,
  LUNGHEZZA_CAMMINO_VISIBILE);

  dsp_cammino[LUNGHEZZA_CAMMINO_VISIBILE]=0;

  printf("\x1b[%d;%dH\x1b[38;5;94m%s\x1b[0m",
  Y_CAMMINO,1,dsp_cammino);

  fflush(stdout);

  /* Passo avanti*/
  avanzamento_cammino+=1;
  gctr_tuki.pos_xold=gctr_tuki.pos_x;
  gctr_tuki.pos_x+=1;
  gctr_tuki.stato_giocatore.nutrienti-=D_NTRL;
  gctr_tuki.stato_giocatore.energia-=D_NRGY;


  /*Verifica collisione*/
  if(ob[oggetto_prossimo].pos_x == gctr_tuki.pos_x+TUKI_DIM-1)
  {
    azione a = turno_tuki
    (ob[oggetto_prossimo].pxl, gctr_tuki.stato_giocatore);

      model_azione(a);

      if(!ob[oggetto_prossimo].exists){
        view_nuvoletta(ob+oggetto_prossimo,r);
      }
    }


  if(ob[oggetto_prossimo].exists &&
  (ob[oggetto_prossimo].pos_x == gctr_tuki.pos_x+TUKI_DIM-1))
  {
    printf("\x1b[%d;1HAzione: <Nessuna azione>   ",
    MENU_ROW+1);

    model_punteggio(&gctr_tuki,ob+oggetto_prossimo);

    (ob+oggetto_prossimo)->exists=0;

    model_distruggi(ob+oggetto_prossimo,r);
```

```
    }

}

void mostra_presentazione()
{
  write(STDOUT_FILENO, "\x1b[2J", 4);
  write(STDOUT_FILENO, "\x1b[H", 3);
  for(int i=1;i<ALTEZZA_T;i++)
    for(int j=1;j<LARGHEZZA_T;j++)
      printf("\x1b[%d;%dH ",i,j);
  fflush(stdout);
  delay(1000);
  char i1[40];
  strcpy(i1,"@scuola_sisini presents:");
  char i2[30];
  strcpy(i2,"Il duro cammino");
  char i3[70];
  strcpy(i3,"Un gioco programma di pattern recognition");
  char i4[30];
  strcpy(i4,"Copyleft @scuola_sisini");
  char i5[50];
  strcpy(i5,"find more games on facebook.com/scuolasisini");
  char i6[120];
  strcpy(i6,"Premi un tasto per continuare o <CTRL>-q per uscire");

  printf("\x1b[4;%dH%s",(int)(MARGIN(i1)),i1);
  fflush(stdout);
  delay(200000);
  printf("\x1b[5;%dH%s",(int)(MARGIN(i2)),i2);
  fflush(stdout);
   printf("\x1b[7;%dH%s",(int)(MARGIN(i3)),i3);
   delay(200000);
  fflush(stdout);
   printf("\x1b[9;%dH%s",(int)(MARGIN(i4)),i4);
   delay(200000);
  fflush(stdout);
  printf("\x1b[11;%dH%s",(int)(MARGIN(i5)),i5);
   delay(200000);
  fflush(stdout);
  printf("\x1b[13;%dH%s",(int)(MARGIN(i6)),i6);
   delay(200000);
  fflush(stdout);
  char chs=0;

  while (1) {
    chs=  processa_tasto();
    break;
  }
  cancella_schermo();
  for(int i=0;i<(int)MARGIN("XXXXXXXX")-3;i++){

    (*g_gctr).pos_x=i+1;
    view_stampa_giocatore(g_gctr,0);
    (*g_gctr).pos_xold=(*g_gctr).pos_x;
    delay(100000);
  }
  delay(1000000);
  strcpy(i6,"Ciao sono Tuki e voglio raggiungere LadyB");
  printf("\x1b[1;%dH%s",(int)(MARGIN(i6)),i6);
  fflush(stdout);
  delay(1500000);
  strcpy(i6,"Ho bisogno di cibo ed energia, ma temo le tossine");
  printf("\x1b[2;%dH%s",(int)(MARGIN(i6)),i6);
  fflush(stdout);
  delay(1500000);
  strcpy(i6,"Sono guidato dal tuo algoritmo...");
  printf("\x1b[3;%dH%s",(int)(MARGIN(i6)),i6);
  fflush(stdout);
  strcpy(i6,"                                              ");
  printf("\x1b[1;%dH%s",(int)(MARGIN(i6)),i6);
  delay(1500000);
```

```
  strcpy(i6,"che deve riconoscere quello che trovo");
  printf("\x1b[4;%dH%s",(int)(MARGIN(i6)),i6);
  fflush(stdout);
  delay(1500000);
  cancella_schermo();
  strcpy(i6,"per farmi ragguiungere la mia amica");
  printf("\x1b[1;%dH%s",(int)(MARGIN(i6)),i6);
  fflush(stdout);
  delay(1500000);



  /*Mostra gli oggetti*/
  cancella_schermo();
  oggetto ob;
  //Gemma
  strcpy(i6,"Questa è la GEMMA: <PRENDI> la! ");
  printf("\x1b[5;%dH%s",(int)(MARGIN(i6)),i6);
  int ds=(int)MARGIN("xxxxx");;
  model_oggetto(GEMMA,0.1,0.7,0.2,ds,1,&ob);
  view_oggetto(&ob,0,LUNGHEZZA_CAMMINO_VISIBILE);
  fflush(stdout);
  delay(1000000);
  view_nuvoletta(&ob,0);
  fflush(stdout);
  delay(1000000);
  //Melanzana
  strcpy(i6,"Questa è la MELANZANA: <MANGIA> la!");
  printf("\x1b[5;%dH%s",(int)(MARGIN(i6)),i6);
  model_oggetto(MELANZANA,0.1,0.7,0.2,ds,1,&ob);
  view_oggetto(&ob,0,LUNGHEZZA_CAMMINO_VISIBILE);
  fflush(stdout);
  delay(1000000);
  view_nuvoletta(&ob,0);
  fflush(stdout);
  delay(1000000);
  //Veleno
  strcpy(i6,"Questa è la POZIONE: <SALTA> la!    ");
  printf("\x1b[5;%dH%s",(int)(MARGIN(i6)),i6);
  model_oggetto(POZIONE,0.1,0.7,0.2,ds,1,&ob);
  view_oggetto(&ob,0,LUNGHEZZA_CAMMINO_VISIBILE);
  fflush(stdout);
  delay(1000000);
  view_nuvoletta(&ob,0);
  fflush(stdout);
  delay(1000000);



}


/***termimal***/
void die(const char *s) {
  write(STDOUT_FILENO, "\x1b[2J", 4);
  write(STDOUT_FILENO, "\x1b[H", 3);

  perror(s);
  exit(1);
}

int posizione_cursore(int *rows, int *cols) {
  char buf[32];
  unsigned int i = 0;
  if (write(STDOUT_FILENO, "\x1b[6n", 4) != 4) return -1;
  while (i < sizeof(buf) - 1) {
    if (read(STDIN_FILENO, &buf[i], 1) != 1) break;
    if (buf[i] == 'R') break;
    i++;
  }
  buf[i] = '\0';
  if (buf[0] != '\x1b' || buf[1] != '[') return -1;
```

```
  if (sscanf(&buf[2], "%d;%d", rows, cols) != 2) return -1;
  return 0;
}

int dimensioni_finestra(int *rows, int *cols) {
  struct winsize ws;

  if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1 || ws.ws_col == 0)
   {
    if (write(STDOUT_FILENO, "\x1b[999C\x1b[999B", 12) != 12)
    return -1;
    return posizione_cursore(rows, cols);
  }
  else
   {
    *cols = ws.ws_col;
    *rows = ws.ws_row;
    return 0;
  }
}

char leggi_tasto()
{
  int nread;
  char c;

  while ((nread = read(STDIN_FILENO, &c, 1)) != 1)
    {
      if (nread == -1 && errno != EAGAIN) die("read");
    }
  return c;
}

void  terminale_cucinato()
{
  if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &E.orig_termios) == -1)
    die("tcsetattr");
}
void terminale_crudo()
{

  if (tcgetattr(STDIN_FILENO, &E.orig_termios) == -1)
   die("tcgetattr");

  atexit( terminale_cucinato);

  struct termios raw = E.orig_termios;
  raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
  raw.c_oflag &= ~(OPOST);
  raw.c_cflag |= (CS8);
  raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
  raw.c_cc[VMIN] = 0;
  raw.c_cc[VTIME] = 5;
  if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw) == -1)
   die("tcsetattr");

}

/*** input ***/
int processa_tasto()
{
  char c = leggi_tasto();
  switch (c)
  {
    case CTRL_KEY('q'):
      write(STDOUT_FILENO, "\x1b[?25l", 6);
      write(STDOUT_FILENO, "\x1b[2J", 4);
      write(STDOUT_FILENO, "\x1b[H", 3);
      exit(0);
      break;
   case 'p':
    return 1;
```

```
   case 'a':
    return 2;
   default:
    return 0;
  }
}

/*** init ***/

void inizializza_terminale()
{
  if (dimensioni_finestra(&E.righe, &E.colonne) == -1)
   die("dimensioni_finestra");
  if(E.righe<ALTEZZA_T||E.colonne<LARGHEZZA_T)
    {
      char str[200];
      sprintf(str," Ridimensiona la finestra (%d,%d)",
      ALTEZZA_T,LARGHEZZA_T);
      die(str);
    }
}
```

Listato turno_tuki.c

```
/*_____________________________________
*|
*| turno_tuki.c
*/
#include "giocatore.h"

azione turno_tuki
(char oggetto_incontrato[DIM*DIM], stato st_tuki)
{
  /*Scrivi qui il tuo codice*/

  return MANGIA;
}
```

Listato giocatore.h

```
/*_____________________________________
*|
*| giocatore.h
*/

#define DIM 5
typedef enum {GEMMA,MELANZANA,POZIONE} tipo_oggetto;
typedef enum {MANGIA,SALTA,PRENDI} azione;
typedef struct
{
  float nutrienti;
  float tossine;
  float energia;
} stato;
```

TUKI4: GIULI IL DINAMITARDO

Con *Giuli il dinamitardo* inauguriamo una nuova stagione.
Dopo Tuki 3, in cui Tuki non aveva rivali diretti, riprende la sfida tra Tuki e Giuli ma questa volta la sfida è asimmetrica, cioè i due giocatori non si sfidano l'un l'altro sulla stessa prova.
Questo gioco è stato ispirato del crescente interesse per gli algoritmi *intelligenti*, cioè algoritmi capaci di assolvere il proprio compito imparando dall'esperienza. Questi si differenziano dagli algoritmi pensati per risolvere un problema specifico, i cos' detti *task specific algorithm* (TSA). Gli algoritmi di *path finding* o di *sorting* visti in Tuki 1 e 2 sono esempi di TSA. Tuki 3 può essere risolto sia implementando un TSA che un algoritmo intelligente. La sfida di Tuki 4 invece chiama ad alta voce un algoritmo capace di apprendere. Vediamola!

La sfida

A prima vista può sembrare davvero un rompicapo, ma, con un po' di pazienza, si capisce che è una sfida ragionevole e che può anche essere vinta. In cosa consiste?
Giuli questa volta ha preso una brutta passione, salito su un albero si è messo a lanciare dei pacchetti a Tuki, solo che non sono pacchetti del tutto innocui, infatti alcuni di questi contengono una bomba mentre gli altri una mela. Il fatto è che dall'esterno i pacchetti sono uguali sia che contengano l'una, sia che contengano l'altra: non c'è modo di distinguerli.
Il problema è che se Tuki afferra un pacchetto contenente una bomba, questa gli scoppia in faccia e lui si fa male. Tuki potrebbe decidere di lasciare andare tutti i pacchetti, ma come nel gioco Tuki 3, ogni turno perde energia che può recuperare solo mangiando le mele.
Sembra impossibile da risolvere vero? Non proprio, infatti Giuli non lancia i pacchi con le bombe e le mele in modo completamente casuale. Sebbene il giocatore non sappia che schema segua Giuli, questi ne segue uno. Giuli ha diviso mentalmente il campo da gioco in zone in cui lancerà bombe e in zone in cui lancerà mele, e un po' qua e un po' la, in base al proprio sentimento, lancia mele e bombe attenendosi a tale schema.
Questa è la sfida di Tuki 4, capire lo schema prima di subire troppi danni a causa delle bombe o di rimanere senza energie a causa della fame!

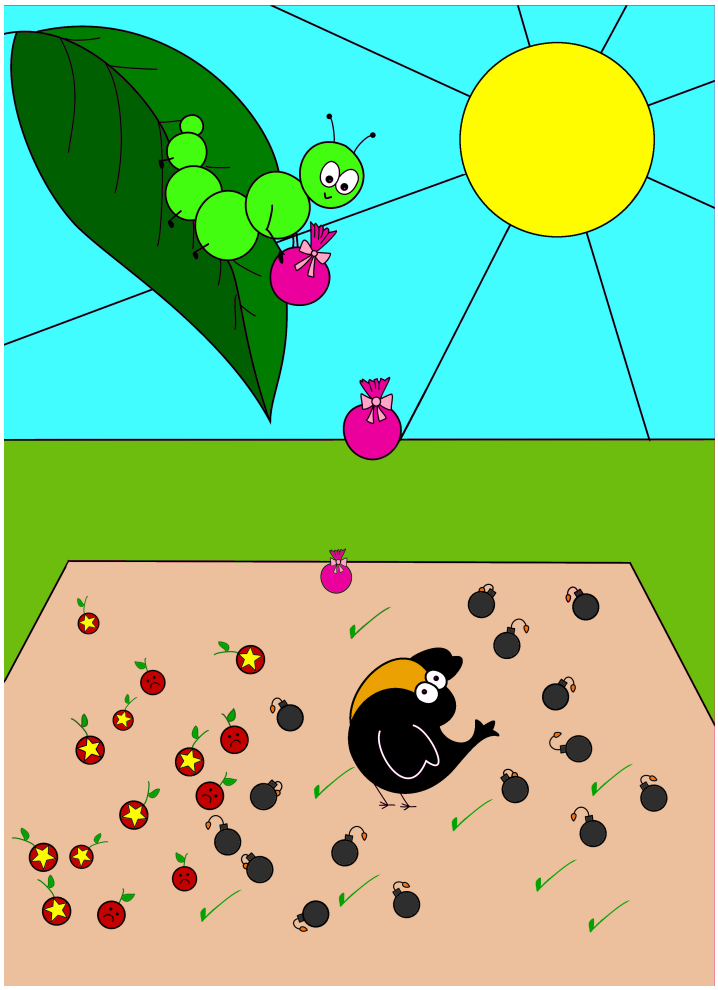


Fig. 9 - La figura mostra la bozza iniziale da cui è stati ispirato: Giuli il dinamitardo

Analisi di Tuki 4

Il model di Tuki 4

La logica del **model** orbita intorno al concetto di **evento** che è l'elemento più importante del modello stesso. L'evento rappresenta la sequenza del turno di Giuli e della risposta di Tuki, in pratica incapsula le informazioni su cosa Giuli ha tirato e su come Tuki ha risposto. La struttura dati per l'evento è la seguente:

```
typedef struct {
  lancio l;
  azione az;
} evento;
```

Ogni evento avviene in un punto preciso del campo. Questo è un rettangolo di dimensioni ALTEZZA e LARGHEZZA. Il concetto di evento è il concetto chiave della fisica relativistica, e anche in questo gioco incapsula a fisica del campo. Le coordinate dell'evento sono incapsulate nel tipo `lancio`:

```
typedef struct  {
  int x;
  int y;
  pacco proiettile; //cosa ha lanciato
} lancio;
```

Il model ha la responsabilità di generare le coordinate dell'evento. Quando Giuli e Tuki hanno contribuito a completare l'evento: Giuli stabilendo cosa contiene il pacco legato a quell'evento e Tuki stabilendo l'azione di risposta al pacco, al model spetta di applicare le regole del gioco per determinare il punteggio dei giocatori.

Regole di Tuki 4

Abbiamo già visto con Tuki 3 che la *matematica* delle regole è molto importante perché segna il confine tra un gioco complesso ma divertente e uno troppo semplice e banale. Per questo bisogna dedicare attenzione e una giusta quantità di analisi a questo aspetto. In Tuki 4, le regole servono a stabilire cosa succede a Tuki per ciascuna delle quattro possibili eventualità che si generano per ogni evento:

1. il pacco contiene una bomba e Tuki la distrugge
2. il pacco contiene una bomba e Tuki e la mangia
3. Il pacco contiene una mela e Tuki la distrugge
4. il pacco contiene una mela e Tuki la mangia

Nello stabilire queste regole, cioè nel determinare come varia lo *score* di Tuki, bisogna conoscere bene la sfida e sapere che Tuki non può agire con l'azione adeguata fin dall'inizio.
La creatività e l'impulso creativo danno il meglio quando sono illuminati dalla ragione, in questo caso la ragione deve farci comprendere a pieno la natura del problema e impostare le regole in modo che il gioco sia si difficile, ma non impossibile. In altre parole, quello che vogliamo dire è che l'algoritmo di *scoring* deve lasciare tempo a Tuki di `capire` la logica con cui Giuli sta lanciando i suoi pacchi. La logica di scoring è incapsulata nella funzione `model_aggiorna_stato` il cui codice è presentato qui nel seguito:

```
if(e.l.proiettile==MELA){
    if(e.az==MANGIA)
      {
        //Hai fatto bene
        energia+=1;
      }
    else
      {
        ;//Non fa nulla
      }
  }else{
    if(e.az==MANGIA)
      {
        danni+=2;
      }
    else
      {
        danni-=1;
        energia+=1;
      }
  }

  /* Aggoirnamento */
  lanci_totali++;
  energia-=1;

  if(danni&amp;amp;gt;MAXDANNI||energia&amp;amp;lt;0) return 0;
  return 1;
```

I parametri `aggiustabili` per rendere il gioco divertente sono l'energia iniziale e il numero massimo di danni sopportabili da Tuki prima di dare fourfait. È regolando questi due valori che il gioco acquisisce o perde di giocabilità.

Il view di Tuki 4

Dal punto di vista grafico, bisogna ammettere che questo gioco è un po' monotono. Il **viewer** ha il compito molto semplice di visualizzare gli eventi sullo schermo dopo che il controller ha fuso insieme il lancio e la reazione. Per stuzzicare un po' gli animi sensibili, sono stati usati dei caratteri UTF che per rappresentare le quattro diverse eventualità previste. La bomba distrutta è rappresentata con un segno di spunta, mentre la bomba mangiata con un cerchio nero. La mela distrutta con un faccini triste (perché ha perso la mela) e la mela mangiata con una stella dentro ad un cerchietto.

Il controller di Tuki 4

La semplicità di questo gioco permette al **controller** di implementare un comportamento esemplare, mantenendosi nei limiti delle proprie competenze senza rinunciare alle proprie responsabilità. Come il per Tuki 3, anche il 4 ha una durata massima stabilita, cioè il numero

predefinito di lanci operati da Giuli. Il **controller** ha anzitutto la responsabilità di ciclare sui turni dei giocatori fino alla fine del gioco, e, per ogni turno, passare il controllo a Giuli in modo che stabilisca il contenuto del pacco, quindi passarla a Tuki perché veda dove sta `atterrando` il pacco e prenda la sua decisione:

```
lancio l = model_genera();

pacco pac = turno_giuli(l);

l.proiettile = pac;

azione a=turno_tuki(
                    model_energia(),
                    model_danni(),
                    l.x,
                    l.y);
```

Raccolte queste informazioni le impacchetta insieme e le passa al **model** perché applichi le regole viste sopra:

```
evento e;

e.l=l;
e.az=a;

int s=model_aggiorna_stato(e);
```

Dopo la valutazione dell'evento è il turno del **viewer**, a cui il **controller** passa lo scettro perché provvede a visualizzare l'evento:

```
view_evento(e);
```

Organizzazione dei file e distribuzione del pacchetto Tuki 4

L'organizzazione dei moduli di Tuki 4 è la seguente. Ci sono i cinque file sorgente:

- tuki4_model.c
- tuki4_view.c
- tuki4_controller.c
- turno_giuli.c
- **turno_tuki.c**

ed i tre file header:

- tuki4_model.h

- tuki4_view.h
- giocatore.h

La distribuzione del gioco è ottenuta compilando i sorgenti con l'opzione `-c`

```
gcc -c  tuki4_view.c tuki4_controller.c tuki4_model.c turno_giuli.c
```

che genera i `.o` e creando un pacchetto con i file seguenti:

- tuki4_model.o
- tuki4_view.o
- tuki4_controller.o
- turno_giuli.o
- giocatore.h
- **turno_tuki.c**

Il giocatore apporterà le proprie modifiche e compilerà come segue:

```
gcc -c  turno_tuki.c
```

quindi:

```
gcc -o tuki4.game tuki4_model.o tuki4_view.o tuki4_controller.o turno_giuli.o t
```

Il codice completo di Tuki 4

Listato tuki4_model.c

```
/*_____________________________
 *|
 *| tuki4_model.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "tuki4_model.h"

/*** section data ***/
int lanci_totali;
int energia;
int danni;


/** SECTION 1: preparazione **/
void model_inizializza()
{
  lanci_totali = 0;
  energia = ENERGIA;
  danni = DANNI;
}

/** SECTION 2: comunicazione **/
int model_energia()
{
  return energia;
```

```
}

int model_danni()
{
  return danni;
}

int model_lanci_totali()
{
  return LANCI;
}

int model_lanci_restanti()
{
  return LANCI - lanci_totali;
}

/*_____________________________________________________________________
 *|
 *| Il model genera pseudo casualmente le coordinate del lancio
 */
lancio model_genera()
{
  double x,y;
  lancio l;

  x = (double)rand()/(double)RAND_MAX*(LARGHEZZA-2);
  y = (double)rand()/(double)RAND_MAX*(ALTEZZA-2);
  x -= (LARGHEZZA-2)/2;
  y -= (ALTEZZA-2)/2;
  l.x = (int)x;
  l.y = (int)y;

  return l;
}


int model_aggiorna_stato(evento e)
{
  if(e.l.proiettile == MELA)
  {
    if(e.az == MANGIA)
      {
        //Hai fatto bene
        energia += 1;
      }
    else
      {
        ;//Non fa nulla
      }
  }
  else
  {
    if(e.az==MANGIA)
      {
        danni += 2;
      }
    else
      {
        danni -= 1;
        energia += 1;
      }
  }

  /* Aggiornamento */
  lanci_totali++;
  energia -= 1;

  if(danni>MAXDANNI||energia<0) return 0;
  return 1;

}
```

TUKI5: TUKI E IL PACMAN

L'idea per il quinto gioco è arrivata casualmente da un amico di *Scuola Sisini* che dopo aver giocato alle versione on-line di Tuki 4 manifestò il desiderio di approfondire lo studio del linguaggio C per scriversi il codice del Pacman.
Fino a quel momento avevamo implementato solo giochi di nostra invenzione e l'idea di crearne uno basato su un videogioco di successo pareva davvero buona. Il gioco venne proposto anche in versione on-line, che permetteva (è ancora attivo al momento della pubblicazione di questo libro) di giocare anche senza scaricare i file oggetto e senza avere un compilatore.
Quasi tutti gli appassionati di videogiochi conoscono Pacman e ci hanno giocato, per questo un gioco programma che riprenda la sua tradizione, offre qualcosa in più: la possibilità di sfidare direttamente gli algoritmi che ci hanno fatto impazzire sul loro stesso piano.

Fig. 10 - Due mesi dall'apertura del livello capra

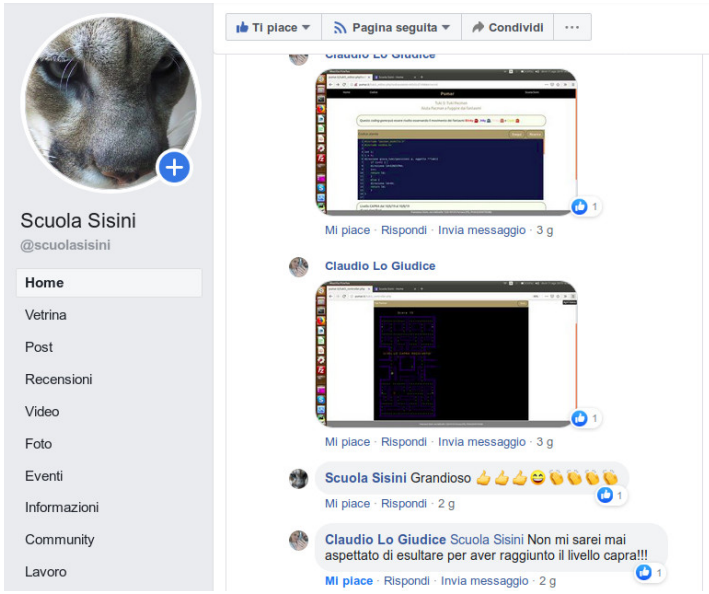


Fig. 11 - Il post con la soluzione

Fig. 12 - Il codice con cui il vincitore ha superato il livello

La sfida

È difficile descrivere la sfida del Pacman aggiungendo qualcosa che non sia già noto alla maggior parte di appassionati di videogiochi. Comunque, per chi non ci avesse mai giocato, diciamo che è un videogioco arcade a labirinto, distribuito dalla Namco nel 1980. Pacman, il protagonista, deve mangiare delle pillole disseminate lungo il percorso del labirinto mentre viene inseguito da quattro fantasmini che cercano di bloccarlo.

Analisi di Tuki 5

Tuki 5 presenta la stessa architettura a MVC già discussa nei capitoli precedenti. Nello sviluppare questo gioco non abbiamo analizzato il sorgente originale di Pacman, ma abbiamo preferito scriverlo ex novo seguendo lo stesso stile già usato per i giochi precedenti. Per chi fosse interessato, è possibile trovare in rete di *disassemblato* di Pacman originale, cioè le istruzioni assembly ottenute con un operazione di *reverse enguneering* dal videogioco originale.

Il model di Tuki 5

Il campo

Il labirinto è composto da una griglia di 31 righe per 28 colonne di caratteri. Ogni celle può essere occupata da un muro, da una pillola, da una pastiglia (energizzante) o essere vuota. Il labirinto è strutturato in modo da essere percorso da corridoi che non sono mai più larghi di una singola cella. Gli elementi del campo sono classificato usando il tipo oggetto:

```
typedef enum {
    a='a',
    A = 'A', ANGOLO ALTO SIN BORDO
    B = 'B', ANGOLO ALTO DES BORDO
    C = 'C', ANGOLO BASSO SIN BORDO
    D = 'D', ANGOLO BASSO DES BORDO
    E = 'E', MURO ORR BORDO
    F = 'F', MURO VER BORDO
    G = 'G', ANGOLO ALTO SIN
    H = 'H', ANGOLO ALTO DES
    I = 'I', ANGOLO ALTO DES
    J = 'J', SPAZIO
    L = 'L', ANGOLO ALTO SIN
    M = 'M', ANGOLO BASSO DES
    N = 'N', ANGOLO BASSO SIN
    O = 'O', ANGOLO BASSO SIN
    P = 'P', ANGOLO ALTO SIN
    Q = 'Q', ANGOLO BASSO DES
    R = 'R', ANGOLO ALTO DES
    S = 'S', MURO VER
    T = 'T',  MURO ORR
    U = 'U', PUNTINO
    V = 'V', PILLOLA
    X = 'X', MURO VER BORDO
    Y = 'Y', ANGOLO BASSO DES
    W = 'W', ANGOLO BASSO SIN
    Z = 'Z', MURO ORZZ BORDO
} oggetto;
```

Il campo è memorizzato in un array di `oggetto`:

```
  oggetto  campo[ALTEZZA][LARGHEZZA]={
  {J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J},
```

```
  {J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J},
  {A,E,E,E,E,E,E,E,E,E,E,E,I,L,E,E,E,E,E,E,E,E,E,E,B},
  {F,U,U,U,U,U,U,U,U,U,U,U,S,S,U,U,U,U,U,U,U,U,U,U,X},
  {F,U,G,T,T,H,U,G,T,T,T,H,U,S,S,U,G,T,T,T,H,U,G,T,T,H,U,X},
  {F,V,S,J,J,S,U,S,J,J,J,S,U,S,S,U,S,J,J,J,S,U,S,J,J,S,V,X},
  {F,U,W,T,T,Y,U,W,T,T,T,Y,U,W,Y,U,W,T,T,T,Y,U,W,T,T,Y,U,X},
  {F,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,X},
  {F,U,G,T,T,H,U,G,H,U,G,T,T,T,T,T,T,H,U,G,H,U,G,T,T,H,U,X},
  {F,U,W,T,T,Y,U,S,S,U,W,T,T,H,G,T,T,Y,U,S,S,U,W,T,T,Y,U,X},
  {F,U,U,U,U,U,U,S,S,U,U,U,U,S,S,U,U,U,U,S,S,U,U,U,U,U,X},
  {C,Z,Z,Z,Z,Z,B,U,S,W,T,T,H,J,S,S,J,G,T,T,Y,S,U,A,Z,Z,Z,Z,D},
  {J,J,J,J,J,F,U,S,G,T,T,Y,J,W,Y,J,W,T,T,H,S,U,X,J,J,J,J,J},
  {J,J,J,J,J,F,U,S,S,J,J,J,J,J,J,J,J,J,J,S,S,U,X,J,J,J,J,J},
  {J,J,J,J,J,F,U,S,S,J,A,E,E,J,J,E,E,B,J,S,S,U,X,J,J,J,J,J},
  {Z,Z,Z,Z,Z,Z,D,U,W,Y,J,F,J,J,J,J,J,J,X,J,W,Y,U,C,E,E,E,E,E},
  {J,J,J,J,J,J,U,J,J,J,F,J,J,J,J,J,J,J,X,J,J,J,U,J,J,J,J,J},
  {E,E,E,E,E,B,U,G,H,J,F,J,J,J,J,J,J,J,X,J,G,H,U,A,Z,Z,Z,Z,Z},
  {J,J,J,J,J,F,U,S,S,J,C,E,E,E,E,E,E,D,J,S,S,U,X,J,J,J,J,J},
  {J,J,J,J,J,F,U,S,S,J,J,J,J,J,J,J,J,J,J,S,S,U,X,J,J,J,J,J},
  {J,J,J,J,J,F,U,S,S,J,G,T,T,T,T,T,T,H,J,S,S,U,X,J,J,J,J,J},
  {A,E,E,E,E,D,U,W,Y,J,W,T,T,H,G,T,T,Y,J,W,Y,U,C,E,E,E,E,B},
  {F,U,U,U,U,U,U,U,U,U,U,U,S,S,U,U,U,U,U,U,U,U,U,U,U,X},
  {F,U,G,T,T,H,U,G,T,T,T,H,U,S,S,U,G,T,T,T,H,U,G,T,T,H,U,X},
  {F,U,W,T,H,S,U,W,T,T,T,Y,U,W,Y,U,W,T,T,T,Y,U,S,G,T,Y,U,X},
  {F,V,U,U,S,S,U,U,U,U,U,U,U,J,J,U,U,U,U,U,U,U,S,S,U,U,V,X},
  {O,T,H,U,S,S,U,G,H,U,G,T,T,T,T,T,T,H,U,G,H,U,S,S,U,G,T,Q},
  {P,T,Y,U,W,Y,U,S,S,U,W,T,T,H,G,T,T,Y,U,S,S,U,W,Y,U,W,T,R},
  {F,U,U,U,U,U,U,U,S,S,U,U,U,U,S,S,U,U,U,U,S,S,U,U,U,U,U,X},
  {F,U,G,T,T,T,T,Y,W,T,T,H,U,S,S,U,G,T,T,Y,W,T,T,T,T,H,U,X},
  {F,U,W,T,T,T,T,T,T,T,T,Y,U,W,Y,U,W,T,T,T,T,T,T,T,Y,U,X},
  {F,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,X},
  {C,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,D},
  {J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J},
  {J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J,J}
};
```

Questo array viene passato al giocatore come argomento della funzione `gioca_tuki`, che quindi,differentemente da Tuki 1, **ha la percezione completa della forma del labirinto** e della disposizione delle pillole da mangiare.

La **Fisica** del gioco è molto semplice, se non per un singolo aspetto... l'effetto Pacman, che discuteremo tra breve. A parte questo effetto, la fisica si limita alla reazione vincolare offerta dai muri del labirinto. Il **model** calcola la posizione dei giocatori e controlla le eventuali collisioni con i muri del labirinto e, in caso di collisione, si limita ad impedire il movimento nella specifica direzione. Sempre il **model** ha il compito di verificare se i fantasmi vengono in contatto con Tuki, e nel caso, di informare il controller dell'avvenuta collisione.

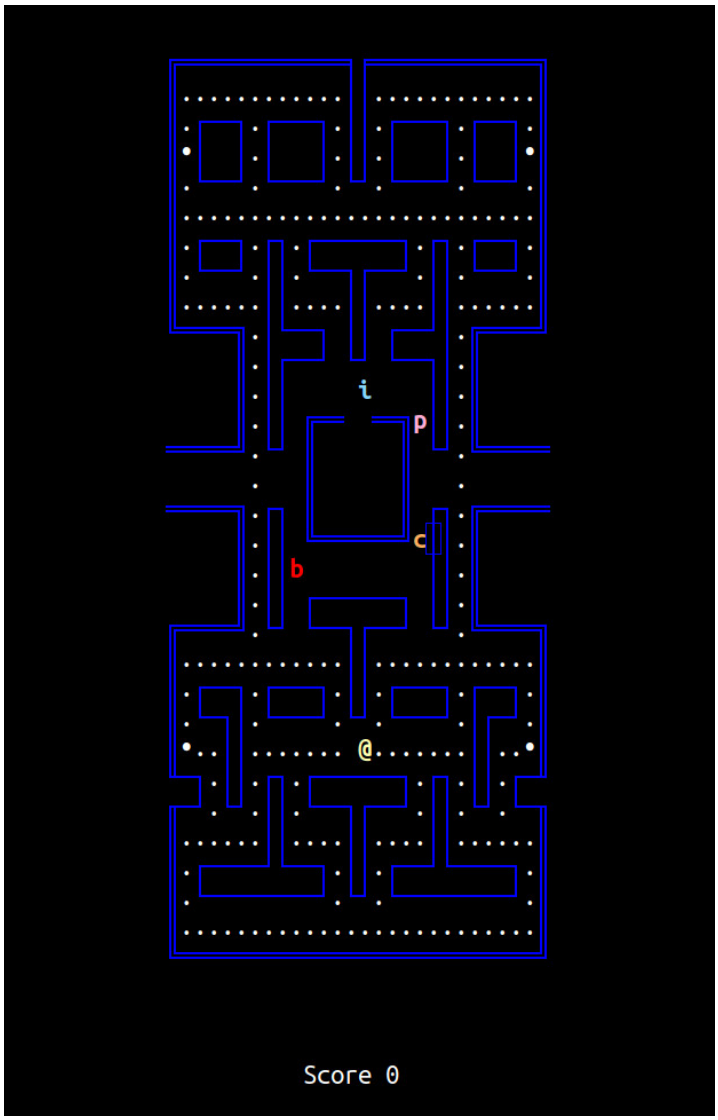


Fig. 13 - La figura mostra il labirinto del gioco Tuki 5

Le regole

Le regole di Pacman sono molto intuitive. Una delle caratteristiche dei giochi arcade che venivano distribuiti negli anni '80 era proprio la loro semplicità e capacità di essere compresi senza intermediazioni con il progettista. In Tuki 5, che si ispira a Pacman al limite del plagio (se non per il fatto che Tuki 5 è no profit e open source), abbiamo cercato di preservare la semplicità e l'immediatezza del gioco originale, anche se sono state apportate alcune semplificazioni:

1. Se Tuki viene in contatto con uno dei fantasmini perde la partita
2. Se Tuki mangia una pillola aumenta il punteggio di 1
3. Se Tuki mangia una pillola energetica può venire in contatto con i fantasmini senza perdere la partita
4. Tuki vince se mangia tutte le pillole presenti nel labirinto

Queste sono le regole del gioco.

Il view di Tuki 5

Le idee e l'implementazione del **view** per questo gioco non si discostano da quelle già incontrate in precedenza. Una funzione interessante però è la `terminale_str` che codifica in caratteri UTF il modello astratto del labirinto definito nel **model**. La sua implementazione (parziale) è la seguente:

```
     if(e  == B)
    return "\u2557";

  if(e  == A)
    return "\x1b[38:5:21m\u2554";

  if(e  == J)
    return " ";

  if(e  == C)
    return "\u255a";

  if(e  == D)
    return "\u255D";

  if(e  == E)
    return "\u2550";

  if(e  == F)
    return "\u2551";

  if(e  == G)
    return "\u250c";

  if(e  == H)
    return "\u2510";

  if(e  == K)
    return "\u2596";

  if(e  == I)
    return "\u2555";

  if(e  == L)
    return "\u2552";
```

Un ultimo aspetto da vedere, che non è mai stato incontrato negli altri giochi, è la possibilità offerta dai terminali di gestire il *blinking*, cioè il lampeggiamento, dei caratteri stampati. Usando la sequenza `\x1b[5m` i caratteri che vengono stampati continuano a lampeggiare sotto il

diretto controllo del terminale, liberando l'applicativo da questa responsabilità.

Il controller di Tuki 5

Il **controller** chiede al **model** le posizioni dei cinque giocatori sul campo: quattro fantasmini e Tuki. Questi gliele passa con una variabile di tipo `posizioni` definita come segue:

```
typedef struct  {
  int tuki_x, tuki_y;
  int blinky_x, blinky_y;
  int inky_x, inky_y;
  int pinky_x, pinky_y;
  int clyde_x, clyde_y;
} posizioni;
```

È importante notare che è il **model** e non il controller a mantenere il controllo delle posizioni dei giocatori sul campo. Il **controller** passa una copia delle posizioni ai giocatori, insieme ad una copia dell'array con lo schema del labirinto e delle pillole presenti sul campo, in modo che ogni giocatore possa usare queste informazioni per stabilire la propria strategia di gioco:

```
direzione t = gioca_tuki(p, lab);

direzione b = gioca_blinky(p, lab);

direzione i = gioca_inky(p, lab);

direzione pi = gioca_pinky(p, lab);

direzione c = gioca_clyde(p, lab);
```

A questo punto il controller ha raccolto tutte le direzioni di moto scelte dai cinque giocatori e deve comunicarle al model perché esegua le verifiche fisiche (collisioni) e applichi le regole del gioco:

```
char go = mdl_passo(t,b,i,pi,c);

int pnt = mdl_punteggio();

char inblu=mdl_superpacman();
```

Il model deve anche verificare *l'effetto Pacman*: nel caso le coordinate di Pacman coincidano con quelle dell'imbocco del tunnel, deve trasformarle nelle coordinate corrispondenti all'uscita opposta del tunnel stesso

```
//-verifica imbocco tunnel
   if(g.tuki_x == 23 && g.tuki_y==16)
     {
       g.tuki_x = 3;
       muovi(DESTRA,&g.tuki_x,&g.tuki_y,0);
     }
```

```
if(g.tuki_x == 2 && g.tuki_y==16)
 {
   g.tuki_x = 22;
   muovi(DESTRA,&g.tuki_x,&g.tuki_y,0);
 }
```

Dopo questo il controller chiama il **viewer** perché rappresenti graficamente lo stato del campo dei giocatori:

```
view_punteggio(pnt);

view_giocatori(p,lab,inblu);
```

I fantasmini Blinky, Pinky, Inky e Clyde

Tuki 5 non è il Pacman, questo è chiaro. Pacman è un gioco molto bello che ha fatto la storia dei videogame, e Tuki 5 si ispira ad esso per proporsi come gioco programma a chi ha voglia di sperimentare la programmazione anche come attività creativa o didattica.
Detto questo, nel limite del possibile, Tuki 5 cerca di replicare alcuni aspetti di Pacman, proprio per permettere a chi già vi ha giocato di trasformare in algoritmo la propria strategia di gioco. Per questo il labirinto è stato riprodotto con la massima fedeltà all'originale, e, entro limiti ragionevoli, abbiamo cercato di replicare il comportamento dei fantasmini, basandoci più sulla descrizione ufficiale che non sulla nostra esperienza personale nel gioco.
Di seguito sono descritti i comportamenti dei fantasmi che sono implementati nella versione del codice presente in questo libro. Nel paragrafo successivo verranno inoltre analizzate le principali strutture del codice che realizzano i comportamenti descritti qui nel seguito.

Lo spirito dei fantasmini

"To bring some tension into the game, I wanted the monsters to surround Pac-Man at a certain stage of the game. But I felt that it would create stress for the player if he were constantly surrounded by ghosts. Therefore, I had the monsters surround him in waves: first attack, then retreat. When they regrouped, the attack began again. It seemed to me more natural than a constant attack. "
- Toru Iwatani, creator of Pac-Man
Il lavoro dei fantasmini è di impedire a Tuki di completare il labirinto. Per raggiungere questo obiettivo hanno due possibilità: la prima consiste nell'attaccare direttamente Tuki in modo da fermare il gioco, la seconda invece nel presidiare la zona verso cui Tuki si dirige per

salvaguardare le pastiglie. Per implementare questi comportamenti nei fantasmini è necessario infondergli un po' di spirito e ovviamente l'unico modo che conosciamo è di scrivere un algoritmo che ne determini il movimento. Ora stabiliamo cosa vogliamo ottenere da ciascuno dei fantasmini, poi vedremo con quale algoritmo possiamo realizzarlo.
L'aspetto fondamentale della nostra implementazione è che per ogni fantasmino ci baseremo sullo stesso algoritmo di movimento. In pratica il movimento avviene fissando una cella di partenza che è quella in cui si trova il fantasmino e una di arrivo. La scelta della cella di arrivo determina la strategia di difesa del territorio fantasmino. Ovviamente la scelta è influenzata anche dallo stato del fantasmino.

La scelta della cella obiettivo

La cella obiettivo è la cella verso cui si dirige il fantasmino. Ogni fantasmino sceglie la cella obiettivo in base alla propria strategia. Ovviamente la cella obiettivo cambia durante l'esecuzione del gioco, ma non ad ogni turno, bensì solo in corrispondenza dei punti di decisione.

I punti di decisione

I fantasmini non pensano spesso, anzi una volta imboccato un corridoio non cambiano più finché non raggiungono un punto di diramazione, cioè un **nodo** del **grafo**.
Una volta raggiunto il **nodo** stabiliscono la la cella obiettivo in base alla loro `personalità` e quindi decidono il nodo successivo per raggiungere l'obiettivo. In questa seconda scelta non mettono tanta intelligenza. Anzi tutto danno per scontato che il labirinto sia *connesso* cioè che esita sempre un *cammino* tra la propria posizione e la cella obiettivo. Poi stabiliscono quale arco del grafo seguire semplicemente calcolando la distanza *euclidea* che intercorre tra la prima cella di ognuno dei due archi e la cella obiettivo. Ovviamente in questo modo corrono il rischio di imboccare la strada più lunga!

Grafi

Per chi non li avesse mai incontrati, i grafi sono un concetto molto importante in informatica, e in questo gioco abbiamo deciso di usarli per implementare il movimento dei fantasmi. L'idea è molto semplice.

Come si può vedere dalla figura del labirinto, la maggior parte delle celle presenti nei corridoi, ha un unico predecessore ed un unico successore. In pratica quando un giocatore si trova su questo tipo di cella non può far altro che proseguire o tornare indietro.
Alcune tra le celle (per esattezza sono trentaquattro) si trovano invece in punti di biforcazione: in pratica queste celle sono connesse a più di due celle. Chiamiamo queste celle **nodi**. La prima cosa che si vede è che i nodi sono uniti tra loro: ogni nodo è unito almeno ad altri tre nodi e al massimo a quattro (questo è vero nel gioco in questione, non è una proprietà generale). Nel codice i nodi sono rappresentati dalla seguente struttura:

```
typedef struct {
  int riga;
  int colonna;
  int indice;
  int n_sx;
  int n_dx;
  int n_su;
  int n_giu;
} nodo;
```

La riga e la colonna si riferiscono all'array del campo visto prima. L'indice è un progressivo che si assegna ai nodi, partendo da 0, da in alto e proseguendo da sinistra a destra e dall'alto verso l basso. I successivi quattro membri della struttura sono gli indici dei nodi a cui è collegato il nodo in oggetto. Per fare un esempio, il primo nodo del grafo è rappresentato dai valori seguenti:

```
  ...riga = 3;
  ...colonna = 6;
  ...indice = 0;
  ...n_sx = 2;
  ...n_dx = 5;
  ...n_su = -1;
  ...n_giu = 3;
```

Il valore -1 viene usato (in questo gioco) per indicare che manca il collegamento. Nel caso in questione il nodo 0 non ha un collegamento verso l'alto.
La rappresentazione del labirinto come **grafo** permette di usare diversi algoritmi di *path finding* basati sui grafi. Nel nostro caso, l'uso del grafo è piuttosto semplice e si basa su una regole e un assunto:

- ogni fantasmino si muove solo da nodo a nodo, cioè, una volta che ha stabilito di raggiungere un nodo non cambia più decisione finché non lo ha raggiunto (nessuna inversione di marcia)
- il labirinto è completamente connesso: è possibile raggiungere qualsiasi nodo dato un nodo di partenza

Questo approccio al movimento dei fantasmi ha come effetto che essi appaiono scorrere *eleganti* e senza inversioni improvvise, un po' come accade durante un inseguimento di auto.

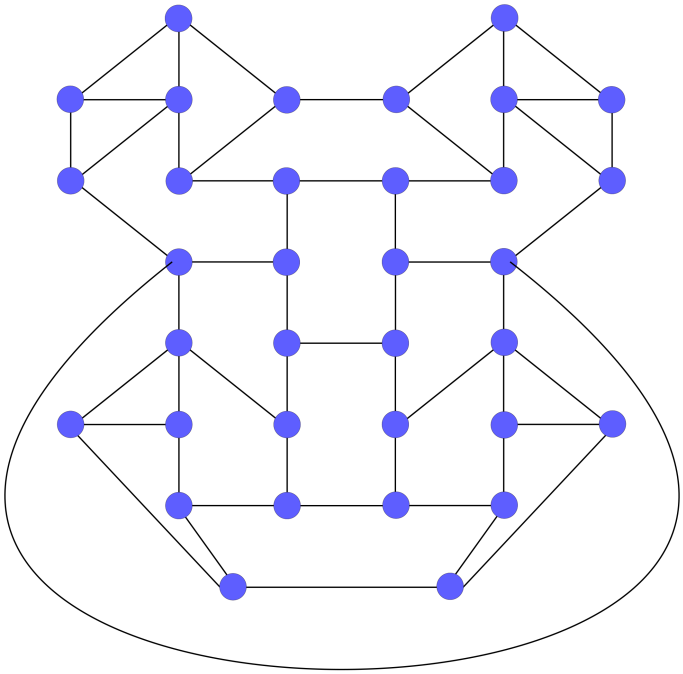


Fig. 14 - La figura mostra il grafo ottenuto dai nodi del labirinto di Tuki 5

Le quattro strategie

Blinky, il fantasmino rosso punta diretto alla cella di Tuki. Come specificato sopra, i suoi cambi di direzione vengono stabiliti solo in corrispondenza dei punti di decisione e inoltre (come abbiamo già scritto), durante la sua caccia non gli è permesso di invertire il verso del moto. La strategia di Blinky lo porta ad essere il più aggressivo e temuto dei quattro.
Pinky, quello rosa, punta alla cella che si trova quattro posizioni avanti a quella di Tuki. Come specificato già per Blinky, anche per Pinky e per gli altri fantasmini vale il principio per cui il cambio di direzione viene preso solo in corrispondenza dei punti di decisione, per questo principio, non è escluso che Pinky catturi Tuki anche se punta sempre davanti a lui e non direttamente la sua cella.
Inky, il fantasma azzurro, ha un comportamento più complesso da

```
      return "\u255c";

    if(e==R)
      return "\u2556";

    if(e==S)
      return "\u2502";

    if(e==T)
       return "\u2500";

    if(e==W)
      return "\u2514";

    if(e==X)
      return "\u2551";

    if(e==Y)
      return "\u2518";

    if(e==V) //PIllola
      return "\x1b[38:5:231m\u2022\x1b[38:5:21m";

    if(e==U) //Trifogli
      return "\x1b[38:5:231m.\x1b[38:5:21m";

    if(e==Z)
     return "\u2550";

    if(e==a)
     return "\u2598";

  return st;
}



void view_labirinto(oggetto **campo)
{
  int x,y;
  int rof=ROFFSET(ALTEZZA*1);
  int cof=COFFSET(LARGHEZZA);
  char str[25];

  for(int i=0;i<ALTEZZA;i++)
    for(int j=0;j<LARGHEZZA;j++)
      {
        x=j+cof;
        y=i+rof;
        sprintf(str,
        "\x1b[%d;%dH%s",y,x,terminale_str(*(*(campo+i)+j)));
        write(STDOUT_FILENO, str, strlen(str));
      }
  printf("\x1b[%d;%dH",y,1);
}

int view_sfondo(int r, int c,oggetto **campo)
{
  int x,y;
  int rof=ROFFSET(ALTEZZA*1);
  int cof=COFFSET(LARGHEZZA);
  char str[25];

  x=c+cof;
  y=r+rof;
  sprintf(str,"\x1b[%d;%dH%s",y,x,terminale_str(*(*(campo+r)+c)));
  write(STDOUT_FILENO, str, strlen(str));

  return 0;
}

void view_mangiato(posizioni g)
```

```
{
  int x,y;
  int rof=ROFFSET(ALTEZZA);
  int cof=COFFSET(LARGHEZZA);
  char str[80];
  char* str_lst="@*[]\\||//#+^@";

  x=g.tuki_x+cof;
  y=g.tuki_y+rof;
  for(int i=0;i<strlen(str_lst);i++)
    {
      sprintf(str,
      "\x1b[38:5:229m\x1b[%d;%dH%c",y,x,*(str_lst+i));
      write(STDOUT_FILENO, str, strlen(str));
      delay(50);
    }


}



int view_giocatori(posizioni g, oggetto ** lab,char in_blue)
{
  static posizioni g_prec;
  static char init=0;
  char pu[20];
  /* colori dei fantasmi */
  int bl,in,pi,cl;
  if(in_blue)
    {
      strcpy(pu,"\x1b[5m");
      bl=in=pi=cl=57;
    }
  else
    {
      strcpy(pu,"");
      bl=196;
      in=117;
      pi=218;
      cl=215;
    }

  if(init==0)
    {
      g_prec=g;
      init=1;
    }

  int x,y;
  int rof=ROFFSET(ALTEZZA);
  int cof=COFFSET(LARGHEZZA);
  char str[80];


  /** Tuki */
  // Ripristina il campo
  x = g_prec.tuki_x;
  y = g_prec.tuki_y;
  sprintf(str,"\x1b[38:5:229m\x1b[%d;%dH%s",y+rof,x+cof,terminale_str(*(*(lab+y
  write(STDOUT_FILENO, str, strlen(str));

  //Stampa Tuki
  x=g.tuki_x+cof;
  y=g.tuki_y+rof;
  sprintf(str,"\x1b[38:5:229m\x1b[%d;%dH%s",y,x,terminale_str(TUKI));
  write(STDOUT_FILENO, str, strlen(str));

  /** Blinky */
  // Ripristina il campo
  x = g_prec.blinky_x;
  y = g_prec.blinky_y;
```

```
  sprintf(str,"\x1b[38:5:229m\x1b[%d;%dH%s",y+rof,x+cof,terminale_str(*(*(lab+y
  write(STDOUT_FILENO, str, strlen(str));

  // stampa Blinky
  x=g.blinky_x+cof;
  y=g.blinky_y+rof;
  sprintf(str,"%s\x1b[38:5:%dm\x1b[%d;%dH%s\x1b[0m",pu,bl,y,x,terminale_str(BLI
  write(STDOUT_FILENO, str, strlen(str));

  /** Inky */
   // Ripristina il campo
  x = g_prec.inky_x;
  y = g_prec.inky_y;
  sprintf(str,"\x1b[38:5:229m\x1b[%d;%dH%s",y+rof,x+cof,terminale_str(*(*(lab+y
  write(STDOUT_FILENO, str, strlen(str));

  // stampa inky
  x=g.inky_x+cof;
  y=g.inky_y+rof;
  sprintf(str,"%s\x1b[38:5:%dm\x1b[%d;%dH%s\x1b[0m",pu,in,y,x,terminale_str(INK
  write(STDOUT_FILENO, str, strlen(str));

  /** Pinky */
  // Ripristina il campo
  x = g_prec.pinky_x;
  y = g_prec.pinky_y;
  sprintf(str,"\x1b[38:5:229m\x1b[%d;%dH%s",y+rof,x+cof,terminale_str(*(*(lab+y
  write(STDOUT_FILENO, str, strlen(str));

  //stampa pinky
  x=g.pinky_x+cof;
  y=g.pinky_y+rof;
  sprintf(str,"%s\x1b[38:5:%dm\x1b[%d;%dH%s\x1b[0m",pu,pi,y,x,terminale_str(PIN
  write(STDOUT_FILENO, str, strlen(str));

  /** Clyde */
  // Ripristina il campo
  x = g_prec.clyde_x;
  y = g_prec.clyde_y;
  sprintf(str,"\x1b[38:5:229m\x1b[%d;%dH%s",y+rof,x+cof,terminale_str(*(*(lab+y
  write(STDOUT_FILENO, str, strlen(str));

  // stampa clyde
  x=g.clyde_x+cof;
  y=g.clyde_y+rof;
  sprintf(str,"%s\x1b[38:5:%dm\x1b[%d;%dH%s\x1b[0m",pu,cl,y,x,terminale_str(CLY
  write(STDOUT_FILENO, str, strlen(str));

  /** Memorizza le posizioni del turno appena visualizzato */
  g_prec=g;


}

void view_punteggio(int punti)
{
  char str2[50];
  char str[60];
  sprintf(str2,"Score %d ",punti);
  int l=strlen(str2);
  int rof=ROFFSET(ALTEZZA*1);
  int cof=COFFSET(l);
  sprintf(str,"\x1b[%d;%dH%s",ALTEZZA+2,cof,str2);
  write(STDOUT_FILENO, str, strlen(str));
}

void view_gameover(char * message)
{
  char str2[50];
  char str[60];

  int l=strlen(message);
```

```
  int rof=ROFFSET(ALTEZZA*1);
  int cof=COFFSET(l);
  sprintf(str,"\x1b[%d;%dH%s\r\n\r\n",rof+12+1,cof,message);
  write(STDOUT_FILENO, str, strlen(str));
  //Set cursor at the begin of last line

  printf("\x1b[%d;%dH",rows,1);
}

void delay(int milliseconds){
  long pause;
  clock_t now,then;

  pause = milliseconds*(CLOCKS_PER_SEC/1000);
  now = then = clock();
  char c;
  while( (now-then) < pause )
    {
      now = clock();
    }
}

void fading( char * str,int r, int c)
{
  float gray=232;
  float inc=0.03;
  float vrs=1;
  while(gray<255)
    {
      printf("\x1b[%d;%dH\x1b[38;5;%dm%s\x1b[m",r,c,(int)gray,str);
      if(gray<255){
        vrs=+1;
      }
      if(gray<=232){
        //vrs=1;
      }
      delay(1);
      gray+=vrs*inc;
    }
  fflush(stdout);


}


void view_presentazione(){
  char str2[50];
  char str[60];
  char * message[]={"TUKI e Pacman",
  "Coding-game prodotto e distribuito",
  "da Scuola Sisini",
  "visita http://pumar.it"};
  int l,rof,cof;

  printf("\x1b[%d;%dH",1,1);
  write(STDOUT_FILENO, "\x1b[2J", 4);
  for(int i=0;i<4;i++)
    {
      l=strlen(message[i]);
      rof=ROFFSET(ALTEZZA*1.5);
      cof=COFFSET(l);
      fading(message[i],rof+ALTEZZA/2+i,cof);
      delay(100);
    }
  printf("\x1b[%d;%dH",1,1);
  delay(1250);
  write(STDOUT_FILENO, "\x1b[2J", 4);

}
```

Listato tuki5_controllo.c

```
/*_____________________________
*|
*| tuki5_controllo.c
*/

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>

#include "tuki5_modello.h"
#include "tuki5_visore.h"

/** DESCRIZIONE */
/**
   - Nel campo di gioco sono presenti trifogli, pillole, muri

   - La distribuzione degli oggetti nel campo di gioco è memorizzata in un arra

   - La posizione dei 5 giocatori è memorizzata in una singola struttura

   - Ad ogni personaggio del gioco è associata una funzione con il nome: gioca_
   che torna la mossa che il personaggio fa in un turno di gioco.

*/
/**
   -game-controller.c
   Il controller offre ai giocatori le 5 funzioni di controllo
   a turno chiama le 5 funzioni per permettere ad ogni giocatore di giocare
   il controller chiama poi il model comunicando la mossa del giocatore

   -game-model.c
   il model elabora la mossa del giocatore valuatando la nuova posizione e veri
   1) se avviene una collisione tra giocatori
   2) se avviene una collisione con il muro
   3) se viene mangiato un trifoglio
   4) se viene mangiata una pillola

   -game-viewer.c
   stampa sullo schermo lo stato del gioco

*/

/* MACRO  */
#define MAJOR    0
#define MINOR    1
#define RELEASE  1

/* GLOBALS  */

/* PROTOTIPI  */
direzione gioca_tuki(posizioni p, oggetto** lab);
direzione gioca_blinky(posizioni p, oggetto** lab);
direzione gioca_inky(posizioni p, oggetto** lab);
direzione gioca_pinky(posizioni p, oggetto** lab);
direzione gioca_clyde(posizioni p, oggetto** lab);



int main(int argc, char **argv)
{

  /* Presentazione */
  view_init();
```

```
  //view_presentazione();

  /* Inizio gioco */
  int inizio=  mdl_genera_campo();

  if(!inizio) exit(1);

  /* Labirinto */
  oggetto ** lab = mdl_campo();
  view_labirinto(lab);


  /**
   * START GAME
   */
  int r=1;
  while(r){

    posizioni p=mdl_posizioni();

    if(lab==0) exit(1);

    direzione t = gioca_tuki(p, lab);

    direzione b = gioca_blinky(p, lab);

    direzione i = gioca_inky(p, lab);

    direzione pi = gioca_pinky(p, lab);

    direzione c = gioca_clyde(p, lab);

    char go = mdl_passo(t,b,i,pi,c);
    int pnt = mdl_punteggio();
    char inblu=mdl_superpacman();

    /** mostra campo e giocatori */
    r+=1;
    view_punteggio(pnt);
    view_giocatori(p,lab,inblu);
    delay(150);

    /** Pacman è stato mangiato **/
    if(go==0)
      {
        view_mangiato(p);
        view_gameover("GAME OVER");
        exit(0);
      }

    /** Pacman ha finito le pastiglie 245 */
    if(pnt == 244)
      {
        view_gameover("Ha vinto TUKI!");
        exit(0);
      }

  }
  /*end of the game*/

}
```

Listato tuki5_modello.h

```
/*_________________________________
*|
*| tuki5_modello.h
*/

/*** SECTION: battlefiled  ***/
#define ALTEZZA 35
#define LARGHEZZA 28

/**
 * Gli oggetti nel campo
 */
typedef enum {
  //Angoli
   a='a',
   A = 'A', //ANGOLO ALTO SIN BORDO
   B = 'B', //ANGOLO ALTO DES BORDO
   C = 'C', //ANGOLO BASSO SIN BORDO
   D = 'D', //ANGOLO BASSO DES BORDO
   E = 'E', //MURO ORR BORDO
   F = 'F', //MURO VER BORDO
   G = 'G', //ANGOLO ALTO SIN
   H = 'H', //ANGOLO ALTO DES
   I = 'I', //ANGOLO ALTO DES
   J = 'J', //SPAZIO
   K = 'K',
   L = 'L', //ANGOLO ALTO SIN
   M = 'M', //ANGOLO BASSO DES
   N = 'N', //ANGOLO BASSO SIN
   O = 'O', //ANGOLO BASSO SIN
   P = 'P', //ANGOLO ALTO SIN
   Q = 'Q', //ANGOLO BASSO DES
   R = 'R', //ANGOLO ALTO DES
   S = 'S', //MURO VER
   T = 'T', //MURO ORR
   U = 'U', //PUNTINO
   V = 'V', //PILLOLA
   X = 'X', //MURO VER BORDO
   Y = 'Y', //ANGOLO BASSO DES
   W = 'W', //ANGOLO BASSO SIN
   Z = 'Z', //MURO ORZZ BORDO
   TRIFOGLIO = 'O'
} oggetto;


/**
 * I 5 giocatori
 */
typedef enum {
  //Personaggi
  BLINKY='b',
  INKY='i',
  PINKY='p',
  CLYDE='c',
  TUKI='@',

} giocatore;


#ifndef PLAYER
typedef enum  {SINISTRA,SU,DESTRA,GIU,FERMO} direzione;
#define PLAYER
#endif

typedef struct  {
  int tuki_x, tuki_y;
  int blinky_x, blinky_y;
  int inky_x, inky_y;
  int pinky_x, pinky_y;
  int clyde_x, clyde_y;
} posizioni;
```

```
/*** interfaccia***/

/* elabora l'evoluzione del gioco e torna 0 quando il tuki è mangiato */
int mdl_passo
(direzione tuki,
 direzione blinky,
 direzione inky,
 direzione pinky,
 direzione clyde);

/* torna una copia del campo ad uso dei giocatori */
oggetto **  mdl_campo();

/* copia delle posizioni dei giocatori, ad uso dei giocatori */
posizioni mdl_posizioni();

/* crea il campo, ad uso del controller */
int mdl_genera_campo();

/* libera la memoria del campo */
void mdl_libera_campo();

/* torna il punteggio del tuki */
int mdl_punteggio();

/* torna lo stato di superpacman */
char mdl_superpacman();
```

Listato tuki5_visore.h

```
/*_________________________________
*|
*| tuki5_visore.h
*/

void view_init();

/** Stampa il labirinto */
void view_labirinto(oggetto **labirinto);

/** stampa l'oggetto (se) presente in posizione r e c */
int view_sfondo(int r, int c,oggetto **labirinto);

/** stampa i giocatori sul campo */
int view_giocatori(posizioni giocatori, oggetto ** labirinto, char inblue);

/** stampa il punteggio */
void view_punteggio(int punti);

/** stampa la fine del gioco */
void view_gameover(char * messaggio);

/** stamp la presentazione */
void view_presentazione();

/** stampa la fine di pacman */
void view_mangiato(posizioni g);

void delay(int millis);
```

Listato gioca_fantasmi.c

```
/*_____________________________
*|
*| gioca_fantasmi.c
*/

#include "tuki5_modello.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>


/*_____________________________________________________________
 *|
 *| Nodo del grafo che rappresenta il labirinto
 */
typedef struct {
  int riga;
  int colonna;
  int indice;
  int n_sx;
  int n_dx;
  int n_su;
  int n_giu;
} nodo;

/*_____________________________________________________________
 *|
 *| I nodi del grafo el labirinto
 */
nodo grafo[34];



/*_____________________________________________________________
 *|
 *| Crea un grafo corrispondente al labirinto del tuki
 */
void collega_nodi()
{
  grafo[0].riga = 3;
  grafo[0].colonna = 6;
  grafo[0].indice = 0;
  grafo[0].n_sx = 2;
  grafo[0].n_dx = 5;
  grafo[0].n_su = -1;
  grafo[0].n_giu = 3;

  grafo[1].riga = 3;
  grafo[1].colonna = 21;
  grafo[1].indice = 1;
  grafo[1].n_sx = 6;
  grafo[1].n_dx = 9;
  grafo[1].n_su = -1;
  grafo[1].n_giu = 8;

  grafo[2].riga = 7;
  grafo[2].colonna = 1;
  grafo[2].indice = 2;
  grafo[2].n_sx = -1;
  grafo[2].n_dx = 3;
  grafo[2].n_su = 0;
  grafo[2].n_giu = 10;

  grafo[3].riga = 7;
  grafo[3].colonna = 6;
  grafo[3].indice = 3;
  grafo[3].n_sx = 2;
  grafo[3].n_dx = 4;
  grafo[3].n_su = 0;
  grafo[3].n_giu = 10;
```

```
grafo[4].riga = 7;
grafo[4].colonna = 9;
grafo[4].indice = 4;
grafo[4].n_sx = 3;
grafo[4].n_dx = 5;
grafo[4].n_su = -1;
grafo[4].n_giu = 12;

grafo[5].riga = 7;
grafo[5].colonna = 12;
grafo[5].indice = 5;
grafo[5].n_sx = 4;
grafo[5].n_dx = 6;
grafo[5].n_su = 0;
grafo[5].n_giu = -1;

grafo[6].riga = 7;
grafo[6].colonna = 15;
grafo[6].indice = 6;
grafo[6].n_sx = 5;
grafo[6].n_dx = 7;
grafo[6].n_su = 1;
grafo[6].n_giu = -1;

grafo[7].riga = 7;
grafo[7].colonna = 18;
grafo[7].indice = 7;
grafo[7].n_sx = 6;
grafo[7].n_dx = 8;
grafo[7].n_su = -1;
grafo[7].n_giu = 13;

grafo[8].riga = 7;
grafo[8].colonna = 21;
grafo[8].indice = 8;
grafo[8].n_sx = 7;
grafo[8].n_dx = 9;
grafo[8].n_su = 1;
grafo[8].n_giu = 11;

grafo[9].riga = 7;
grafo[9].colonna = 26;
grafo[9].indice = 9;
grafo[9].n_sx = 8;
grafo[9].n_dx = -1;
grafo[9].n_su = 1;
grafo[9].n_giu = 11;

grafo[10].riga = 10;
grafo[10].colonna = 6;
grafo[10].indice = 10;
grafo[10].n_sx = 2;
grafo[10].n_dx = -1;
grafo[10].n_su = 3;
grafo[10].n_giu = 14;

grafo[11].riga = 10;
grafo[11].colonna = 21;
grafo[11].indice = 11;
grafo[11].n_sx = -1;
grafo[11].n_dx = 9;
grafo[11].n_su = 8;
grafo[11].n_giu = 17;

grafo[12].riga = 13;
grafo[12].colonna = 12;
grafo[12].indice = 12;
grafo[12].n_sx = 15;
grafo[12].n_dx = 13;
grafo[12].n_su = 4;
grafo[12].n_giu = -1;
```

```
grafo[13].riga = 13;
grafo[13].colonna = 15;
grafo[13].indice = 13;
grafo[13].n_sx = 12;
grafo[13].n_dx = 16;
grafo[13].n_su = 7;
grafo[13].n_giu = -1;

grafo[14].riga = 16;
grafo[14].colonna = 6;
grafo[14].indice = 14;
grafo[14].n_sx = -1;
grafo[14].n_dx = 15;
grafo[14].n_su = 10;
grafo[14].n_giu = 20;

grafo[15].riga = 16;
grafo[15].colonna = 9;
grafo[15].indice = 15;
grafo[15].n_sx = 14;
grafo[15].n_dx = -1;
grafo[15].n_su = 12;
grafo[15].n_giu = 18;

grafo[16].riga = 16;
grafo[16].colonna = 18;
grafo[16].indice = 16;
grafo[16].n_sx = -1;
grafo[16].n_dx = 17;
grafo[16].n_su = 13;
grafo[16].n_giu = 19;

grafo[17].riga = 16;
grafo[17].colonna = 21;
grafo[17].indice = 17;
grafo[17].n_sx = 16;
grafo[17].n_dx = -1;
grafo[17].n_su = 11;
grafo[17].n_giu = 23;

grafo[18].riga = 19;
grafo[18].colonna = 9;
grafo[18].indice = 18;
grafo[18].n_sx = -1;
grafo[18].n_dx = 19;
grafo[18].n_su = 15;
grafo[18].n_giu = 21;

grafo[19].riga = 19;
grafo[19].colonna = 18;
grafo[19].indice = 19;
grafo[19].n_sx = 18;
grafo[19].n_dx = -1;
grafo[19].n_su = 16;
grafo[19].n_giu = 22;

grafo[20].riga = 22;
grafo[20].colonna = 6;
grafo[20].indice = 20;
grafo[20].n_sx = 30;
grafo[20].n_dx = 21;
grafo[20].n_su = 14;
grafo[20].n_giu = 24;

 grafo[21].riga = 22;
grafo[21].colonna = 9;
grafo[21].indice = 21;
grafo[21].n_sx = 20;
grafo[21].n_dx = 26;
grafo[21].n_su = 18;
grafo[21].n_giu = -1;
```

```
 *| nodo_a <- nodo_dest
 */

/* Nodo temporaneo */
nodo n_d;

/* calcola la distanza tra i prossimi nodi
e la cella intermedia tra Tuki e Blinky */
double d_sx = 10000, d_dx = 10000, d_su = 10000, d_giu = 10000;

cella s;
cella tuki, blinky;
tuki.riga = p.tuki_y;
tuki.colonna = p.tuki_x;
blinky.riga = p.blinky_y;
blinky.colonna = p.blinky_x;

int m_colonna, m_riga;
m_colonna = (tuki.colonna + blinky.colonna)/2;
m_riga = (tuki.riga + blinky.riga)/2;

/* stabilisco la cella tra Tuki e Blinky */

tuki.riga = m_riga;
tuki.colonna = m_colonna;

if(nodo_a.n_sx >= 0)
  {
    n_d = grafo[nodo_a.n_sx];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_sx = distanza_celle(s, tuki);
      }
  }

if(nodo_a.n_dx >= 0)
  {
    n_d = grafo[nodo_a.n_dx];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_dx = distanza_celle(s, tuki);
      }
  }

if(nodo_a.n_su >= 0)
  {
    n_d = grafo[nodo_a.n_su];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_su = distanza_celle(s, tuki);
      }
  }

if(nodo_a.n_giu >= 0)
  {
    n_d = grafo[nodo_a.n_giu];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_giu = distanza_celle(s, tuki);
      }
  }

nodo_da =  nodo_a;
```

```
      int ind;
      /* cerco il nodo più vicino a tuki */
      if(d_sx<=d_dx && d_sx<=d_su && d_sx<=d_giu)
        {
          ind = nodo_a.n_sx;
          nodo_a = grafo[ind];

          ld = SINISTRA;
        }
        else if(d_dx<=d_sx && d_dx<=d_su && d_dx<=d_giu)
        {
          ind = nodo_a.n_dx;
          nodo_a = grafo[ind];

          ld = DESTRA;
        }
        else if(d_su<=d_sx && d_su<=d_dx && d_su<=d_giu)
        {
          ind = nodo_a.n_su;
          nodo_a = grafo[ind];

          ld = SU;
        }
        else if(d_giu<=d_sx && d_giu<=d_dx && d_giu<=d_su)
        {
          ind = nodo_a.n_giu;

          nodo_a = grafo[ind];

          ld = GIU;
        }
      colonna_old = colonna;
      riga_old = riga;

      return ld;
    }

  return ld;

}

direzione gioca_clyde(posizioni p, oggetto **lab)
{
  static int turno = 0;

  static int colonna_old,riga_old;

  turno+=1;

  /* indica l'edge su cui si sta muovendo il fantasma */
  static nodo nodo_da, nodo_a;

  int riga = p.clyde_y;
  int colonna = p.clyde_x;

  direzione ld =  FERMO;

  if(turno == 1)
    {
      collega_nodi();
    }

  /* uscita dalla casetta dei fantasmi */
  if(turno<7)
    {
      if(turno<5)
        return SU;

      nodo_a = grafo[13];
      nodo_da = grafo[13];

      colonna_old = -1;
```

```
    colonna = nodo_a.colonna;

    riga_old = -1;
    riga = nodo_a.riga;
    return DESTRA;
  }

/* Sono su un nodo? */
int inx = indice_nodo_cella(riga,colonna);

/* cella su cui si trova e cella precedente */
cella c, pr;
c.riga = riga;
c.colonna = colonna;
pr.riga = riga_old;
pr.colonna = colonna_old;

if(inx<0)
  {
    /*_____________________________________________________
     *|  Non sono su un nodo. Il percorso è obbligato
     *| sono definiti: nodo_da, nodo_a, x e y _old
     *| satbilisce il prossimo passo come l'unico
     *| possibile in modo che non sia ne un muro ne la
     *| posizione attuale
     */

    ld = prossima_cella(c, pr, lab);

    colonna_old = colonna;
    riga_old = riga;

    return ld;

  }
else
  {
    /*_____________________________________________________
     *|
     *| Sono su un nodo, devo decidere il prossimo
     *| edge su cui muovermi
     *| se non ci sono stati errori inx ==  nodo_a.indice
     *| in base alla posizione di tuki.
     *| Al termine di questo blocco devo:
     *| nodo_da <- noda_a
     *| nodo_a <- nodo_dest
     */

    /* Nodo temporaneo */
    nodo n_d;

    /* calcola la distanza tra i prossimi nodi e tuki */
    double d_sx = 10000, d_dx = 10000, d_su = 10000, d_giu = 10000;

    cella tuki,clyde;
    tuki.riga = p.tuki_y;
    tuki.colonna = p.tuki_x;
    clyde.riga  = p.clyde_y;
    clyde.colonna = p.clyde_x;

    double ds = distanza_celle(tuki,clyde);

    if(ds <= 8)
      {
        tuki.riga = 28;
        tuki.colonna = 3;
      }

    cella s;

    if(nodo_a.n_sx >= 0)
      {
```

```
    n_d = grafo[nodo_a.n_sx];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_sx = distanza_celle(s, tuki);
      }
  }
if(nodo_a.n_dx >= 0)
  {
    n_d = grafo[nodo_a.n_dx];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_dx = distanza_celle(s, tuki);
      }
  }
if(nodo_a.n_su >= 0)
  {
    n_d = grafo[nodo_a.n_su];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_su = distanza_celle(s, tuki);
      }
  }
if(nodo_a.n_giu >= 0)
  {
    n_d = grafo[nodo_a.n_giu];
    if(n_d.indice != nodo_da.indice && n_d.indice >= 0)
      {
        s.riga = n_d.riga;
        s.colonna = n_d.colonna;
        d_giu = distanza_celle(s, tuki);
      }
  }
    nodo_da =  nodo_a;

int ind;

/* cerco il nodo più vicino a tuki */
if(d_sx<=d_dx && d_sx<=d_su && d_sx<=d_giu)
  {
    ind = nodo_a.n_sx;
    nodo_a = grafo[ind];

    ld = SINISTRA;
  }
  else if(d_dx<=d_sx && d_dx<=d_su && d_dx<=d_giu)
  {
    ind = nodo_a.n_dx;
    nodo_a = grafo[ind];

    ld = DESTRA;
  }
  else if(d_su<=d_sx && d_su<=d_dx && d_su<=d_giu)
  {
    ind = nodo_a.n_su;
    nodo_a = grafo[ind];

    ld = SU;
  }
  else if(d_giu<=d_sx && d_giu<=d_dx && d_giu<=d_su)
  {
    ind = nodo_a.n_giu;
    nodo_a = grafo[ind];

    ld = GIU;
  }
```

```
      colonna_old = colonna;
      riga_old = riga;

      return ld;

    }

  return ld;
}
```

Listato gioca_tuki.c

```
/*_____________________________
*|
*| gioca_tuki.c
*/

#include "tuki5_modello.h"
#include <stdio.h>
#include <unistd.h>

direzione gioca_tuki(posizioni p, oggetto **labx)
{
  direzione dir = FERMO;

  return dir;
}
```

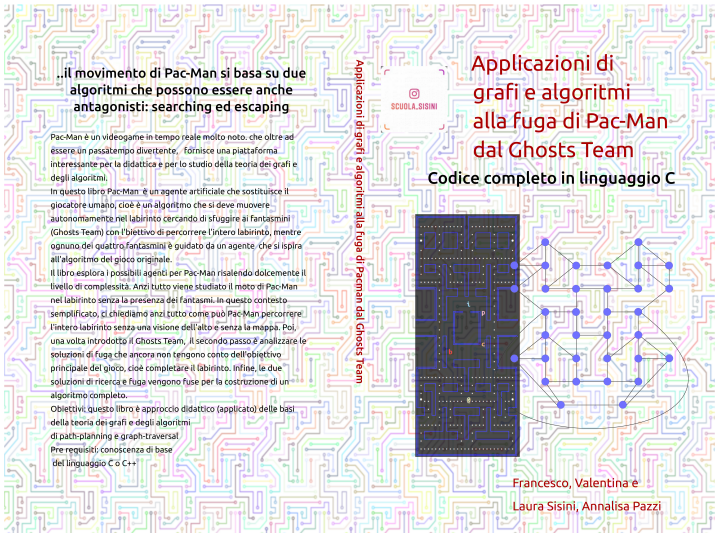
ULTIME CONSIDERAZIONI

Potrebbe sembrare una cosa eccessiva, ma il motivo per cui accettiamo delle sfide è quello di capire fin dove siamo capaci di spingerci nella vita e quale sia il significato della vita per noi.
Nei casi peggiori, ci accontentiamo di sfide affidate al caso e alla speranza di essere aiutati dalla fortuna, ma la dea bendata e cieca e il gioco d'azzardo non paga e non da soddisfazione.
Altre volte accettiamo sfide perché sappiamo a priori di poterle superare. Sono sfide che non fanno crescere.
La sfida migliore è quella che costringe a conoscere meglio se stessi perché presenta situazioni nuove che necessitano di sviluppare nuove abilità . I gioco-programmi fanno parte di queste ultimi. Sebbene limitati alla sfera intellettiva, costituiscono un valore che può essere speso per formare sé stessi ma anche gli altri.
A questo proposito, sono state interessanti le conversazioni avute con Michele Piattella, psicologo del lavoro, che ha suggerito molte nuove idee interessanti per l'applicazione dei giochi nell'ambito del problem solving.
Ci auguriamo che dalla nostra conoscenza e amicizia, possa nascere una collaborazione che magari sfocerà in nuovi giochi e prodotti.

BIBLIOGRAFIA E RIFERIMENTI

- Algorithm of ghost behavior in the game Pac-Man: https://weekly-geekly.github.io/articles/109406/index.html

ALTRE PUBBLICAZIONI DI SCUOLA SISINI



Applicazioni di grafi e algoritmi alla fuga di Pac-Man dal Ghost-Team

Introduzione alle reti neurali con esempi in linguaggio C

Reti neurali non supervisionate: il cognitrone di Fukushima