
**Universitá degli studi di Padova**
Facoltá di ingegneria

Corso di laurea in Ingegneria Informatica

# *Snap!: un efficace ambiente web-based per introdurre la programmazione*

**Relatore:** Prof. Michele Moro

**Laureando:** Fabio Maiani

Matricola: 611703 IF

Anno accademico 2012 - 2013

# Indice

# Introduzione

L'obbiettivo di questa tesi è di illustrare l'ultima versione del programma BYOB, nota come Snap!, attualmente ancora disponibile solo in versione beta, e di mostrare come tutti i programmi che derivano dal vecchio ma ancora ben noto linguaggio Logo, Snap! compreso dunque, possano fornire un approccio introduttivo al campo della robotica. Di seguito viene riportata una descrizione del contenuto dei capitoli.

- Inizialmente verrà presentata una breve cronologia storica per mostrare i vari passi che sono stati fatti per arrivare alla creazione di Snap!, a partire dal suo linguaggio padre, ovvero il famoso Logo, linguaggio che ancora ai giorni nostri influenza molti linguaggi di programmazione, progettati soprattutto a scopo educativo.

- Si proseguirà con un'analisi sulle sostanziali differenze fra i linguaggi derivati da Logo, di ultima generazione, ovvero Scratch, BYOB e Snap!, incentrandosi ovviamente di più su quest'ultimo, per meglio capire le ultime innovazioni che lo riguardano.

- Vista la sua ampiezza e importanza, verrà dedicato un intero capitolo ai vari metodi di salvataggio che Snap! permette.

- Verrà poi illustrata una veloce panoramica su come i linguaggi discendenti da Logo, possono essere utili nell'apprendimento dei passi fondamentali all'introduzione di una scienza che trova applicazione in molteplici contesti: la robotica. Essa viene definita come la scienza che si ispira alla natura e si occupa di studiare e sviluppare metodologie. che permettano ad un robot di eseguire dei compiti specifici. Nonostante la robotica sia una branca dell'ingegneria, in essa confluiscono approcci di molte discipline sia di natura umanistica, come linguistica e psicologia, che scientifica: biologia, fisiologia, automazione, elettronica, fisica, informatica, matematica e meccanica.

- Infine verrà illustrato come esempio applicativo, all'interno del quale si potranno scorgere alcuni degli aspetti mostrati riguardo all'ottica dei sensori presentata nel punto precedente ed ovviamente molte altre potenzialità dei linguaggi discendenti da Logo, una versione del noto videogame Pacman.

Questa tesi vuole dunque anche cercare di avvicinare il lettore ad un ambito tutt'altro che semplice come quello della robotica, mostrando con semplici esempi, che grazie a linguaggi come Scratch, BYOB e Snap!, ma soprattutto grazie al loro antenato Logo da cui tutti derivano, è possibile sviluppare un'ottica e una modesta esperienza che permettono a chi volesse proseguire il proprio studio in quell'ambito, di affrontarlo con minor difficoltà, avendo per lo meno inquadrato alcuni aspetti di base di tale scienza.

# Capitolo 1

# La storia di Snap!

## 1.1 Logo

Partendo dagli albori dell'informatica, troviamo il primo vero e proprio antenato di Snap!: Logo. Logo è un linguaggio di programmazione che è stato creato, nella sua prima versione, al MIT (Laboratorio di Intelligenza Artificiale) di Boston nel 1967 da un gruppo di ricercatori in cui operava anche il matematico Seymour Papert, già collaboratore dello psicologo svizzero Jean Piaget, studioso dei processi di apprendimento. Logo si basa sulla filosofia del costruttivismo, che concepisce l'apprendimento come un processo di esplorazione, di creazione e di costruzione e lo sviluppo della conoscenza come interazione con le altre persone e col mondo circostante. Logo è un linguaggio molto potente, che permette agli operatori esperti di creare progetti sofisticati e complessi, e di avere nel contempo una bassa soglia di accesso che ne consente l'uso anche ai bambini della scuola primaria. Fra gli ambienti di apprendimento che Logo offre, quello più conosciuto e sicuramente più usato nelle scuole, anche in Italia, è la Geometria della Tartaruga. Originariamente la Tartaruga era un robot che si muoveva su una superficie tramite comandi impartiti attraverso un computer. In seguito divenne uno strumento grafico, fu trasferita sul monitor del computer ed usata per disegnare e creare immagini, come ad esempio quella di figura 1.1.

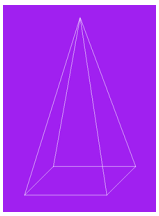

Figura 1.1: Esempio di immagine creata con Logo

Col passare degli anni Logo è stato prodotto in varie versioni ed ha subito alterne vicende in relazione all'evolversi del mondo dell'informatica e dei personal computers. A partire dal 1980, anche a seguito del successo del libro Mindstorm pubblicato dal suo ideatore Papert, Logo è stato tradotto in diverse lingue e si è diffuso in molte scuole, dal Nord America, al Sud America, al Giappone, all'Europa. In quell'anno fu fondata anche la società LCSI (Logo Computer System Inc.) in cui confluirono vari ricercatori e studiosi di questo linguaggio. Venne prodotta una nuova versione di Logo per Apple ed una per altri computers. Tuttavia a partire dal 1985 Logo è stato in buona parte soppiantato anche nel mondo della scuola dall'allora nuovo sistema MS-DOS. Nel 1993 LCSI ha diffuso MicroWorlds, una nuova versione innovativa di Logo per Apple alla quale erano state aggiunte funzioni multimediali che consentivano di inserire nei progetti svariati tipi di file, come immagini, suoni, video, testi e altro ancora. Alcuni anni dopo è stata creata anche la versione di MicroWorlds per Windows. Alla base di quest'ultimo sistema multimediale appena accennato vi è proprio Logo: circa 250 primitive con le quali si possono costruire procedure i cui nomi possono poi essere inseriti in ulteriori procedure. Le primitive Logo sono raccolte nel Vocabolario Logo che mostra anche la sintassi ed esempi per l'uso. L'insieme delle primitive spaziava un po' su tutto: grafica, oggetti, videoscrittura, gestione dello schermo, accesso al disco, logica e controllo del flusso, variabili, matematica, input, la musica, il tempo e lo spazio di lavoro. Alcuni comandi risultano essere molto semplici, al punto che possono essere usati con efficacia anche da bambini di scuola elementare, mentre altri comandi non lo sono affatto: l'uso di raffinate funzioni logico-matematiche e quello di variabili, consentono di realizzare progetti elaborati e complessi.

## 1.2 Scratch

Un vero e proprio erede di Logo, in quanto ne è stato molto influenzato, è Scratch. Scratch è un linguaggio di programmazione visuale, ovvero un linguaggio che consente la programmazione tramite la manipolazione grafica di elementi e non invece tramite la classica sintassi scritta. Proprio per questo esso risulta molto semplice da usare, soprattutto ai livelli iniziali per usi didattici, dove gli studenti non devono necessariamente conoscere la sintassi di un linguaggio per poterlo usare, basta solo un po' di logica. La prima versione di Scratch fu sviluppata nel 2006 dal Lifelong Kindergarten group, guidato da Mitchel Resnick al laboratorio del MIT. Scratch permette lo sviluppo di semplici programmi interattivi, come animazioni, giochi e molto altro. La strategia utilizzata è quella del *drag and drop*, attraverso il quale è possibile comporre frammenti di codice, semplicemente trascinando determinati blocchi di codice già pronti ed unendoli fra loro. Attualmente

la versione in uso è Scratch 1.4, ma è già comunque in progettazione una versione successiva, ovvero Scratch 2.0.

## 1.3 BYOB

Byob è una vera e propria estensione di Scratch, allargata ed arricchita in modo che il risultato, cioè BYOB stesso, sia un linguaggio di programmazione più potente. Quali sono queste innovazioni? A questa domanda verrà risposto successivamente, nella sezione 2.2, cercando di dare una panoramica globale di questi nuovi contenuti non eccessivamente lunga, ma comunque fondamentale per poter poi capire molti meccanismi nell'associazione di questo linguaggio con l'ambiente della robotica. L'ultima versione definitiva di BYOB è stato BYOB 3.1, ancora comunque largamente utilizzato, dato che Snap! invece è ancora in versione di beta.

## 1.4 Snap!

Qualcuno all'inizio potrebbe pensare che Snap! sia a sua volta un'estensione di BYOB: ebbene non è così. Snap! è semplicemente una nuova versione di BYOB, non è una versione più estesa. Come spiega Brian Harvey (professore alla Berkeley ed uno dei principali sviluppatori di BYOB assieme a Jens Monig) nel forum dedicato a BYOB, la versione che sarebbe stata BYOB 4.0 è stata invece rinominata come Snap! 4.0. Harvey spiega che ciò è semplicemente dovuto al fatto che a due professori nel team, Harvey si astiene nel nominarli, non piaceva il nome BYOB in quanto questo acronimo ha svariati significati(alcuni anche non proprio belli per così dire) e per questo essi hanno fortemente lottato per il cambiamento del nome. Harvey pur di non creare scompiglio e malumori all'interno del team ha accettato, non senza però esprimere il proprio disappunto sul fatto di non riuscire a capire il problema in questione, in quanto un cambio di nome di sicuro non è la soluzione a tutti i problemi del mondo. Dopo questa breve parentesi è bene sapere che Snap! è stato scritto in Javascript, a differenza di BYOB e Scratch che invece sono stati scritti in Squeak.

# Capitolo 2

# Differenze ed innovazioni fra i linguaggi

Credo sia utile, al fine di avere una maggior comprensione dei prossimi argomenti, avere ben chiaro in testa come rispettivamente si differenziano fra loro i linguaggi Scratch, BYOB e Snap!. In questa sezione verrà dunque fornita una breve descrizione di quali sono le funzioni permesse, e quali invece no, da ognuno dei linguaggi prima citati. Questa illustrazione dovrebbe servire a farsi un'idea di quanto potente risulta ciascun linguaggio ed in più intuirne così i suoi limiti e/o funzionalità esclusive che esso può utilizzare. Così facendo, una volta che ci si trova di fronte ad un problema si tenterà di risolverlo cercando di usare la migliore delle alternative che ognuno dei linguaggi dispone.

## 2.1 Scratch, l'origine

Dei tre linguaggi in questione, si può intuire che Scratch è quello da cui partire. Esso permette la costruzione di determinati script, volendo anche abbastanza complessi, ed inoltre permette anche il parallelismo fra sprite, ovvero la possibilità di un'esecuzione parallela fra gli script di due o più sprite diversi o di più script appartenenti ad uno stesso sprite. Tuttavia Scratch risulta privo della funzione di ricorsione, uno strumento spesso utilissimo nell'informatica, il che limita decisamente le sue potenzialità. Ciò non toglie comunque il fatto che Scratch sia un linguaggio di tutto rispetto, con il quale si possono realizzare progetti notevoli anche nel campo della robotica.

## 2.2 BYOB estende Scratch

BYOB ha portato molte nuove funzionalità con la sua nascita. La sua innovazione più importante è senza ombra di dubbio la possibilità di poter creare un proprio blocco personalizzato (da cui ne deriva anche il nome, Build Your

Own Blocks), in base ad una qualsiasi esigenza si abbia: per farlo si crea uno script (prototipale), dotato di parametri specifici ed eventualmente di un valore di ritorno, utilizzando blocchi già forniti di base oppure blocchi creati precedentemente. Una volta creato un blocco è possibile usarlo, modificarlo e nel caso non fosse più utile anche eliminarlo. Assieme a questo concetto è stata anche colmata la lacuna di Scratch, in quanto BYOB permette la costruzione di blocchi ricorsivi. Altro particolare per cui BYOB si distingue da Scratch è il fatto di considerare il blocco lista come tipo di dato di prima classe. Si ricorda che, in un linguaggio di programmazione, un tipo di dato di prima classe è un dato che può essere:

1. il valore di una variabile
2. l'input di una procedura
3. il valore restituito da una procedura
4. un membro di un aggregato di dati
5. anonimo (cioè un dato a cui non è ancora stato assegnato un nome identificativo)

Su BYOB inoltre sono state aggiunte tre tipologie di parametro, ovvero un parametro può essere anche una lista, una procedura o un oggetto. A sua volta una procedura (che può essere un blocco o uno script di blocchi) su BYOB è un dato di prima classe, ovvero una procedura può anche essere passata come input ad un blocco. Sulla base dell'affermazione precedente, dove si nominava un oggetto, si intuisce che BYOB permette addirittura la programmazione ad oggetti, strumento molto utile e spesso usato nell'associazione di BYOB con la robotica. E' bene tenere presente però che molti dei nuovi elementi di BYOB non sono delle vere e proprie novità. BYOB ha introdotto rispetto a Scratch, svariate funzionalità, molte delle quali comunque erano già usate diversi anni prima dai vari ambienti di Logo disponibili.

## 2.3 L'arrivo di Snap!

Grossolanamente si può dire che Snap!, essendo alla fine semplicemente la versione più recente di BYOB, possiede certamente delle innovazioni, ma non profondissime come nel passaggio da Scratch a BYOB. Vediamo dunque di seguito cos'è cambiato.

### 2.3.1 Gli anelli usati in modo esplicito

L'anello è un elemento che già esisteva con BYOB (veniva fuori automaticamente, trascinando i blocchi in una determinata posizione nello script),

ma non poteva venire usato direttamente come invece è possibile su Snap!. Gli anelli con Snap!, sono infatti ora direttamente reperibili nella categoria *Operators* del menu di Snap!.

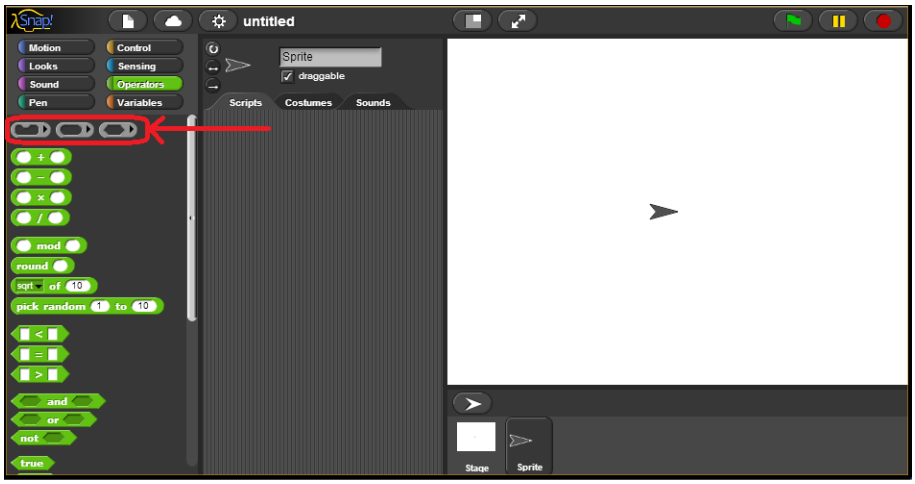


Figura 2.1: Gli anelli disponibili su Snap!

Oltre a questo particolare, il funzionamento di un anello rimane esattamente analogo a quello che aveva su BYOB.

### 2.3.2 Le continuazioni

Tramite l'uso di anelli è possibile definire un ulteriore nuovo concetto che Snap! mette a disposizione: le continuazioni.

La continuazione di un blocco all'interno di uno script è ciò che rimane da eseguire, per terminare l'intero script, al termine dell'esecuzione del blocco in questione. Volendo si possono vedere le continuazioni come dei sottoinsiemi dell'intera esecuzione di uno script. In generale, le continuazioni vengono rappresentate come degli script all'interno di un anello.
Inizialmente, per capire meglio il concetto di continuazione, si consideri uno script privo di cicli: a sinistra in figura 2.2 vi è lo script di esempio, mentre a destra è riportata la continuazione del blocco [*move 100 steps*].

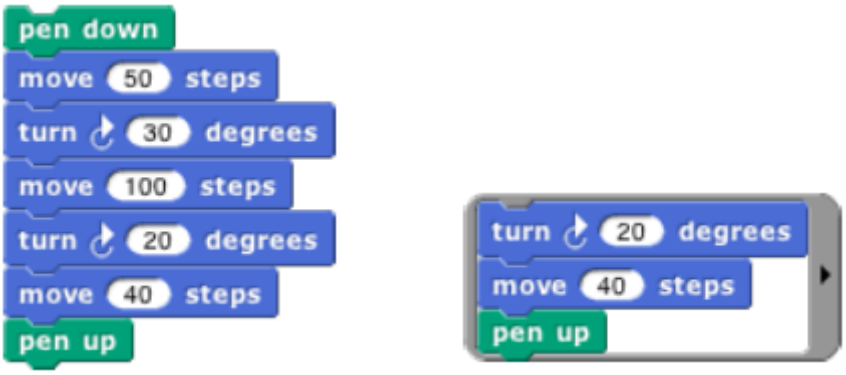


Figura 2.2: Continuazione di [move 100 steps] dello script di esempio

E' facile notare infatti che ciò che rimane da eseguire nello script appena fornito, dopo che il blocco [*move 100 steps*] ha terminato il suo lavoro, sono esattamente quei tre blocchi specificati dalla continuazione.

Le cose si complicano leggermente, quando si ha a che fare con la continuazione di un blocco contenuto in un ciclo. Questo perchè il blocco in questione, essendo all'interno del ciclo, verrà sicuramente eseguito più volte prima del termine dello script. Dunque, ogni volta che tale blocco verrà eseguito, la sua relativa continuazione sarà diversa, perchè ogni volta la parte computazionale rimanente sarà inferiore (rimarrà un ciclo in meno da compiere). Per comprendere meglio, si consideri lo script di figura 2.3.


Figura 2.3: Esempio di script con un ciclo

In relazione allo script appena fornito sopra, in figura 2.4 sono mostrate le continuazioni del blocco [*turn 50 degrees*] nella prima esecuzione del ciclo (a sinistra) e nella terza esecuzione del ciclo (a destra).

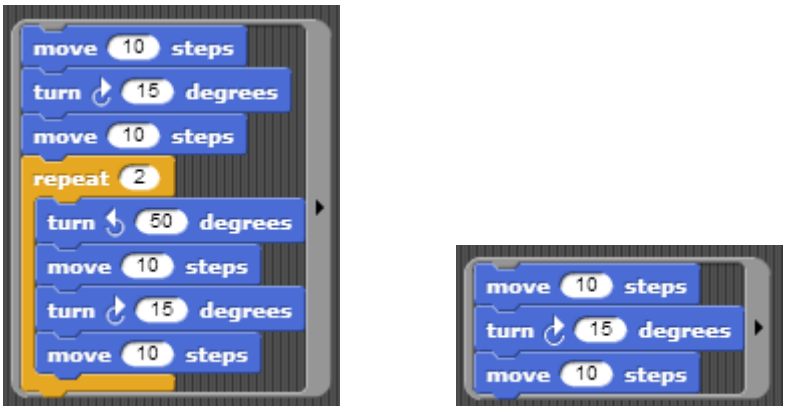


Figura 2.4: Prima e terza continuazione di [turn 50 degrees] dello script precedente

In pratica, considerando la successione di tutte le continuazioni del blocco [*turn 50 degrees*], il ciclo *repeat* ha, di volta in volta, un'esecuzione in meno da attuare (nella terza ed ultima continuazione il ciclo sparisce perchè non ci sono più ripetizioni da eseguire). Infatti quello che conta non è esattamente ciò che viene dopo il blocco in questione, bensì quale computazione rimane dopo di esso per completare l'intera esecuzione dello script.

Un'altra complicazione può insorgere: si immagini che un utente crei un suo blocco personalizzato, ovvero ne definisca lo script del prototipo. Una volta creato, si supponga che questo blocco venga inserito in un altro script (come spesso accade). Ora quale sarebbe la continuazione di un blocco del prototipo?
Questo tipo di continuazioni è particolare perché esse di norma sono costituite da blocchi appartenenti a script diversi; in questo caso ci saranno alcuni blocchi appartenenti allo script del prototipo e altri appartenenti allo script esterno in cui il blocco personalizzato è inserito. In figura 2.5, a sinistra sono presenti il prototipo e lo script esterno, mentre a destra è mostrata la continuazione del blocco [*move 50 steps*] del prototipo.

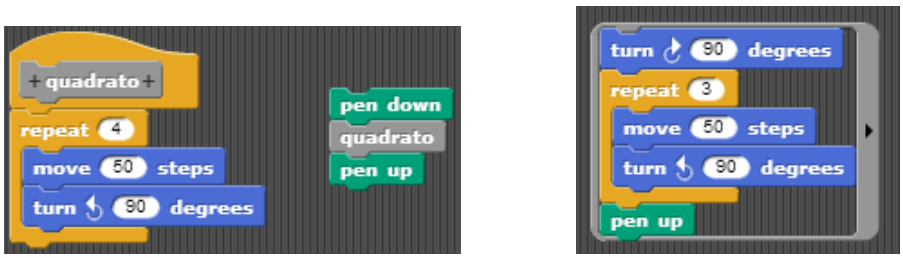


Figura 2.5: Continuazione di [move 50 steps] dello script di esempio

La spiegazione di quanto appena visto è molto semplice: nell'esecuzione

dello script esterno, quando arriverà il momento dell'esecuzione del blocco personalizzato *quadrato*, il programma comincerà ad eseguire il suo relativo prototipo. Terminata l'esecuzione del prototipo il programma ritornerà ad eseguire ciò che rimane dello script esterno. Quindi se si vuole la continuazione di un blocco del prototipo (che non sia l'ultimo), per forza ciò che rimane da eseguire, dopo di esso, sarà la parte del prototipo successiva al mio blocco scelto, con in più l'aggiunta della parte di script esterno che rimane da completare dopo l'esecuzione del mio blocco personalizzato. Ovviamente, nel caso si scelga di volere la continuazione dell'ultimo blocco di un prototipo, la sua continuazione sarà invece costituita da blocchi appartenenti solo allo script esterno (a meno che, nella parte computazionale rimanente dello script esterno, non si abbia fatto uso di ulteriori blocchi personalizzati).

Fin qui sono stati presentati esempi di continuazioni di command blocks, le quali, come si può notare, in pratica non hanno nessun input. Non è invece il caso dei reporter blocks. Questi blocchi sono usati all'interno degli script con lo scopo che, dopo una loro elaborazione interna, restituiscano un certo risultato. Tale risultato verrà poi utilizzato nella parte dello script successiva alla loro elaborazione. Quindi se si vuole una continuazione di un reporter block, si deve come sempre considerare la parte successiva alla loro esecuzione, ma ad essa andrà passato, tramite una variabile di input, l'esatto valore che restituirebbe il reporter block. La variabile ovviamente andrà ad occupare nello script l'esatta posizione che prima occupava il reporter block preso in considerazione. Come al solito un esempio semplifica la comprensione: in figura 2.6, in alto vi è uno script di esempio con dei reporter blocks, mentre subito sotto è mostrata la continuazione del blocco (*8 + 3*). La variabile *risultato* conterrà in questo caso il valore finale dell'elaborazione del blocco (*8 + 3*), ovvero 11.


Figura 2.6: Continuazione di (8 + 3) dello script di esempio

Ovviamente il nome della variabile di input della continuazione è assolutamente arbitrario, in quanto, nella normale esecuzione dello script, sarebbe una variabile interna nascosta.

### 2.3.3 Continuation Passing Style

Nel caso si volessero rappresentare con Snap! delle vere e proprie espressioni matematiche, risulta abbastanza ovvio il fatto che si avrà a che fare con l'uso di svariati reporter blocks (addetti alle operazioni matematiche), dove ognuno di questi restituirà un proprio risultato. Spesso però si avranno anche delle composizioni annidate di reporter block, o meglio reporter blocks all'interno di altri.

Ad esempio si consideri l'espressione 

. Leggendola normalmente sarebbe " tre per quattro più cinque ". Ma ovviamente Snap! prima somma 4 e 5, e poi moltiplica il tutto per 3. Quindi le operazioni sono svolte partendo dal blocco più interno fino ad arrivare a quello più esterno, al contrario di come sarebbe invece leggendo normalmente l'espressione da sinistra verso destra. Un altro modo di leggere l'espressione di prima potrebbe essere: prendi la somma di 4 e 5, e moltiplica tale somma per 3. In questo modo si leggerebbe l'espressione nell'esatto ordine con cui Snap! procede con i calcoli.
Ora in una piccola espressione ciò può sembrare abbastanza inutile, ma si immagini di essere al telefono con un amico al quale si deve comunicare l'espressione di figura 2.7.


Figura 2.7: Espressione complessa

Se si comunicasse: "il fattoriale di tre volte il fattoriale di due più due più cinque", l'amico dall'altra parte del telefono potrebbe capire un'espressione sbagliata, come una qualsiasi di quelle specificate in figura 2.8.

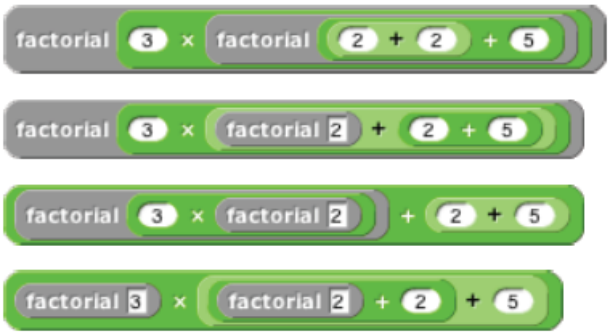


Figura 2.8: Espressioni malamente interpretate

La comprensione al telefono risulterebbe sicuramente migliore se si dicesse "Somma due più due, fai il fattoriale di questa somma, a questo aggiungi cinque, moltiplica il tutto per tre, e infine fai il fattoriale di tutto il risultato".

Su Snap! è possibile ottenere un simile riordinamento delle espressioni: bisognerà definire delle nuove versioni di tutti i reporter blocks usati per le operazioni. Queste nuove versioni riceveranno, oltre ai loro soliti parametri per il calcolo (addendi,fattori,ecc.), anche le loro continuazioni come input esplicito. Le nuove versioni create non saranno più dei reporter blocks, bensì saranno dei command blocks. La loro ridefinizione è mostrata di seguito.

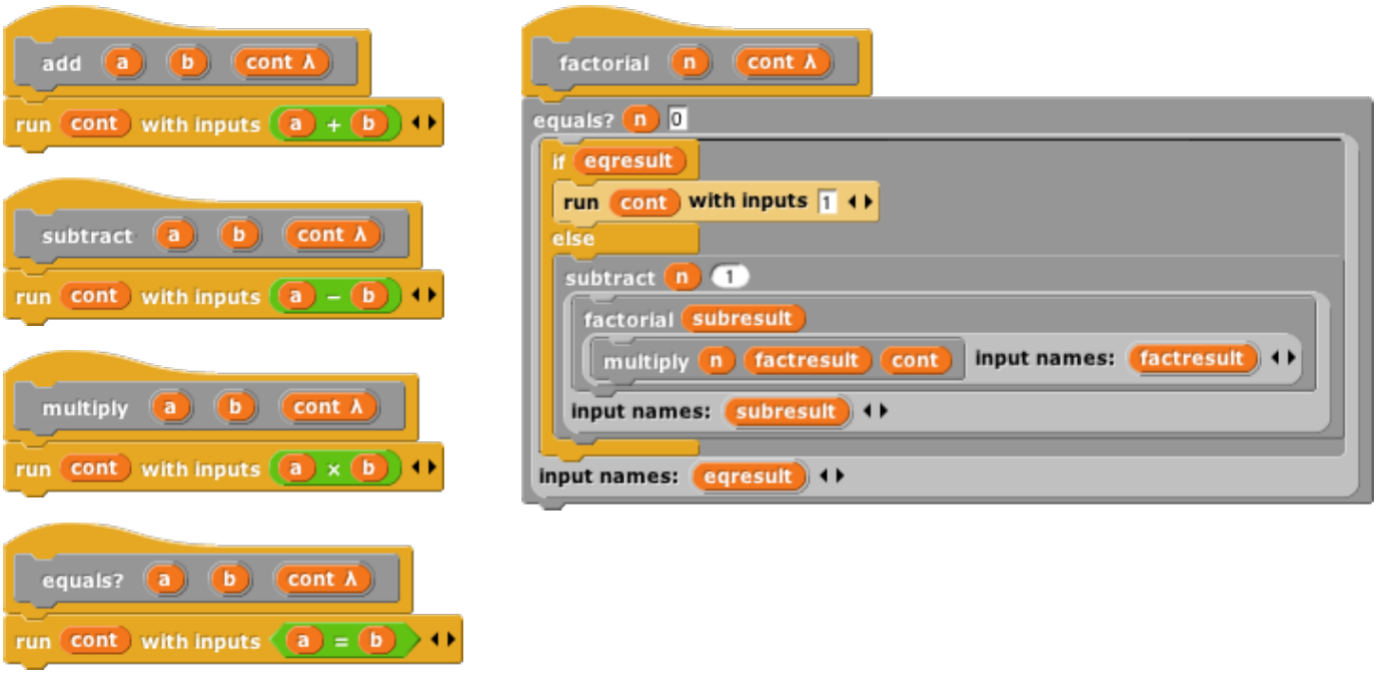


Figura 2.9: Ridefinizione dei blocchi per i calcoli

Prendiamo ad esempio il blocco *add*: facendo riferimento ancora a figura 2.9, esso riceve in input due addendi $a$ e $b$ ed una continuazione. In sostanza esso esegue una chiamata alla continuazione, fornendole come input il risultato della somma dei due addendi ricevuti. Analogo funzionamento per i blocchi *subtract*, *multiply* e *equals?* (ovviamente non si avrà più l'operazione di somma fra i due numeri $a$ e $b$ ricevuti, ma rispettivamente una differenza, un prodotto ed una verifica di uguaglianza).
Il comportamento del blocco *factorial* è invece leggermente diverso, e risulterà più comprensibile fra poco. Per il momento lo si prenda per buono.

Ora l'espressione complessa (quella in figura 2.7), può ora essere rappresentata nel seguente modo.

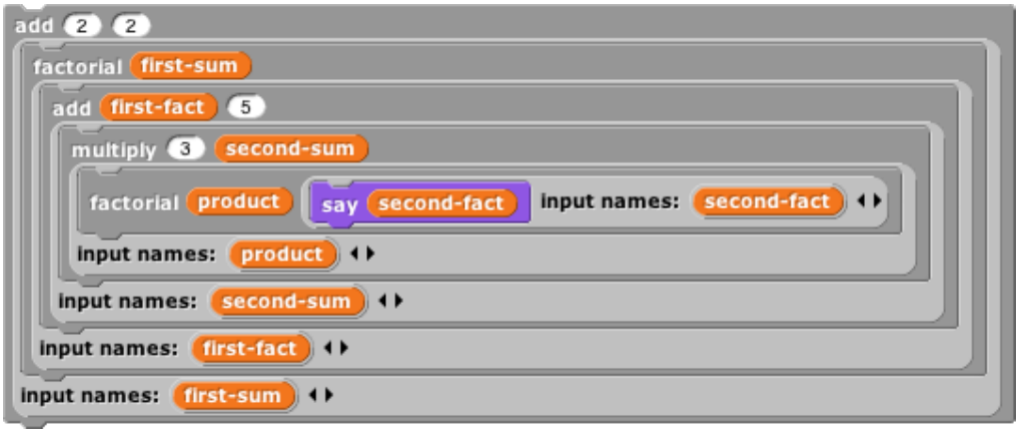


Figura 2.10: Script piu leggibile dell'espressione

Si nota subito che la lettura dello script, dall'alto verso il basso, è esattamente ordinata in modo da avere una lettura più comprensibile, come prima segnalato. Si può vedere il funzionamento in questo modo:

- il blocco *add* più esterno, prende i due numeri forniti (2 e 2), calcola la loro somma, e ne passa il risultato, contenuto nella variabile *first-sum*, al blocco *factorial*
- il blocco *factorial* eseguirà la sua operazione di fattoriale sulla variabile *first-sum*, scriverà il risultato sulla variabile *first-fact*, e passerà tale variabile al blocco più interno
- questo sistema si ripete fino a raggiungere il blocco *say* che mostrerà sullo schermo il risultato finale di tutta l'espressione.

Questo metodo di costruzione di script, in cui ogni command block riceve una continuazione come suo input, è detto *continuation-passing style*. Cer-

tamente risulta orribile a prima vista, ma ha comunque i suoi pregi per la comprensione. Un pregio importante, ad esempio, è che ogni script eseguito è in sostanza costituito da un singolo blocco, il quale, una volta completata la sua specifica operazione, delega tutto il resto del problema ad un blocco più interno.

Tornando ora al blocco *factorial* (sempre in figura 2.9), ora che si conosce il funzionamento generale di un *continuation passing style*, si noterà che all'interno del blocco *factorial* viene proprio usata questa tecnica. L'unica differenza sta nel fatto che al suo interno, durante l'uso analogo della tecnica del *continuation-passing style*, vi è una chiamata ricorsiva al blocco *factorial* stesso.

### 2.3.4 Blocchi specifici per le continuazioni

Come appena visto, per poter usufruire del *continuation passing style* è però necessario crearsi tutti i blocchi che eseguano le operazioni volute, come ad esempio il blocco *add*, il blocco *multiply*, il blocco *factorial* e così via.
Si prenda per esempio la creazione di un blocco ricorsivo per realizzare l'operazione di moltiplicazione, il quale riceve in input una lista di numeri e ne restituisce il loro prodotto, illustrato in figura 2.11.


Figura 2.11: Blocco prodotto con ricorsione

Per migliorare l'efficienza dello script, come si può notare, è stato inserito un controllo per verificare se uno dei numeri avuti nella lista di input vale zero; in questo caso il risultato da restituire sarebbe sempre nullo ovviamente, indipendentemente da quali fossero gli altri fattori. Tuttavia questo script non è poi così efficiente come sembra: nel caso, ad esempio, che esso riceva come lista di input la sequenza (1,2,3,4,0,5), lo zero verrà rilevato solo alla quarta chiamata ricorsiva, come primo elemento della sublist (0,5). Quindi

lo script continua la sua esecuzione, quasi fino al termine della lista, poi, trovando lo zero, verrà eseguito il blocco *report 0*.

Ora qual'è la continuazione del blocco *report 0* dello script di figura 2.11, se la lista di input è appunto (1,2,3,4,0,5)?
Essa è mostrata in figura 2.12: si noti il fatto che i quattro reporter blocks annidati sono il frutto di tutte le chiamate ricorsive avvenute fino al rilevamento del numero 0 nella sequenza fornita in input (le chiamate ricorsive successive non avvengono). Dentro alla variabile *risultato* ci sarà invece, come visto in precedenza per le continuazioni dei reporter block, il valore restituito dal blocco *report 0*, ovvero esattamente zero.


Figura 2.12: Continuazione del blocco report 0

Quindi nonostante si sappia che il risultato sarà comunque zero, si andranno a fare ben quattro moltiplicazioni non necessarie, dovute a tutte le chiamate ricorsive avvenute prima di trovare l'elemento nullo.

Si può migliorare questo aspetto ridefinendo il blocco *prodotto* come in figura 2.13 e con l'utilizzo di un blocco ausiliario chiamato *aiuto prodotto* (in figura 2.14.


Figura 2.13: Blocco prodotto con continuazione

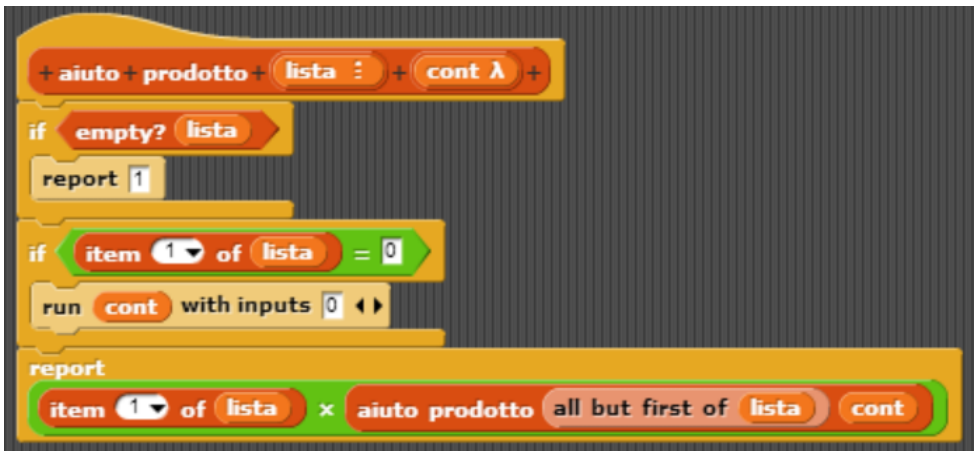


Figura 2.14: Blocco ausiliario per la funzione prodotto

Il blocco *aiuto prodotto* ha sostanzialmente lo stesso scopo dello script precedente di figura 2.11, tuttavia è leggermente diverso, in quanto in questo caso vi è l'utilizzo delle continuazioni, cosa che verrà chiarita fra poco.
La novità è sostanzialmente il blocco *call with continuation*, presente nel nuovo script del blocco *prodotto*. Il suo funzionamento generale è il seguente: esso riceve in ingresso un certo script, e andrà ad eseguirlo dandogli come parametro d'ingresso proprio la continuazione del medesimo script. Ora, facendo riferimento a figura 2.13, la continuazione dello script fornito come ingresso al blocco *call with continuation* è la seguente.


Figura 2.15: Continuazione dello script di input del blocco run with continuation

La variabile *risultato* contiene il valore finale dell'intera elaborazione che avviene con la chiamata del blocco *aiuto prodotto*; tale chiamata è effettuata dal blocco *prodotto*, come si vede dallo script di figura 2.13. La continuazione appena vista riporterà il risultato a chiunque abbia chiamato il blocco *prodotto* dall'esterno.

In pratica il funzionamento degli script *prodotto* e *aiuto prodotto* è questo:

1. se nella lista di input non c'è nessuno zero allora il sistema funziona esattamente come la sua versione precedente, in quanto non verrà mai usata la continuazione di figura 2.15, dato che il blocco *run* dello script *aiuto prodotto* non verrà mai eseguito.

2. se nella lista di input c'è invece uno zero allora

   - innanzitutto verranno fatte le chiamate ricorsive per gli elementi della lista (non nulli) che in ordine posizionale vengono prima dello zero (se lo zero è in prima posizione non verrà fatta alcuna chiamata ricorsiva ovviamente). Nel caso della nostra solita sequenza (1,2,3,4,0,5), verranno fatte quattro chiamate ricorsive prima di individuare lo zero.
   - quando verrà individuato lo zero, si entrerà nel secondo *if* del blocco *aiuto prodotto*, e si eseguirà, tramite il blocco *run*, la continuazione vista in figura 2.15, fornendole 0 come input.
   - la continuazione passerà immediatamente lo zero a chiunque abbia chiamato il metodo *prodotto*, senza effettuare tutte le moltiplicazioni non necessarie.

### 2.3.5 Uscite immediate dai cicli

In molti linguaggi di programmazione si usa un comando, di solito chiamato *break*, per permettere l'uscita da cicli di vario tipo come *while*, *for*, *repeat*, ecc. Snap! mette a disposizione un blocco appositamente per questo scopo, ovvero il blocco *catch*. Esistono due forme del blocco *catch*: un *catch* di tipo *command* ed un *catch* di tipo *reporter*.
Iniziamo con il primo dei due (mostrato in figura 2.16). Questo blocco ha come input una upvar (e una procedura che però in pratica è lo script che il blocco *catch* contiene). Lo scopo della upvar è questo: essa contiene come valore la continuazione esatta di tutto ciò che viene dopo il blocco *catch*; quindi effettuando una chiamata a questa variabile, tramite ad esempio il blocco *run*, si esce immediatamente dal *catch* e si continua l'esecuzione di tutto ciò che viene dopo. Questa chiamata può ovviamente essere fatta in un qualsiasi punto all'interno del *catch*.


Figura 2.16: Il blocco command catch

Un esempio è mostrato in figura 2.17. Vengono visualizzati i numeri 1,2,3,4 e poi si esce dal ciclo eseguendo l'output del blocco *say*.

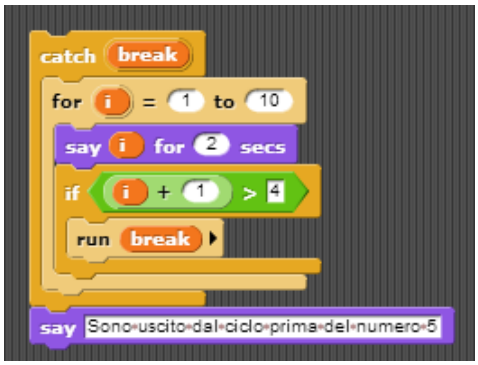


Figura 2.17: Esempio con il blocco command catch

Vediamo invece ora il secondo blocco *catch* (mostrato qui sotto).


Figura 2.18: Il blocco reporter catch

Per prima cosa è bene notare che, essendo in questo caso il *catch* un blocco di tipo reporter, esso dovrà restituire un certo valore dopo la sua elaborazione (il *catch* precedente, essendo di tipo command, non doveva restituire nulla). Ad esso viene associato un determinato blocco *throw*: questo blocco ha come parametri la solita upvar, per poter uscire dal *catch* in modo esattamente analogo a quanto visto prima, e un altro parametro (solitamente definito dall'utente), che sarà esattamente il valore restituito dall'intero *catch* nel caso il blocco *throw* venisse eseguito. Quindi se, durante l'esecuzione dello script all'interno del *catch*, non viene eseguito il blocco *throw*, il risultato sarà il calcolo esatto di tutta l'espressione fornita al blocco *catch*; in caso contrario, il risultato del *catch* sarà il valore specificato nel *throw*., ignorando l'intera espressione. Il seguente esempio ne è una prova.

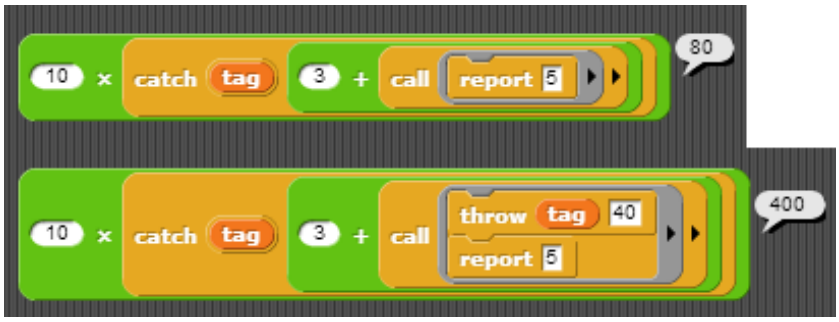


Figura 2.19: Esempio con il blocco reporter catch

Nel primo script di figura 2.19, dove non c'è il comando *throw*, avvengono questi procedimenti:

1. il blocco *report* restituisce 5
2. il blocco + restituisce 8
3. il blocco x restituisce 80

Invece nel secondo script di figura 2.19, dove si usa il comando *throw*, avviene questo:

1. il blocco *throw* restituisce 40 e si esce completamente dal blocco *catch*
2. il blocco x restituisce 400

### 2.3.6 Thread Systems tramite continuazioni

Come già anche su BYOB si poteva fare, anche con Snap! si possono eseguire più script all'interno di uno stesso sprite e, più in generale, più script in diversi sprite. Ovvio poi che l'esecuzione di tutti gli script non è esattamente contemporanea: Snap! infatti esegue una parte di script alla volta, saltando spesso da uno script all'altro.
Difatti alla fine di ognuno dei blocchi che rappresentano un ciclo (*for*, *forever*, *repeat*, ecc.) c'è un comando implicito ed invisibile che consiste in una vera e propria "cessione" dell'esecuzione (chiamata *yield* nei threading systems) ad un altro script. Ogni volta che avviene questa operazione (perché avvenga è necessario che ci siano più script su uno stesso sprite, altrimenti non succede nulla), Snap! deve ovviamente tenere in memoria lo stato dello script (in pratica i valori delle variabili calcolati fino ad allora) al momento della chiamata del comando prima citato.
C'è un ulteriore comando implicito alla fine di ogni script, una segnalazione di *end thread*, con il quale Snap! riconosce che lo script in questione ha terminato la sua esecuzione e quindi il programma può passare ad eseguire

un altro script senza però dover mantenere in memoria lo stato di quello che è appena terminato.
Tutto questo sistema descritto fin qui risulta automatico su Snap!: l'utente utilizza dei veri e propri thread systems senza doverne necessariamente conoscere il loro funzionamento. E' comunque bene conoscere questi particolari ed in più sapere che anche in questo contesto c'entrano le continuazioni. Infatti il procedimento seguito da Snap! è sostanzialmente quello di creare una lista inizialmente vuota, nella quale, ogni volta che sarà terminato un ciclo, verrà aggiunta la continuazione dello script considerato, a partire dal primo blocco successivo al ciclo. Successivamente verrà poi preso in considerazione l'elemento(la continuazione) della lista che aspetta da più tempo (il primo della lista), e ne sarà avviata l'esecuzione, segnando come prossimo elemento in attesa di essere eseguito, l'elemento successivo nella lista a quello appena selezionato. Infine il comando implicito di *end thread* semplicemente non salva nulla nella lista e verrà quindi presa ancora una volta la continuazione più vecchia della lista. Il processo si ripete fino a quando tutti gli script non hanno terminato la propria esecuzione.
Un altro particolare degno di nota, sempre riguardante i thread systems, è l'opzione *thread safe scripts*. Tale opzione è selezionabile dal menù opzioni di Snap!, ed è stata introdotta già con BYOB (su Scratch invece non era presente). Su Scratch infatti vi erano dei problemi sulla gestione della concorrenza dei messaggi broadcast: in pratica se, durante l'esecuzione di uno script, dall'esterno arrivava un messaggio broadcast, tale script veniva bloccato nella sua esecuzione (ancora incompleta) e successivamente veniva ricominciata la sua esecuzione da zero.
Spuntando invece l'opzione *thread safe scripts*, su BYOB o su Snap!, quando avvengono due eventi sovrapposti, il secondo viene semplicemente ignorato, permettendo così al primo di terminare normalmente la sua esecuzione. Questa soluzione non è perfetta ma almeno evita di lasciare il progetto in uno stato inconsistente.

### 2.3.7 L'esecuzione interamente su browser

La più rilevante di tutte le differenze è senz'altro il fatto che Snap! non necessita di una installazione su disco, in quanto funziona interamente su un qualsiasi browser si abbia a disposizione. Basta semplicemente cliccare sul link `http://snap.berkeley.edu/snapsource/snap.html#open:http://snap.berkeley.edu/snapsource/tools.xml` e vi si aprirà la schermata di lavoro di Snap!.

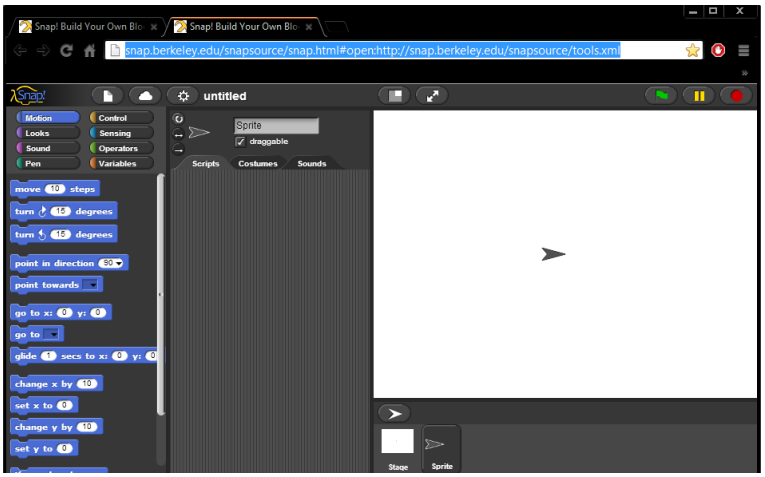


Figura 2.20: Interfaccia su browser di Snap!

Ciò risulta essere indubbiamente molto utile, in quanto basta semplicemente disporre di una connessione ad Internet per poter lavorare ad un vostro progetto, indipendentemente da dove vi troviate o che tipo di dispositivo usiate per continuare il vostro lavoro. Come già menzionato precedentemente, Snap! è stato implementato usando Javascript, il quale è stato progettato con lo scopo di limitare la possibilità di danneggiamento dei dati da parte di software browser based, quindi c'è sicurezza anche nell'esecuzione dei progetti da parte di altre persone.
Vista questa particolare caratteristica di Snap! può però sorgere un dubbio. Cosa succede quando devo salvare i miei progetti? Quali metodi di salvataggio Snap! permette?
Ciò è proprio il tema principale del prossimo capitolo.

# Capitolo 3

# Metodi di salvataggio progetti su Snap!

E' di fondamentale importanza conoscere i possibili metodi con cui un utente di Snap! può salvare il proprio lavoro: questa fra l'altro risulta essere un'altra delle caratteristiche principali con cui Snap! si differenzia da BYOB . Innanzitutto c'è una distinzione da fare: Snap! lascia decidere se si vuole salvare il proprio progetto direttamente sul proprio computer, oppure se lo si vuole salvare online, nella pagina web di Snap!. La prima opzione, come si vedrà in seguito, si differenzia ulteriormente in altri due metodi di procedimento. Il vantaggio della seconda opzione è ovviamente il fatto che si può accedere al proprio progetto anche da un diverso pc oppure anche da un mobile device come un tablet o uno smartphone. L'altra scelta viene invece adottata per casi particolari di sicurezza, come per esempio l'assenza temporanea di Internet.

## 3.1 Salvataggio locale

### 3.1.1 Salvataggio locale tramite browser

Con questo metodo il progetto verrà salvato in un file speciale sul computer, il quale potrà essere letto solo ed esclusivamente dallo stesso computer ,con lo stesso browser, connesso alla stessa pagina web di Snap!: ecco come javascript protegge i progetti salvati da eventuali malware. Al momento del salvataggio del progetto si potranno salvare assieme ad esso delle note o appunti a riguardo con lo scopo di facilitare il lavoro la prossima volta che si riaprirà il progetto.
Il procedimento è il seguente:

1. cliccare sull'icona  del menù File di Snap!
2. selezionare l'opzione *Save as...* e vi apparirà la finestra di figura 3.1 con l'opzione *browser* già selezionata di base.

3. Inserire il nome voluto nell'apposita casella di testo,cliccare sul tasto *Save* ed il gioco è fatto!

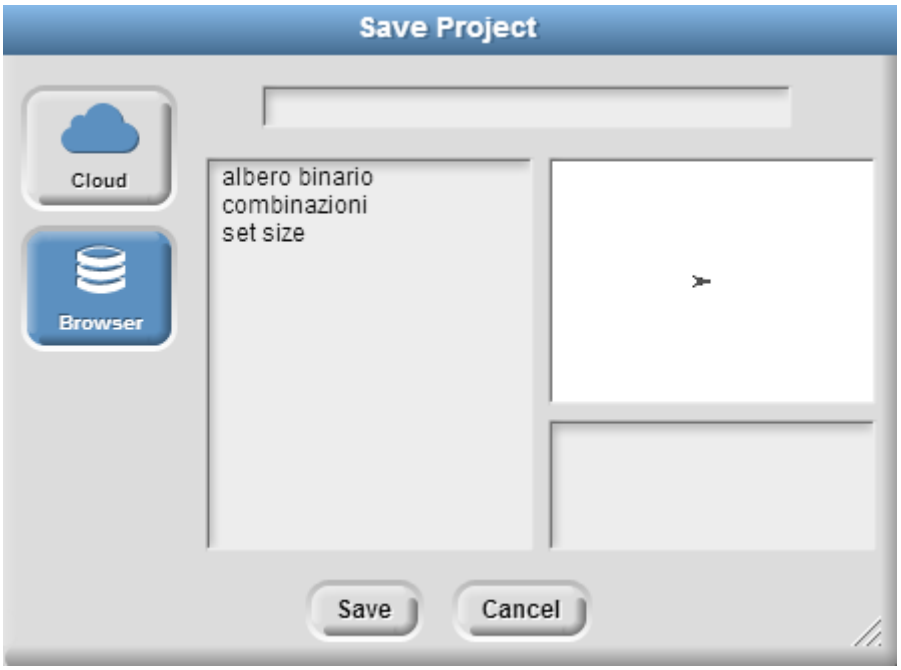


Figura 3.1: Il salvataggio locale su Snap!

Come si può notare sempre dalla figura 3.1, tutti i vostri progetti salvati saranno visibili appena sotto alla casella di testo dove avete inserito il nome del progetto. Se si ha un progetto al quale si lavora individualmente e solo esclusivamente da una postazione di lavoro fissa, questo è senz'altro il metodo indicato per essere ed è sicuramente comodo in certi casi. Tuttavia questo metodo di salvataggio presenta un inconveniente piuttosto scomodo: il limite di memoria del browser! Si può ovviamente cambiare questa opzione nel menù delle Preferenze del browser, ma anche così facendo, il metodo di salvataggio locale è possibile solo per un piccolo numero di progetti. Oltretutto se il browser fosse impostato con l'opzione di bloccare i cookies dai siti web, allora il salvataggio locale non sarebbe nemmeno possibile!

Per poter effettuare il passaggio contrario al salvataggio, ovvero il caricamento di un progetto, basterà selezionare l'opzione *Open...* dal solito menù File, selezionare il progetto desiderato e cliccare sul tasto *Open*. Nel caso ci si volesse invece disfare di uno dei vostri prodotti, il passaggio è analogo solo che si clicca sul tasto *Delete*.

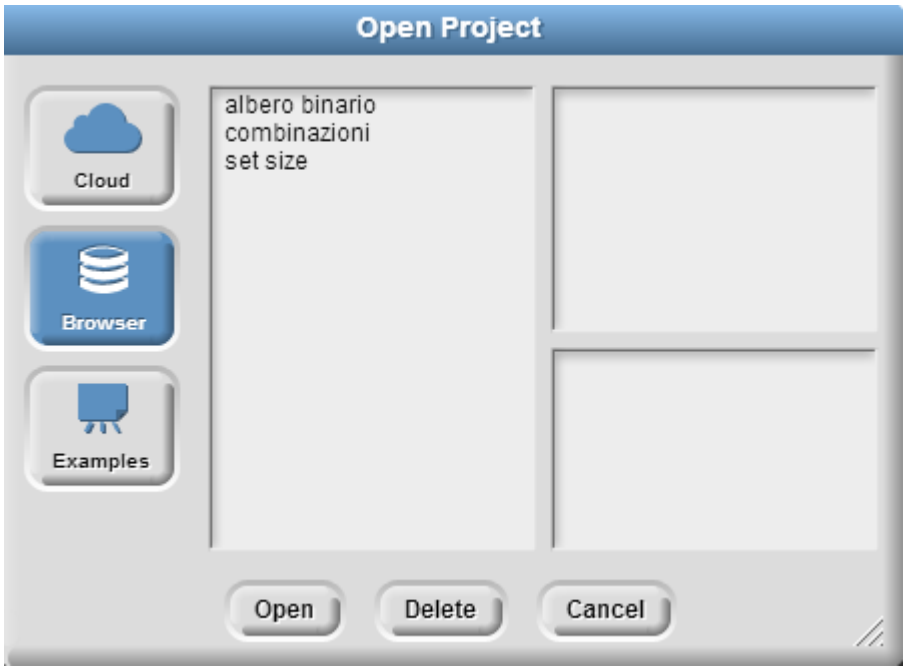


Figura 3.2: Il caricamento locale su Snap!

### 3.1.2 Esportazione XML

Scegliendo l'opzione *Export Project...* sempre dal menù File verrà aperta una nuova finestra di dialogo mostrata in figura 3.3, la quale richiede di dare un nome al vostro progetto.

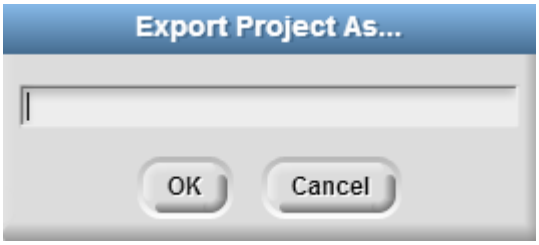


Figura 3.3: Finestra esportazione XML

Una volta confermato il tutto si aprirà questa volta una nuova finestra (figura 3.4) del vostro browser, la quale conterrà il progetto svolto finora, in linguaggio XML.

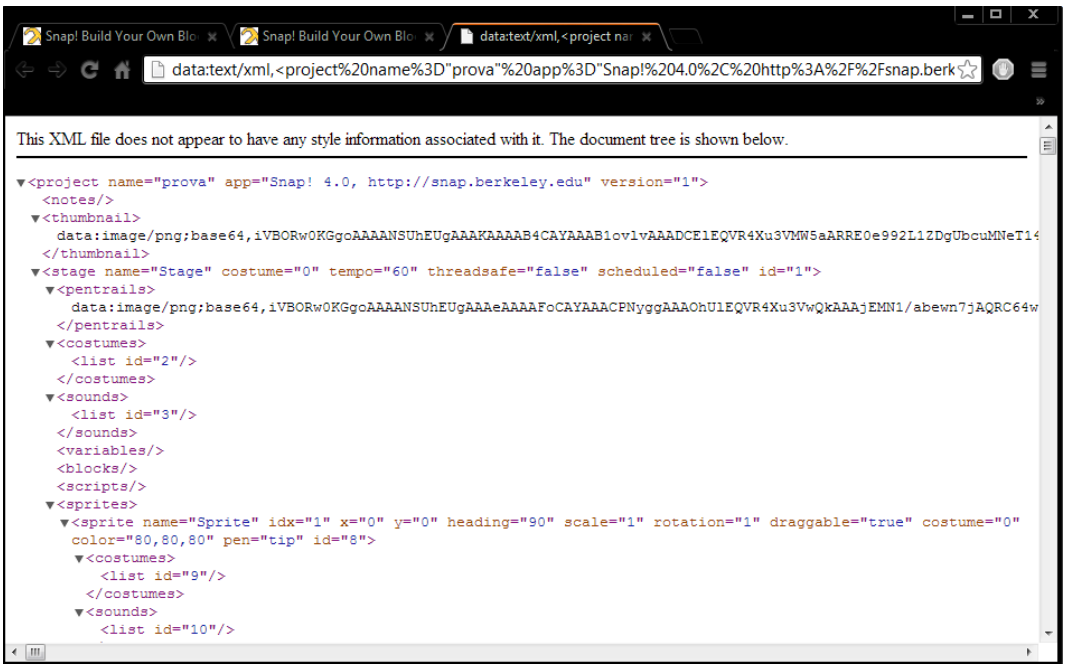


Figura 3.4: Codice XML

Ora basterà solo salvare questa pagina (anche con il classico comando di salvataggio CTRL + S del browser), selezionare la cartella di destinazione di salvataggio e dare un nome al vostro file. Questo secondo metodo non soffre delle limitazioni del precedente: i progetti in questo modo diventeranno dei normalissimi file nel computer e potranno essere condivisi con amici, essere aperti con un qualsiasi browser; oltretutto non ci sarà più nessun vincolo di memoria (a parte la dimensione massima del vostro hard disk, certo!).

Per il caricamento qui bisogna selezionare la voce *Import...* dal menù File e poi selezionare il file XML voluto. In alternativa, per far prima, basta trascinare il file XML nella finestra del browser di Snap!.

## 3.2 Memorizzazione online

C'è infine la possibilità di salvare tutto tramite Cloud nella pagina web di Snap!. In questo caso è richiesta la creazione di un piccolo account, ma è un'operazione facile e veloce, e che ovviamente va fatta solo la prima volta che si usa questo metodo di salvataggio. Basta cliccare sull'icona  e selezionare la voce *Signup...*


Figura 3.5: Finestra di signup di Cloud

Notare che nella finestra di registrazione al Cloud (figura 3.5), dato che Snap! è attualmente solo una versione beta, c'è un avvertimento del fatto che potrebbero esserci momenti di downtime del server di Snap!, nei quali ovviamente non sarà possibile raggiungere le vostre risorse salvate online. Molto probabilmente una volta che sarà rilasciata una versione definitiva di Snap! questi problemi saranno risolti.
Una volta inserito tutti i dati vi verrà inviata la password di login al vostro indirizzo di posta elettronica. Dopodichè basta andare nel menù Cloud , selezionare questa volta *Login* e inserite i vostri dati d'accesso. Un messaggio apparirà al centro dello schermo segnalandovi il successo dell'operazione di collegamento con il server di Snap!. Ora basterà semplicemente andare nel menù File, selezionare la voce *Save as...* ed automaticamente sarà resa disponibile l'opzione di salvataggio tramite Cloud. Si veda figura 3.6.

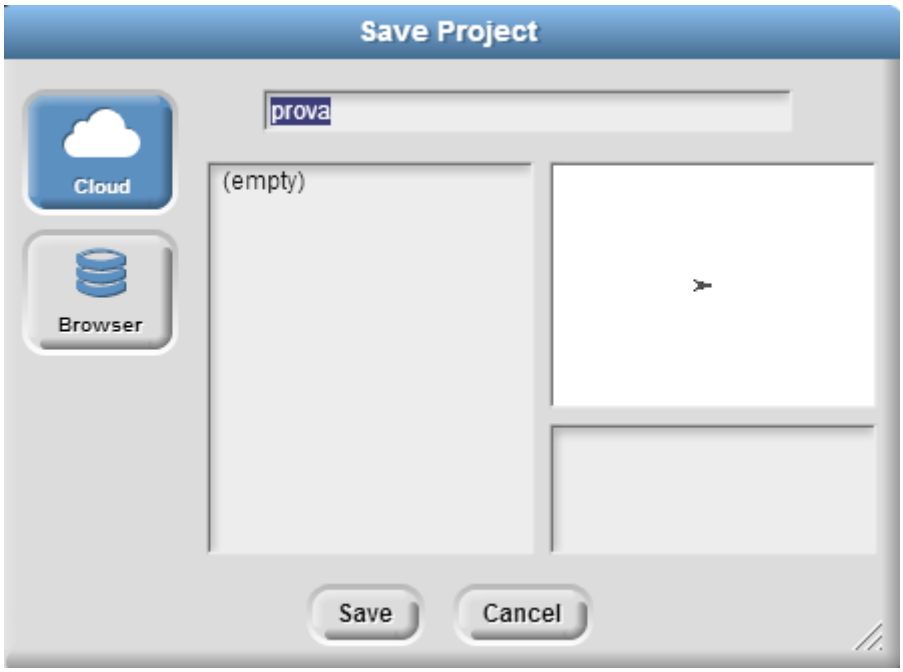


Figura 3.6: Il salvataggio online

Un progetto salvato in questo modo sarà accessibile da qualunque computer che abbia accesso ad Internet. Questo tipo di salvataggio è molto utile nel caso in cui si lavorasse spesso da postazioni diverse, come ad esempio uno studente che lavora in un'aula scolastica e che poi continua il lavoro da casa, o viceversa.

Anche il caricamento è analogo: basta selezionare la voce *Open...* del menù File e scegliere il file da aprire.

## 3.3 Esportazione di blocchi

Ora che si conoscono i metodi con cui salvare un certo progetto è utile sapere come Snap! si comporta con i blocchi creati e personalizzati dall'utente, ovvero i blocchi non di base, i quali invece si trovano ogni volta che apra Snap! tramite browser. In sostanza quando si salva un progetto (con un qualsiasi metodo di quelli illustrati precedentemente) tutti i i vostri blocchi personalizzati verranno a loro volta salvati con esso. Quindi se si riaprisse il medesimo progetto, si troverebbero comunque tutti i blocchi creati. Tuttavia, quando sono stati creati un certo numero di blocchi, a volte può risultare utile salvare solo ed esclusivamente questa collezione di blocchi (non tutto il progetto), in modo da poterli usare anche da altri progetti, senza dover

ogni volta ricrearli da zero. Ad esempio un utente ha creato tutti i blocchi necessari per gestire la struttura dati di uno stack; però può succedere che anche in un altri progetti gli capiti di avere a che fare con il medesimo tipo di struttura, e quindi diventerebbe molto noioso e dispendioso in termini di tempo, doversi rifare ogni volta i blocchi. A questo fine Snap! permette di salvare vere e proprie librerie di blocchi che possono poi essere importate in uno qualsiasi dei progetti desiderati. Le versioni precedenti di BYOB permettevano di esportare gli sprite con tutti i loro relativi blocchi, ma non di creare degli insiemi di blocchi da poter caricare in un qualsiasi progetto.

La creazione di una libreria di blocchi avviene cliccando sull'opzione *Export blocks...* dal menù File. In seguito si aprirà la seguente finestra, dove saranno visualizzati tutti i vostri blocchi personalizzati.


Figura 3.7: Finestra esportazione blocchi

Si potrà ora decidere, spuntando o meno il relativo checkbox, esattamente di quali blocchi dev'essere formata la vostra libreria. Premendo sul tasto *OK* si aprirà, come nel caso del salvataggio tramite XML (vedere figura 3.4),

una nuova finestra del browser contenente il codice XML dei blocchi scelti. Ora basterà salvare il tutto con un CTRL+S, fornendo una destinazione di salvataggio, e la procedura è terminata.

Come nel caso di salvataggio di XML, per caricare una libreria sarà sufficiente usare l'opzione *Import...* del menù File, oppure semplicemente trascinare il file XML all'interno della finestra del browser di Snap!.

Ora che sono state chiarite quali sono le differenze tra Snap! e i suoi "parenti" BYOB e Scratch, è possibile vedere con più chiarezza come con essi ci possa affacciare al mondo della robotica.

# Capitolo 4

# La discendenza di Logo nel campo della robotica

## 4.1 Motivazioni

Innanzitutto ci si potrebbe chiedere perché i linguaggi citati finora, vengano tuttora utilizzati nel campo della robotica, dato che spesso essi non lavorano su veri e propri robot fisici.

La risposta risiede nella semplicità che essi forniscono per apprendere i concetti base della robotica, per aiutare coloro che per la prima volta si affacciano a questa scienza tutt'altro che semplice, e che non sanno ancora come cimentarsi in essa. Ad esempio molti giovani studenti spesso trovano molto difficile usare, direttamente, sofisticati simulatori per riprodurre oggetti in 3D, con tutti i loro parametri fisici.

Ci si lascia dunque alle spalle alcuni problemi, che, pur essendo consci del fatto che nella realtà fisica di un robot esistono, si possono apprendere anche successivamente, una volta acquisita una certa esperienza con altri concetti. Con un cosiddetto "robot virtuale" si possono tralasciare tutti i problemi di incertezza di cui invece soffre un robot fisico, quando si andrà a tradurre un comando assegnato come una vera e propria azione da far eseguire al robot in questione. C'è una bella differenza tra i valori dei parametri di una struttura reale di un robot ed i parametri teorici di un modello di robot, questo è poco ma sicuro. Si tralasciano inoltre altri particolari riguardanti accuratezze di sensori vari, limitazioni di natura energetica e molto altro. Si consideri ad esempio un robot fisico ed uno virtuale: con entrambi, partendo da una determinata posizione, si vuole percorrere un certo percorso a forma di esagono perfetto, ritornando alla fine esattamente al punto di partenza. Il robot fisico difficilmente lo farà, ma non a causa di un errore nella fase di formulazione del compito assegnato, cioè nelle fasi di codifica e decodifica

del programma, bensì a causa dell'imprecisione o accuratezza che è spesso fonte di problemi dei robot poco sofisticati.

Quindi soprattutto a scopo educativo, evitare questo tipo di problemi significa poter almeno capire più facilmente i processi costruttivi della robotica semplice. Una volta acquisita una certa esperienza con questi, sarà poi possibile passare a livelli più avanzati, tenendo conto di più variabili o vari fattori, ai quali prestare attenzione durante l'esecuzione di determinati procedimenti; tutto questo magari incontrando meno difficoltà, che invece sarebbero sorte approcciandosi direttamente.

## 4.2 Il ruolo di Logo

Molti potrebbero pensare che la scienza della robotica sia una novità riguardante solo ed esclusivamente Snap!, o perlomeno ad una sua versione precedente di BYOB o tutt'al più a Scratch. Invece non è affatto così. Già molti anni fa infatti, Logo è stato molto usato in questo ambito, applicando un approccio costruzionista, spesso considerato il miglior metodo per introdurre le basi della robotica e cominciare a scoprirne le sue potenzialità. Vi sono infatti due approcci per poter cominciare ad apprendere la robotica, di cui uno è l'estensione dell'altro:

- **approccio costruttivista**: in esso l'apprendimento è inteso come costruzione di modelli cognitivi attraverso la progressiva interiorizzazione delle azioni. Ogni conoscenza è ottenuta non per semplice trasferimento di informazione ma come processo di costruzione fondato sulla conoscenza già acquisita

- **approccio costruzionista**: aggiunge al costruttivismo la convinzione che l'accrescimento di conoscenza avviene in modo più felice ed appropriato se colui che apprende ha la consapevolezza di essere coinvolto nella costruzione di qualcosa di tangibile e di condivisibile, sia essa un castello di sabbia o la teoria dell'universo.

Tornando a Logo, un paio di esempi a suo riguardo sono:

- la famosa *tartaruga* di Papert, la quale in sostanza si trattava di un semplice robot disegnatore su schermo, al quale potevano essere forniti semplici comandi

- altri robot basilari di scopo educativo come Bee-bot e Pro-bot, i quali sono stati programmati con delle primitive molto simili a Logo.

Comunque anche in molti altri casi, nei semplici esempi di movimento di un robot si può notare la decisa somiglianza con i comandi che venivano

impartiti per far muovere la tartaruga di Logo.

Il puro linguaggio Logo è tuttavia in un certo senso limitato. Le implementazioni più comuni di Logo forniscono uno scenario di lavoro dove per così dire se ne possono controllare gli "attori". Logo permette interazioni tra loro, però non tiene conto delle interazioni che invece essi hanno con l'ambiente esterno. Nel caso di Logo, l'ambiente esterno altro non è che uno stage da decorare, un disegno creato su schermo cambiando colore a determinati pixel. Un attore (ovvero una *tartaruga*) può solamente sapere il colore del pixel dove si trova attualmente. Ad esempio se si volesse disegnare una certa figura geometrica, una volta assegnato questo compito alla tartaruga, essa eseguirà il tutto senza essere influenzata in qualche modo da altri particolari presenti nello stage (ovvero nell'ambiente). Di conseguenza le primitive di Logo, più comunemente usate, sono vari comandi e funzioni aritmetiche per il movimento della tartaruga ed il comando *repeat*; risulta invece più raro, visto il problema detto in precedenza, l'uso del comando *if*.

Esistono però ovviamente situazioni più complesse, dove si raggiunge un secondo livello di astrazione, nel momento in cui si valuteranno ipotesi di condizioni comportamentali più evolute, cioè ipotesi che esprimono le interazioni con l'ambiente esterno. Proprio per ovviare a questo tipo di situazioni i robot vengono dotati di svariati sensori, i quali interagiscono continuamente con l'ambiente e forniscono informazioni a suo riguardo se interrogati.

In questo contesto Scratch e BYOB (e di conseguenza anche Snap!) permettono la realizzazione di svariati sensori, tramite i quali è possibile interrogare il nostro robot-sprite e decidere, sulla base della risposta restituita, il suo comportamento. Le prossime sezioni tratteranno proprio di questo.

## 4.3 La robotica con Scratch

Come già visto, Scratch è una versione molto più limitata di Snap!, ma tuttavia ciò non vuol dire che tale programma non possa permettere un esperienza introduttiva alla robotica.

Saranno presentati dei semplici esempi di sensori basilari di un robot virtuale. Questi esempi sono utili per avere un'introduzione dei sensori che BYOB e Snap! permettono di creare, dato che il meccanismo di realizzazione è molto simile. Inoltre verranno illustrati i principali blocchi utili per tale scopo, dei quali è fondamentale conoscere il comportamento. Questi blocchi, verranno usati anche nella sezioni successive a quella di Scratch, ma non verranno rispiegati, in quanto il loro comportamento rimane sempre lo stesso.

### 4.3.1 Semplici movimenti

Scratch fornisce sostanzialmente due strumenti per controllare l'esecuzione di un certo movimento di uno sprite: il blocco *glide* ed il supporto di un timer. Il blocco *glide* infatti riceve in input un tempo e una coppia di valori che rappresentano delle determinate coordinate (asse x e asse y) nel piano dello stage: con questi input il comando *glide* farà raggiungere allo sprite la posizione selezionata nel tempo specificato (quindi tanto più alto sarà il valore del tempo, tanto più lento sarà lo spostamento; viceversa, più basso sarà il valore temporale, più veloce sarà lo spostamento).

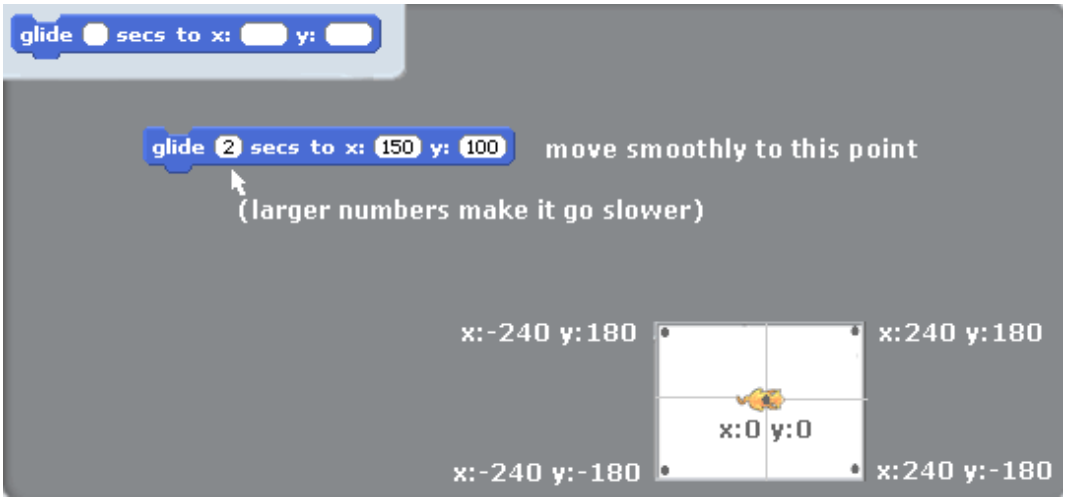


Figura 4.1: Funzionamento blocco glide

Un semplice esempio di movimento, tramite l'uso del blocco appena accennato, può essere la situazione di un autobus che deve effettuare una sosta di un certo tempo ad ogni fermata; si supponga, per semplicità, che tutte le fermate siano posizionate lungo una retta, ed in più ogni fermata si trovi ad una certa distanza fissata rispetto ad un'altra. Un particolare riguardante la distanza: Scratch,BYOB e Snap! come unità di misura della distanza usano gli step.

Si definiscono le seguenti variabili:

- *numStops*=numero di fermate
- *waitStop*=tempo che l'autobus deve attendere in sosta ad una fermata
- *stopDis*=distanza fissa fra una fermata e l'altra
- *busSpeed*=velocità dell'autobus (in step/s)

Dunque per calcolare il timer da dare in input al blocco *glide*, basterà usare la formula inversa della velocità per calcolare il tempo, ovvero tempo=spazio/velocità. Un blocco *wait*, che riceverà in input la variabile *waitStop*, effettuerà la sosta. Lo spostamento (blocco *glide*) e l'attesa (blocco *wait*) saranno inclusi in un ciclo (blocco *repeat*); tale ciclo sarà effettuato tante volte quante sono le fermate specificate nella variabile *numStops*. Tutto ciò è mostrato in figura 4.2.

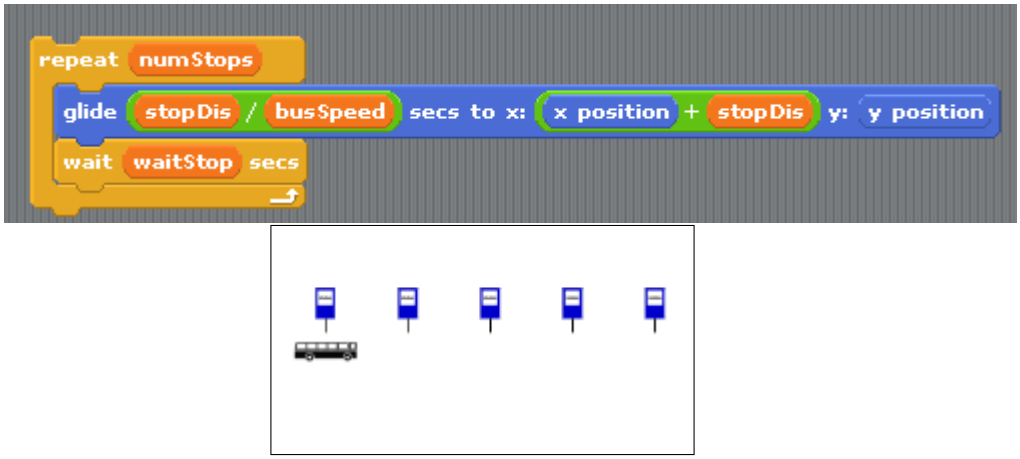


Figura 4.2: Fermate autobus in linea retta

Notare il fatto che la posizione data in ingresso al blocco *glide* è fornita dai due reporter blocks *x position* e *y position*. In particolare il blocco *x position* è stato immesso in un blocco di somma assieme al blocco della variabile *stopDis*: in questo modo, ad ogni ciclo, verrà aggiunta alla posizione x corrente dello sprite, l'esatta distanza da compiere per arrivare alla prossima fermata.

Se si prendesse però il caso in cui le fermate non siano così ben disposte come in precedenza, ma bensì fossero sparse casualmente nello stage, sorgerebbe allora un problema: il blocco *glide* purtroppo non ha una versione in cui si possa specificare sia la distanza che la direzione (quest'ultima nell'esempio precedente è stata omessa infatti). Bisognerà dunque ricorrere all'aiuto di due ulteriori blocchi: il blocco *point towards* ed il blocco *distance to*. Il primo consente di poter orientare lo sprite verso la prossima fermata, fornita in input come nome di uno sprite(*stopName* nell'esempio successivo), mentre il secondo riceve in ingresso il nome di uno sprite esistente (sempre di una fermata) e ne restituisce la distanza da esso, rispetto allo sprite in cui è eseguito tale blocco. L'esempio di figura 4.3 illustra quanto appena detto.

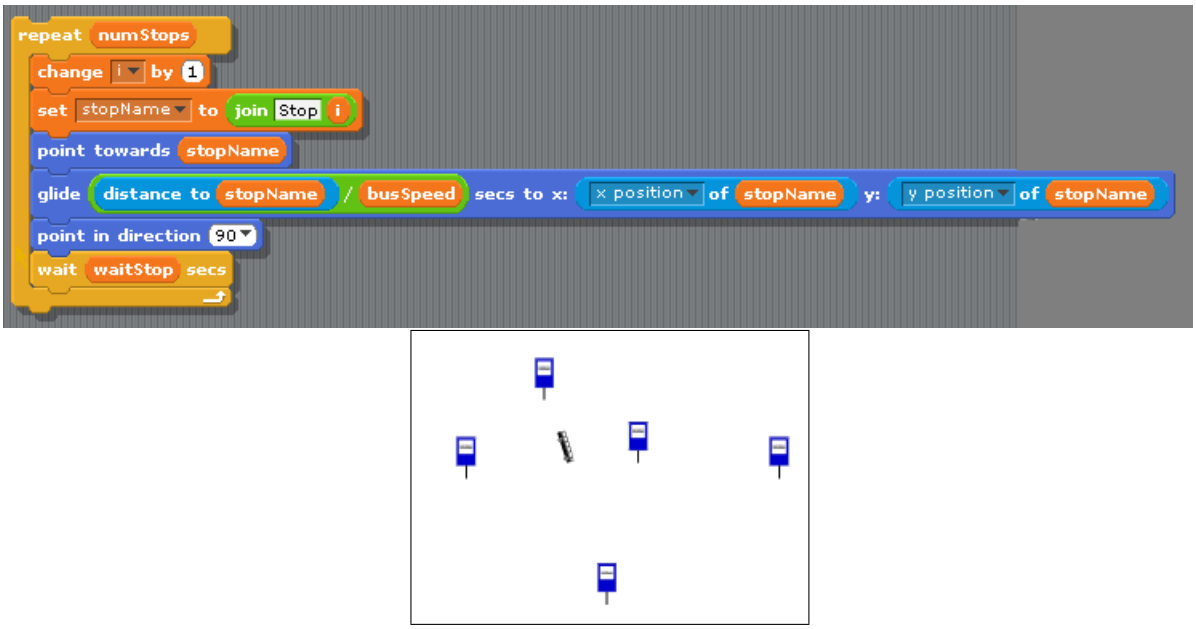


Figura 4.3: Fermate autobus disposte casualmente

Le coordinate questa volta sono ottenute tramite l'uso di particolari blocchi situati nella categoria *Sensing*: uno per la posizione dell'ascissa e l'altro per la posizione dell'ordinata, e in essi si specifica il nome dello sprite voluto, proprio come mostrato in figura 4.4.


Figura 4.4: Prelevamento coordinate esempio precedente

Il blocco *point in direction* è semplicemente un'altra versione del blocco *point towards*. Nell'esempio appena visto serve per riaggiustare l'orientamento dello sprite dell'autobus, una volta che esso ha raggiunto una fermata, in modo che punti in orizzontale verso destra. La figura 4.5 mostra le possibile combinazioni dell'uso di questo blocco.

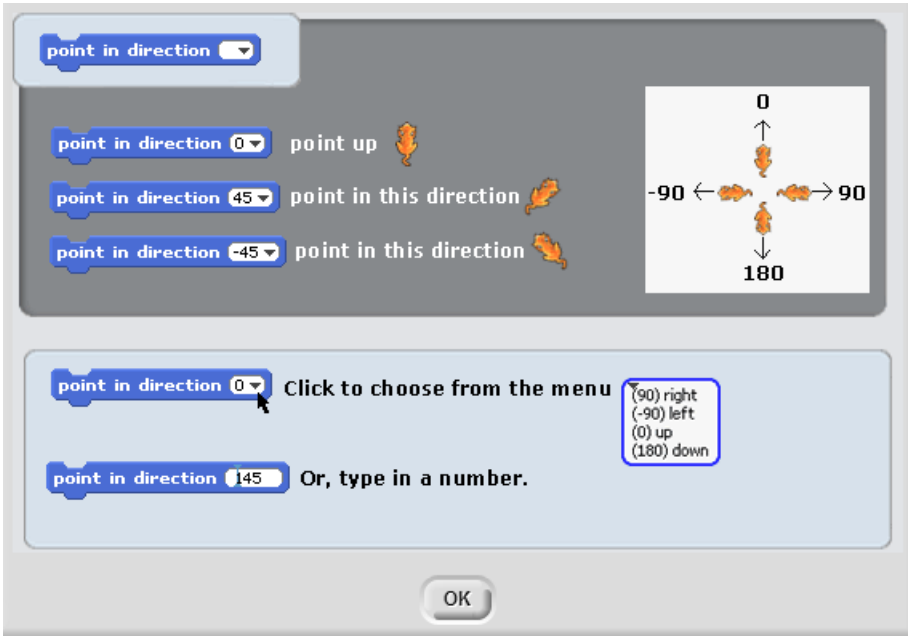


Figura 4.5: Funzione blocco point in direction

Un altro blocco che può tornare utile in questo ambito è sicuramente il blocco *color <color1> is touching <color2> ?*. Fra poco si avrà una dimostrazione dell'uso di questo blocco.


Figura 4.6: Blocco per verificare il contatto tra due colori

Rimettiamoci nel caso del primo esempio, dove tutte le fermate erano in linea retta, ma questa volta si supponga che la distanza tra una fermata e l'altra non sia né fissa, né nota. Di sicuro non c'è il problema dell'orientamento, ma tuttavia non ho nessuna indicazione su quando lo sprite dell'autobus si deve fermare. Uno stratagemma efficace consiste nell'aggiungere all'immagine dello sprite dell'autobus, detta costume, un piccolo rettangolo di uno specifico colore, ad esempio rosso (si veda figura 4.7). Così facendo, si può sfruttare il blocco appena visto in figura 4.6: quando il rettangolo rosso toccherà il colore nero della parte inferiore di una fermata, quel blocco restituirà il valore *true*, ed in quel caso si può decidere di fermare lo sprite dell'autobus. Per il resto il programma è analogo a come visto in precedenza.

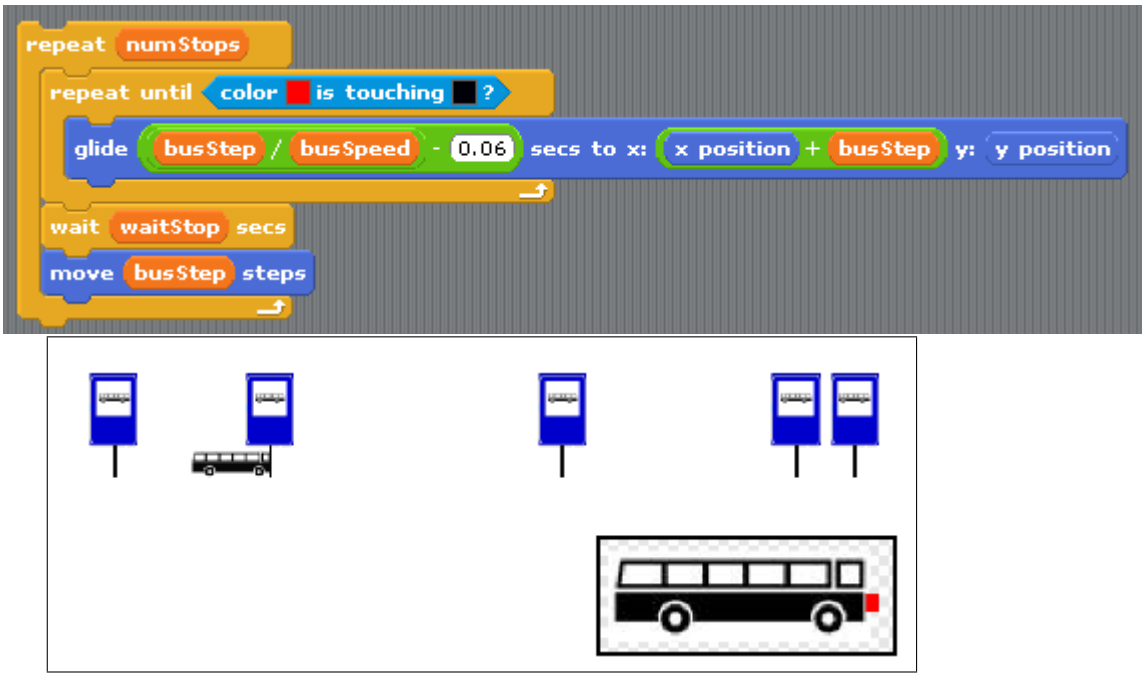


Figura 4.7: Autobus con sensore e fermate in linea retta

E' stata usata una variabile *busStep*, che altro non è che una distanza fissata (non troppo grande) usata come spostamento dell'autobus ad ogni esecuzione ciclica del blocco *repeat*, fino al verificarsi della condizione della condizione di uscita del ciclo. Naturalmente non è proprio esatto fermare l'autobus appena la condizione del blocco *color <color1> is touching <color2> ?* è verificata, perché lo sprite si fermerebbe un po' prima della fermata. Per risolvere tale problema basta specificare un certo ritardo al timer nel blocco *glide*(nell'esempio di 0.06), permettendo quindi all'autobus di spostarsi quanto basta per meglio centrare la sosta alla fermata.

### 4.3.2 Evitare ostacoli

Se si conosce il nome dello sprite che funge da ostacolo, tramite il blocco booleano *touching <objectsprite>?* si può fare in modo che un certo sprite in movimento eviti degli ostacoli. Infatti questo blocco restituisce il valore *true* se i costumi di due sprite (uno è quello nel cui script si usa il suddetto blocco, mentre l'altro è fornito in input tramite il menù a tendina del blocco stesso) vengono a contatto. Una volta verificata questa condizione si può fare in modo di virare il movimento dello sprite affinchè l'ostacolo in questione venga evitato.


Figura 4.8: Blocco per verificare contatto tra sprite

Nell'esempio successivo di figura 4.9, lo sprite di Scratch inizialmente avanza verticalmente verso l'alto, grazie al ciclo *repeat* più esterno, finchè non trova lo sprite d'ostacolo (il rettangolo nero); quando accade ciò, si entrerà nel blocco *repeat* più interno e lo sprite, finchè rimarrà a contatto con il rettangolo nero, si sposterà lateralmente verso destra, aumentando continuamente la sua coordinata x. Quando lo sprite non sarà più a contatto con l'ostacolo, si uscirà dal blocco *repeat* interno e si continuerà l'esecuzione di quello esterno, aumentando la coordinata y, e quindi facendo salire di nuovo verticalmente lo sprite, fino a quando la sua coordinata y non avrà raggiunto il valore 180.

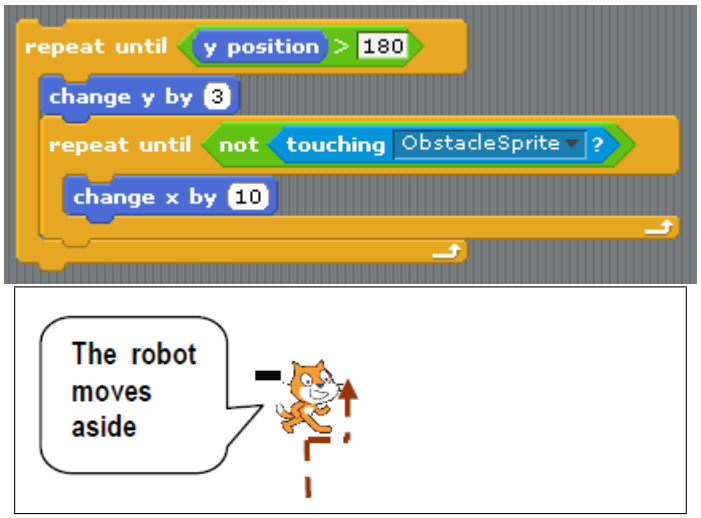


Figura 4.9: Esempio per evitare ostacoli

Non avrebbe molto senso che lo sprite in movimento vada fuori dai confini dello stage, dove non si può prestare attenzione al suo comportamento. Il blocco booleano *touching edge?* (il secondo in figura 4.10) permette di evitare questo problema: quando lo sprite avrà raggiunto un bordo dello stage, tramite questa condizione, si può modificare il suo comportamento nel modo voluto dall'utente, come ad esempio cambiare traiettoria, fermarsi o altro. Altra complicazione, nel caso di voler evitare gli ostacoli con Scratch, può essere il caso in cui non si conosce il nome degli sprite di ostacolo, e di conseguenza non si può conoscere l'esatta posizione degli ostacoli, esattamente come il caso precedente dell'autobus con le fermate disposte in modo casuale nello stage. Ed infatti anche qui si risolve questo inconveniente con una

versione simile di un blocco visto in precedenza: il blocco *touching color <color>?* (il primo in figura 4.10), per verificare quando si andrà a toccare, con lo sprite in movimento, un ostacolo di un certo colore.


Figura 4.10: Altri blocchi utili

Nell'esempio di figura 4.11 si procede proprio con questo meccanismo: lo sprite di movimento in questione è una semplice pallina rossa, mentre gli ostacoli sono dei mattoncini neri, di dimensioni e posizioni diverse.

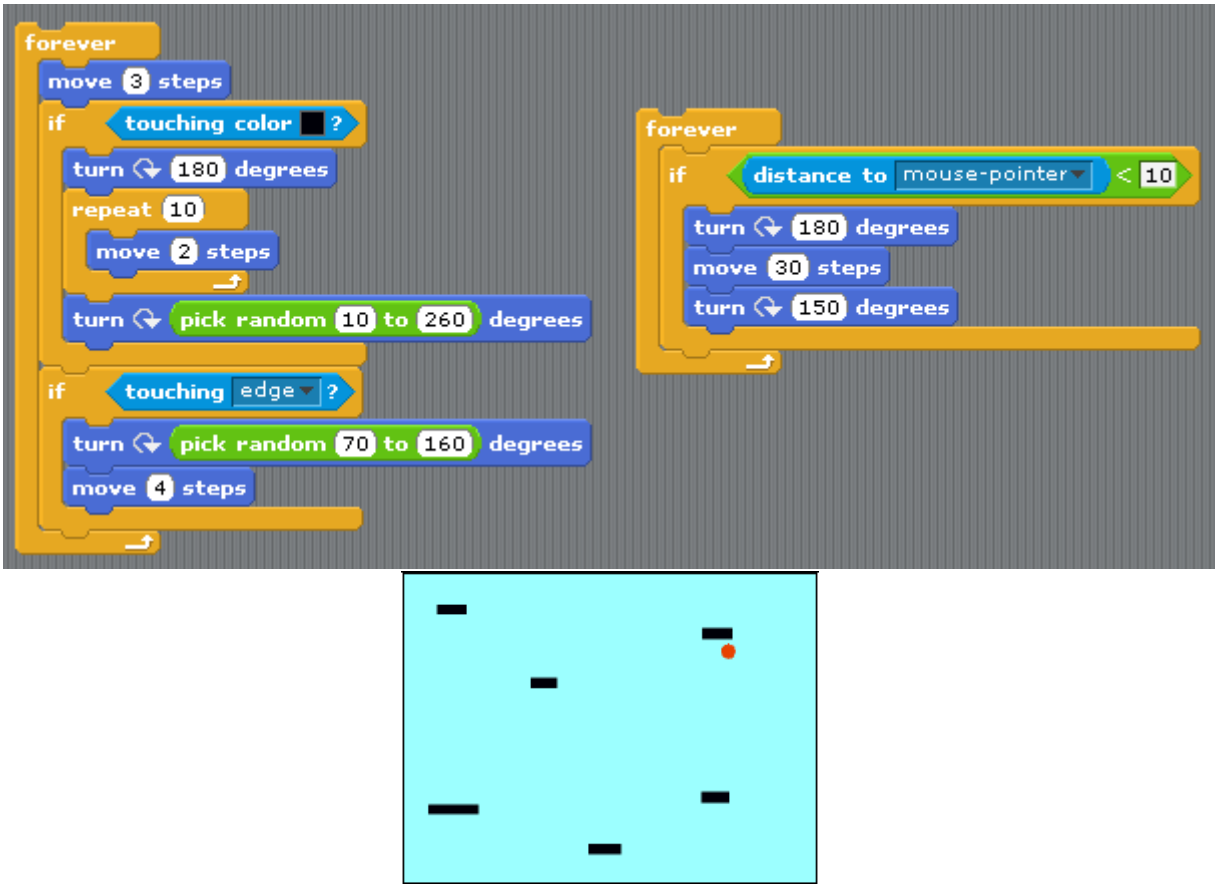


Figura 4.11: Esempio piu complesso per evitare ostacoli

Degna di nota è la presenza questa volta di ben due script, indipendenti l'uno dall'altro, ed eseguiti in modo concorrente, una volta cliccata la classica bandierina verde. Lo script di sinistra regola tutto il procedimento spiegato finora, ovvero la possibilità di evitare i mattoncini, con l'aggiunta del fatto

che la pallina rimbalzi sui bordi dello stage, senza uscirne, grazie al secondo blocco *if*.
Lo script di destra è semplicemente un'ulteriore possibilità di interazione fornita all'utente. Infatti grazie alla funzione *mouse pointer*, inserita nella condizione del blocco *if*, si può specificare un altro evento nello stage qualora il puntatore del mouse si trovi in prossimità della pallina rossa. La differenza sta nel fatto che qui è proprio l'utente che decide direttamente se scatenare tale evento o meno. Nell'esempio, se il cursore del mouse si trovasse nelle vicinanze della pallina rossa, questa effettuerà un piccolo salto.

## 4.4 La robotica con BYOB e Snap!

Essendo BYOB e Snap! delle estensioni di Scratch ovviamente permettono anch'essi la realizzazione di svariati sensori nell'ambito robotico. Tutti i relativi comandi prima presentati nella piccola sezione dedicata a Scratch esistono ed hanno esattamente la medesima funzione anche su BYOB e Snap!. Qui tuttavia è possibile ampliare il comportamento di determinati sensori oppure crearne alcuni più specifici, utilizzando sia dei blocchi già presenti su Scratch e sia blocchi che invece non ne facevano parte.

Si intuisce comunque, che spesso nel caso di BYOB o Snap! ogni blocco creato costituisce un sensore a sé stante, utilizzabile, ogni volta sia necessario, in altri script riguardanti il nostro robot virtuale. Per meglio simulare la situazione che si avrebbe connettendo effettivamente un robot fisico, si può utilizzare una particolare interfaccia basata sul concetto principale di "porta". Si effettua una configurazione di base dove il nostro robot virtuale è connesso al computer tramite un certo numero di porte. Tali porte sono indicizzate tutte con un numero intero diverso, di solito consecutivo. Solitamente si crea un blocco ***config***, mostrato in figura 4.12, per ogni sprite, che inizializza tutte le porte a cui lo sprite stesso deve fare riferimento.

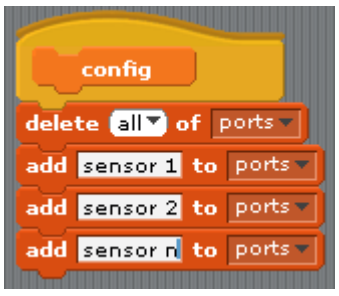


Figura 4.12: Il blocco config

In generale, le porte altro non sono che un mezzo di comunicazione per permettere lo scambio di informazioni fra il programma ed i sensori del nostro robot. I sensori hanno infatti il compito di fornire determinati parametri in base alla situazione momentanea del robot. Per questo i blocchi creati che fungono da sensori ricevono sempre come input una porta, un intero specifico, per aprire la connessione per un determinato servizio richiesto. Successivamente, nello script dei blocchi stessi, è fondamentale inserire un controllo di porta: se la porta inserita in input è corretta per la funzione voluta, la connessione avviene con successo e il blocco esegue il suo script; se invece fosse stata fornita una porta sbagliata, verrà emesso un messaggio di errore.

Verranno ora presentati alcuni fra i principali sensori realizzabili sia con BYOB che con Snap!.

### 4.4.1 Sensori embedded

Sono i primi strumenti di cui di solito si necessita; sensori che sono facilmente realizzabili, in quanto esiste, solitamente, già un blocco di base che fornisce l'informazione voluta. Si può dunque creare un sensore che possa restituire la posizione attuale di un nostro sprite nelle solite coordinate dello stage, oppurre creare un sensore che restituisca in che modo lo sprite stesso è orientato, per poter poi definire altri comandi sulla base del suo orientamento momentaneo.
Nella figura 4.13 è mostrato lo script di un sensore chiamato ***gps***, il quale fornisce la posizione del nostro sprite-robot nello stage, mentre in figura 4.14 è fornito lo script del sensore ***compass*** che ne fornisce l'orientamento.

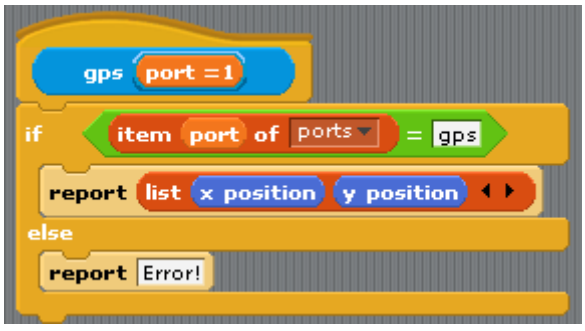


Figura 4.13: Sensore per la posizione

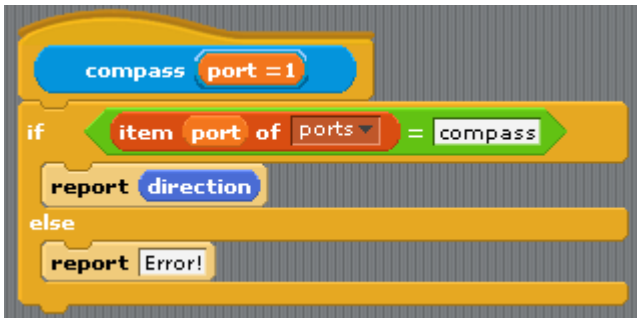


Figura 4.14: Sensore per l'orientamento

Analogamente, se si volesse realizzare un sensore sonoro, lo script sarebbe esattamente lo stesso di figura 4.14, tranne per il nome ovviamente, e per il fatto che al posto del blocco *direction* restituirebbe il blocco *loudness*, già fornito di base dal programma.

I blocchi appena visti funzionano però solo per lo sprite stesso per il quale essi sono dichiarati. E' possibile tuttavia anche realizzare una loro versione remota, ovvero sensori che forniscano ad uno sprite, informazioni riguardanti un altro sprite. Si può immaginare il tutto come se ci fosse un robot che chiede informazioni ad un suo pari, tramite un qualche tipo di connessione. Per questi metodi però non è sufficiente l'input di una porta: è necessario conoscere anche il nome dello sprite a cui si vuole fare riferimento (la voce *myself* indica lo sprite stesso in cui è eseguita la chiamata di un blocco-sensore). Le versioni remote dei sensori ***gps*** e ***compass*** sono rispettivamente forniti nelle figure 4.15 e 4.16.

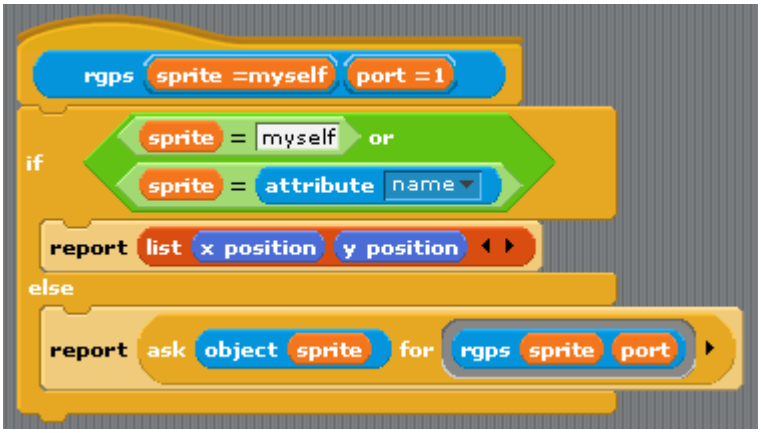


Figura 4.15: Sensore per la posizione remoto

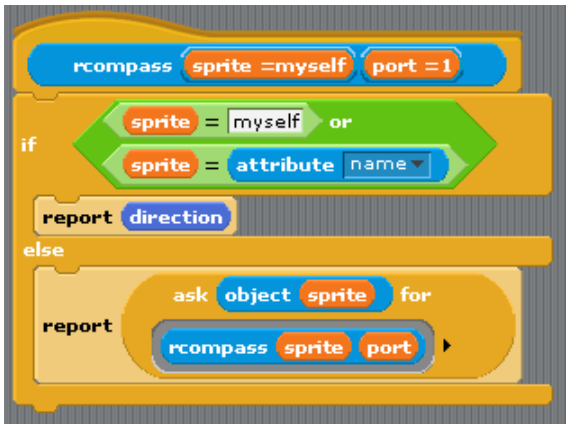


Figura 4.16: Sensore per l'orientamento remoto

In essi, per la chiamata remota, è inoltre utilizzato un blocco ausiliario ***ask***, mostrato in figura 4.17.


Figura 4.17: Il blocco ask

I sensori **rgps** e **rcompass**, in pratica fanno questo ragionamento:

- inizialmente verificano se il campo *sprite*, ricevuto in input, corrisponde al nome stesso dello sprite (oppure vi è la voce *myself*). In caso questo fosse vero, il loro funzionamento è esattamente uguale alle loro versioni non remote.
- nel caso invece in cui venisse fornito un nome di sprite diverso da quello in cui è avvenuta la chiamata del blocco-sensore, allora si farà uso del blocco *ask* per effettuare la chiamata remota allo sprite specifico (se esiste), con gli stessi parametri di input ricevuti (nome dello sprite e porta). Nel caso in cui il nome di sprite specificato non esistesse, verrà restituito un messaggio di errore.

Negli esempi appena visti è stato omesso il controllo di porta, in quanto esattamente analogo a quanto visto prima. Lo stesso sarà fatto per gli esempi successivi.

Un altro tipo di sensori, che in molte applicazioni risultano fondamentali, sono quelli per verificare il contatto con un altro oggetto nello stage. Sulla base che ci sia o meno il contatto, si può indicare il comportamento voluto dal nostro robot. Pure qui, come su Scratch, il trucco in sostanza sta nell'aggiungere al costume dello sprite un piccolo sensore, di qualsiasi forma si voglia, ma che abbia uno specifico colore. Si andrà poi proprio ad agire su condizioni che riguardano tale colore: finchè il colore del sensore non sarà a contatto con il colore di un certo ostacolo si avrà un comportamento, mentre quando i due colori saranno a contatto allora si svolgerà un'altra azione. Nell'esempio di figura 4.18, al costume dello sprite (l'automobile) sono stati aggiunti due piccoli rettangoli, uno rosso ed uno blu. Per ognuno dei due viene realizzato un apposito blocco sensore (chiamati ad esempio ***bumperred*** e ***bumperblue***) per verificare l'impatto con i muri grigi dello stage.

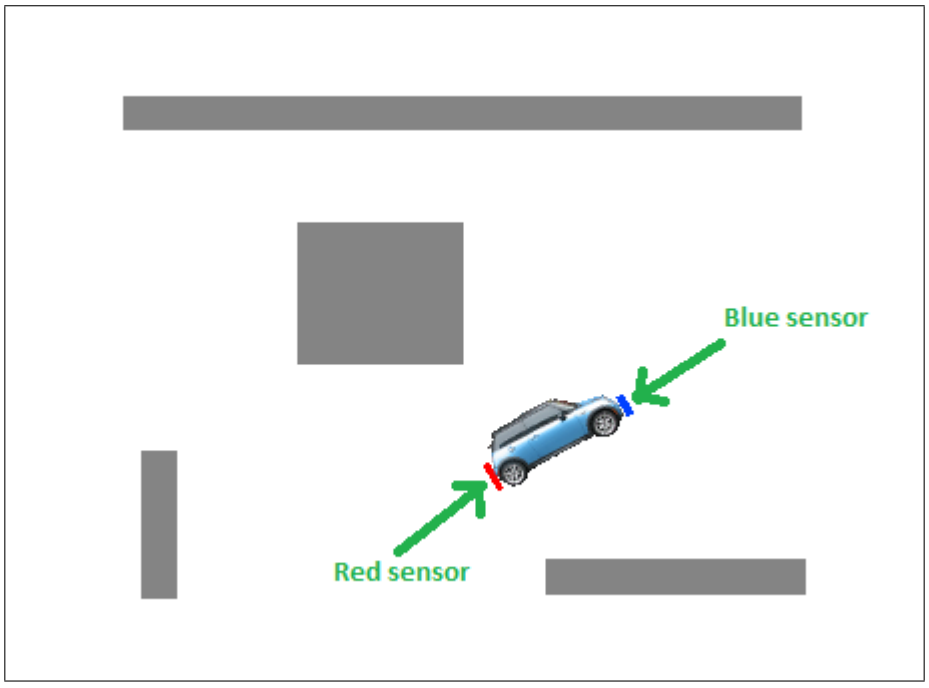


Figura 4.18: Esempio con sensori di contatto

In figura 4.19 è mostrato lo script per il sensore rosso; quello per il blu è esattamente uguale a parte il nome, il colore di confronto e l'uso di una diversa variabile booleana di stato.

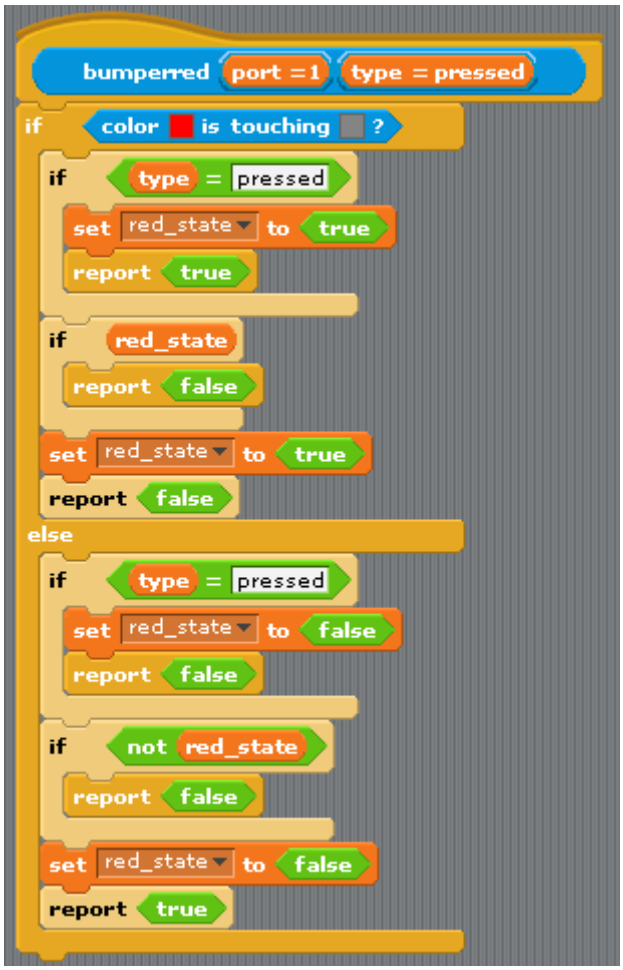


Figura 4.19: Sensore rosso di contatto

In questo caso il fulcro dei blocchi ***bumperred*** e ***bumperblue*** appena visti è ancora il blocco *color <color1> is touching <color2>?*, blocco già visto anche nell'ambito di Scratch.
La variabile di input *type* si assume che possa accettare solo due valori: *pressed* e *bumped*. La funzione del primo tipo risulta ovvia, mentre quella del secondo è un po' diversa, in quanto esso ha memoria: tiene cioè conto dei valori precedentemente assunti dalla variabile *red state*. Con il tipo *bumped*, il sensore restituisce il valore *true*, quando si ha una transizione fra gli stati *pressed* e *not pressed*.

I due blocchi sensori creati possono essere utilizzati in un qualsiasi script volto a regolare le azioni dello sprite. Con lo script di figura 4.20, ogni volta che uno dei due sensori tocca uno dei muri, la traiettoria dello sprite-automobile

viene modificata di 90°, in senso orario per un sensore, in senso antiorario per l'altro. E' stato inoltre aggiunto un ulteriore blocco sensore: il suo script è analogo a quello di figura 4.19, solo che come condizione del primo *if* viene inserito il blocco <*if key [space] is pressed?*>, che verifica se da tastiera viene premuto il tasto di spaziatura o meno (inoltre è opportuno definire una nuova variabile di stato). In caso positivo il sensore restituirà come al solito il valore *true* e lo script di figura 4.20 farà girare automaticamente la nostra macchina sempre di 90°.

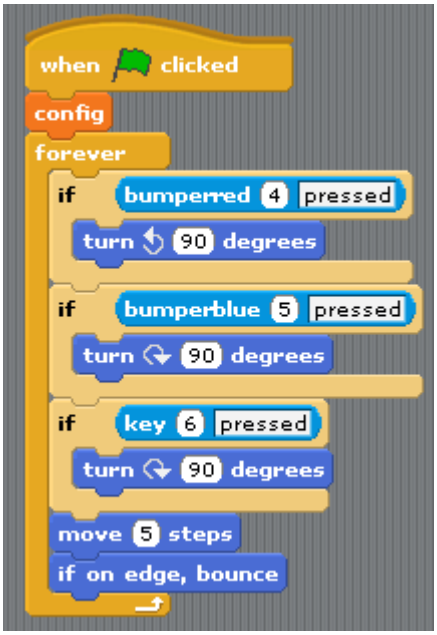


Figura 4.20: Script per l'esempio dell'automobile

Altro blocco utile è il blocco *if on edge bounce* (ultimo nello script della figura 4.20), che appunto fa invertire la rotta della macchina nel caso essa toccasse un bordo dello stage.

### 4.4.2 Sensori di luminosità e di colore

Finora si sono visti sensori che possono essere direttamente simulati tramite l'ambiente di sviluppo di BYOB o Snap!, come ad esempio l'orientamento e la posizione di uno sprite, proprietà implicite di ognuno degli oggetti in questione. Tuttavia alcuni sensori non appartengono a tale categoria e non possono essere quindi rappresentati nello stesso modo. Serviranno come minimo una o anche più variabili, che tengano conto del valore di una caratteristica dello sprite; questo valore verrà modificato in base a determinate azioni eseguite dall'utente, nel caso più semplice, ad esempio, da un normale

input da tastiera.

Mettiamo il caso che si voglia realizzare un sensore che ci riferisca il colore, rappresentato nel caso classico da una stringa esadecimale, di un determinato sprite con il quale il sensore è a contatto. Si crea il blocco ***colorcode*** (figura 4.21), il quale si suppone funzioni con la porta 7 per il nostro sprite; questo blocco altro non fa che restituire il valore della variabile locale *colorvar* di un determinato sprite (il significato di questa variabile verrà chiarito fra poco). Oltre a questo, si crea poi il blocco ***color*** (figura 4.22), supponendo che funzioni con la porta 8, e che specifica con che colore il nostro sensore è a contatto.


Figura 4.21: Script blocco colorcode

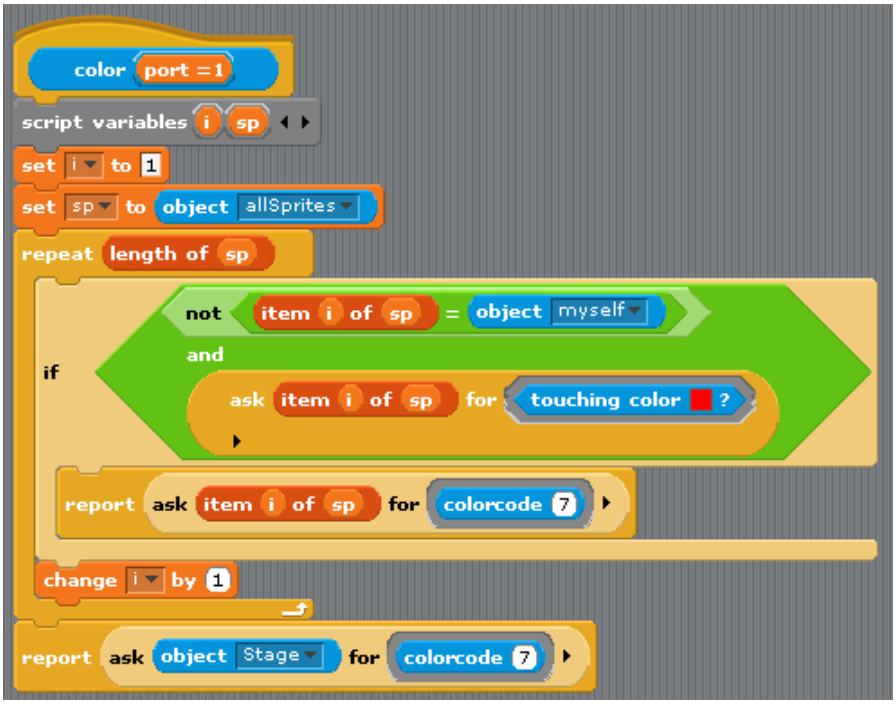


Figura 4.22: Script blocco color

Lo script del blocco *color* controlla se esiste uno sprite (che non sia lo stesso in cui è avvenuta la chiamata del blocco *color*, quindi escludendo la voce *myself*) che sia a contatto con il colore del sensore: se trova tale sprite, allora ne stampa il relativo colore in stringa esadecimale.
Nell'esempio di figura 4.23, lo sprite principale che contiene il sensore è un rettangolo nero; il sensore è il rettangolino rosso su di esso. Sono stati inoltre creati altri due sprite di colore omogeneo: *Sprite1* è un ellisse giallo, mentre *Sprite2* un altro rettangolo ma questa volta di colore blu. Infine, è stata creata una variabile, locale ad ogni sprite, chiamata *colorvar*: per ogni sprite questa variabile è stata settata manualmente con la stringa esadecimale corrispondente al colore dello sprite stesso.

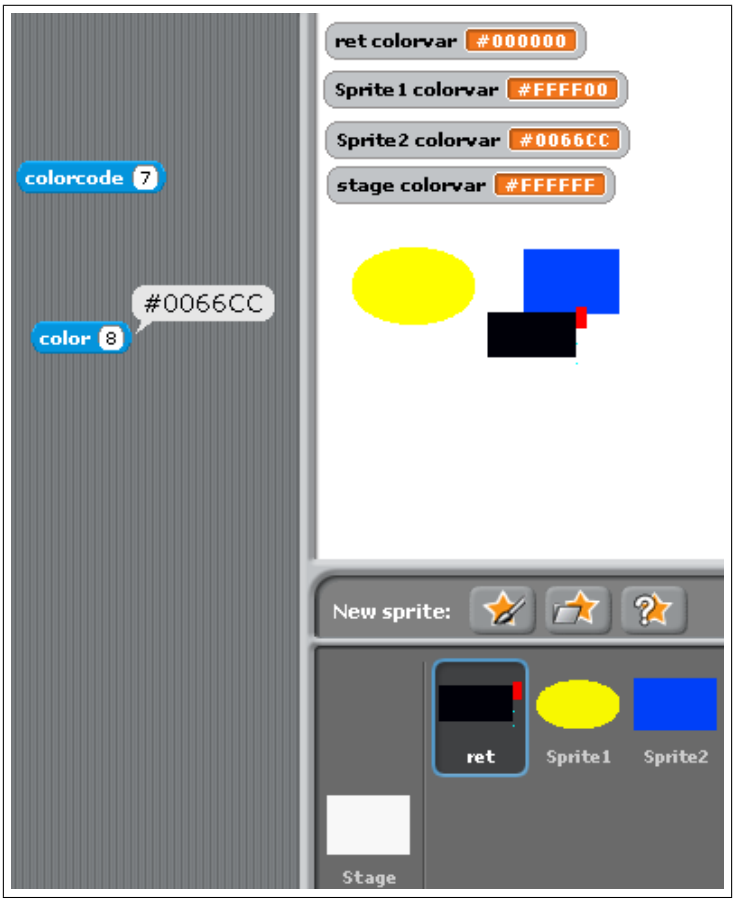


Figura 4.23: Output sensore a contatto con uno sprite

Se il sensore non è a contatto con nessuno sprite allora restituisce il colore dello stage (sempre assegnato alla sua variabile *colorvar* in precedenza), come si può notare dalla figura 4.24.

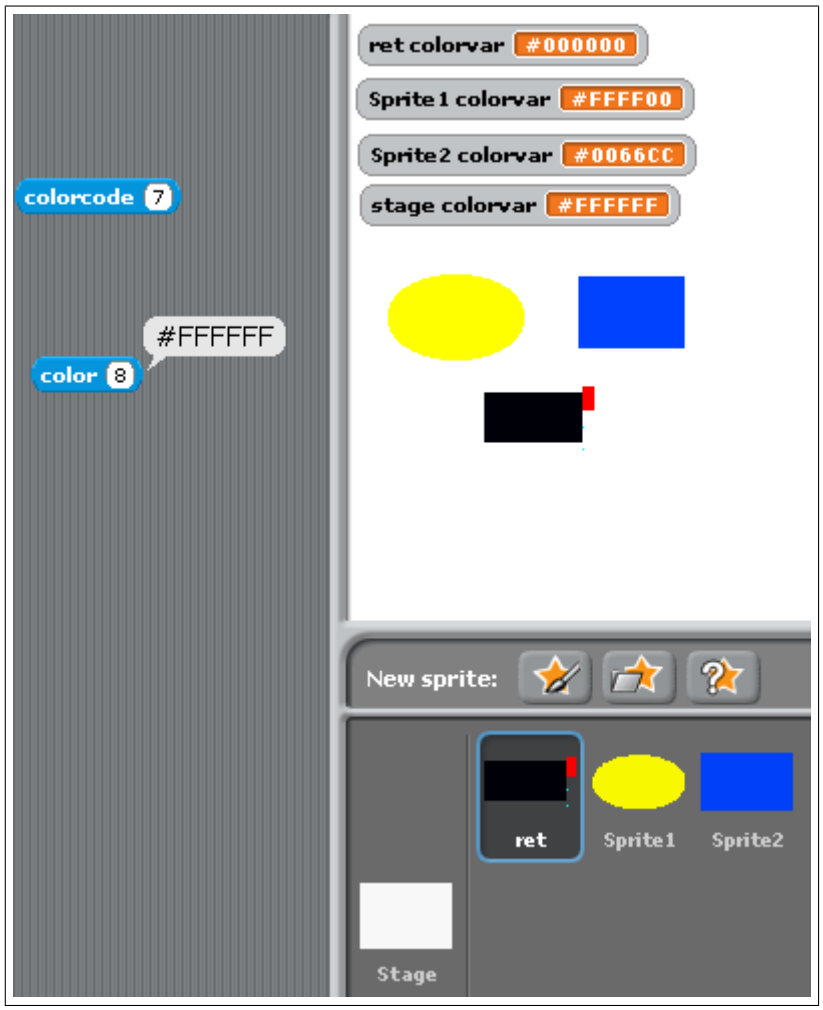


Figura 4.24: Output sensore non a contatto con uno sprite

In maniera esattamente analoga può essere realizzato un sensore di luminosità: considerando lo stesso sprite nero col rettangolino rosso precedente, si crea una variabile *lightvar* (solitamente un intero nel range 1-100) e se ne setta il valore per ogni sprite nel programma; dopodiché si creano due blocchi sensore analoghi a quelli presentati prima (nel caso specifico ***lightlev*** e ***light***). Dato che, in questo caso, la variabile associata al nostro sensore è un normale numero intero, è facile, e spesso utile, impostare delle condizioni che, se verificate, possono modificare il valore di questa variabile, magari tramite la pigiatura di un tasto. Ad esempio con il tasto *up arrow* si può fare in modo di incrementare di un'unità la variabile *lightvar*, mentre con il tasto *down arrow* invece la si può decrementare di uno.
Le condizioni riguardanti input vari da tastiera sono molto frequenti in varie realizzazioni di applicazioni con uno dei qualsiasi programmi visti finora in

questa tesi. Il caso in cui si agisca solo su una variabile è il caso più semplice: si può infatti anche adottare un intero comportamento diverso, per un nostro sprite, solo con la pigiatura di un tasto; quindi, in tal caso, si parla anche di eseguire uno o più script, dove al loro interno si possono modificare variabili, accertarsi di certe condizioni tramite sensori vari e molto altro ancora.

### 4.4.3 Configurazione Lego NXT tramite Snap!

LEGO Mindstorms NXT è un kit robotico programmabile rilasciato dalla Lego alla fine di luglio 2006. Il componente principale del kit è il computer a forma di mattone chiamato NXT brick: esso può ricevere in input fino ad un massimo di quattro sensori e controllare fino a tre motori elettrici, attraverso cavi RJ12, molto simili ma incompatibili con i cavi del telefono RJ11. Programmi molto semplici possono essere scritti usando il menù dell'NXT; programmi invece più complicati e file sonori possono essere scaricati usando la porta USB o senza fili usando il Bluetooth. Oltre a queste generalità, grazie al lavoro di Connor Hudson, ora Lego NXT ha la possibilità di essere configurato, e quindi poi utilizzato, mediante Snap!. Vediamo dunque una traccia di come questo è possibile.
Innanzitutto è necessario scaricare l'apposito pacchetto di file necessario per la configurazione. Tali contenuti si possono trovare all'indirizzo `http://technoboy10.github.io/snap-nxt/`. Ora è necessario aprire Snap! in una finestra del browser ed importare, tramite l'uso (già visto in precedenza) del comando *import* del menù *File* di Snap!, il file *nxt.xml*, il quale è contenuto nel suddetto pacchetto. Se l'operazione ha avuto successo, nella sezione *Variables* di Snap! si dovrebbero ora trovare i blocchi di figura 4.25.

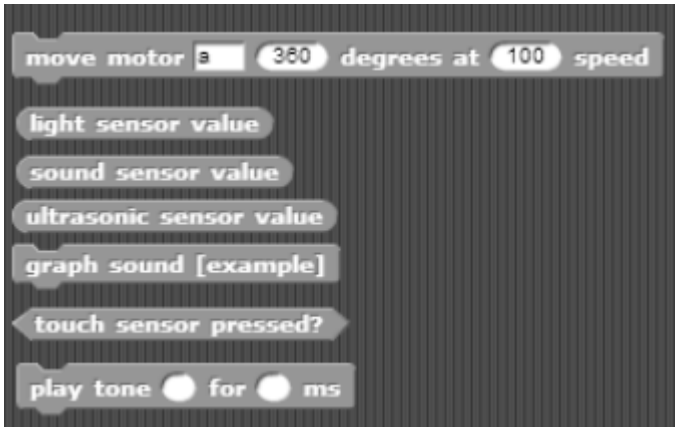


Figura 4.25: Blocchi aggiunti per Lego NXT

Infatti, all'interno del file appena caricato, sono stati definiti, in codice XML, tutti i blocchi relativi a servizi destinati all'uso di Lego NXT. Tali servizi, nell'ordine esatto con cui sono menzionati nel file *nxt.xml*, sono:

- move
- nxttouch
- nxtlight
- nxtsound
- nxtultrasonic
- tone
- nxtbattery
- nxtilluminate

Vediamo brevemente il meccanismo di definizione di uno dei blocchi, ad esempio di quello destinato al servizio *nxtlight*.

```
- <block-definition category="other" type="reporter" s="light sensor value">
    <inputs/>
  - <script>
      - <block s="doReport">
          - <block s="reportURL">
              <l>localhost:1330/nxtlight</l>
            </block>
        </block>
    </script>
</block-definition>
```

Figura 4.26: Definizione del blocco per il sensore di luce

Come si può notare dalla definizione del sensore appena mostrata, lo script di tale blocco è sostanzialmente composto da un blocco *report*, all'interno del quale vi uno speciale blocco *http*, che a sua volta contiene un determinato URL (localhost, ovvero 127.0.0.1) con tanto di porta (1330). La porta 1330 infatti è riservata esclusivamente per la comunicazione con il robot di Lego NXT. Quindi ciò fa intuire che il protocollo usato per l'utilizzo dei servizi prima specificati è esattamente HTTP.

Proprio a questo scopo, Snap! ha messo a disposizione un apposito blocco, mostrato qui sotto.

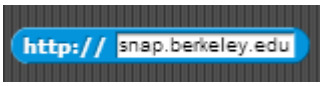


Figura 4.27: Blocco http

La conferma di quanto detto fin qui, arriva direttamente dal prototipo, del medesimo blocco di cui si è parlato finora (ovvero *light sensor value*), aperto su Snap!, mostrato in figura 4.28.


Figura 4.28: Prototipo blocco light sensor value

Riepilogando, in sostanza viene riportato il risultato del reporter block *http*, all'interno del quale sono specificati i parametri menzionati prima. Come si può vedere la sintassi è questa: **indirizzo:porta/servizio**.
Gli altri reporter blocks, relativi a Lego NXT, funzionano in maniera esattamente analoga a questo.

Vediamo ora invece un blocco più complicato, di tipo command, ad esempio il blocco riguardante il regolamento del motore.

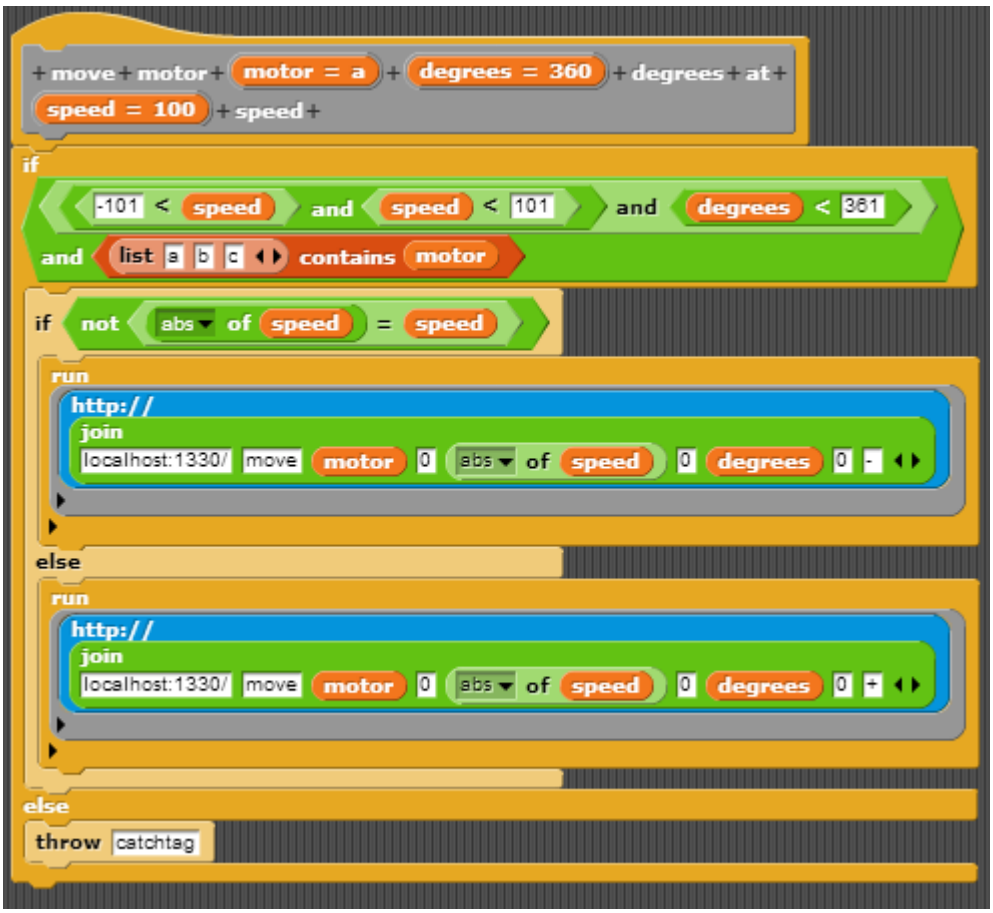


Figura 4.29: Blocco per il regolamento del motore

Come già detto all'inizio di questo paragrafo, Lego NXT può controllare fino ad un massimo di tre motori elettrici. Supponendo che questi si chiamino (a,b,c), il primo parametro ricevuto dal blocco appena visto è esattamente l'identificativo di un determinato motore. Oltre a questo gli input sono i gradi e una velocità: per i primi ovviamente si effettuerà un controllo per non inserire un numero maggiore di 360, mentre riguardo la velocità vengono accettati valori compresi fra -100 e 100. Valori di velocità negativa indicano che il motore sta girando al contrario.
Vengono poi divisi i casi in cui si abbia una velocità positiva o negativa: il controllo viene fatto sull'*if*, guardando se il modulo del valore ricevuto è uguale al valore stesso della velocità (caso velocità positiva) oppure se sono diversi (caso velocità negativa).
In ognuno dei due casi, grazie all'uso del blocco *join*, viene creata l'esatta sequenza da immettere nel blocco *http* per richiedere il servizio voluto. La sintassi questa volta è leggermente diversa (gli spazi messi qui in realtà non

devono esserci, sono aggiunti solo per una migliore lettura):
**indirizzo:porta/servizio IDmotore 0 abs(velocità) 0 gradi 0 segno-velocità**.
Gli zeri servono per differenziare un campo da un altro, *abs(velocità)* indica il valore assoluto della velocità ricevuta in input, e *segnovelocità* può avere come valori +, se la velocità è positiva, o **-**, se la velocità è negativa.
L'ultima parte dello script riguarda solamente la gestione di eccezioni, in caso di parametri non passati correttamente.

Ora che ci si è fatta un'idea generale di come funzionano i blocchi importati per NXT, possiamo tornare alla nostra procedura iniziale di configurazione di Lego NXT, al passo successivo all'importazione del file *nxt.xml*.
Si dovrà ora collegare il robot al computer attraverso un cavo USB.
Ora attraverso un prompt dei comandi (Windows) oppure un terminale (Linux), da riga di comando, è necessario spostarsi nella cartella contenente i file scaricati all'inizio di tutto il procedimento di configurazione. Una volta fatto questo si deve dare il comando **python Snap-NXT.py**.

Soffermandoci un momento su questo fatto. Risulta chiara un'ulteriore cosa: Snap! utilizza una libreria di Python, appositamente creata, per l'utilizzo di Lego NXT. Infatti, il file **Snap-NXT.py**, sempre all'interno del solito pacchetto scaricato prima, contiene il codice necessario per l'uso di Lego NXT. Se si prova ad aprire tale file, con un editor qualsiasi, si potrà osservare in che modo avviene questa configurazione.
Inizialmente, si noterà che il programma si concentra nell'assegnare, a determinate variabili appositamente create, i valori dei parametri ricevuti in input. Questi valori vengono ricavati da una determinata richiesta HTTP ricevuta (con il tipo di sintassi visto in precedenza). Ovviamente i parametri variano in base a che tipo di servizio è stato chiamato.
Vediamo ad esempio la richiesta di un servizio *move*: si ricavano i parametri di velocità, gradi e l'identificativo del motore, i quali vengono salvati rispettivamente nelle variabili power, degrees e motor (sulla velocità viene fatto il solito controllo per vedere se è positiva o negativa). Successivamente, in base a che identificativo del motore è stato ricevuto, viene chiamata la funzione *turn*, sullo specifico motore. Quanto appena detto si trova nella figura sottostante.

```
if 'move' in path:
    regex = re.compile("\/move([abc])0([0-9]+)0([0-9]+)0([+-])")
    m = regex.match(path)
    if m.group(4) == '-':
        power = -1 * int(m.group(2))
    elif m.group(4) == "+":
        power = int(m.group(2))
    degrees = int(m.group(3))
    motor = m.group(1)
    if motor == "a":
        m_a.turn(power, degrees)
    elif motor == "b":
        m_b.turn(power, degrees)
    if motor == "c":
        m_c.turn(power, degrees)
```

Figura 4.30: Codice servizio move

La struttura appena vista è analoga anche per i servizi *nxtilluminate* e *tone*. Ad esempio il codice riguardante quest'ultimo servizio, consiste nell' assegnazione dei parametri di input (suono e tempo di riproduzione) a determinate variabili (tone e time), ed alla fine l'esecuzione della specifica funzione python per riprodurre il suono voluto, ovvero *play tone and wait(tone,time)*.

Negli altri servizi rimanenti non vi sono parametri di input, ed inoltre essi devono restituire un certo valore di ritorno (cosa quindi specifica di reporter o predicate blocks), in base sempre a che servizio è stato invocato. Per meglio capire cosa succede, sono utili alcune nozioni base sul protocollo HTTP. Per il protocollo HTTP esistono due tipi di richieste e di risposte: Simple Request e Simple Response, Full Request e Full Response. La differenza fra il tipo Simple ed il tipo Full, sta nel fatto che nel primo tipo non sono presenti gli header, mentre nel secondo sì. Gli header sono determinati parametri che danno più informazioni sulla richiesta o sulla risposta; essi sono divisi in due parti: nome dell'header e valore dell'header. Un esempio di header, presente nel codice in questione, è: **Content-type, ctype** (dove ctype è una variabile che contiene il valore dell'header). Una volta che, in uno script qualsiasi di Snap!, viene eseguito il blocco *http*, contenuto in uno qualsiasi dei prototipi degli appositi blocchi per NXT, avviene il seguente procedimento:

- il blocco *http* invia una Simple Request HTTP, con i parametri specificati al suo interno, al nostro programma python

- il programma python verifica la richiesta ricevuta: se corretta, salva i parametri dei sensori del robot richiesti e successivamente costruisce una Full Response HTTP, specificando vari header e salvando tutto ciò su un file; tale file viene rimandato al mittente
- il reporter block *http*, come suo valore finale di ritorno, restituisce esattamente la risposta vista nel punto precedente, con tutti i parametri specifici del servizio

Quanto descritto finora è mostrato in figura 4.31.

```
elif path == '/nxtlight': #light
    f = open(ospath + '/nxtreturn', 'w+')
    f.write(str(l.get_sample()))
    f.close()
    f = open(ospath + '/nxtreturn', 'rb')
    ctype = self.guess_type(ospath + '/nxtreturn')
    self.send_response(200)
        self.send_header("Content-type", ctype)
        fs = os.fstat(f.fileno())
        self.send_header("Content-Length", str(fs[6]))
        self.send_header("Last-Modified", self.date_time_string(fs.st_mtime))
        self.send_header("Access-Control-Allow-Origin", "*")
        self.end_headers()
        return f
```

Figura 4.31: Codice servizio nxtlight

Il numero 200 è un codice, sempre della risposta HTTP, per indicare che la comunicazione è andata a buon fine (viene chiamato status code; ogni codice diverso implica una diversa situazione. Ad esempio il classico codice 404, indica una *bad request*, ovvero una comunicazione non andata nel modo aspettato). Gli altri servizi funzionano con lo stesso meccanismo.

Vediamo ora il corpo del *main* del programma. Dopo l'importazione delle apposite librerie, il programma controlla se esiste effettivamente un dispositivo NXT collegato; in caso non fosse così viene lanciata un'eccezione. Dopodichè avviene il settaggio delle quattro porte del robot; tale settaggio lo si può trovare sempre all'indirizzo http://technoboy10.github.io/snap-nxt/, ed è esattamente questo:

- PORTA 1: TOUCH
- PORTA 2: SOUND

- PORTA 3: LIGHT
- PORTA 4: ULTRASONIC

Inoltre avviene il settaggio delle porte per i tre motori:

- PORTA A: primo motore
- PORTA B: secondo motore
- PORTA C: terzo motore

```
try:
    b = nxt.locator.find_one_brick()
except:
    print "NXT brick not found. Please connect an NXT brick and try again."
    quit()
try:
    t = Touch(b, PORT_1)
except:
    pass
try:
    s = Sound(b, PORT_2)
except:
    pass
try:
    l = Light(b, PORT_3)
except:
    pass
try:
    u = Ultrasonic(b, PORT_4)
except:
    pass
m_a = Motor(b, PORT_A)
m_b = Motor(b, PORT_B)
m_c = Motor(b, PORT_C)
```

Figura 4.32: Impostazioni porte per Lego NXT

Successivamente viene aperto un socket (il canale di comunicazione fra richieste e risposte) con protocollo TCP (HTTP è usato al livello applicazione, mentre TCP al livello di trasporto); tale socket funzionerà sulla porta 1330, porta il cui scopo è già stato presentato precedentemente.
Infine, se l'intera configurazione ha avuto successo, viene stampato un output (che sarà visibile direttamente su Snap!); l'ultima istruzione specifica di lasciare il socket attivo per sempre, o meglio fino ad un'interruzione voluta dall'utente.

```
PORT = 1330

Handler = CORSHTTPRequestHandler
#Handler = SimpleHTTPServer.SimpleHTTPRequestHandler

httpd = SocketServer.TCPServer(("", PORT), Handler)

print "serving at port", PORT
print "Go ahead and launch Snap!."
print "<a>http://snap.berkeley.edu/snapsource/snap.html#open:http://localhost:1330/Snap-NXT</a>"
httpd.serve_forever()
```

Figura 4.33: Avvio di connessione con Lego NXT

Ora, se tutte queste operazioni sono avvenute con successo, Lego NXT è pronto per essere comandato tramite Snap!.

### 4.4.4 Esempio di comunicazione tra Snap! ed un web server locale

Un possibile esempio di come avverrebbero le comunicazioni fra Snap! ed un piccolo web server locale, utile al fine di comprendere meglio ciò che si è visto nel paragrafo precedente (richieste e risposte HTTP con Lego NXT), verrà brevemente presentato in questo paragrafo.
Il server, realizzato in Java, si mette in ascolto sulla porta 8000 (scelta a caso, l'importante è che non sia già riservata per qualcosa) e quando Snap!, tramite sempre il blocco *http*, manda una Simple Request HTTP, il server invia come risposta il valore attuale di uno slider, regolato a piacere dall'utente.
Anche lo slider è stato realizzato in Java, con una sua interfaccia grafica a finestra, gestita come al solito con la classica libreria *swing* di Java.

```
package com.example;

import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;

import com.example.Test.MyHandler;
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpServer;


import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class CreateSlider{
  static JSlider slider;
  JLabel label;
  public static void main(String[] args) throws Exception{
      HttpServer server = HttpServer.create(new InetSocketAddress(8000), 0);
      server.createContext("/slider", new MyHandler());
      server.setExecutor(null); // creates a default executor
      server.start();
  CreateSlider cs = new CreateSlider();
  }
```

Figura 4.34: Prima parte programma

```
static class MyHandler implements HttpHandler {
    public void handle(HttpExchange t) throws IOException {
        String response = "<html><head><h3>Valore attuale dello slider</h3>
                          </head><body>"+ slider.getValue()+"</body></html>";
        t.sendResponseHeaders(200, response.length());
        OutputStream os = t.getResponseBody();
        os.write(response.getBytes());
        os.close();
    }
}
public CreateSlider(){
JFrame frame = new JFrame("Slider");
slider = new JSlider();
slider.setValue(50);
slider.addChangeListener(new MyChangeAction());
label = new JLabel("50");
JPanel panel = new JPanel();
panel.add(slider);
panel.add(label);
frame.add(panel, BorderLayout.CENTER);
frame.setSize(400, 100);
frame.setVisible(true);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
public class MyChangeAction implements ChangeListener{
public void stateChanged(ChangeEvent ce){
int value = slider.getValue();
String str = Integer.toString(value);
label.setText(str);
}
}
```

Figura 4.35: Seconda parte programma

La richiesta sarà strutturata in questo modo: **localhost:8000/slider**
Nella figura successiva vi è un esempio di esecuzione.


Figura 4.36: Esempio di esecuzione

Tale codice è stato testato ed è funzionante con un qualsiasi browser; tuttavia su Snap! non viene rilevata (e quindi non viene visualizzata) nessuna risposta, nemmeno in caso di errore.
Sono state inoltre controllate, in entrambi i casi (ovvero browser e Snap!), sia le richieste che le risposte sul programma Java, ed esse sono risultate uguali.
Attualmente il blocco *http* infatti, utilizzando diversi semplici URL, non sempre fornisce una risposta. Forse necessita di ulteriori modifiche, oppure funziona correttamente con determinate opzioni che però non mi sono note.

# Capitolo 5

# Una versione semplificata di Pacman

La realizzazione di questo videogame contiene alcune delle nozioni sui sensori viste nel capitolo precedente, in particolare quelle riguardanti movimenti, ostacoli e l'uso congiunto di variabili per la segnalazione di altri comportamenti da prevedere. Tale gioco è realizzabile con uno qualsiasi dei programmi visti in questa tesina.
Le regole sono poche e semplici:

- lo sprite di pacman (a sinistra in figura 5.1) si deve muovere all'interno del labirinto dello stage, cercando di mangiare tutte la palline disposte su quest'ultimo e nel contempo di evitare il contatto con i quattro fantasmi (a destra in figura 5.1)che si muoveranno sempre nel labirinto a guardia di queste.

Figura 5.1: Lo sprite di pacman e di uno dei quattro fantasmi

- una volta mangiate tutte le palline si passerà al livello successivo, contenente più componenti da raccogliere, fino al livello finale (in questo caso il terzo, ma si può impostare a scelta)
- completando il livello finale si finisce il gioco vincendo
- ad ogni contatto con un fantasma si perde una vita
- se si perdono tutte le vite prima di aver completato l'ultimo livello, il gioco finisce con una sconfitta

Vediamo ora di quali oggetti, con i loro relativi script, è composto il gioco.

## 5.1 Script di un fantasma

Ognuno dei quattro fantasmi presenti nello stage possiede tre script:

- un primo script per far eseguire un determinato giro nel labirinto dello stage, presente in figura 5.2. Le coordinate sono state ottenute posizionando di volta in volta il fantasma nelle posizioni volute in maniera consecutiva, ovvero ogni blocco *glide* corrisponde ad un movimento con un orientamento diverso mentre tutti i blocchi *glide* formano un vero e proprio percorso. Le coordinate saranno ovviamente diverse per ogni fantasma, sulla base di che percorso gli si vuole far eseguire

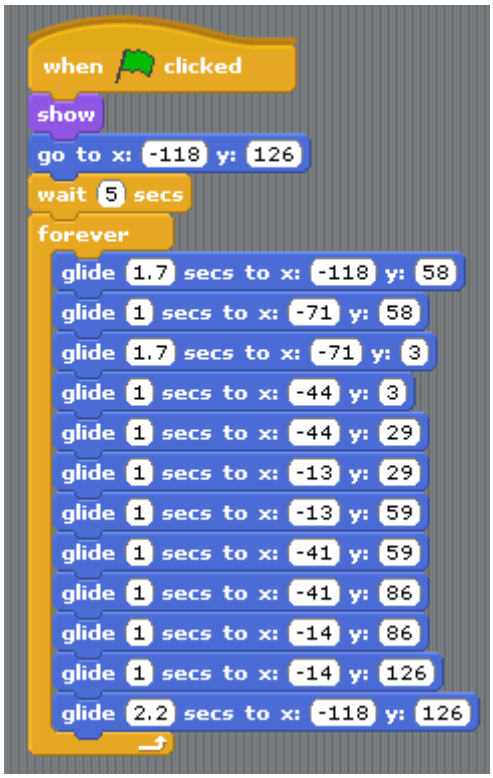


Figura 5.2: Movimento di un fantasma

- uno script per segnalare che lo sprite di pacman è stato colpito (e quindi una vita sarà andata persa) e per far ripartire il fantasma dalla sua posizione iniziale quando invece un livello del gioco è stato completato

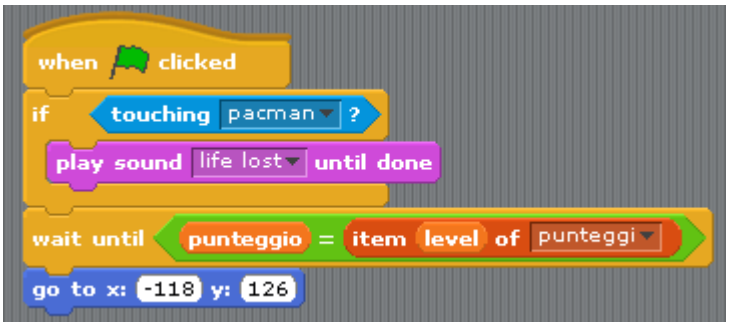


Figura 5.3: Contatto con pacman e cambio di livello

- un piccolo script per nascondere il fantasma quando il gioco sarà completato, concludendo tutti i livelli (ovvero raggiungendo l'ultimo elemento dell'array *punteggi* precedentemente creato)

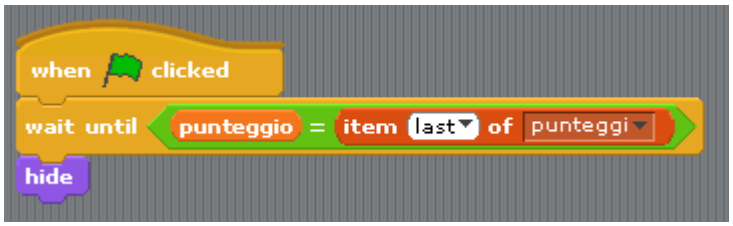


Figura 5.4: Fantasma nascosto alla fine del gioco

## 5.2 Script delle palline

Ci sono due tipi di sprite di questo tipo:

1. palline del livello base
2. palline che compaiono in determinati livelli successivi

La prima classe possiede uno script (figura 5.5)per incrementare la variabile *punteggio* se c'è il contatto con lo sprite di pacman; in tal caso tramite il blocco *hide* tale sprite uscirà di scena per il momentaneo livello attuale di gioco.

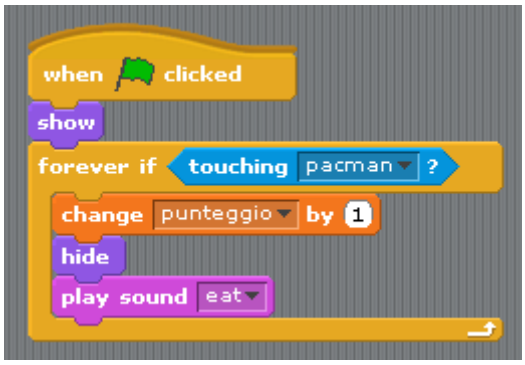


Figura 5.5: Contatto pallina con sprite pacman

Serviranno poi ovviamente degli altri script per far ricomparire le palline in ogni livello successivo. Ad esempio per far ricomparire uno di questi sprite nel secondo livello si deve comporre lo script di figura 5.6.

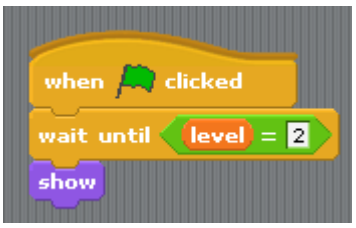


Figura 5.6: Ricomparsa pallina nel secondo livello

Per quanto invece riguarda la seconda classe degli sprite in questione, lo script di contatto con pacman è leggermente diverso: all'inizio lo sprite è nascosto e rimane in questo stato finchè non si raggiunge il livello in cui si vuole far comparire la pallina. Quando accadrà questo lo sprite diventerà attivo tramite il blocco *show* e per ogni livello successivo il procedimento sarà poi analogo a quello delle palline di base. Ad esempio in figura 5.7 vi è lo script di una pallina che compare dal secondo livello in poi fino all'ultimo livello (il terzo nell'esempio).

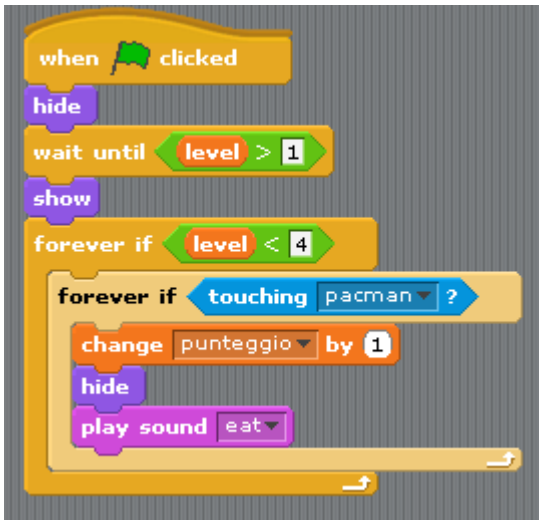


Figura 5.7: Palline che compaiono in livelli successivi

Ovviamente anche qui saranno necessari degli script per far ricomparire tali sprite ad ogni livello successivo (tranne per le palline dell'ultimo livello): rispetto a prima cambia solo il fatto che ce ne saranno appunto sempre meno, man mano che si procede verso il livello finale.
Finora le palline viste incrementavano il punteggio di una sola unità, ma normalmente si creano anche delle palline di categoria più elevata, cioè che valgono un punteggio più alto se conquistate. Basta modificare il costume dello sprite di una pallina di base (magari cambiando dimensioni oppure il colore) e specificando nello script di figura 5.7 che il punteggio verrà incrementato di una certa quantità (ad esempio 5 punti).

## 5.3 I portali

Sono stati creati due portali con lo scopo di teletrasportare pacman da una parte all'altra dello stage se toccati. Tali sprite non possiedono alcuno script, servono solo come riferimento per lo sprite di pacman. Sono stati realizzati con un colore molto simile allo sfondo per renderli invisibili e sono stati chiamati rispettivamente *left* e *right*.

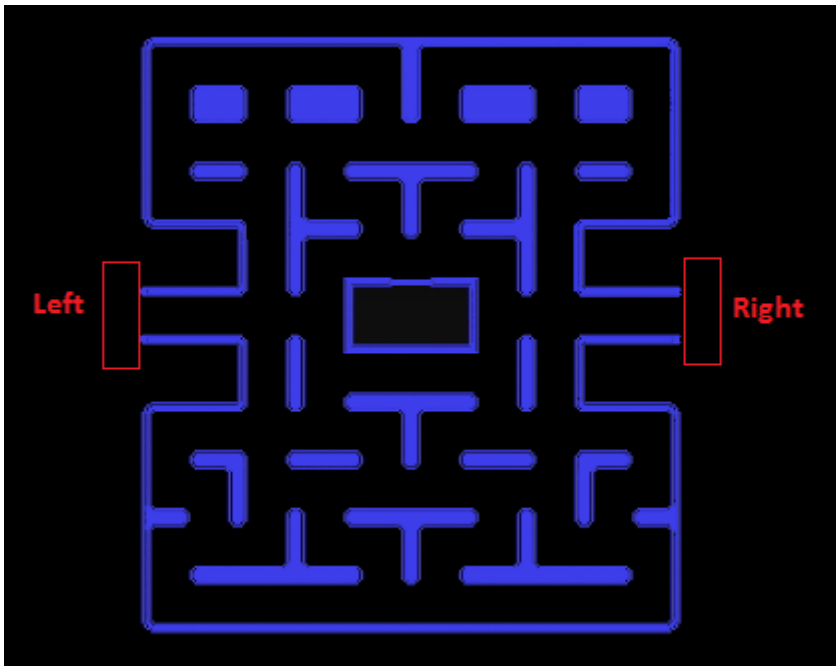


Figura 5.8: I portali

## 5.4 Sprite di notifica

### 5.4.1 Caso di vittoria

Un semplice sprite, che altro non è che un immagine con il testo You win!, è stato creato per essere visualizzato quando e se si riesce a completare il gioco. I suoi script sono molto semplici e sono mostrati in figura 5.9.
In sostanza la scritta rimane nascosta all'inizio e compare solo nel caso in cui la variabile *punteggio* non ha raggiunto il punteggio massimo fissato. Inoltre questo sprite rimane in attesa di un messaggio di broadcast chiamato FINE, il quale sarà lanciato dallo sprite di pacman (si veda sezione 5.5); quando avverrà ciò, verrà fermata l'esecuzione di tutti gli script del programma.

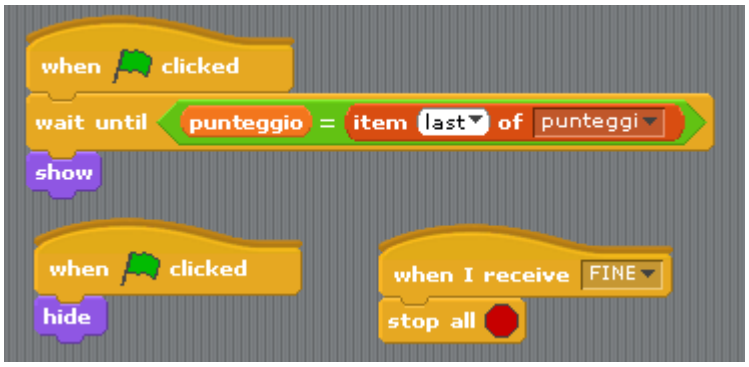


Figura 5.9: I tre script della vittoria

### 5.4.2 Caso di game over

Analogamente allo sprite appena visto nella sezione precedente, vi è uno sprite che mostra la frase GAME OVER!, nel caso il gioco termini con una sconfitta. Tale condizione avviene se la variabile *vite* diventa negativa. A parte un ciclo infinito che controlla questa condizione, il tutto è molto simile al caso di vittoria, tranne per il fatto che qui non c'è nessun messaggio di broadcast.

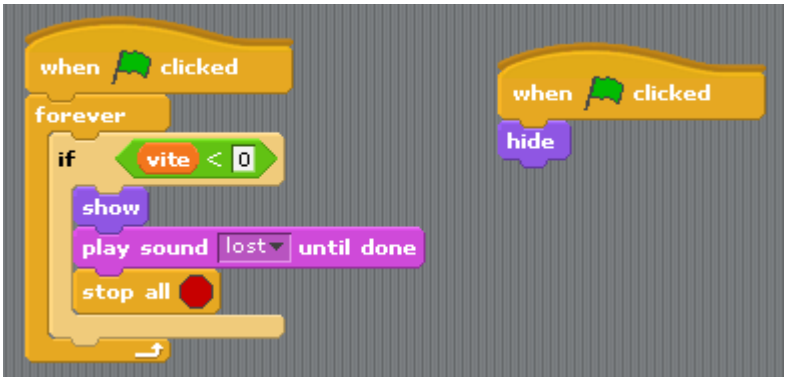


Figura 5.10: I due script della sconfitta

### 5.4.3 I livelli

Ancora altro testo da visualizzare nei momenti opportuni. All'inizio del gioco verrà visualizzata la scritta GET READY!, assieme in alto al testo LEVEL 1. Il loro script banale è in figura 5.11.

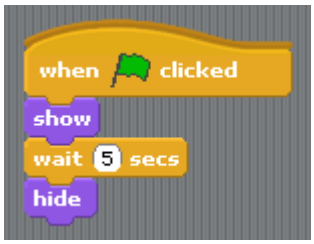


Figura 5.11: Le scritte di inizio gioco

Dopodichè ci sarà uno sprite per ogni livello successivo che segnale quando esso è stato raggiunto. Ad esempio per il secondo livello avrò lo script di figura 5.12.

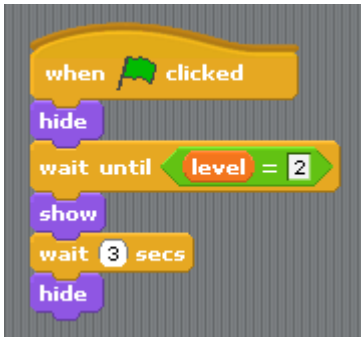


Figura 5.12: Segnalazione secondo livello

### 5.4.4 Le vite

E' un fatto classico del videogame di pacman segnalare sullo schermo quante vite rimangono al giocatore, dove ognuna di esse è identificata da un'immagine simile allo sprite di pacman. Quando si perde una vita semplicemente una di quelle immagini scompare. Di norma si hanno tre vite di scorta oltre a quella iniziale. Serviranno dunque tre sprite per ognuna di esse che saranno visibili o nascosti a seconda dell'attuale valore della variabile *vite*. In figura è mostrato lo script per la seconda vita, gli altri sono analoghi.

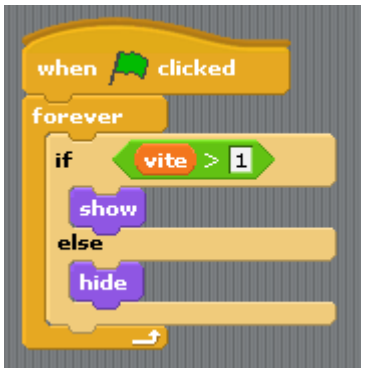


Figura 5.13: La seconda vita

## 5.5 Sprite centrale di Pacman

Come si potrà immaginare questo sprite è la colonna portante di tutto il sistema (il nostro sprite robot). In esso andranno fatti la maggior parte dei controlli e degli aggiornamenti delle variabili, con inoltre la gestione dei movimenti.
Innanzitutto si parte con la definizione e configurazione iniziale di una lista chiamata *punteggi*, dove in essa verranno salvati, in ordine crescente rispetto al livello e sulla base di quanti di questi siano stati creati, tutti i punteggi massimi di ogni rispettivo livello. Tale lista sarà poi utile qui ed anche in altri sprite per verificare se un determinato livello è stato cmpletato o meno. La configurazione di esempio (con tre livelli) è mostrata in figura 5.14.

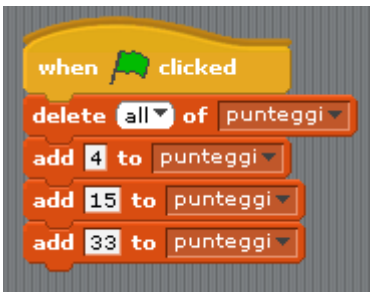


Figura 5.14: Configurazione iniziale punteggi

Questo script dovrà essere eseguito da solo almeno una volta prima di poter avviare il gioco.

Fatto ciò è il turno di gestire i movimenti. Conviene in questo caso selezionare l'opzione *only face left-right* (in alto nel programma, vicino alla figura

dello sprite), per far sì che tutti i costumi di Pacman funzionino correttamente, senza capovolgere completamente l'immagine dello sprite. Detto questo, saranno necessari ben quattro costumi, uno per ogni tasto di movimento (le frecce direzionali). Ognuno di questi sarà selezionato in base a che tasto direzionale sarà premuto dall'utente, grazie ad un opportuno script; ognuno di questi quattro script utilizzerà il blocco *<key [input v] pressed?>*, già visto anche nella sezione dedicata alla robotica. In caso positivo il blocco *point in direction* orienterà nel corretto modo lo sprite. Uno script di esempio per il tasto *right arrow* è mostrato in figura 5.15.

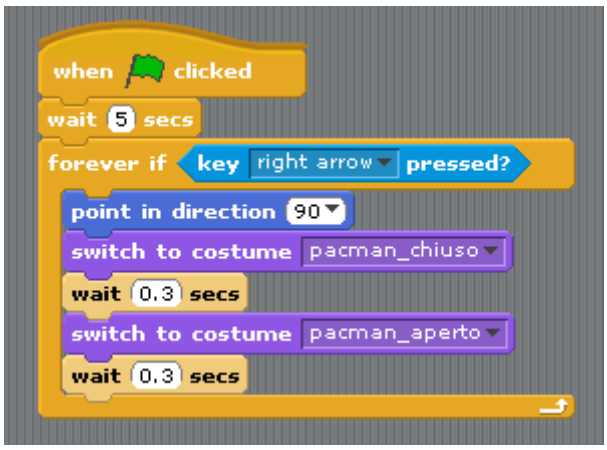


Figura 5.15: Movimento verso destra di Pacman

Ed ora è il momento dello script di controllo. Essendo piuttosto grande, verrà spezzato in più figure consecutive.

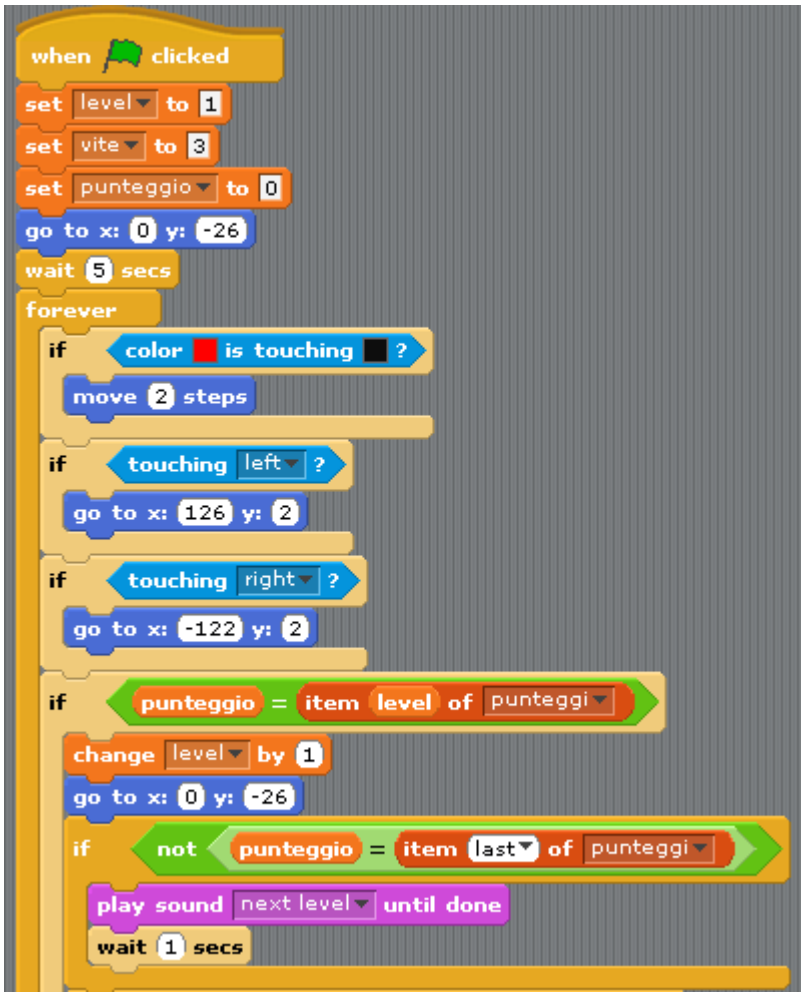


Figura 5.16: Prima parte script centrale di pacman

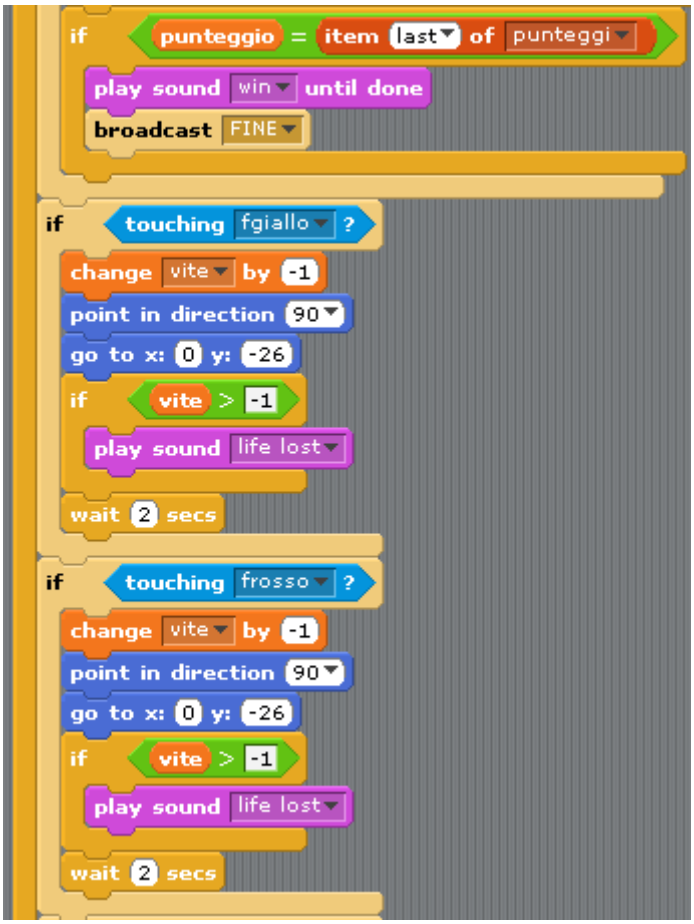


Figura 5.17: Seconda parte script centrale di pacman

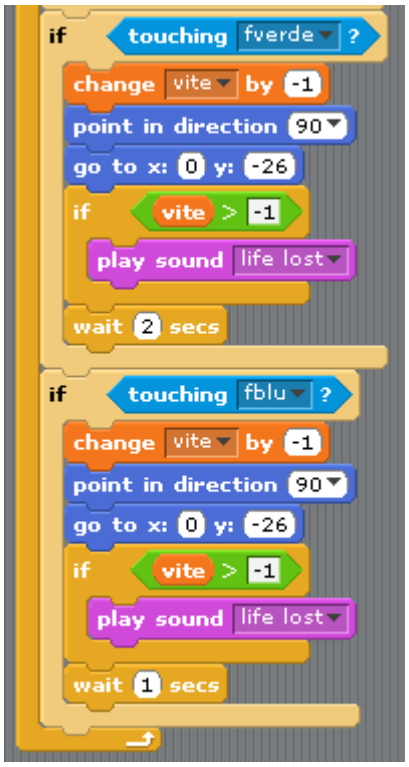


Figura 5.18: Terza parte script centrale di pacman

Dopo un breve settaggio delle variabili necessarie, vediamo la spiegazioni dei componenti all'interno del ciclo *forever*:

- il primo *if* fa praticamente la stessa cosa che avevano visto nella robotica. Lo sprite di pacman ha un rettangolo rosso (invisibile perchè troppo piccolo una volta aperto il videogame), tramite il cui colore si può specificare se muovere o meno lo sprite
- il secondo ed il terzo *if* esegue il teletrasporto da una parte all'altra dello stage nel caso si toccasse un portale
- l'*if* successivo verifica se è stato completato il livello. In caso positivo incrementa la variabile *level* e riporta lo sprite di pacman alla posizione iniziale. Dopodichè controlla se si tratta della fine di un normale livello o del livello finale, ed in questo ultimo caso invia un messaggio di broadcast per avvisare
- tutti gli *if* successivi verificano, sempre tramite l'uso di un blocco sensore, se lo sprite di pacman viene o meno a contatto con un fantasma. Nel caso fosse così si decrementa la variabile *vite* e si riporta pacman alla sua posizione iniziale.

Con tutti gli elementi fin qui descritti il videogame è pronto per essere testato.

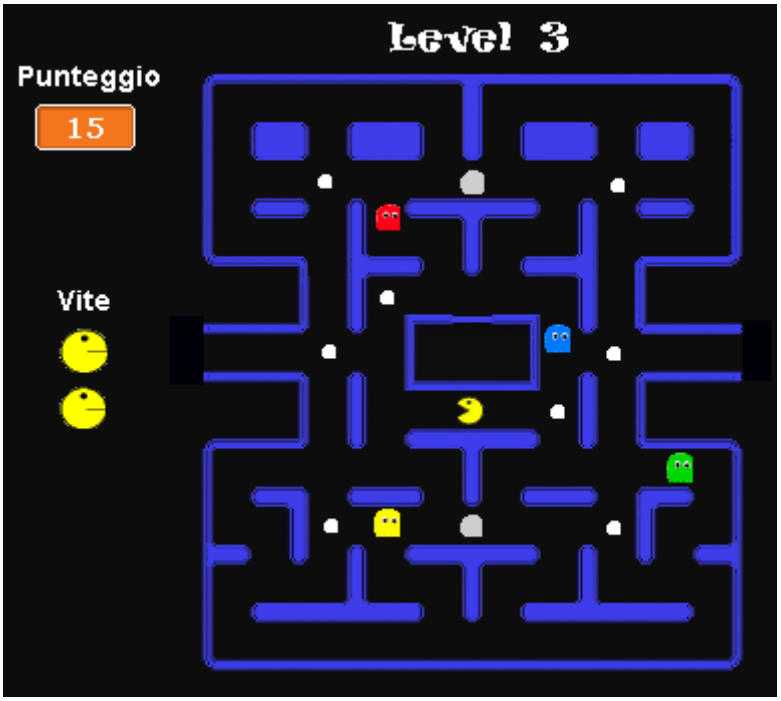


Figura 5.19: Gioco di Pacman completo

# Conclusioni

Quanto descritto finora mostra come i linguaggi Scratch,BYOB e Snap! possano essere usati in svariati ambiti educativi. Snap! è l'ultimo arrivato nel processo evolutivo di questa famiglia di programmi e dimostra di possedere molte utili funzionalità che sicuramente poi verrano a sua volta ampliate nel corso degli anni. Gli ideatori hanno cercato di creare un programma che mantenesse tutte le numerose funzionalità delle versioni precedenti di BYOB, ma aggiungendo tratti di ulteriore versatilità, in modo da semplificare maggiormente l'adattamento all'uso da parte di più utenti in più ambiti e di conseguenza anche facilitare il loro lavoro, nascondendo loro svariati aspetti piuttosto complicati di basso livello che comunque il programma contiene.

Tutti i programmi visti discendenti da Logo hanno una loro possibile associazione al campo della robotica: grazie alla connessione di determinati dispositivi come le Picoboard, le schede Arduino oppure anche semplicemente tramite l'utilizzo di strumenti ormai diventati di uso comune come un microfono o una webcam, si possono controllare mediante l'uso di sensori appropriati e creati da uno di questi programmi, alcuni dei loro parametri di funzionamento.

# Bibliografia


[1] *http://snap.berkeley.edu/*

[2] *http://scratch.mit.edu/forums/.*

[3] *http://snap.berkeley.edu/SnapManual.pdf*

[4] Javier Arlegui,Michele Moro,Alfredo Pina. *How to enhance the robotic experience with Scratch.* Constructionism 2012, Athens, Greece

[5] Javier Arlegui,Michele Moro,Alfredo Pina. *Simulation of robotic sensors in BYOB.*

[6] *http://it.wikipedia.org/*