Programmare giochi per Atari 2600 in Batari Basic

Versione 0.1 Dicembre 2025

By E-Paper Adventures

epaperadventures@gmail.com

Sommario

Introduzione

Nel lontano 1977, nelle case di tutto il mondo, apparve una piccola scatola di legno e plastica che avrebbe cambiato per sempre il modo di giocare: l'**Atari 2600**. In un'epoca senza smartphone e senza Internet, dove i televisori avevano schermi curvi e i telefoni erano attaccati al muro con un filo, questa console era pura ingegneria creativa. Permetteva a chiunque di controllare mondi fatti di blocchi colorati e suoni elettronici direttamente dal salotto di casa.

Foto: Sergey Galyonkin, CC BY-SA 2.0.

L'Atari 2600 non era solo un prodotto; fu un fenomeno culturale che definì un'intera generazione, vendendo oltre **30 milioni di unità** nel mondo. Per questa console furono scritti più di 500 giochi ufficiali, con centinaia di milioni di cartucce vendute, tra cui capolavori come *Pitfall!* che da solo superò i 4 milioni di copie.

Per riuscirci, i programmatori di allora compirono veri e propri miracoli. Immagina di dover costruire un grattacielo con una manciata di mattoncini LEGO. L'Atari 2600 aveva solo **128 byte di memoria RAM**. Non megabyte, non kilobyte. *Byte*. Per darti un'idea, uno smartphone moderno ha una quantità di memoria RAM almeno **30 milioni di volte superiore**. Le cartucce dei giochi contenevano tipicamente 2 o 4 kilobyte di ROM (memoria di sola lettura), mentre un gioco moderno può superare i 50 gigabyte. La bravura di quei pionieri non consisteva nell'usare una potenza infinita, ma nel creare divertimento, avventura ed emozione dal quasi nulla.

Questo libro è il tuo biglietto per quel mondo. Non stai solo per imparare a scrivere codice. Stai per imparare a pensare come i pionieri dei videogiochi, armato solo di ingegno e di una manciata di byte. Scoprirai che programmare per l'Atari 2600 è una delle sfide più gratificanti che esistano. Nell'era moderna, dove la potenza di calcolo è quasi illimitata, è facile perdersi in grafiche fotorealistiche e mondi sconfinati. Ma su questa console, le regole sono diverse. Con

così poche risorse a disposizione, non puoi affidarti alla tecnologia per creare il divertimento. Devi inventarlo. Qui, il vero "sale" del game design emerge nella sua forma più pura. L'ingegno non è un'opzione, è l'unico strumento che hai. Ogni byte risparmiato, ogni trucco per simulare un movimento fluido, ogni scelta di colore per rendere leggibile un nemico, diventa una piccola vittoria. Il focus si sposta inevitabilmente dalla complessità visiva alla giocabilità: un gioco Atari ha successo solo se è divertente, immediato e intelligente. La sfida sta nel distillare un'idea fino alla sua essenza, creando un'esperienza coinvolgente con quasi nulla. È un'arte che ti costringerà a diventare un programmatore più creativo e consapevole.

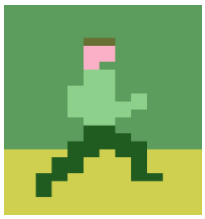
Stai per scoprire i segreti di una macchina leggendaria e usare i suoi stessi limiti come fonte di ispirazione per creare un videogioco tutto tuo. E non sei solo: ancora oggi, una vivace community di appassionati, chiamata *homebrew*, continua a creare giochi nuovi di zecca per questa console, spingendo i suoi limiti oltre ogni immaginazione. Molti di questi nuovi giochi vengono persino venduti su cartucce fisiche, a dimostrazione della vitalità immortale di una piattaforma dove l'inventiva conta più di ogni altra cosa.

Le Icone speciali del manuale

Mentre esploreremo Batari Basic, vedrai spesso dei ritagli grafici del gioco leggendario: **Pitfall!** Creato nel 1982 dal geniale programmatore David Crane per Activision, *Pitfall!* non era solo un gioco: era una rivoluzione tecnologica. In un'epoca in cui i giochi erano spesso limitati a una singola schermata, *Pitfall!* presentava un mondo vasto e interconnesso di moltissime schermate, pieno di giungle, sabbie mobili e tesori. Il suo protagonista, Pitfall Harry, mostrava un'animazione fluida che sembrava impossibile per l'hardware di allora. *Pitfall!* è diventato il simbolo di ciò che è possibile ottenere quando l'ingegno del programmatore supera i limiti dell'hardware. È la nostra stella polare in questo viaggio.

In questo manuale incontrerai delle icone speciali:

	**Pitfall!**	La testa di un coccodrillo ti avverte di una trappola pericolosa: un errore comune, una limitazione hardware o un concetto difficile. Presta la massima attenzione!
	**Consiglio prezioso**	Un lingotto d'oro rappresenta un consiglio prezioso, un "tesoro" di conoscenza. È un trucco del mestiere o un suggerimento che renderà il tuo codice più elegante ed efficiente.
	**Approfondimento Tecnico**	Una scaletta che scende indica che stiamo per analizzare un concetto in profondità, svelando i meccanismi interni dell'hardware o del software.



Prova Tu!

La figura del nostro eroe, Pitfall Harry, ti invita all'azione. È un esercizio pratico, una sfida per mettere alla prova le abilità che hai appena imparato.

Perchè questo manuale?

Un tempo, soprattutto negli anni '80, generazioni di programmatori sono nati così: collegando un computer al televisore di casa e iniziando a digitare comandi in un linguaggio chiamato BASIC. Quel primo *PRINT "CIAO"* su uno schermo a tubo catodico era un *imprinting* incredibile. Era il momento in cui si scopriva di poter dare ordini a una macchina, di poter tradurre un pensiero in un'azione. Si imparava sul campo il concetto di algoritmo, il flusso di un programma, l'arte della caccia al "bug" e l'importanza di tenere il manuale sempre a portata di mano.
Oggi, l'approccio iniziale alla programmazione è spesso visuale, fatto di blocchi colorati da trascinare e risultati grafici immediati. Ma scrivere codice testuale su uno "schermo nero" ha qualcosa di ancestrale, un potere unico. Costringe a visualizzare il flusso nella propria mente, a tracciare lo stato delle variabili, a trasformare un'idea in una sequenza logica di istruzioni. È un esercizio che non insegna solo a programmare, ma a pensare come un programmatore, costruendo fondamenta logiche solide.
Programmare in Batari Basic l'Atari 2600, con le sue incredibili limitazioni, è la palestra perfetta per forgiare l'ingegno:

- Una risoluzione grafica bassissima
- Solo due sprite (*player0*, *player1*), due missili (*missile0*, *missile1*) e una palla (*ball*).
- Uno sfondo (*Playfield*) a blocchi
- Due canali audio con suoni molto caratteristici
- Appena 26 variabili intere (*byte*) per tutta la logica del tuo gioco.
- Poco più di un centinaio di righe per il tuo codice

In un mondo di abbondanza tecnologica, queste limitazioni potrebbero sembrare insormontabili. Invece, sono la nostra più grande opportunità. Con così poche armi a disposizione, non puoi affidarti alla grafica mozzafiato; devi concentrarti su ciò che rende un gioco davvero interessante: la giocabilità, la fantasia, l'inventiva. La sfida non è creare un gioco nonostante i limiti, ma creare un bel gioco grazie a essi, spremendo ogni byte e scoprendo trucchi incredibili per superare ciò che sembra impossibile.
E la ricompensa finale è qualcosa che l'era del "tutto e subito" ha quasi dimenticato: la soddisfazione di creare, con fatica e ingegno, qualcosa di tangibile. E il grande vantaggio di Batari Basic è che il tuo programma può anche diventare una vera cartuccia, da inserire nella console più importante della storia, da giocare sul televisore con amici e familiari.

Ringraziamenti, Fonti e Licenza

Nel preparare questo manuale, si è attinto alla conoscenza collettiva della community Atari, un tesoro accumulato in decenni di passione. Si desidera ringraziare in particolare le seguenti due fantastiche fonti, che sono state un punto di riferimento indispensabile:

- Il sito *Random Terrain's Batari Basic Page*, l'enciclopedia definitiva su Batari Basic, curata con dedizione e competenza.
- La serie di articoli "Programmare il 2600" di Giorgio Balestrieri sulla rivista *RETROMAGAZINE*, una fonte preziosa di informazioni.

Nel manuale sono inoltre presenti diversi listati di programmi trovati online e riadattati. Laddove possibile, i relativi autori sono stati citati.

Licenza d'Uso e Disclaimer

Parte 1: Le Basi della Programmazione in Batari Basic

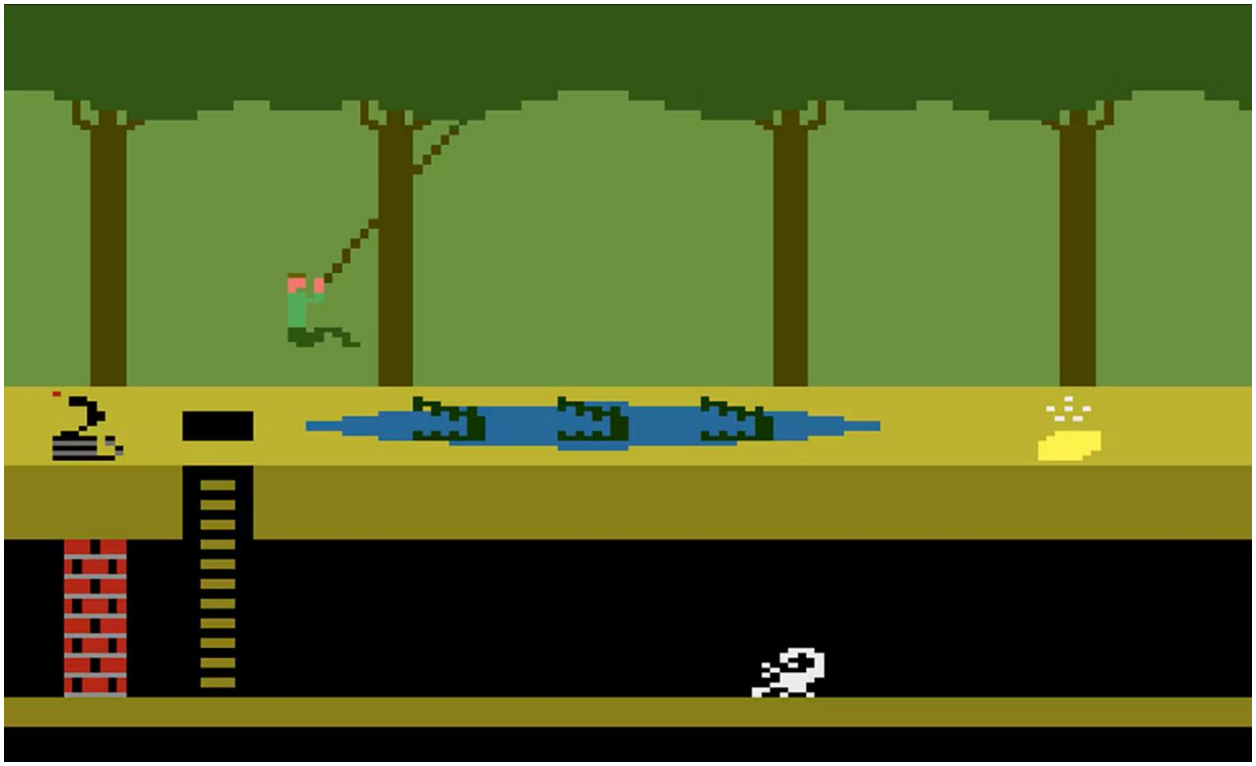


Immagine del gioco Pitfall! di David Crane per Activision

Capitolo 1 – I Tuoi Attrezzi

Prima di partire per il passato, hai bisogno della sua attrezzatura. In questo capitolo, prepareremo insieme la nostra "officina digitale": un luogo speciale sul tuo computer moderno per programmare l'Atari 2600. Non preoccuparti, anche se la nostra destinazione è "vintage", i nostri strumenti saranno moderni, potenti e facili da usare.

Foto: Sergey Galyonkin, CC BY-SA 2.0.

1.1 – Un Salto nel Tempo

Immagina di tornare nel 1977. I computer personali sono ancora un sogno per pochi e i videogiochi sono una novità esplosiva confinata nelle sale giochi. In quell'anno, Atari lancia una scatola magica che cambierà tutto: l'**Atari Video Computer System**, che il mondo imparerà a conoscere come **Atari 2600**.
Per la prima volta, intere famiglie potevano giocare a titoli come *Pac-Man*, *Space Invaders* e *Pitfall!* direttamente sul televisore di casa, grazie a delle cartucce intercambiabili. La 2600 non era solo una console, era un fenomeno culturale che ha definito un'intera generazione.

Oggi, a decenni di distanza, potresti pensare che sia solo un pezzo da museo. E invece no! Un'incredibile comunità di appassionati, chiamati **homebrew developer** (sviluppatori casalinghi), continua a creare giochi nuovi di zecca per questa console, spingendo i suoi limiti oltre ogni immaginazione. E tu stai per diventare uno di loro.

Il nome in codice originale dell'Atari 2600 era "Stella". Si dice che fosse il nome della bicicletta di uno degli ingegneri. Anche se il nome ufficiale divenne un altro, quel nomignolo è rimasto nel cuore degli appassionati. Non a caso, il più famoso programma per provare i giochi Atari 2600 sul computer si chiama proprio **Stella**!

1.2 – Il linguaggio Batari Basic

Per dare ordini alla nostra console, abbiamo bisogno di un linguaggio che possa capire. L'Atari 2600 parla solo un linguaggio numerico (fatto di 0 e 1) molto complesso, quasi incomprensibile. Noi, invece, parliamo una lingua umana. Come facciamo a comunicare?
Usando un traduttore speciale: il linguaggio **Batari Basic**.

Se hai già sentito parlare del linguaggio BASIC, forse lo associ a computer come il *Commodore 64*. Quei BASIC erano come dei traduttori simultanei: ascoltavano un comando e lo traducevano all'istante. L'Atari 2600, però, è una creatura molto più semplice e non ha abbastanza potenza per una traduzione dal vivo. Ha bisogno di ricevere istruzioni già perfettamente tradotte.

Ecco perché il Batari Basic (spesso abbreviato in **bB**) è così speciale:

- **Parla la lingua dell'Atari.** Invece di tradurre al momento, il Batari Basic agisce come un traduttore che prepara un intero "libro di istruzioni" (il nostro gioco) in un file che una vera cartuccia Atari 2600 (o un emulatore) può eseguire. Questo processo si chiama **compilazione**.
- **Conosce le regole della macchina.** La sua sintassi, a volte un po' strana, è stata creata appositamente per "dialogare" con i chip della console, rispettando le sue incredibili limitazioni.
- **È un ponte tra la semplicità e la complessità.** Non devi essere un genio dell'assembly (il linguaggio a bassissimo livello) per iniziare, ma mentre programmi in bB, impari a "sentire" come ragiona la macchina.

1.3 – Assemblare l'Officina

Anche se la console è antica, la nostra officina sarà modernissima. Useremo due strumenti principali che lavoreranno insieme.

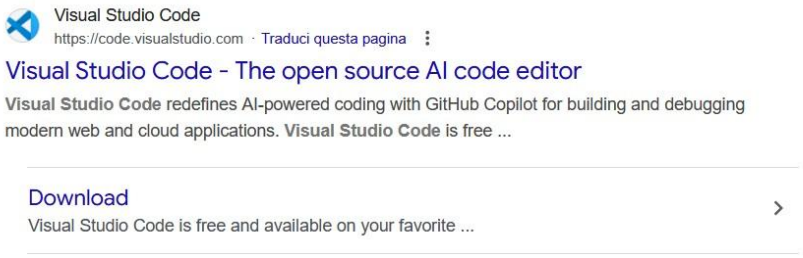
- **L'IDE (Visual Studio Code + Atari Dev Studio): La nostra cassetta degli attrezzi digitale.** Un Ambiente di Sviluppo Integrato (IDE) è un programma che contiene tutti gli strumenti di cui abbiamo bisogno. Useremo **Visual Studio Code**, un editor di testo molto popolare, con un'estensione speciale chiamata **Atari Dev Studio**. Insieme, ci daranno:
  - Un editor intelligente che colora il codice e ci suggerisce i comandi.
  - Il "traduttore" Batari Basic integrato.
  - Un pulsante magico (F5) per compilare il gioco e avviarlo all'istante!
- **L'Emulatore (Stella): Un simulatore per la nostra macchina del tempo.** Invece di usare una vera console Atari 2600, useremo un **emulatore**: un programma che *finge* di essere una console Atari sul tuo computer. Il migliore è **Stella** (proprio come il nome in codice!), e *Atari Dev Studio* lo userà automaticamente per lanciare i tuoi giochi.

È ora di assemblare la nostra officina. I passi principali sono:

- **Scarica e installa Visual Studio Code** dal suo sito ufficiale (cerca "Visual Studio Code" sul tuo motore di ricerca preferito) per il sistema operativo del tuo computer (Windows, MacOS, …)
- **Aprilo.** Sulla barra laterale sinistra, cerca un'icona con dei quadratini: è la sezione "**Estensioni**". Cliccaci sopra.
- Nella barra di ricerca che appare, digita "**Atari Dev Studio**" e premi Invio.
- Clicca sul pulsante "**Installa**" accanto all'estensione.
- Una volta finita l'installazione, **riavvia Visual Studio Code**.

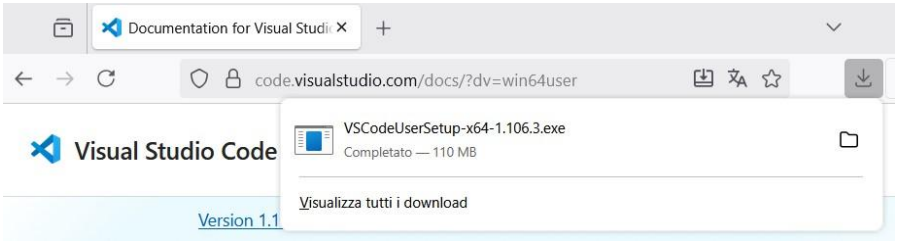
Qui di seguito la procedura in dettaglio per Windows.

Vai sulla sezione download del sito di Visual Studio Code:



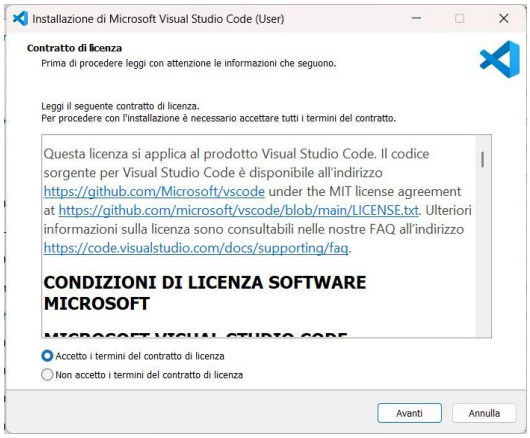
Scarica la versione per Windows:

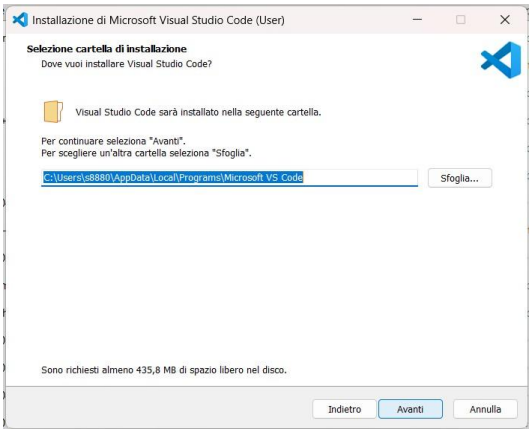
Attendi che il download si completi:

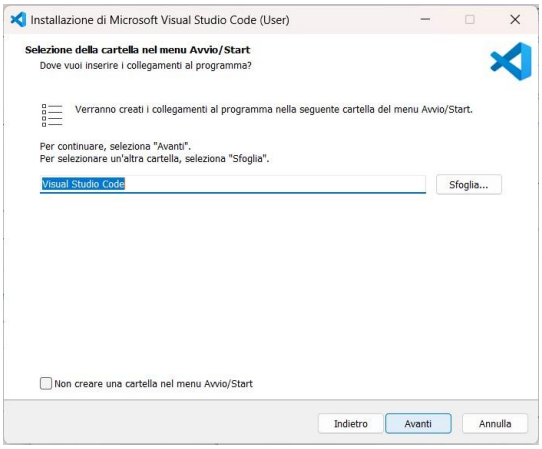


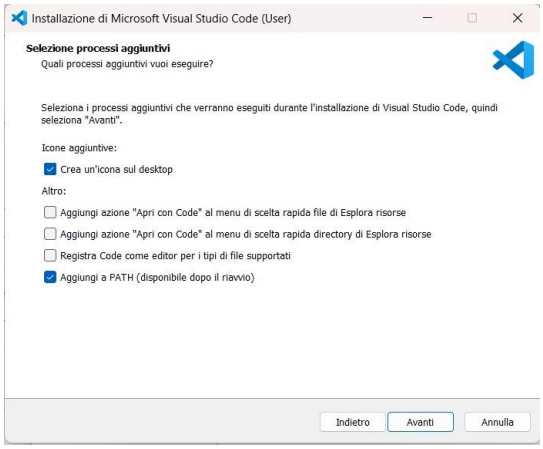
Procedi con l'installazione come indicato nelle immagini qui di seguito.

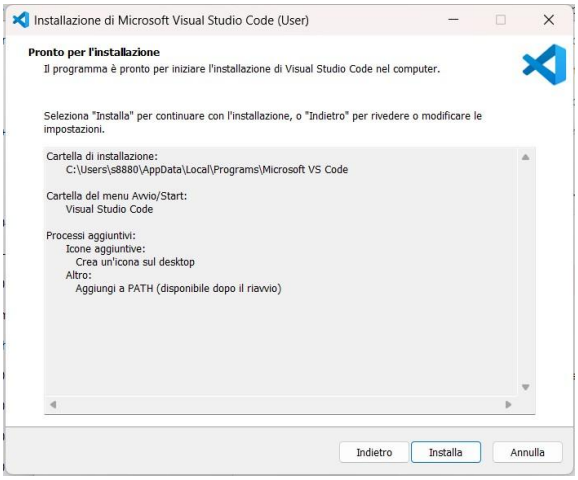
Durante l'installazione Windows potrebbe richiedere ulteriori conferme per procedere (controlla se vi è un icona lampeggiante nella barra delle applicazioni in basso e in caso clicca su di essa e dai l'OK nella finestra che appare).

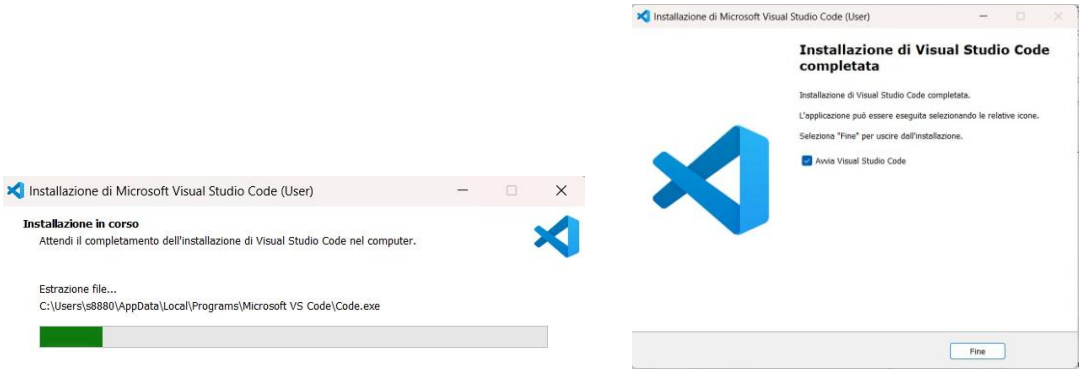




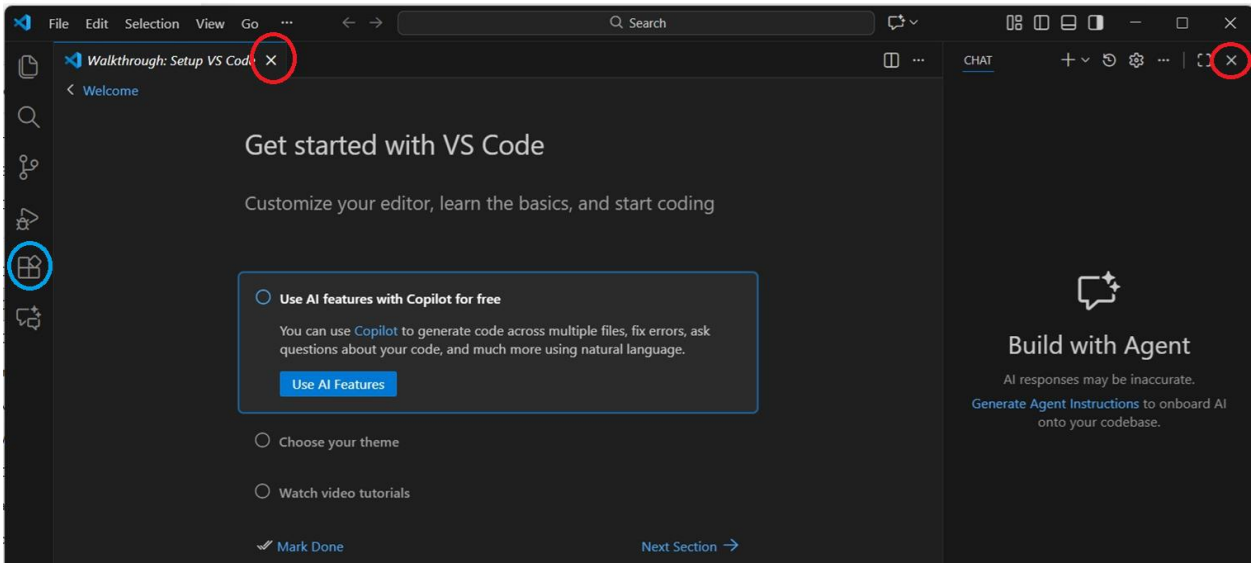




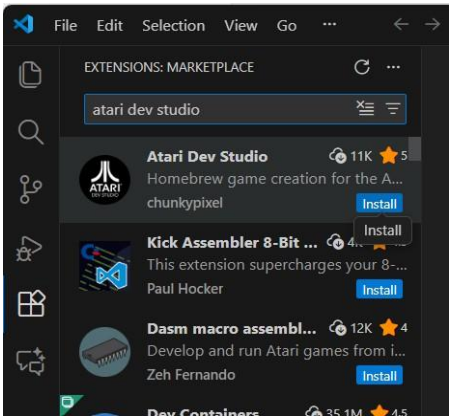




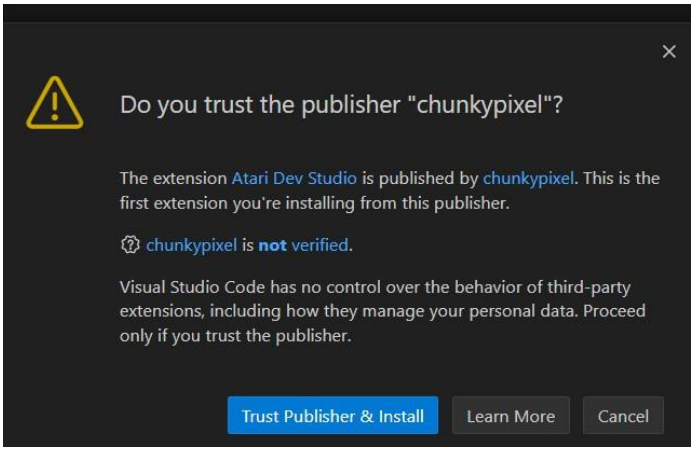
Una volta installato, Visual Studio Code verrà lanciato. Puoi chiudere subito la finestra di Welcome e quella relativa agli agenti AI, cliccando sulle relative “X” (circoletto rosso).



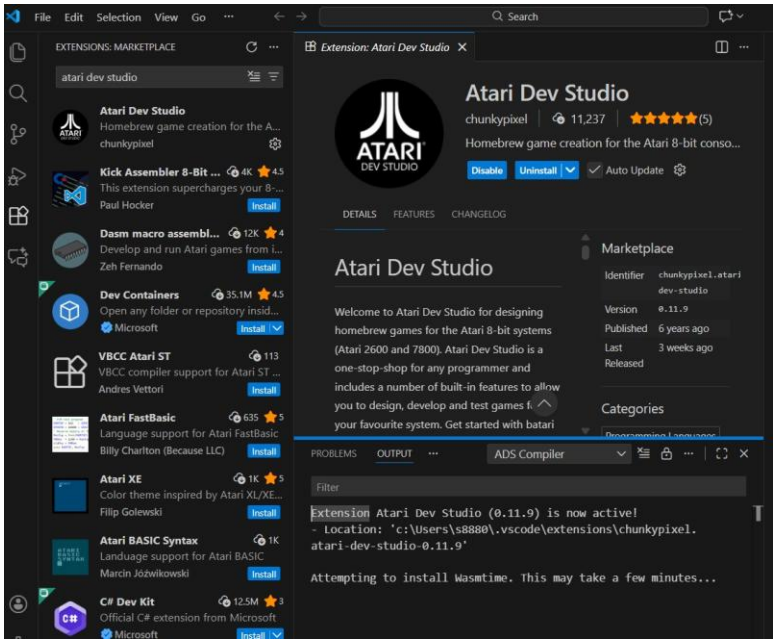
Clicca sull’icona estensioni a sinistra (circoletto blu) e nella finestra che apparirà inserisci “atari dev Studio” e successivamente premi su “install”:

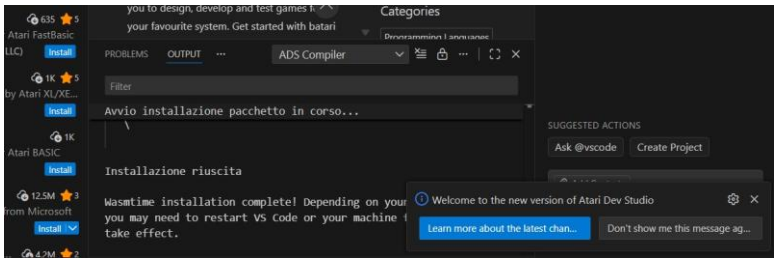


Accetta come credibile l’autore dell’estensione:



Controlla in basso che l’installazione proceda e termini correttamente:





Chiudi Visual Studio Code e cerca sul desktop l’icona per lanciarlo nuovamente:

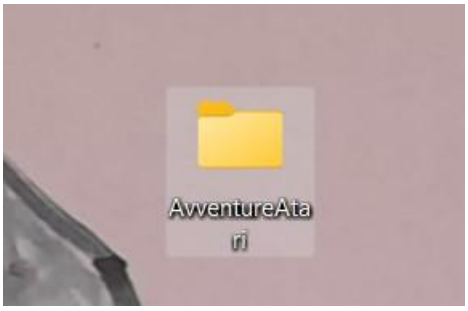
Appena si apre la finestra, puoi nuovamente chiudere il TAB di Welcome e quello relativo agli agenti AI.

Capitolo 2 – Cominciamo a programmare!

La nostra officina digitale è pronta. Il nostro primo obiettivo è senmplice, ma fondamentale: vogliamo ottenere un segnale stabile. Un semplice schermo nero, immobile e silenzioso, sarà la prova che abbiamo stabilito un contatto con il 1977.

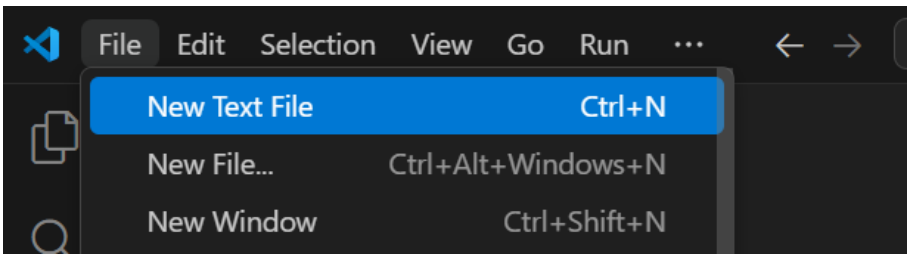
2.1 – Lo Scheletro di un programma batari basic

È ora di sporcarsi le mani! Apri Visual Studio Code. Per prima cosa, crea una cartella sul tuo computer (ad esempio sul desktop) dove conserverai tutti i tuoi progetti per l'Atari 2600. Chiamala, ad esempio, *AvventureAtari*.



Ora segui questi passi:

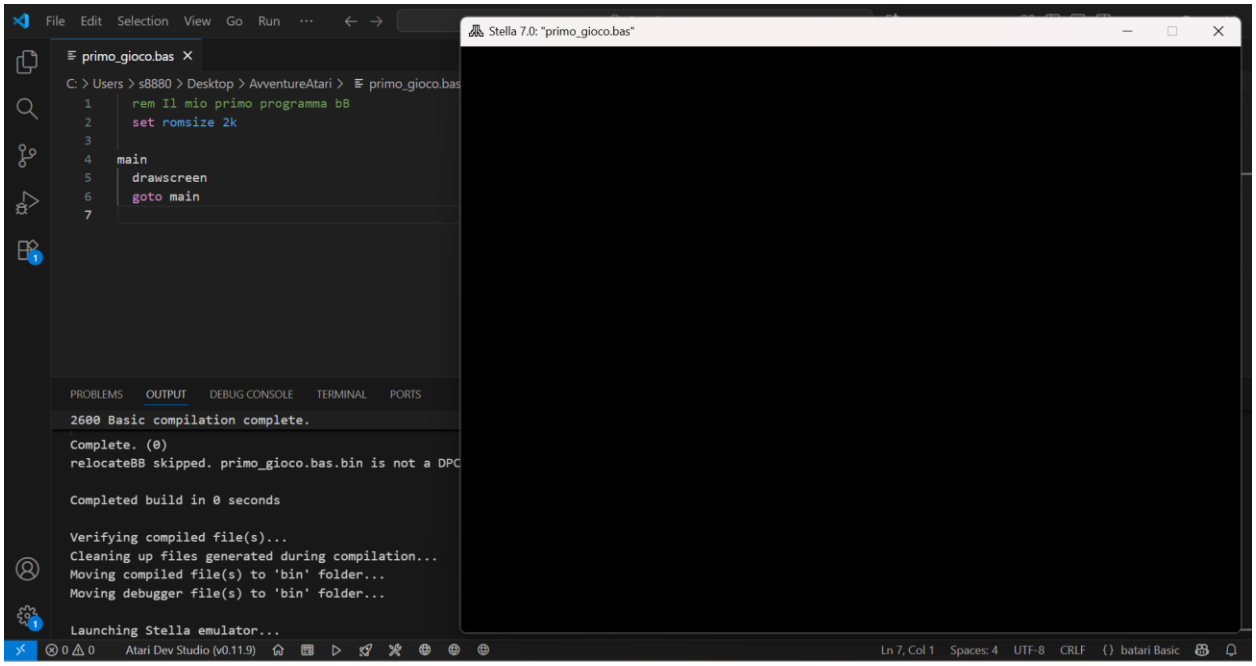
1. In Visual Studio Code, vai su **File > Nuovo File di Testo** (New Text File)



2. Vai subito su **File > Salva con nome…** (Save As). Naviga fino alla tua cartella AvventureAtari e salva il file con il nome *primo_gioco.bas*. L'estensione .bas è molto importante, perché dice ad Atari Dev Studio che questo è un file Batari Basic!
3. Scrivi queste poche righe di codice nel file. Fai molta attenzione a dove metti gli spazi e a cosa scrivi all'inizio della riga! La sintassi è come una formula: ogni simbolo deve essere al posto giusto, **spazi a inizio riga compresi**!

```
  rem Il mio primo programma bB
  set romsize 2k

main
  drawscreen
  goto main
```

Ora, premi il tasto **F5** sulla tua tastiera.



Se tutto è andato per il verso giusto, l'emulatore Stella dovrebbe aprirsi e mostrarti… uno schermo nero!

Aspetta, non ti preoccupare! Non hai rotto niente. Anzi, hai appena compiuto il primo, grande passo. Se vedi uno schermo nero, stabile, che non trema e non "rotola" su se stesso, significa che **tutto ha funzionato**! Hai appena creato e avviato il tuo primo programma per Atari 2600. Ben fatto!

Il Processo di Compilazione

Quando scrivi il tuo codice in Batari Basic (un file con estensione *.bas*), stai scrivendo in un linguaggio "ad alto livello", fatto di parole che possiamo capire come *if, goto, player0x*. Ma la CPU dell'Atari 2600, il MOS 6507, non capisce queste parole. Comprende solo il **linguaggio macchina**, una sequenza di numeri che corrispondono a operazioni molto semplici.

È qui che entra in gioco il processo di compilazione.

La compilazione è l'atto di tradurre il tuo **codice sorgente** (*.bas*) in un file eseguibile in linguaggio macchina. Quando premi F5 in Visual Studio Code, l'estensione Atari Dev Studio avvia un programma chiamato compilatore che fa esattamente questo.

Il compilatore legge il tuo file *bas* dall'inizio alla fine e poi converte ogni comando Batari Basic nel suo equivalente in linguaggio macchina 6507. Ad esempio, la riga player0x = 80 viene tradotta in una sequenza di istruzioni numeriche che dicono alla CPU: *"Prendi il numero 80 e mettilo nell'indirizzo di memoria che controlla la posizione X di player0"*.

Il risultato di questa traduzione è un nuovo file, di tipo *bin*. Questo file *.bin* (da "binario") è la tua cartuccia di gioco virtuale. Non contiene più parole, ma solo la sequenza pura di 0 e 1 (rappresentati come numeri) che la CPU dell'Atari 2600 può eseguire direttamente.

L’estensione Atari Dev Studio, per mantenere le cose ordinate, crea una sottocartella chiamata *bin* all’interno della cartella del tuo progetto. È lì che troverai il file *.bin* pronto per essere eseguito.

Quando l’emulatore Stella si avvia, non sta eseguendo il tuo file *.bas*. Quello è solo il tuo “progetto”. Stella carica ed esegue il file *.bin* che il compilatore ha creato. È quel file binario che contiene le vere istruzioni che danno vita al tuo gioco.

Capire questa distinzione è fondamentale: noi scriviamo in un linguaggio umano, il compilatore lo traduce, e la console esegue solo il risultato finale di quella traduzione.

E se qualcosa va storto?

Prima che Stella si apra, noterai del testo apparire nella parte bassa di Visual Studio Code, in una finestra chiamata “OUTPUT”. Questo è il diario di bordo della nostra officina: ci racconta cosa sta succedendo “sotto il cofano”.

Quando tutto va bene

Se hai scritto il codice correttamente, vedrai un messaggio simile a questo:

Starting build of primo_gioco.bas

batari Basic v1.9 (c)2025

2600 Basic compilation complete.

607 bytes of ROM space left

Completed build in 0 seconds

...

Launching Stella emulator...

Le righe importanti sono **“2600 Basic compilation complete”** e **“Completed build”**. Significano che il “traduttore” (il compilatore) ha capito le tue istruzioni e ha creato con successo il file di gioco. Subito dopo, l’estensione lancerà l’emulatore Stella.

Quando qualcosa va storto

Se hai commesso un piccolo errore di battitura (ad esempio, hai messo uno spazio prima di main), il traduttore non capirà e si fermerà. Vedrai un messaggio di errore:

Starting build of primo_gioco.bas

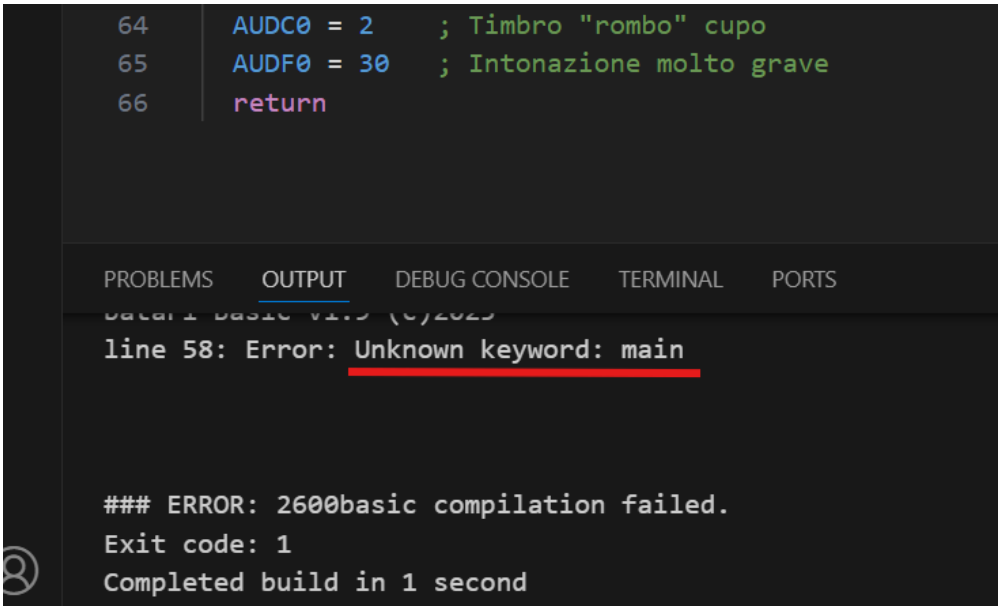
batari Basic v1.9 (c)2025

line 4: Error: Unknown keyword: main

ERROR: 2600basic compilation failed.

L’emulatore Stella non si avvierà. Il messaggio Error: *Unknown keyword: main* sembra strano: “main” non è una parola chiave sconosciuta!

Questo ci insegna una lezione fondamentale su Batari Basic: **i messaggi di errore spesso indicano *dove* si trova il problema, ma non sempre spiegano *chiaramente* quale sia**. In questo caso, l’errore non è la parola main, ma lo spazio che la precede, che viola la regola delle etichette in colonna 0.



2.2 – Anatomia dello Scheletro: Il Codice Spiegato

Quello che hai appena scritto è lo "scheletro" di ogni gioco per Atari 2600. È la struttura fondamentale che tiene tutto insieme. Analizziamola riga per riga.

rem Il mio primo programma bB → rem (che sta per *remark*, "osservazione") è un **commento**. È una nota per te, l'essere umano. Tutto ciò che scrivi dopo rem su una riga viene completamente ignorato dal computer. Usalo per prendere appunti e ricordare cosa fa il tuo codice!

set romsize 2k →Questa è una **direttiva** per il nostro "traduttore" (il compilatore). Gli stiamo dicendo: "Prepara una 'cartuccia' virtuale da 2 kilobyte". È la dimensione più piccola possibile, perfetta per i nostri primi esperimenti.

main → Questa è un'**etichetta o label**. Pensa a un segnalibro. Deve stare sempre all'inizio della riga (in colonna 0, senza spazi prima) e serve come punto di riferimento, un luogo a cui possiamo dire al programma di "saltare".

drawscreen → Questo è il **comando principale**. È il cuore pulsante del nostro programma. Ogni volta che il programma esegue questo comando, dice all'hardware dell'Atari: "Ok, per ora ho finito di preparare tutto. Disegna un fotogramma sullo schermo!". In questo caso, non avendo preparato nulla, disegna semplicemente uno schermo vuoto (nero).

goto main → goto significa "vai a". Questa istruzione dice al programma: "Salta immediatamente indietro fino all'etichetta chiamata main".

Cos'è un Programma? Il Flusso delle Istruzioni

Pensa a un programma come a una ricetta di cucina per il computer. È una lista di istruzioni semplici e precise, scritte in un linguaggio che la macchina può capire.

Proprio come tu segui una ricetta passo dopo passo, il computer esegue il tuo programma un'istruzione alla volta, dall'alto verso il basso. Questo percorso sequenziale è chiamato **flusso di esecuzione**.

Comandi come *goto* o *if...then* sono gli strumenti che ci permettono di creare cicli, prendere decisioni e deviare dal semplice percorso dall'alto verso il basso, dando vita a programmi complessi e interattivi. Li vedremo presto!

Due Tipi di Commenti, *rem* e ;

In Batari Basic, hai due modi per lasciare note nel tuo codice: *rem* e il punto e virgola (;). Sebbene entrambi servano a scrivere commenti, hanno un uso stilistico e pratico diverso che ti aiuterà a mantenere il codice ordinato.

rem (Remark): Per commenti a riga intera.
rem deve trovarsi all'inizio di un'istruzione (dopo l'indentazione). Tutto ciò che segue su quella riga è un commento. È ideale per creare titoli di sezione o per descrivere in dettaglio un blocco di codice complesso.

; (Punto e virgola): Per commenti a fine riga.
Il punto e virgola può essere inserito dopo un comando. Tutto ciò che segue il ; fino alla fine della riga viene ignorato. È perfetto per aggiungere brevi note che spiegano cosa fa una singola riga di codice, senza interrompere il flusso. **Fai attenzione che in alcune piattaforme di sviluppo per Atari 2600 che utilizzano batari basic il ; non è accettato come commento valido. Se vuoi scrivere codice 100% compatibile, usa solo *rem* da solo su righe di codice dedicate!**

Evitare errori: suggerimenti

1. **Copia gli esempi con precisione**: All'inizio, il 99% degli errori deriva da piccole imprecisioni. Batari Basic è molto severo sulla sintassi! Assicurati di copiare gli esempi esattamente come sono scritti, **prestando la massima attenzione a spazi, due punti (:) e parole chiave, a maiuscole e minuscole**. Ecco alcuni suggerimenti:

- l'errore "**Error: Unknown keyword: …**" è quasi certamente dovuto a ":" usati dopo una label (etichetta) oppure a dei ***rem* messi ad inizio riga** (serve almeno uno spazio prima!)

- le **label** non vogliono ":" alla fine, ma alcune parole speciali come *playfield* e *player0* si! Fai sempre attenzione al ":" alla fine della riga.

- un altro errore tipico sono gli **"end" che non sono collocati ad inizio riga**

- aggiungi sempre una riga vuota come ultima linea del tuo programma

2. **Isola il Problema**: Se aggiungi un nuovo blocco di codice e il programma smette di compilare, l'errore è quasi certamente lì. Una tecnica da detective è "commentare" le nuove righe (mettendo rem all'inizio di ognuna) e provare a ricompilare. Se ora funziona, sai che il problema è in una di quelle righe.

3. **Compila senza Avviare**: Premere F5 fa due cose: compila il gioco e, se ha successo, avvia Stella. A volte, potresti voler solo controllare se il codice compila. Puoi farlo premendo Ctrl+Shift+B (o andando su "Terminale > Esegui attività di compilazione"). Questo eseguirà solo la compilazione e ti mostrerà eventuali errori nella finestra di OUTPUT, senza lanciare l'emulatore.

Man mano che procederemo, questo manuale ti indicherà gli errori più tipici (i Pitfall!) a cui prestare attenzione. Tecniche di debug più avanzate, per scovare errori non di sintassi ma di logica (i cosiddetti "bug"), saranno introdotte nel Capitolo 12. Per ora, la tua migliore amica è la precisione!

2.3 – Il Ciclo Infinito: Il Motore del Tempo

Mettendo insieme *main* e *goto main*, abbiamo creato un **ciclo infinito**. Il programma parte da main, esegue drawscreen e poi *goto main* gli dice di tornare subito all'inizio. E poi di nuovo, e di nuovo, circa 60 volte al secondo! Questo ciclo continuo è ciò che mantiene lo schermo stabile e impedisce all'immagine di "crollare". È il motore che fa girare il nostro gioco.

2.4 – I Registri del TIA: Il Cruscotto della Console

È ora di mettere in pratica la teoria! Abbiamo detto che il chip **TIA** è l'artista della console. Diamogli il nostro primo ordine diretto. Diciamogli di cambiare il colore dello sfondo. Per farlo, comunicheremo con uno dei suoi "cassetti" speciali, chiamati **registri**.
Apri il tuo file *primo_gioco.bas* e modificalo così:

```
  rem Sfondo semplice
  set romsize 2k

main
  COLUBK = $86 ; Imposta lo sfondo a un bel blu
  drawscreen
  goto main
```

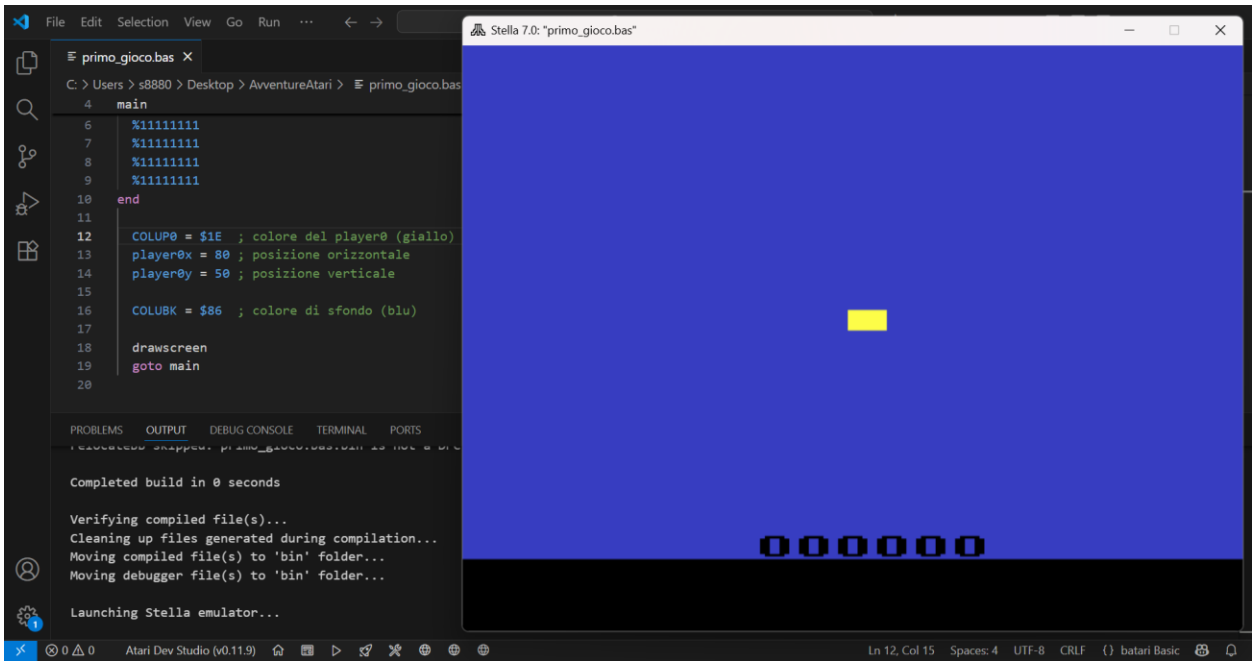
Cosa fa questo nuovo codice?
COLUBK = $86 → Questa è la nuova riga. *COLUBK* è il nome del registro del TIA che controlla il colore dello sfondo (*COlor LUMINOSITY Background*). Stiamo scrivendo in quel registro il valore $86. Il simbolo $ indica che è un numero **esadecimale**, un modo di contare molto usato dai programmatori. Per ora, ti basta sapere che $86 corrisponde a un bel blu sulla tavolozza di colori dell'Atari.
Ora, premi **F5**.
Il tuo schermo nero dovrebbe essere diventato… **blu**! Hai appena dato il tuo primo comando diretto all'hardware dell'Atari! Stai parlando la sua lingua.

2.5 – Missione: "Hello, Player!"

Basta con gli schermi vuoti. È il momento di creare il nostro primo attore, un piccolo eroe fatto di quadrati luminosi. Daremo vita al nostro primo oggetto grafico, imparando a definirne la forma, la posizione e il colore.
Modifica il tuo file *primo_gioco.bas*. Come sempre, fai molta attenzione all'indentazione!

```
  rem Primo programma bB - Hello Player!
  set romsize 2k

main
  player0:
  %11111111
  %11111111
  %11111111
  %11111111
end

  COLUP0 = $1E  ; colore del player0 (giallo)
  player0x = 80 ; posizione orizzontale
  player0y = 50 ; posizione verticale

  COLUBK = $86  ; colore di sfondo (blu)

  drawscreen
  goto main
```

Premi **F5**. Vedrai un piccolo quadrato giallo apparire al centro dello schermo blu. Ce l'hai fatta! Hai appena evocato il tuo primo **sprite** dal freddo silicio della console!



Screenshot dell'emulatore Stella che mostra un semplice quadrato giallo su sfondo blu, come risultato del codice.

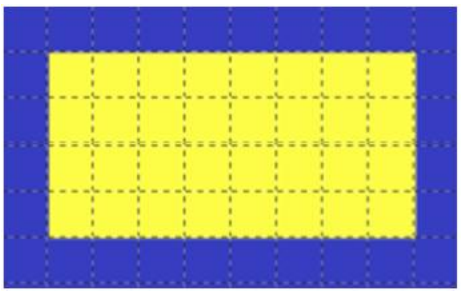
2.6 – Il Codice Spiegato

Questo codice è più complesso. Analizziamolo per capire i segreti che nasconde.

player0 → è un'etichetta speciale che definisce la grafica per il primo oggetto mobile, chiamato **Player 0**.

```
  player0:
  %11111111
  %11111111
  %11111111
  %11111111
end
```

Le righe che iniziano con % rappresentano i dati **binari** (On/Off) che disegnano lo sprite. Ogni riga è una fetta orizzontale di 8 pixel. 1 significa "pixel acceso" (visibile), 0 significa "pixel spento". In questo caso, stiamo creando un blocco solido.



end → segnala la fine della definizione grafica.

```
  COLUP0 = $1E  ; colore del player0 (giallo)
  player0x = 80 ; posizione orizzontale
  player0y = 50 ; posizione verticale
```

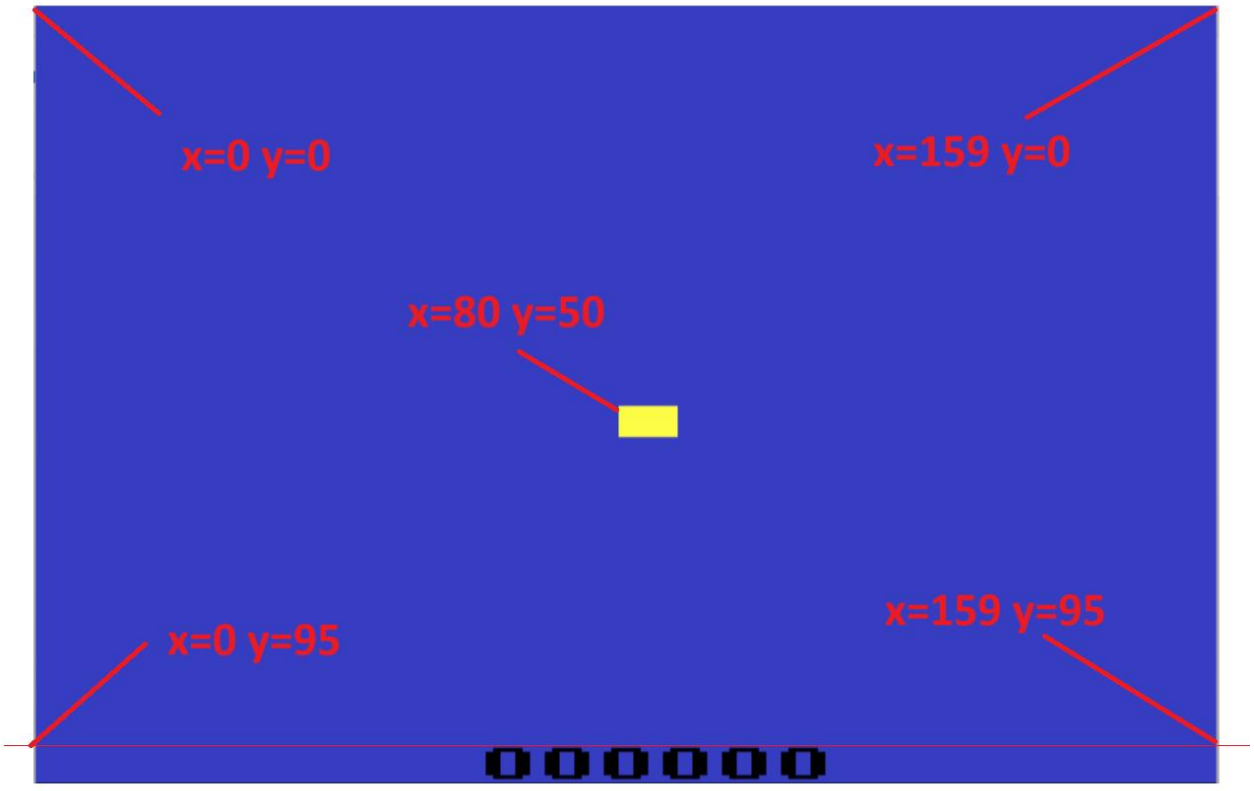
COLUP0 = $EA → È il registro del colore per il player0 (*COlor LUMINOSITY Player 0*). Il valore $1E corrisponde a un bel giallo brillante.
player0x = 80 → Imposta la coordinata **orizzontale (x)** dello sprite.
player0y = 50 → Imposta la coordinata **verticale (y)** dello sprite.

Ma cosa significano esattamente questi numeri? Per capirlo, dobbiamo conoscere la mappa del nostro universo digitale.

2.7 – Il Sistema di Coordinate dell'Atari

Pensa allo schermo dell'Atari 2600 come a una mappa. Ogni punto su questa mappa ha delle coordinate, proprio come in una battaglia navale.

- **L'Origine (0, 0):** Il punto di partenza è l'angolo **in alto a sinistra** dello schermo.
- **L'Asse X (Orizzontale):** I valori di *player0x* aumentano da sinistra verso destra. L'intervallo visibile va circa da **0** (bordo sinistro) a **159** (bordo destro).
- **L'Asse Y (Verticale):** I valori di *player0y* aumentano dall'alto verso il basso. L'intervallo visibile va circa da **0** (bordo superiore) a **95** (bordo inferiore).



Il Punto di Origine dello Sprite

Quando imposti *player0x* e *player0y*, a quale pixel dello sprite ti riferisci? La regola è: le coordinate (x, y) si riferiscono sempre all'**angolo in alto a sinistra** del tuo sprite.

Quindi, *player0x = 80* e *player0y = 50* posiziona l'angolo in alto a sinistra del nostro quadrato giallo al centro dello schermo.

Coordinate diverse per sprite e playfield

Attenzione a non confonderti! Il sistema di coordinate per gli sprite (0-159 in orizzontale, 0-95 in verticale) è diverso da quello usato per manipolare lo sfondo (il *Playfield*) con comandi che vedremo più avanti come *pfpixel* e *pfread*. Il Playfield usa un sistema a "blocchi" molto più piccolo (da 0 a 31 in orizzontale e da 0 a 10 in verticale). Parleremo di questo nel Capitolo 4. Per ora, ricorda che *player0x* e le coordinate del Playfield sono due cose diverse!

I colori dell'ATARI 2600

Un colore sull'Atari 2600 è definito da un singolo byte (un valore da 0 a 255). La tabella che segue mostra la tavolozza di 128 colori disponibile sullo standard televisivo NTSC (Nord America, Giappone). I valori sono in esadecimale (un modo più compatto di scrivere i numeri). Per trovare un colore, incrocia la la riga della **Tonalità** (la prima cifra, $X-) con la colonna della **Luminosità** (la seconda cifra, $-Y). Ad esempio $1E è un bel giallo brillante ($1E corrisponde a 30).

	0	2	4	6	8	A	C	E
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								
A								
B								
C								
D								
E								
F								

Capitolo 3 – Muovere l'Eroe

Il nostro piccolo eroe giallo è sul palco. È definito, colorato e sa dove stare. Ma c'è un problema: è immobile, come una statua. Un gioco non è veramente un gioco finché il giocatore non può interagire. È il momento di dare al nostro personaggio il dono più prezioso di tutti: **il movimento**. In questo capitolo, collegheremo il mondo fisico al nostro universo digitale. Prenderemo i segnali elettrici (virtuali) di un joystick e li trasformeremo in azioni sullo schermo.

3.1 – Ascoltare il Giocatore: Leggere il Joystick

Come fa il nostro programma a sapere se stai spingendo la levetta del joystick? Il Batari Basic rende questo compito incredibilmente semplice. Ci fornisce dei comandi speciali che, usati all'interno di una condizione if, si comportano come delle domande dirette alla console.
I comandi principali per il primo joystick (chiamato joy0) sono:

- *joy0up* (su)
- *joy0down* (giù)
- *joy0left* (sinistra)
- *joy0right* (destra)
- *joy0fire* (il pulsante di fuoco rosso)

Questi comandi diventano "veri" solo quando il giocatore sta effettivamente compiendo quell'azione. Possiamo usarli per creare delle logiche molto semplici, come: "**SE** il giocatore preme a destra, **ALLORA** fai qualcosa".

3.2 – Primi Passi

Mettiamo subito in pratica questa conoscenza. Modifichiamo il nostro codice *primo_gioco.bas* per far muovere il quadrato a destra e a sinistra. Aggiorna la sezione *main* in questo modo:

```
main
  player0:
  %11111111
  %11111111
  %11111111
  %11111111
end

  if joy0left then player0x = player0x - 1
  if joy0right then player0x = player0x + 1

  COLUP0 = $1E  ; colore del player0 (giallo)
  player0y = 50 ; posizione verticale (per ora fissa)

  COLUBK = $86  ; colore di sfondo (blu)

  drawscreen
  goto main
```

Cosa c'è di nuovo?

- *if joy0left then player0x = player0x - 1*: Questa è la nostra logica di movimento. Dice: "**SE** il joystick è spinto a sinistra, **ALLORA** prendi il valore attuale di player0x, sottrai 1 e salva il nuovo risultato in player0x". Questo sposta lo sprite di un pixel a sinistra.
- *if joy0right then player0x = player0x + 1*: Fa la stessa cosa, ma aggiungendo 1 per spostare lo sprite a destra.

Operatori e Parentesi

Nel tuo viaggio, avrai costantemente bisogno di fare calcoli per muovere personaggi, aggiornare timer o gestire punteggi. Batari Basic ti mette a disposizione gli operatori matematici fondamentali, ma con alcune regole specifiche che devi conoscere.

Gli Operatori di Base: + - *

+ (Addizione): Somma due numeri.

- (Sottrazione): Sottrae un numero da un altro.

* (Moltiplicazione): Moltiplica due numeri.

La Divisione Intera: /

Qui devi prestare molta attenzione. A differenza della matematica a cui sei abituato, la divisione in Batari Basic è solo intera. Questo significa che il risultato perde qualsiasi parte decimale.

10 / 2 darà come risultato 5 (corretto).

10 / 3 darà come risultato 3, non 3.333.... Il resto viene semplicemente scartato.

5 / 2 darà come risultato 2, non 2.5.

L'Ordine delle Operazioni e le Parentesi ( )

Batari Basic segue le regole matematiche standard per l'ordine delle operazioni: la moltiplicazione (*) e la divisione (/) vengono eseguite prima dell'addizione (+) e della sottrazione (-).

risultato = 5 + 2 * 3 darà 11 (perché 2 * 3 viene calcolato prima).

Per forzare un ordine diverso, devi usare le parentesi ( ). Tutto ciò che è all'interno delle parentesi viene calcolato per primo.

risultato = (5 + 2) * 3 darà 21 (perché 5 + 2 viene calcolato prima).

La Magia dei Numeri Negativi

Hai appena scritto *player0x = player0x - 1*. Semplice, vero? Ma come fa un programma **che conosce solo numeri da 0 a 255** a capire cosa significa “sottrarre”? La risposta è uno dei trucchi più geniali della programmazione a 8 bit, chiamato **complemento a due**. Pensa a un contachilometri che arriva solo fino a 255. Se sei a 0 e vai indietro di 1, cosa succede? Fa il giro al contrario e va a 255! Per la CPU dell’Atari, quindi, fare “0 – 1” è la stessa identica cosa che ottenere 255. Questo significa che il numero **255** si comporta esattamente come **-1**. Non devi memorizzare tutto, ma ricorda: i numeri “alti” (vicino a 255) possono comportarsi come piccoli numeri negativi. È una tecnica fondamentale che useremo spesso!

Quando hai dubbi sull'ordine in cui verranno eseguiti i calcoli, usa sempre le parentesi. Non costano nulla in termini di performance e rendono il tuo codice infinitamente più chiaro e meno soggetto a bug.

Premi **F5** per avviare l'emulatore. Ora hai il controllo! Ma come muovi il tuo eroe senza un vero joystick? L'emulatore Stella ti permette di usare la tastiera del tuo computer per simulare i joystick e i tasti della console. Di default, i controlli sono mappati come segue:

Giocatore 1 (*joy0*) → Tasti Freccia: Per muovere la levetta nelle quattro direzioni (su, giù, sinistra, destra). **Barra Spaziatrice:** Per premere il pulsante di fuoco (*joy0fire*).

Giocatore 2 (*joy1*) → Tasti F, R, D, G: Per muovere la levetta (F=su, R=destra, D=giù, G=sinistra). **Tasto A (o tasto 0 del tastierino numerico):** Per premere il pulsante di fuoco (*joy1fire*).

Per provare il codice, usa i tasti freccia sinistra e destra sulla tua tastiera. Vedrai il tuo quadrato muoversi!

Per controllare lo stato del joystick, non devi mai usare il segno di uguale (=). Il comando stesso è la condizione.

if joy0right then ... ← **Corretto!**

if joy0right = 1 then ... ← **Errato!**

In Batari Basic, le condizioni if...then devono stare su **una sola riga**.

if joy0right then player0x = player0x + 1 ← **Corretto!**

if joy0right then

player0x = player0x + 1 ← **Errato!**

Questo errore di sintassi è una delle trappole più comuni per chi inizia. Tienilo a mente!

E se vuoi controllare se una direzione *non* è premuta? Usa il punto esclamativo ! (che significa NON):

if !joy0fire then ... (SE il pulsante di fuoco NON è premuto…)

3.3 – Il Ponte di Comando: Joystick e Interruttori della Console

L'Atari 2600 era famosa per il suo iconico joystick nero con un singolo pulsante rosso. La console supportava due giocatori, ognuno con il proprio controller, identificati nel nostro codice come *joy0* (giocatore 1, collegato alla porta sinistra) e *joy1* (giocatore 2, collegato alla porta destra).

Gli Interruttori della Console: Tasti "Software"

Oltre ai joystick, la console aveva una fila di interruttori metallici sul pannello frontale. Una delle genialità dell'Atari 2600 è che la funzione di questi tasti non era "cablata" nell'hardware, ma era **definita dal software**. Questo significa che un programmatore poteva decidere a cosa servisse ogni interruttore, rendendoli estremamente versatili. Ecco i principali e il loro uso più comune:

- ***switchreset* (Game Reset):** Solitamente usato per riavviare il gioco dall'inizio, tornando alla schermata del titolo.
- ***switchselect* (Game Select):** Usato per ciclare tra le diverse modalità di gioco prima di iniziare (es. 1 giocatore vs 2 giocatori, facile vs difficile).
- ***switchbw* (Color / B&W):** Usato per passare dalla modalità a colori a quella in bianco e nero. Molti programmatori, in modo creativo, lo riutilizzarono come **tasto di pausa**!
- ***switchleftb* / *switchrightb* (Difficulty A/B):** Due interruttori per impostare la difficoltà (A=Advanced, B=Beginner) separatamente per il giocatore 1 e 2. Spesso cambiavano la velocità dei nemici, la dimensione delle racchette (in *Pong*), o altri parametri di gioco.

Nell'emulatore Stella, questi interruttori sono mappati su tasti funzione → **F1:** Game Reset ; **F2:** Game Select ; **F3 / F4:** Difficoltà Giocatore 1 / 2 (Sinistra / Destra) ; **F5:** Colori / Bianco e Nero

L'Esperienza Autentica: L'Atari 2600+

Mentre l'emulatore è uno strumento fantastico per lo sviluppo, nulla batte la sensazione di giocare con un vero joystick. Oggi, grazie a console moderne come l'**Atari 2600+**, è possibile rivivere quell'esperienza. Questa console è una riproduzione fedele dell'originale, ma con un'uscita HDMI per i televisori moderni. Viene fornita con un joystick CX40+ che ricrea perfettamente il feeling del controller classico. Grazie a speciali "cartucce flash" (come la Harmony Cartridge), potrai persino caricare e giocare i giochi che creerai con questo manuale sulla tua console Atari 2600+, chiudendo il cerchio del tuo viaggio nel tempo!

3.4 – Clamping: I Muri Invisibili del Mondo

Prova a muovere il quadrato tutto a sinistra o tutto a destra. Sparisce! È uscito dai confini dello schermo e si è perso nel vuoto digitale. Dobbiamo dargli dei limiti, come se ci fossero dei muri invisibili ai lati del mondo. Questa tecnica si chiama **clamping**.
Modifichiamo le nostre righe if per aggiungere un controllo sui bordi, usando l'operatore AND "&&" che significa "E".

```
if joy0left && player0x > 8 then player0x = player0x - 1
if joy0right && player0x < 152 then player0x = player0x + 1
```

Ora le nostre condizioni sono più complesse:

- "**SE** il joystick è a sinistra **E** la posizione player0x è maggiore di 8, **ALLORA** muoviti a sinistra."
- "**SE** il joystick è a destra **E** la posizione player0x è minore di 152, **ALLORA** muoviti a destra."

Questo impedisce al programma di aggiornare la posizione se lo sprite è già arrivato al bordo, bloccandolo efficacemente all'interno dell'area di gioco visibile. (I valori 8 e 152 sono scelti per tenere conto della larghezza dello sprite).

Gli Operatori Logici && (E) e || (OPPURE)

Quando vogliamo verificare che due o più condizioni sono vere contemporaneamente abbiamo bisogno degli operatori logici.

&& (AND logico - E): Restituisce "vero" solo se tutte le condizioni che collega sono vere. È perfetto per creare requisiti stringenti.

Esempio: *if joy0left && player0x > 8 then ...*

Significato: “Esegui il comando solo SE il joystick è a sinistra E la posizione x è maggiore di 8.”

|| **(OR logico - OPPURE)**: Restituisce “vero” se almeno una delle condizioni che collega è vera. È ideale per controllare se si verifica una tra più possibilità.

Esempio: *if joy0left || joy0right then ...*

Significato: “Esegui il comando SE il joystick è a sinistra OPPURE SE è a destra.”

Questi operatori sono i mattoni fondamentali per creare una logica di gioco complessa e reattiva.

Non Mischiare && e || nello Stesso if!

Questa è una delle limitazioni più importanti e contro-intuitive di Batari Basic. A differenza dei linguaggi moderni, **non puoi usare && e || insieme all’interno della stessa condizione *if***. Se lo fai, otterrai un comportamento imprevedibile o errato! Se necessario devi sempre scomporre la logica in più istruzioni *if*, una per ogni “gruppo” di condizioni che ti servono.

Non usare più di un || in un if!

A differenza di &&, non utilizzare mai più di un || nello stesso *if* altrimenti il programma non funzionerà.

3.5 – Un Tocco di Stile: Riflettere lo Sprite con REFP0

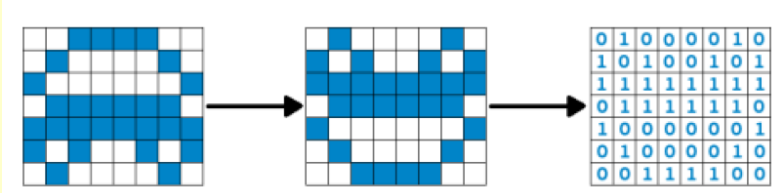
Il nostro quadrato si muove, ma è un po’ noioso. Diamo al nostro personaggio un po’ più di vita. Invece di un quadrato, disegniamo una semplice navicella. Sostituisci il blocco *player0*: con questo:

```
  player0:
  %01111100
  %00111111
  %01111100
end
```

Costruire dal basso verso l’alto
La prima riga di dati (%01111100) disegna in realtà la riga più bassa dello sprite! Il TIA (il chip grafico) infatti legge e disegna i dati dello sprite in ordine inverso rispetto al codice batari basic.

Qui sotto hai l’esempio di un automobile. Il tuo codice per questa automobile sarebbe:

```
 player0:
 %01000010
 %10100101
 %11111111
 %01111110
 %10000001
 %01000010
 %00111100

end
```



Premi **F5** e ricompila tutto. Ora, quando ti muovi, la “punta” della navicella è sempre rivolta a destra. Non sarebbe bello se si “girasse” per guardare a sinistra quando ci muoviamo in quella direzione?
Possiamo farlo con un altro registro del TIA: **REFP0** (*REFlect Player 0*). Questo registro agisce come un interruttore per uno specchio.

Molti registri del TIA, incluso *REFP0,* sono **volatili**. Questo significa che il loro valore viene automaticamente azzerato (resettato a 0) dopo ogni *drawscreen*. Se impostiamo *REFP0* solo quando premiamo il joystick, l’effetto durerà un solo frame e poi svanirà! Per mantenere un effetto persistente, come la riflessione dello sprite, dobbiamo usare una **variabile** per “ricordare” lo stato di riflessione desiderato. Poi, ad ogni ciclo del *main loop*, assegneremo il valore di quella variabile a *REFP0* subito prima di disegnare lo schermo. Vedremo subito un esempio di questa tecnica!

La lista dei registri volatili e non la trovi nell’appendice B.

Le Variabili a-z

Batari Basic ti mette a disposizione 26 variabili predefinite, nominate con una singola lettera dalla a alla z, che possono contenere un valore da 0 a 255. Pensa a loro come a 26 scatole vuote, etichettate da **a** a **z**, pronte per essere usate.

Una variabile è un contenitore per un'informazione che cambia durante l'esecuzione del gioco. La posizione del giocatore, il suo punteggio, il numero di vite rimaste, il tempo su un timer: tutti questi sono valori che devono essere costantemente aggiornati.

L'Assegnazione (=): Mettere un Valore nella Scatola

L'operazione con cui si inserisce o si aggiorna un valore in una variabile si chiama assegnazione. In Batari Basic (e in quasi tutti i linguaggi di programmazione), l'assegnazione è rappresentata dal simbolo di uguale (=), ma il suo significato è molto diverso da quello matematico.

In programmazione, il segno = non significa "è uguale a", ma piuttosto "riceve il valore di". È un'azione, un ordine che dice al computer: "Calcola tutto quello che c'è a destra e metti il risultato finale nella variabile che si trova a sinistra".

Vediamo questo processo in azione, passo dopo passo, immaginando di eseguire le seguenti operazioni:

a = 4 ; Metti il numero 4 nella scatola 'a'.

b = a ; Prendi il valore che c'è dentro 'a' (che è 4) e copialo nella scatola 'b'.

a = a + 1 ; Calcola a+1 e metti il risultato in a

Assegnazione Semplice:

a = 4 ; Metti il numero 4 nella scatola 'a'.

La variabile a ora contiene il valore 4.

Assegnazione da un'altra Variabile:

b = a ; Prendi il valore che c'è dentro 'a' (che è 4) e copialo nella scatola 'b'.

Ora sia a che b contengono il valore 4.

Assegnazione con Calcolo (L'operazione più importante!):

a = a + 1

Questa riga è il cuore della programmazione e va letta in due tempi, sempre da destra verso sinistra.

Calcola la parte destra: Il computer prende il valore attuale di a (che è 4), ci aggiunge 1 e ottiene il risultato 5.

Assegna alla parte sinistra: Il computer prende questo risultato finale (5) e lo "salva" nella variabile a, sovrascrivendo il vecchio valore.

Dopo questa operazione, a conterrà 5, mentre b conterrà ancora 4.

Assegnazione con Espressioni Complesse:

*c = (a + b) * 2*

Anche qui, il processo è lo stesso:

Prendi il valore di a (5)

Prendi il valore di b (4).

Calcola l'espressione tra parentesi: 5 + 4 = 9.

Moltiplica il risultato per 2: 9 * 2 = 18.

Metti il risultato finale (18) nella variabile c.

Pensa sempre al = come a una freccia che va da destra a sinistra (←). Prima il computer risolve completamente l'espressione a destra, trasformandola in un singolo numero, e solo alla fine deposita quel numero nella variabile a sinistra. Capire a fondo questo meccanismo è la chiave per controllare il flusso e lo stato del tuo gioco.

Modifichiamo il nostro codice per gestire correttamente la riflessione.

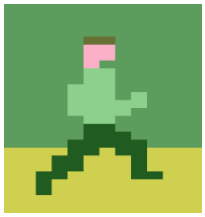
```
 rem Sprite che si specchia, clamp ai bordi
 set romsize 2k

 rem 'a' memorizza lo stato di riflessione (0=normale, 8=specchiato)
 a = 0

 rem posizione iniziale x del player0
 player0x = 80

main
  rem --- Definizione Grafica ---
  player0:
  %01111100
  %00111111
  %01111100
end

  rem --- Logica di Movimento e Riflessione ---
  if joy0left && player0x > 8 then player0x = player0x - 1 : a = 0
  if joy0right && player0x < 152 then player0x = player0x + 1 : a = 8

  rem --- Inizializzazione Registri prima del disegno ---
```

```
REFP0 = a      ; Applica lo stato di riflessione memorizzato in 'a'
COLUP0 = $1E   ; colore del player0 (giallo brillante)
player0y = 50  ; posizione verticale (per ora fissa)
COLUBK = $86   ; colore di sfondo (blu ciano)

rem --- Disegno ---
drawscreen


goto main
```

Analizziamo il nuovo codice:

- ***a = 0***: All'inizio del programma, inizializziamo la nostra variabile a a 0. Questo significa che lo sprite inizierà guardando nella sua direzione normale (in questo caso, a sinistra).
- ***if joy0left ... : a = 0***: Quando ci muoviamo a sinistra, oltre a cambiare la posizione player0x, impostiamo la nostra variabile di stato a a 0. Lo sprite non deve essere riflesso.
- ***if joy0right ... : a = 8***: Quando ci muoviamo a destra, impostiamo la nostra variabile a a 8. Questo "ricorda" al programma che lo sprite *dovrebbe* essere specchiato. Il valore 8 è il numero che attiva la riflessione orizzontale in REFP0.
- ***REFP0 = a***: Questa è la riga cruciale. Ad ogni singolo ciclo del main loop, poco prima di chiamare *drawscreen*, diciamo al registro REFP0 di assumere il valore che abbiamo salvato nella nostra variabile a. Se a è 0, non ci sarà riflessione. Se a è 8, lo sprite verrà specchiato per quel frame. In questo modo, l'effetto diventa stabile e persistente.

Premi **F5**. Ora la tua navicella si gira correttamente nella direzione in cui si muove, e rimane girata! Hai appena imparato una delle tecniche fondamentali per gestire gli stati grafici sull'Atari 2600.



Movimento Verticale e Diagonale

Ora che hai il pieno controllo, sperimenta con il movimento.

Movimento Verticale: Aggiungi la logica per muovere lo sprite su e giù. Ricorda di usare *player0y* e di aggiungere il clamping anche per i bordi superiore e inferiore (i limiti sono circa 10 e 90).

La Sfida della Diagonale: Riesci a far muovere lo sprite in diagonale? (*Suggerimento: Devi controllare se due direzioni sono premute contemporaneamente, ad esempio if joy0up && joy0right then ...*).

Cambia Velocità: Come potresti far muovere lo sprite più velocemente? Prova a cambiare *player0x = player0x + 1* in *player0x = player0x + 2*

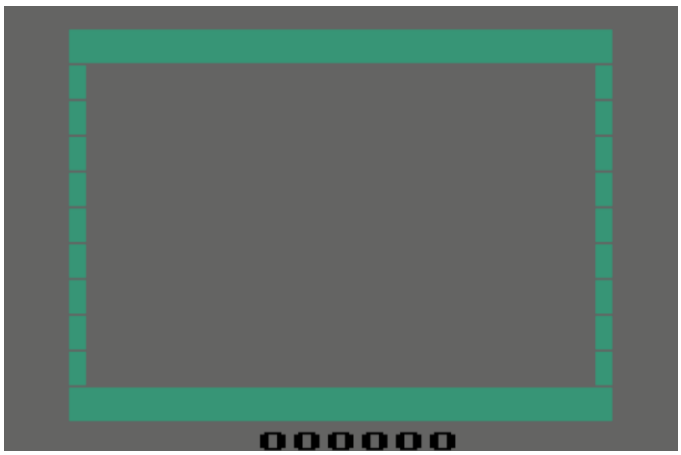
Capitolo 4 – Costruire lo Scenario: Il Playfield

Il nostro eroe si muove liberamente, ma fluttua in uno spazio vuoto e senza confini. Un'avventura ha bisogno di un mondo da esplorare. Ha bisogno di muri, pavimenti, piattaforme, labirinti e ostacoli. In questo capitolo impareremo a usare uno degli strumenti più potenti e creativi dell'Atari 2600: il **Playfield**. Il Playfield è il nostro set di mattoncini digitali, una griglia su cui possiamo disegnare lo sfondo statico del nostro gioco.

4.1 – La Geometria del Playfield

A differenza degli sprite (come player0), che sono oggetti dinamici e mobili, il Playfield è una "tela" fissa. Ha delle caratteristiche molto particolari:

- **È a bassa risoluzione:** È una griglia di blocchi larghi e squadrati. Ogni blocco del Playfield è largo 4 pixel e alto 8 pixel.
- **Non copre tutto lo schermo:** Per limiti hardware e di batari basic, il Playfield occupa solo la parte centrale dello schermo. Le fasce laterali, superiori e inferiori appartengono al "background" o allo score (punteggio) e non possono essere utilizzate. Qui sotto un esempio che mostra le dimensioni massime del playfield (in verde) rispetto alle dimensioni massime dello schermo (in grigio scuro).



Per disegnare questo sfondo verde rettangolare, usiamo un blocco di codice che inizia con l'etichetta *playfield*: e finisce con *end*.

Le Sottili Linee "Vuote" nel Playfield

Hai notato quelle sottili linee orizzontali che separano le righe di mattoncini del *playfield*:? Non è un errore, ma una caratteristica intrinseca dell'hardware dell'Atari 2600. Il chip TIA, per ragioni di timing e semplicità hardware, inserisce automaticamente una linea vuota di 1 pixel (del colore di sfondo, *COLUBK*) tra una riga di Playfield e la successiva. Questo crea l'effetto visivo di "mattoni" separati da una fuga, simile a un muro di mattoni reale. Esistono tecniche avanzate per eliminare queste linee di separazione ma invece di vederle come un limite, considerale un elemento stilistico caratteristico dell'estetica Atari 2600. Molti giochi classici le hanno sfruttate per dare ai loro scenari un aspetto più strutturato e definito.

4.2 – La Nostra Prima Stanza

Costruiamo una semplice cornice, una "stanza" che contenga il nostro eroe. La struttura del nostro codice cambierà leggermente: d'ora in poi, definiremo la grafica all'inizio del file, per avere una migliore organizzazione.

```
  rem Il Mondo di Mattoni Digitali
  set romsize 2k

  a = 0 ; Variabile per la riflessione dello sprite

  rem --- Definizioni Grafiche ---
  player0:
  %01111100
  %00111111
  %01111100
end

  playfield:
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  X..............................X
  X..............................X
  X..............................X
  X..............................X
  X..............................X
  X..............................X
  X..............................X
  X..............................X
  X..............................X
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end

  rem --- Posizione iniziale del giocatore ---
  player0x = 80
  player0y = 50

main
  rem --- Inizializzazione Registri per il frame ---
  COLUBK = $04  ; sfondo: Grigio scuro
  COLUPF = $B6  ; playfield: Verde giungla
  COLUP0 = $1E  ; player: Giallo brillante
  REFP0 = a     ; Applica lo stato di riflessione

  rem --- Logica di Gioco ---
  if joy0left && player0x > 18 then player0x = player0x - 1 : a = 8
```

```
if joy0right && player0x < 142 then player0x = player0x + 1 : a = 0


rem --- Disegno ---
drawscreen


goto main
```

Cosa significano "X" e "."? All'interno del blocco playfield:, questi simboli sono i nostri mattoni: "**X**" significa "blocco acceso" ovvero disegna un mattone solido usando il colore di COLUPF. "." significa "blocco spento" ovvero lascia quell'area trasparente, mostrando il colore di sfondo di COLUBK.
Premi **F5** per compilare e lanciare il programma. Dovresti vedere il tuo personaggio all'interno di una cornice verde. Hai appena costruito la tua prima struttura! Prova a muoverti: noterai che il clamping ora ha più senso, perché impedisce allo sprite di toccare i muri.

Fuori o dentro del *main*?

Hai notato che abbiamo spostato i blocchi *player0*: e *playfield*: all'inizio del file, fuori dal main loop? C'è una ragione precisa e legata all'efficienza.

Definizioni Statiche: se nessuno le modifica, il compilatore Batari Basic necessita di leggere le definizioni di grafica (player0:, playfield:) una sola volta. Non è necessario (né efficiente) che il programma torni a leggere queste definizioni 60 volte al secondo. Posizionarle fuori dal *main loop* ci aiuta a mantenere la logica del gioco (quella che cambia ad ogni frame) separata dalle risorse che sono "statiche".

Registri Persistenti: Lo stesso principio vale per i registri TIA che non sono volatili. Ad esempio, una volta che hai impostato il colore di sfondo (*COLUBK*) nel tuo *main loop*, l'hardware non dimenticherà questo valore. Potremmo spostare l'assegnazione di *COLUBK* (e dei registri non volatili che non cambiano mai, come il colore del Playfield *COLUPF* e del Player *COLUP0*) in una sezione di inizializzazione all'inizio del programma, fuori dal ciclo *main*. In questo modo, eseguiamo l'operazione costosa di assegnazione solo una volta, risparmiando cicli preziosi in ogni frame!

Se un'informazione (come la grafica o un colore) non deve cambiare durante il gioco, definiscila il più lontano possibile dal cuore pulsante del *main loop* per snellire il tuo codice e risparmiare preziosa CPU!

4.3 – Davanti o Dietro? La Priorità con CTRLPF

Hai notato? Il tuo sprite si muove **davanti** ai muri del Playfield. Questo è il comportamento di default. Ma cosa succede se vuoi che il tuo personaggio passi **dietro** a un pilastro o a un albero per creare un effetto di profondità?
Puoi farlo! C'è un altro registro magico nel TIA chiamato **CTRLPF** (*ConTRoL PlayField*), che agisce come un interruttore di priorità.

Aggiungi questa riga nella sezione di inizializzazione del tuo main loop, insieme agli altri registri:

```
  CTRLPF = %00000100
```

Come funziona? Il valore %00000100 "accende" un bit specifico nel registro CTRLPF che dice al TIA: "Disegna i blocchi del Playfield sopra gli sprite". Per tornare al comportamento normale, basta impostare CTRLPF = 0.
Premi **F5**. Ora, quando muovi lo sprite sopra un blocco X del Playfield, **scomparirà dietro di esso!** I blocchi del Playfield hanno la priorità. Questa tecnica è fondamentale per dare profondità e un aspetto più "reale" ai tuoi mondi di gioco.

4.4 – Scontrarsi con i Muri: La Funzione collision

Il nostro eroe può passare dietro i muri, ma non possiamo ancora usarli come ostacoli solidi. Come facciamo a sapere se il nostro sprite sta toccando un muro?
Usando un'altra domanda speciale che possiamo fare al Batari Basic: il comando **collision()**. Questo comando ci permette di sapere se due oggetti grafici si stanno toccando, restituendo "vero" se c'è un contatto. La sua sintassi è*: if collision(oggettoA, oggettoB) then ...*

Quando un personaggio si muove velocemente, potrebbe "passare attraverso" un muro per un singolo frame. Per creare collisioni a prova di bomba, si usa una tecnica tanto semplice quanto efficace: invece di respingere il giocatore, lo blocchiamo riportandolo all'ultima posizione "sicura".

4.5 – Muri Solidi con la Tecnica "Salva e Ripristina"

Mettiamo in pratica la gestione delle collisioni. Avremo bisogno di due variabili per salvare l'ultima posizione sicura del giocatore prima di ogni movimento. Ecco come fare:

1. **Salva:** All'inizio del *main_loop*, si salvano le coordinate del giocatore in x e y.

2. **Muovi:** Permetti al giocatore di muoversi come al solito.

3. **Controlla e Ripristina:** Dopo drawscreen, si controlla se c'è una collisione. Se sì, si riporta le coordinate del giocatore ai valori salvati in x e y.

Ecco il codice completo e funzionante.

```
  rem Il Mondo di Mattoni Digitali - con collisioni!
  set romsize 2k

  a = 0 ; Variabile per la riflessione dello sprite

  rem --- Definizioni Grafiche ---
  player0:
  %01111100
  %00111111
  %01111100
end

  playfield:
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  X..............................X
  X..............................X
```

```
  X.............................X
  X.............................X
  X.............................X
  X.............................X
  X.............................X
  X.............................X
  X.............................X
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end

  rem --- Posizione iniziale del giocatore ---
  player0x = 80
  player0y = 50

main
  rem 1. SALVA la posizione sicura
  x = player0x
  y = player0y

  rem --- Inizializzazione Registri ---
  COLUBK = $04  ; Grigio scuro
  COLUPF = $B6  ; Verde giungla
  COLUP0 = $1E  ; Giallo brillante
  REFP0 = a     ; Applica lo stato di riflessione

  rem 2. MUOVI il personaggio
  if joy0left && player0x > 18 then player0x = player0x - 1 : a = 8
  if joy0right && player0x < 142 then player0x = player0x + 1 : a = 0
  if joy0up && player0y > 10 then player0y = player0y - 1
  if joy0down && player0y < 85 then player0y = player0y + 1

  rem --- Disegno ---
  drawscreen

  rem 3. CONTROLLA e RIPRISTINA
  if collision(player0, playfield) then player0x = x : player0y = y

  goto main
```

Premi **F5**. Prova a sbattere contro i muri da qualsiasi direzione. Vedrai che il tuo personaggio si ferma di colpo, come se fossero solidi. Hai appena implementato la fisica di base del tuo mondo!

"Racing the Beam" e l'Ordine degli Eventi

Come fa l'Atari 2600 a sapere che due oggetti si toccano? La risposta è legata al modo stesso in cui la console disegna l'immagine sullo schermo.

Come abbiamo accennato, l'Atari 2600 non ha abbastanza memoria per costruire un'intera immagine e poi inviarla al televisore. Deve letteralmente "inventare" l'immagine riga per riga, in perfetta sincronia con il raggio di elettroni (il "beam") del televisore. Questo processo è chiamato "Racing the Beam" (correre contro il raggio).

Il chip TIA non si limita a disegnare; mentre il raggio passa su un pixel, il TIA controlla se più di un oggetto sta cercando di essere disegnato in quel preciso punto. Se questo accade, il TIA "alza una bandierina", ovvero imposta un bit di collisione nel suo hardware.

Perché *collision*() va dopo *drawscreen*?

Prima di *drawscreen*: I bit di collisione contengono ancora le informazioni relative al frame precedente. La CPU non ha ancora chiesto al TIA di disegnare nulla di nuovo, quindi il TIA non ha avuto modo di rilevare nuove collisioni basate sulle nuove posizioni degli oggetti.

Durante *drawscreen*: La CPU passa le nuove coordinate (*player0x*, *player0y*, *ecc.*) al TIA. Mentre il TIA disegna il nuovo frame, riga per riga, aggiorna i suoi bit di collisione in tempo reale.

Dopo *drawscreen*: I bit di collisione del TIA sono finalmente aggiornati con le informazioni del frame che hai appena visto disegnare. Questo è il momento giusto per interrogare l'hardware con la funzione *collision*() e ottenere una risposta accurata.

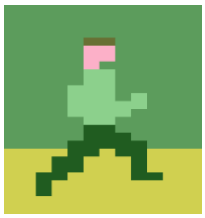
Pensa al *main loop* in questo ordine logico:

Calcola: Aggiorna le posizioni degli oggetti.

Disegna: Chiama *drawscreen* per far sì che il TIA disegni il nuovo frame e rilevi le collisione.

Controlla: Chiama *collision*() per leggere i risultati del disegno appena completato.

Se inverti questo ordine, il tuo gioco avrà sempre un frame di ritardo nel rilevare le collisioni, causando bug e comportamenti imprevedibili.



Libera il tuo Architetto Interiore

Ora che i muri sono solidi, è il momento di renderli più interessanti.

Costruisci un Labirinto: Modifica il blocco playfield: per creare un semplice labirinto con dei corridoi.

Capitolo 5 – La Voce della Console: Suoni ed Effetti Speciali

Il nostro mondo ora ha un aspetto: c'è un eroe, ci sono dei muri, c'è un'avventura che aspetta di essere vissuta. Ma è un'avventura silenziosa. Manca il *beep* di un laser, il *boop* di un oggetto raccolto, il *crunch* di una collisione. Un gioco senza suono è un gioco a metà.
In questo capitolo, impareremo a dare una voce all'Atari 2600, manipolando i suoi registri audio per creare effetti sonori. Nonostante la sua apparente semplicità, il chip TIA nasconde un generatore di suoni sorprendentemente versatile. È ora di fare un po' di rumore!

5.1 – L'Anatomia del Suono Atari

Il TIA, il nostro chip "artista", non si occupa solo della grafica. È anche un musicista. Ha **due canali audio indipendenti** (Canale 0 e Canale 1), il che significa che può produrre due suoni diversi contemporaneamente.
Ogni canale è controllato da un set di tre registri, tre "manopole" che dobbiamo regolare per produrre un suono:

- ***AUDV (Audio Volume)*: La Manopola del Volume.** Controlla quanto forte è il suono, con un valore da **0** (silenzio totale) a **15** (volume massimo). Per il Canale 0 si usa AUDV0, per il Canale 1 AUDV1.

- ***AUDC (Audio Control)*: La Manopola del Timbro.** Controlla la "voce" o la "texture" del suono, con un valore da **0** a **15**. Ogni numero seleziona una forma d'onda diversa, producendo suoni che vanno da toni puri e cristallini (come un flauto) a suoni distorti e "rumorosi" (come il motore di un'auto o un'esplosione).

- ***AUDF (Audio Frequency)*: La Manopola dell'Intonazione.** Controlla la frequenza, ovvero quanto una nota è acuta o grave, con un valore da **0** a **31**. **Attenzione:** qui le cose funzionano al contrario di come ci si aspetterebbe! Valori **bassi** producono suoni più **acuti**, mentre valori **alti** producono suoni più **gravi**.

Per produrre un suono sul Canale 0, dobbiamo impostare tutti e tre i registri: *AUDV0*, *AUDC0*, e *AUDF0*.
Nell'appendice D troverai tutte le informazioni necessarie per scegliere i valori che fanno al caso tuo!

I Suoni non si Fermano da Soli!

I registri audio sono **persistenti**. Una volta che hai impostato un suono, il TIA continuerà a produrlo all'infinito, anche nei frame successivi! È come premere un tasto di un organo che non torna più su. Per fermare un suono, l'unico modo è riportare il suo volume a zero: *AUDV0 = 0*. Ricordalo sempre!

5.2 – Il "Sound Timer": Creare Effetti Sonori a Tempo

Come facciamo a creare un suono breve, come quello di uno sparo, che duri solo una frazione di secondo e poi si fermi? Non possiamo semplicemente accenderlo e spegnerlo subito dopo, perché il nostro *main loop* è troppo veloce!
La soluzione è un pattern di codice fondamentale: il **"Sound Timer"**. È un semplice timer software, un contatore alla rovescia che usa una delle nostre variabili. L'idea è questa:

- Quando vogliamo che il suono inizi, attiviamo l'audio e carichiamo una variabile (il nostro timer) con un numero (es. 10). Questo numero rappresenta per quanti frame il suono durerà.
- Ad ogni ciclo del main loop, decrementiamo il timer di 1.
- Quando il timer raggiunge lo zero, spegniamo il suono impostando il volume a 0.

Prima di vedere un esempio concreto, introduciamo un concetto importantissimo: le **subroutine**.

5.3 – L'Arte dell'Ordine: gosub e return

Man mano che i nostri programmi crescono, il main loop può diventare disordinato. Per mantenere il codice pulito e organizzato, useremo le **subroutine**. Una subroutine è un blocco di codice separato che esegue un compito specifico (come "sparare" o "riprodurre un suono"). Per usare una subroutine abbiamo bisogno di due istruzioni fondamentali, **gosub** e **return**:

- **gosub <etichetta>**: Dice al programma: "Vai a eseguire il codice che si trova all'etichetta <etichetta>, ma ricorda da dove sei partito".
- **return**: Si mette alla fine della subroutine e dice: "Ho finito, torna al punto da cui eri partito".

Questo ci permette di organizzare il codice in blocchi logici e riutilizzabili.

5.4 –Collisione con Suono

Prendiamo il codice del capitolo precedente e aggiungiamo un effetto sonoro ogni volta che il giocatore si scontra con un muro. Avremo bisogno di una nuova variabile (useremo *s*) per il nostro sound timer. Già che ci siamo cambiamo un po' il *playfield*.

```
  rem Collisione con Suono
  set romsize 2k

  a = 0 ; Variabile per la riflessione
  s = 0 ; Variabile per il sound timer (s=sound)

  rem --- Definizioni Grafiche ---
  player0:
  %01111100
  %00111111
  %01111100
end

  playfield:
  XXXXXXXXXXXXXXXXXXXX
  X.....X............X
  X.....X............X
  X....XX.........X..X
  X..............X..X
  X............XXXX..X
  X..................X
  X...X..............X
  XXXXX.........XXXXXX
  X.............X.....
  XXXXXXXXXXXXXXX.....
end

  rem --- Posizione iniziale player0 ---
  player0x = 80
  player0y = 50

main
  rem 1. SALVA la posizione sicura
  x = player0x
  y = player0y
```

```
  rem --- Gestione del Sound Timer ---
  if s > 0 then s = s - 1
  if s = 0 then AUDV0 = 0 ; Spegni il suono quando il timer scade

  rem --- Inizializzazione Registri ---
  COLUBK = $04  ; Grigio scuro
  COLUPF = $B6  ; Verde giungla
  COLUP0 = $1E  ; Giallo brillante
  REFP0 = a     ; Applica lo stato di riflessione

  rem 2. MUOVI il personaggio
  if joy0left && player0x > 18 then player0x = player0x - 1 : a = 8
  if joy0right && player0x < 142 then player0x = player0x + 1 : a = 0
  if joy0up && player0y > 10 then player0y = player0y - 1
  if joy0down && player0y < 85 then player0y = player0y + 1

  rem --- Disegno ---
  drawscreen

  rem 3. CONTROLLA, RIPRISTINA e SUONA
  if collision(player0, playfield) then player0x = x : player0y = y : gosub play_hit_sound

  goto main

  rem --- SUBROUTINE PER IL SUONO ---
play_hit_sound
  s = 5        ; Durata del suono: 5 frame
  AUDV0 = 10   ; Volume
  AUDC0 = 2    ; Timbro "rombo" cupo
  AUDF0 = 30   ; Intonazione molto grave
  return
```

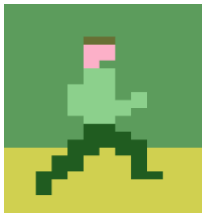
Premi **F5**. Hai appena sincronizzato grafica, input e audio! Come vedi, la **subroutine** *play_hit_sound* viene chiamata quando avviene la collisione attraverso *gosub*. Vengono eseguite le istruzioni "dentro" a *play_hit_sound* che si conclude con *return* che riporta il flusso di esecuzione subito dopo il *gosub play_hit_sound* (ovvero l'istruzione immediatamente successiva sarà *goto main*).

Si può usare un gosub dentro ad una subroutine chiamata da un gosub?

Si. **Ma non andare mai oltre tre "livelli" di gosub** o il programma potrebbe non funzionare correttamente. Ad esempio questo codice (due livelli di *gosub*) è corretto:

```
main
  ; ... logica del gioco ...
  gosub sub1 ;  viene eseguita tutta la sub1
  drawscreen
  goto main


sub1
   ; ... logica della subroutine 1 ...
   gosub sub2 ; viene eseguita tutta la sub2
   ; ... continuazione logica della subroutine 1 ...
   return ; ritorna al main


sub2
```

```
    ; ... logica della subroutine 2 ...
    return ; ritorna alla sub 1
```



Diventa un Sound Designer!

È il momento di sperimentare con i suoni.

Sperimenta con il Timbro: Nella subroutine, prova a cambiare il valore di *AUDC0*. Usa AUDC0 = 12 per un suono più puro, da "raccolta oggetto". Usa *AUDC0 = 14* per un ronzio, ottimo per un motore.

Cambia l'Intonazione: Gioca con *AUDF0*. Prova valori più alti (es. 20) per un suono più grave, o più bassi (es. 5) per un suono ancora più acuto.

Effetto Sonoro di Collisione: Riesci a produrre un suono diverso in base alla direzione del *player0*?

Capitolo 6 – Animazione a Frame Multipli

Il nostro eroe si muove, esplora mondi e interagisce con gli oggetti. Ma c'è ancora qualcosa che non va: si muove in modo rigido, come un pezzo degli scacchi. Manca l'illusione della vita, quel dettaglio che trasforma una semplice forma in un personaggio che corre, salta o attacca.
In questo capitolo, impareremo come animare gli oggetti grafici.

6.1 – Oltre lo Sprite Statico

Come funziona un cartone animato? Non è una singola immagine che si muove, ma una rapida successione di disegni leggermente diversi tra loro. Il nostro cervello, ingannato da questa velocità, percepisce il movimento come fluido.
In Batari Basic, possiamo fare esattamente la stessa cosa. Invece di avere un solo blocco *player0:*, possiamo definirne diversi, ognuno rappresentante un "fotogramma" (o **frame**) della nostra animazione.

L'idea di base:

- Disegniamo diversi sprite per il nostro personaggio (es. gamba destra avanti, gambe unite, gamba sinistra avanti).
- Nel nostro main loop, mostriamo questi sprite in rapida successione.
- Il risultato: il nostro eroe sembrerà correre!

6.2 – La Tecnica del "Cartone Animato": Alternare le Immagini con gosub

Il modo più semplice per gestire più frame è creare una subroutine grafica per ogni disegno. Immaginiamo di voler creare un'animazione di corsa a due frame. Possiamo definire *anim_frame_1* e *anim_frame_2*, ognuna contenente un blocco *player0:* diverso, e chiamarle con *gosub*.
Ma come decidiamo quale subroutine chiamare? Se le alternassimo a ogni ciclo del main loop, l'animazione sarebbe troppo veloce e tremolante (60 cambi al secondo!). Abbiamo bisogno di un metronomo.

6.3 – Il Metronomo del Codice: Usare i Timer per il Ritmo

Per controllare la velocità dell'animazione, usiamo un semplice contatore, una variabile che incrementiamo a ogni ciclo del main loop. La struttura di base del codice è la seguente:

```
  f = 0 ; Variabile per il timer di animazione (f=frame)

main loop
  rem ... logica del gioco ...


  f = f + 1
  gosub animate_player
  drawscreen
  goto main_loop

animate_player

  rem Se il timer è sotto 10, mostra il primo frame
  if f < 10 then gosub anim_frame_1

  rem Se il timer è tra 10 e 19, mostra il secondo frame
  if f >= 10 then gosub anim frame 2

  rem Se il timer arriva a 20, azzeralo per ricominciare il ciclo
  if f = 20 then animation_timer = 0
  return
```

```
anim_frame_1
  rem ... disegno animazione 1 ...
  return

anim_frame_2
  rem ... disegno animazione 2 ...
  return
```

Questo codice mostra il frame_1 per 10 cicli di gioco, poi il frame_2 per altri 10 cicli, e poi ricomincia. L'animazione ora ha un ritmo controllato!

6.4 – Creare un'Animazione di Corsa

Mettiamo tutto insieme. Modifichiamo il nostro programma di movimento per far correre il nostro personaggio, aggiungendo anche un frame "statico" per quando è fermo. Vogliamo:

- Creare due frame di animazione per la corsa e uno per quando il personaggio è fermo.
- Usare un timer per alternare i frame della corsa a un ritmo credibile.
- Mostrare il frame statico e azzerare il timer quando il personaggio si ferma.

```
  rem Animazione di Corsa
  set romsize 2k

  a = 0 ; Variabile per la riflessione
  f = 0 ; Variabile per il timer di animazione (f=frame)

main_loop
  gosub handle_input
  gosub animate_player

  player0x = x
  player0y = y

  COLUP0 = $EA
  COLUBK = $84
  REFP0 = a

  drawscreen
  goto main_loop

  rem --- SUBROUTINES DI GIOCO ---

handle_input
  rem Posizioni iniziali se non definite
  if x = 0 then x = 80 : y = 50

  if joy0left && x > 8 then x = x - 1 : a = 8
```

```
  if joy0right && x < 152 then x = x + 1 : a = 0
  return

animate_player
  rem Se il giocatore si muove, incrementa il timer
  if joy0left || joy0right then f = f + 1

  rem Se il giocatore è fermo, mostra il frame statico e azzera il timer
  if !joy0left && !joy0right then gosub anim_frame_static : f = 0 : return

  rem Logica di alternanza frame
  if f < 10 then gosub anim_frame_1
  if f >= 10 then gosub anim_frame_2
  if f >= 20 then f = 0 ; Azzera il timer per ricominciare il ciclo
  return

  rem --- DEFINIZIONI GRAFICHE ---

anim_frame_static ; Frame per personaggio fermo
  player0:
  %0010100
  %0010100
  %0010100
  %1001001
  %0111110
  %0001000
  %0011100
  %0011100
end
  return

anim_frame_1 ; Corsa - Frame 1
  player0:
  %0010100
  %0010100
  %0010100
  %1001000
  %0111111
  %0001001
  %0011100
  %0011100
end
  return
```

```
anim_frame_2 ; Corsa - Frame 2
  player0:
  %0010000
  %0010000
  %0010100
  %1001001
  %0111110
  %0001000
  %0011100
  %0011100
end
  return
```

Premi **F5**. Il tuo personaggio ora è fermo, in una posa statica. Ma quando spingi il joystick a dest ra o a sinistra, inizierà a "correre", alternando i due frame di animazione!

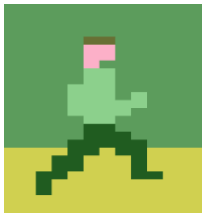
Simulare IF-ELSE con goto e le Etichette

Hai imparato a usare *if ... then gosub...* per eseguire una subroutine se una condizione è vera. Ma cosa succede se vuoi fare una cosa se la condizione è vera, e un'altra cosa completamente diversa se è falsa? I linguaggi moderni usano una struttura chiamata **if-else** per questo. Sebbene in Batari Basic esiste la possibilità di usare *else*, è meglio non usarlo perché il suo funzionamento non è sempre corretto. Tuttavia possiamo ottenere lo stesso risultato combinando *if*, *goto* e le **etichette (label)**.

Immagina di voler eseguire pezzi di codice diversi in base al valore della variabile a, senza scomodare i *gosub*. Ecco come fare:

```
  if a = 0 then goto case_a_0
  if a = 1 then goto case_a_1
  ; .. altri casi ..
  goto end_case_a  ; se nessuna condizione vera
                   ; salto oltre il codice per i vari casi a=0, a=1, ...
case_a_0
  ;...  codice per il caso a = 0
  goto end_case_a

case_a_1
  ;...  codice per il caso a = 1
  goto end_case_a

;... altri casi ...

end_case_a
  ; qui continua il codice del programma
```

La tecnica consiste nel “saltare” ad un blocco di codice specifico per ogni condizione che ci interessa. Basta utilizzare delle **label** con dei nomi diversi. Da tutti i pezzi di codice poi si salta al punto dove prosegue il programma.



Sperimenta con le Tue Abilità di Animatore

Aggiungi più Frame: Riesci a creare un’animazione di corsa più fluida usando 3 o 4 frame invece di 2? Dovrai modificare la logica nella subroutine animate_player per gestire più stati del timer.

Animazione Verticale: Il nostro eroe corre solo a destra e a sinistra. Prova a creare un set di sprite completamente diverso per quando si muove in alto e in basso. (*Suggerimento: Avrai bisogno di un if joy0up || joy0down then ... e di nuove subroutine grafiche*).

Animazione di “Idle”: Molti giochi hanno un’animazione per quando il personaggio è fermo (es. respira, guarda intorno). Modifica la subroutine *anim_frame_static* per alternare due frame molto simili tra loro, dando l’impressione che il personaggio sia vivo anche da fermo.

Capitolo 7 – Progetto Guidato: “Fuga dal Castello Digitale”

In questo capitolo, creeremo un mini-gioco chiamato “Fuga dal Castello Digitale”. L’obiettivo è semplice, ma per realizzarlo dovremo usare quasi tutto ciò che abbiamo imparato: la gestione dell’input, una semplice IA, animazioni, suoni, collisioni, la gestione dello stato di gioco e… un’altro oggetto grafico di cui parleremo nel dettaglio più avanti ma che ora ci serve, la “ball”!

7.1 – Fase 1: La Mappa del Tesoro – Pianificazione e Design

Prima di scrivere una sola riga di codice, un buon game designer pianifica gli elementi essenziali del gioco.

- **Genere:** Mini-avventura a schermata singola.
- **Obiettivo:** Raccogliere una chiave per aprire una porta e fuggire.
- **Personaggi:**
  - *player0*: **L’Eroe**, controllato dal giocatore, con animazioni di movimento.
  - *player1*: **Il Guardiano**, un nemico animato con una semplice IA di pattugliamento.
- **Logica di Gioco:**
  - L’Eroe deve toccare la **Chiave** (rappresentata dall’oggetto **ball**) per raccoglierla, con un suono di conferma.
  - Una volta raccolta la Chiave, l’Eroe deve raggiungere la **Porta** (un’area del castello) per vincere.
  - Se il Guardiano tocca l’Eroe, il gioco finisce (Game Over).
  - L’Eroe e il Guardiano producono suoni di passi. Le collisioni e gli eventi di vittoria/sconfitta hanno effetti sonori dedicati.

7.2 – Fase 2: Le Fondamenta – Mappa delle Variabili e Grafica

La prima cosa da fare è pianificare come useremo le nostre preziose 26 variabili a singola lettera. Assegnare a ogni lettera un ruolo chiaro fin dall’inizio è fondamentale per non perdersi. Poi definiamo la grafica del playfield.

```
  rem Fuga dal Castello Digitale
  set romsize 4k ; Imposta la dimensione della cartuccia virtuale a 4 kilobyte, necessaria per co
ntenere tutto il codice.

  rem --- Mappa delle Variabili ---
  a = 0 ; Memorizza lo stato di riflessione dell'Eroe (0 = guarda a destra, 8 = guarda a sinistra
, specchiato).
  b = 0 ; Timer per l'animazione dell'Eroe. Un contatore che cicla per decidere quale frame di an
imazione mostrare.
  c = 0 ; Timer per l'animazione del Guardiano. Simile a 'b', ma per il nemico.
  d = 0 ; Direzione del Guardiano (0 = si muove a destra, 1 = si muove a sinistra). Usato dall'IA
 di pattugliamento.
  e = 0 ; Stato del gioco (gamestate): 0=Inizializza, 1=In Gioco, 2=Vittoria, 3=Sconfitta. Il "ce
rvello" del gioco.
  f = 0 ; Flag per la chiave: 0 = non posseduta, 1 = posseduta.
  i = 0 ; Flag di movimento: 1 se il giocatore sta muovendo il joystick, 0 se è fermo.
  s = 0 ; Sound Timer per il Canale Audio 0 (usato per l'Eroe).
  t = 0 ; Sound Timer per il Canale Audio 1 (usato per il Guardiano e le collisioni).
  x = 0 ; Posizione orizzontale (coordinata X) corrente dell'Eroe.
  y = 0 ; Posizione verticale (coordinata Y) corrente dell'Eroe.
  u = 0 ; Variabile temporanea per salvare l'ultima posizione X "sicura" dell'Eroe.
  v = 0 ; Variabile temporanea per salvare l'ultima posizione Y "sicura" dell'Eroe.
  z = 0 ; Posizione orizzontale (coordinata X) corrente del Guardiano.
  w = 0 ; Posizione verticale (coordinata Y) corrente del Guardiano.

  rem -- il playfield definisce la grafica statica del castello e non cambia durante il gioco --
  playfield:
```

```
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  ......X........................X
  ......X........................X
  X....XX.....................X..X
  X...........................X..X
  X.........................XXXX..X
  X..............................X
  X...X..........................X
  XXXXX......................XXXXXX
  X..........................X.....
  XXXXXXXXXXXXXXXXXXXXXXXXXXXX.....
end
```

7.3 – Fase 3: La Macchina a Stati – Il Cervello del Gioco

Ora scriviamo il main_loop, il "centralino" che gestisce il flusso del gioco, e tutte le subroutine di stato. Il main_loop controlla la variabile e (gamestate) e chiama la subroutine appropriata per ogni stato del gioco.

```
main loop ; Il cuore del gioco, un ciclo infinito che si ripete circa 60 volte al secondo.
  if e = 0 then gosub state_init_game  ; Se il gioco è nello stato 0, esegui la routine di inizia
lizzazione.
  if e = 1 then gosub state_gameplay ; Se il gioco è nello stato 1, esegui la logica principale d
el gameplay.
  if e = 2 then gosub state_win      ; Se il gioco è nello stato 2, mostra la schermata di vittor
ia.
  if e = 3 then gosub state_game_over; Se il gioco è nello stato 3, mostra la schermata di game o
ver.
  goto main_loop ; Torna all'inizio del ciclo per il prossimo frame.

  rem ========= SUBROUTINES DI STATO =========

state_init_game ; Questa subroutine viene eseguita solo una volta all'inizio di ogni partita.
  x = 30 : y = 80  ; Imposta la posizione di partenza dell'Eroe.
  z = 100 : w = 40 ; Imposta la posizione di partenza del Guardiano.
  f = 0            ; Resetta lo stato della chiave (non posseduta).
  e = 1            ; Cambia lo stato del gioco a "In Gioco".
  i = 0            ; Resetta il flag di movimento del giocatore.
  s = 0            ; Resetta i timer dei suoni.
  return           ; Torna al main_loop.

state_gameplay ; Questa subroutine contiene tutta la logica del gioco attivo.
  gosub handle input      ; Legge il joystick e gestisce il movimento/animazione dell'Eroe.
  gosub update_enemy_ai   ; Muove e anima il Guardiano.
  gosub update sounds     ; Aggiorna i timer dei suoni e li spegne se necessario.
  gosub draw world        ; Disegna tutti gli elementi grafici sullo schermo.
  gosub check_collisions  ; Controlla tutte le interazioni tra gli oggetti.
  return                  ; Torna al main_loop.

state win ; Schermata di vittoria.
  if s = 0 then gosub play win sound ; Suona l'effetto di vittoria, ma solo una volta.
  COLUBK = $9E                   ; Imposta lo sfondo a verde.
  player0y = 200 : player1y = 200  ; Nasconde i personaggi spostandoli fuori dallo schermo.
  if s > 1 then s = s - 1 else AUDV0 = 0 ; Fa durare il suono di vittoria per il suo tempo, poi l
o spegne.
  drawscreen                      ; Continua a disegnare lo schermo per evitare il "roll".
  if joy0fire then e = 0          ; Se il giocatore preme fuoco, riavvia il gioco tornando allo
stato di inizializzazione.
  return

state_game_over ; Schermata di Game Over.
  if s = 0 then gosub play_lose_sound ; Suona l'effetto di sconfitta, ma solo una volta.
  COLUBK = $44                    ; Imposta lo sfondo a rosso.
  player0y = 200 : player1y = 200 ; Nasconde i personaggi.
  if s > 1 then s = s - 1 else AUDV0 = 0 ; Fa durare il suono di sconfitta per il suo tempo, poi
lo spegne.
  drawscreen                      ; Continua a disegnare lo schermo.
  if joy0fire then e = 0          ; Se il giocatore preme fuoco, riavvia il gioco.
  return
```

7.4 – Fase 4: Dare Vita al Mondo – Input, IA, Suoni e Disegno

È il momento di riempire le subroutine di gioco. Iniziamo con l'input del giocatore, l'IA del guardiano, la gestione dei suoni e il disegno del mondo.

```
  rem ========= SUBROUTINES DI GIOCO =========

handle_input
  rem -- salviamo la posizione attuale dell'eroe per la logica "salva e ripristina" --
  u = x : v = y

  i = 0 ; All'inizio di ogni frame, assumiamo che il giocatore sia fermo.
  rem -- Legge il joystick e aggiorna la posizione e la riflessione dell'Eroe --
  if joy0left && x > 0 then x = x - 1 : a = 8 : i = 1   ; Se premi sinistra e non sei al bordo, m
uoviti a sinistra, imposta la riflessione e il flag di movimento.
  if joy0right && x < 159 then x = x + 1 : a = 0 : i = 1 ; Se premi destra...
  if joy0up && y > 0 then y = y - 1 : i = 1              ; Se premi su...
  if joy0down && y < 95 then y = y + 1 : i = 1           ; Se premi giù...

  rem -- Gestisce l'animazione dell'Eroe e il suono dei passi --
  if i = 1 then b = b + 1 : if b > 20 then b = 0 ; Se l'eroe si sta muovendo, incrementa il suo t
imer di animazione.
  if i = 1 && s = 0 then gosub play hero step sound ; Se si muove e il canale audio 0 è libero, r
iproduci il suono del passo.
  if i = 0 then b = 0 ; Se l'eroe è fermo, resetta il suo timer di animazione per mostrare il fra
me statico.

  return

update_enemy_ai
  rem -- Logica di pattugliamento semplice: si muove avanti e indietro tra z = 20 e 120 --
  if d = 0 then z = z + 1      ; Se la direzione è 0 (destra), incrementa la posizione X.
  if z > 120 then d = 1        ; Se raggiunge il limite destro, cambia direzione.
  if d = 1 then z = z - 1      ; Se la direzione è 1 (sinistra), decrementa la posizione X.
  if z < 20 then d = 0         ; Se raggiunge il limite sinistro, cambia direzione.

  rem -- Gestisce l'animazione del Guardiano e il suono dei passi --
  c = c + 1 : if c > 20 then c = 0 ; Incrementa il timer di animazione del guardiano.
  if t = 0 then gosub play_enemy_step_sound ; Se il canale audio 1 è libero, riproduci un passo
  return

update_sounds
  rem -- Gestisce i contatori alla rovescia per entrambi i canali audio --
  if s > 0 then s = s - 1 else AUDV0 = 0 ; Decrementa il timer del canale 0. Se arriva a 0, spegn
e il volume.
  if t > 0 then t = t - 1 else AUDV1 = 0 ; Decrementa il timer del canale 1. Se arriva a 0, spegn
e il volume.
  return

draw world
  rem -- Seleziona e chiama la subroutine grafica corretta per l'Eroe in base al suo timer di ani
mazione 'b' --
  if b = 0 then gosub player0_static
  if b > 0 && b <= 10 then gosub player0_frame1
  if b > 10 then gosub player0_frame2

  rem -- Seleziona e chiama la subroutine grafica corretta per il Guardiano in base al timer 'c'
--
  if c <= 10 then gosub player1_frame1

  if c > 10 then gosub player1_frame2

  rem -- Posiziona la chiave (la 'ball') sullo schermo. Se è stata raccolta (f=1), la sposta fuor
i dall'area visibile --
  ballheight = 4 : if f = 0 then ballx = 120 : bally = 50 else bally = 150

  rem -- Aggiorna le posizioni finali degli sprite e imposta tutti i registri TIA prima di disegn
are --
  player0x = x : player0y = y : player1x = z : player1y = w
  COLUBK = $08 : COLUPF = $1A : COLUP0 = $AE : COLUP1 = $44 : REFP0 = a
```

```
  drawscreen ; Comando che dice al TIA di disegnare l'intero frame.
  return
```

7.5 – Fase 5: Le Regole del Gioco – Collisioni e Logica

Infine, implementiamo la logica che controlla le interazioni: la collisione con il guardiano, l'urto contro i muri, la raccolta della chiave e la fuga finale. E terminiamo con le subroutine dei suoni e delle animazioni.

```
check_collisions
  rem -- Controlla se l'Eroe tocca il Guardiano. Se sì, suona un suono e imposta lo stato a Game
Over --
  if collision(player0, player1) then gosub play_hit_sound : e = 3 : s = 0

  rem -- Controlla se l'Eroe tocca i muri del Playfield. Se sì, ripristina la sua posizione e suo
na un suono --
  if collision(player0, playfield) then x = u : y = v : if t = 0 then gosub play_hit_sound

  rem -- Controlla se l'Eroe tocca la chiave (la 'ball'). Se sì, imposta il flag 'f' a 1 e suona
un suono --
  if f = 0 && collision(player0, ball) then f = 1 : gosub play_pickup_sound

  rem -- Controlla se l'Eroe, con la chiave in mano, raggiunge l'area della porta. Se sì, imposta
 lo stato a Vittoria --
  if f = 1 && x < 20 && y < 28 then e = 2 : s = 0

  return


  rem ====== SUBROUTINES AUDIO (impostano timer e registri audio per ogni effetto) ======

play hero step sound
  s = 3 : AUDV0 = 8 : AUDC0 = 12 : AUDF0 = 25 : return
play enemy step sound
  t = 3 : AUDV1 = 6 : AUDC1 = 14 : AUDF1 = 28 : return
play_hit_sound
  t = 10 : AUDV1 = 12 : AUDC1 = 2 : AUDF1 = 30 : return
play_pickup_sound
  s = 15 : AUDV0 = 15 : AUDC0 = 12 : AUDF0 = 10 : return
play_win_sound
  s = 20 : AUDV0 = 15 : AUDC0 = 12 : AUDF0 = 5 : AUDV1 = 0 : return
play_lose_sound
  s = 20 : AUDV0 = 15 : AUDC0 = 2 : AUDF0 = 25 : AUDV1 = 0 : return


  rem ====== SUBROUTINES GRAFICHE (contengono i dati binari per ogni frame di animazione) ======

player0_static ; Frame per l'Eroe quando è fermo.
  player0:
  %0010100
  %0010100
  %0010100
  %1001001
  %0111110
  %0001000
  %0011100
  %0011100
end
  return

player0_frame1 ; Primo frame dell'animazione di corsa dell'Eroe.
  player0:
  %0010100
  %0010100
  %0010100
  %1001000
  %0111111
  %0001001
  %0011100
  %0011100
```

```
end
  return

player0_frame2 ; Secondo frame dell'animazione di corsa dell'Eroe.
  player0:
  %0010000
  %0010000
  %0010100
  %1001001
  %0111110
  %0001000
  %0011100
  %0011100
end
  return

player1_frame1 ; Primo frame dell'animazione del Guardiano.
  player1:
  %0010100
  %0010100
  %1010101
  %1011101
  %0111110
  %0001000
  %0111110
  %0011100
end
  return

player1_frame2 ; Secondo frame dell'animazione del Guardiano.
  player1:
  %1000001
  %0100010
  %0010100
  %0011100
  %0111110
  %1001001
  %1011101
  %0111110
end
  return
```

Premi **F5**. Il tuo gioco è ora completo e vivo! L'eroe e il guardiano si muovono, i suoni danno vita all'azione e c'è un obiettivo chiaro.

La misteriosa ball

Oltre ai due player, l'Atari 2600 può gestire altri tre oggetti grafici semplici: *missile0*, *missile1* e, appunto, *ball*. La *ball* è un semplice rettangolo il cui colore è legato a quello del Playfield (*COLUPF*). Possiamo controllarne la posizione (*ballx*, bally*)* e le dimensioni. È perfetta per rappresentare oggetti come proiettili, chiavi o tesori. Ne parleremo in dettaglio nel **Capitolo 9**. Per ora, ci basta sapere che è il nostro tesoro da raccogliere.

Il Labirinto delle Variabili a-z

Hai notato quanto può essere difficile tenere traccia di cosa fa ogni variabile? Benvenuto in una delle sfide centrali della programmazione "vecchia scuola"! Con poca memoria a disposizione, i programmatori dovevano essere estremamente metodici. Senza una "mappa" come quella che abbiamo scritto all'inizio, un programma può diventare rapidamente un groviglio indecifrabile. Questo problema della leggibilità è così sentito che Batari Basic offre una soluzione potente: il comando ***dim***. Con *dim*, possiamo dare alle nostre variabili dei nomi significativi

(es. *dim hero_x = a*). Da quel momento in poi, nel codice potremo usare *hero_x* al posto di *a* (ma sono la stessa variabile!) rendendolo più facile da leggere. Abbiamo scelto di non usare *dim* in questo primo progetto per farti "toccare con mano" le sfide originali, ma d'ora in poi lo useremo!

La Necessità di una ROM da 4K

Se avessimo usato all'inizio del programma set romsize 2k avremmo incontrato un errore spaventoso nella finestra di OUTPUT, simile a questo:

-89 bytes of ROM space left …

error: Origin Reverse-indexed.

ERROR: 2600

basic compilation failed.

Questo errore indica il **superamento del limite di memoria della cartuccia**. Cosa significa questo errore? All'inizio dei nostri programmi abbiamo sempre scritto *set romsize 2k*. Questa direttiva dice al compilatore: "Prepara una cartuccia virtuale da 2 kilobyte (2048 byte)". Il nostri programmi finora non sforavano i 2K. Ma "Fuga dal Castello Digitale", con le sue animazioni multiple, le subroutine per la logica e gli effetti sonori, è diventato più grande. L'errore *-89 (meno 89) bytes of ROM space left* è il modo del compilatore di dirti: "Ho finito lo spazio! Il tuo programma è 89 byte più grande di una cartuccia da 2K". L'errore successivo (*Origin Reverse-indexed*) è una conseguenza tecnica di questo sforamento.

La Soluzione: **Una Cartuccia più Grande!** Proprio come nel mondo reale, se un gioco era troppo complesso per una cartuccia da 2K, gli sviluppatori ne usavano una più capiente. La dimensione successiva più comune era quella da 4 kilobyte (4096 byte). Per risolvere il nostro problema, basta comunicare al compilatore che vogliamo usare una cartuccia più grande con *set romsize 4k*. Ora premi F5. Il gioco compilerà senza errori e ti verrà indicato quanti byte di preziosa memoria ROM ti rimangono ancora (1951 bytes)!

2600 Basic compilation complete. 1951 bytes of ROM space left

Gestire le dimensioni della ROM è una parte fondamentale del lavoro di un programmatore dell'Atari 2600. In questo manuale non andremo oltre i 4K, ma potrai trovare informazioni interessanti su ROM più grandi e altro nel capitolo 13.

Un altro errore "strano" : *Branch out of range*

Man mano che i tuoi giochi diventano più complessi, con molte subroutine e una logica articolata, potresti incontrare un errore di compilazione apparentemente strano, simile a questo: "*Error: Branch out of range"*.

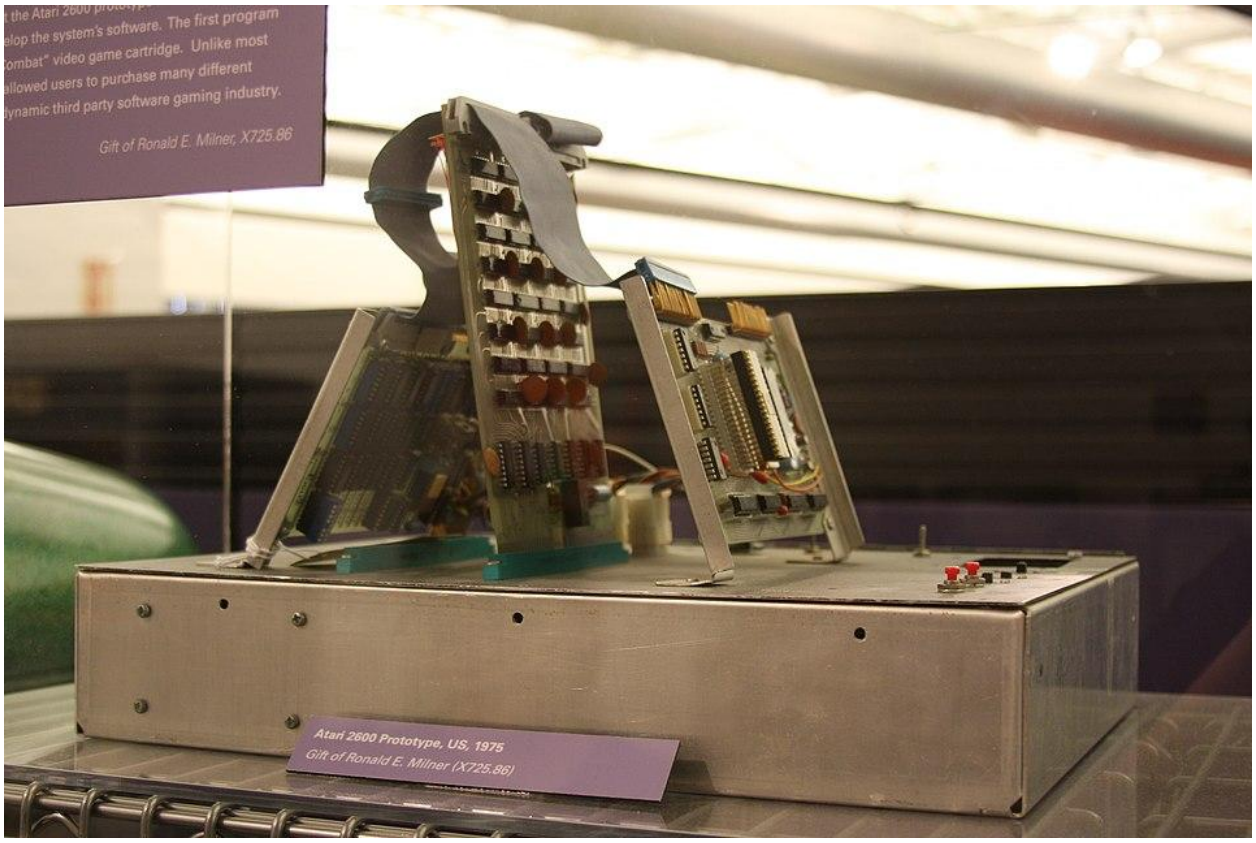
Questo errore non significa che il tuo codice sia sbagliato, ma che stai chiedendo al programma di fare un "salto" (goto o gosub) troppo lungo.

La soluzione è inserire *set smartbranching on* immediatamente dopo *set romsize:*

```
set smartbranching on
```

A questo punto ci penserà batari basic a permettere ai tuoi *goto* e *gosub* di "saltare lontano".

Parte 2: Tecniche Avanzate e Segreti dell'Hardware



Prototipo della console Atari 2600 esposto al Computer History Museum. Foto: Pargon, CC BY 2.0

Capitolo 8 – Alias, palla e missili

In questo capitolo, impareremo a scrivere codice più pulito e a sfruttare altri oggetti grafici che l'Atari 2600 ci mette a disposizione. Passeremo da semplici avventurieri a veri ingegneri del codice.

8.1 – Organizzare il Codice: Gli Alias con dim

Nel capitolo precedente, hai sperimentato in prima persona la difficoltà di tenere traccia di cosa fa ogni variabile da *a* a *z*. Un piccolo errore di distrazione, e un gioco può smettere di funzionare in modi misteriosi. Per risolvere questo problema e rendere il nostro codice infinitamente più leggibile, Batari Basic ci offre un comando fondamentale: **dim**.
dim (che sta per *dimension*) ci permette di creare un **alias**, ovvero un soprannome, per una delle variabili a singola lettera.

La sintassi è: *dim nome_significativo = lettera*

```
  rem Esempio di utilizzo di 'dim'

  dim hero_x = a
  dim hero_y = b
  dim has_key = c

  rem Inizializzazione
  hero_x = 80
  hero_y = 50
  has_key = 0

main
   if joy0left then hero_x = hero_x - 1
   ; ...
```

Come funziona? Dopo aver dichiarato *dim hero_x = a*, ogni volta che userai *hero_x* nel tuo codice, il compilatore lo sostituirà automaticamente con a. Per te, il codice diventa leggibile come un libro; per la console, non cambia assolutamente nulla in termini di performance.

Il comando *dim* è incredibilmente potente, ma nasconde una trappola molto insidiosa. Quando scegli un nome per la tua variabile (un alias), devi assolutamente **evitare di usare esattamente o di iniziare** il nome della tua variabile con:

- parole Chiave di Batari Basic: *rem*, *if*, *then*, *goto*, *end*, ecc.

- nomi di Registri Hardware: *COLUBK*, *player0x*, *REFP0*, *NUSIZ0*, *AUDV0*, ecc.

- variabili Speciali: *score*, *pfscorecolor*, *missile0height*, ecc.

Se nel codice useremo un alias sconosciuto o non corretto (ad esempio: *hero_xx*), verrà segnalato con un errore del tipo:

*primo_gioco.bas.asm (1818): error: **Unknown Mnemonic** 'sta **hero_xx** '.*

Attenzione! **Queste regole si applicano anche alle label**. Mai usare un nome per la label uguale ad una parola chiave del linguaggio o che iniza con essa!

Da questo punto in avanti, useremo **sempre** *dim* per le nostre variabili. È una delle pratiche più importanti per scrivere codice pulito e facile da modificare in futuro. Diremo addio al "labirinto delle variabili a-z" e daremo ai nostri dati dei nomi che abbiano un senso. Scegli sempre nomi unici e descrittivi per le tue variabili, preferibilmente usando il minuscolo e il trattino basso "_" per separare le parole.

8.2 – Oggetti Grafici Semplici: Palla e Missili

Finora abbiamo lavorato principalmente con gli sprite (player0, player1) e lo sfondo (playfield). Ma l'Atari 2600 ha altri tre assi nella manica: la **Palla** (ball) e i due **Missili** (missile0, missile1). Sono oggetti grafici semplici, ma incredibilmente versatili, usati in innumerevoli classici da *Pong* a *Combat*.
Non puoi definirne la forma con dati binari come fai per gli sprite, ma puoi controllarne posizione, dimensione e colore.

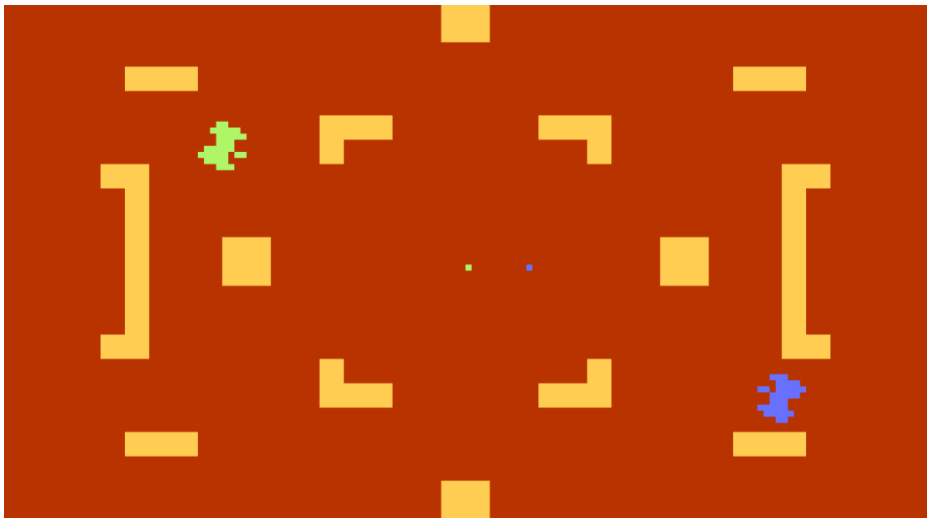
Colore Condiviso: Questi oggetti, per come è stato progettato l'hardware dell'Atari 2600, "prendono in prestito" il colore da altri elementi:

- La ball ha sempre lo stesso colore del Playfield (*COLUPF*).
- missile0 ha sempre lo stesso colore di player0 (*COLUP0*).
- missile1 ha sempre lo stesso colore di player1 (*COLUP1*).

Controllo Dimensioni: La loro altezza e larghezza sono controllate da registri speciali.

Altezza: *ballheight, missile0height, missile1height* (valori da 1 a 8 pixel).

Larghezza: Controllata da bit specifici nei registri *CTRLPF* (per la ball) e *NUSIZ0*/*NUSIZ1* (per i missili). Le larghezze possibili sono 1, 2, 4 o 8 pixel.



Screenshot di Combat: i missili (al centro dello schermo) hanno lo stesso colore dei player

8.3 – La Palla Rimbalzante

Mettiamo subito in pratica queste conoscenze. Creeremo un programma che fa rimbalzare una palla all'interno dello schermo. Questo è il cuore di giochi come *Pong* o *Breakout*. Vogliamo:

- Creare una palla visibile e di dimensioni adeguate (4x4 pixel).
- Darle una velocità iniziale.
- Invertire la sua velocità quando tocca i bordi dello schermo.

```
  rem La Palla Rimbalzante
  set romsize 2k

  dim ball_x = a
  dim ball_y = b
  dim vel_x = c
  dim vel_y = d

  rem --- Inizializzazione ---
  ball_x = 80 ; Posizione iniziale
  ball_y = 50
  vel_x = 1   ; Velocità iniziale
  vel_y = 1

main_loop
  rem --- Aggiorna Posizione ---
  ball_x = ball_x + vel_x
  ball_y = ball_y + vel_y

  rem --- Logica di Rimbalzo sui Bordi ---
  if ball_x < 10 || ball_x > 150 then vel_x = 0 - vel_x ; inverte velocità x
  if ball_y < 10 || ball_y > 85 then vel_y = 0 - vel_y ; inverte velocità y

  rem --- Disegno ---
  ballx = ball_x      ; Assegna la posizione X calcolata al registro hardware
  bally = ball_y      ; Assegna la posizione Y calcolata al registro hardware
  ballheight = 4      ; Altezza di 4 pixel
  CTRLPF = 32         ; Larghezza di 4 pixel (vedi Appendice B)

  COLUBK = $08        ; Sfondo grigio
  COLUPF = $1E        ; Il colore della palla sarà giallo

  drawscreen
  goto main_loop
```

Premi **F5**. Vedrai una palla quadrata verde rimbalzare all'infinito sullo schermo. Hai appena creato il tuo primo motore fisico! Per una guida completa su tutti i valori possibili per i registri *CTRLPF* e *NUSIZx*, consulta l'Appendice B.

8.4 – La Magia dei Missili Orizzontali

Finora abbiamo pensato ai missili come proiettili verticali. Ma come si creano oggetti orizzontali come la spada di un cavaliere in *Adventure* o i laser in *Berzerk*? La risposta è un altro geniale trucco del TIA.
Per creare un missile orizzontale, devi:

- Impostare la sua altezza (*missile0height*) a un valore molto piccolo (di solito **0**). Questo lo trasforma in una linea sottile.
- Usare il registro *NUSIZ0* per dargli una larghezza (fino a 8 pixel).

La Genialità Nata dalla Necessità

Affermare che il Television Interface Adapter (TIA) è "geniale" non è un'esagerazione, ma il riconoscimento di una delle più incredibili opere di ingegneria minimalista nella storia dei videogiochi. Per capire l'Atari 2600, dobbiamo tornare al 1977. L'obiettivo non era creare la console più potente possibile, ma quella più economica possibile. Ogni componente, ogni transistor, ogni singolo centesimo risparmiato sul costo di produzione era fondamentale per rendere la console accessibile alle famiglie. Questa filosofia di design, guidata dal leggendario ingegnere Jay Miner, portò alla creazione di un hardware estremamente limitato, ma incredibilmente flessibile.

Nata per Pong, Preparata per l'Impossibile

Inizialmente, l'hardware dell'Atari 2600 fu concepito per giochi molto semplici, come *Pong* o *Combat*. Il TIA era stato progettato per muovere pochi oggetti (due "racchette", due "proiettili", una "palla") su uno sfondo quasi inesistente. Non esisteva un "framebuffer", ovvero una memoria video dove disegnare un'immagine completa. Tutto doveva essere generato in tempo reale, riga per riga, in sincrono con il pennello elettronico del televisore ("**Racing the Beam**"). Sembrava una condanna a giochi eternamente semplici. E invece, accadde l'incredibile. I programmatori, inizialmente gli stessi ingegneri di Atari e poi quelli delle prime software house come *Activision*, iniziarono a "interrogare" l'hardware. Scoprirono che, cambiando i registri del TIA nel mezzo del disegno di un singolo frame, potevano convincere il TIA a fare cose per cui non era mai stato progettato. Volevano più di due oggetti per riga? Cambiavano la posizione orizzontale di uno sprite "al volo" dopo che era già stato disegnato, per farlo riapparire in un altro punto della stessa riga, creando l'illusione di più oggetti (una tecnica usata per gli alieni di *Space Invaders*). Volevano sfondi complessi e colorati? Cambiavano i registri del colore del playfield a ogni nuova scanline per creare cieli sfumati, orizzonti e terreni. Volevano oggetti complessi e non solo proiettili verticali? Hanno trasformato un missile in una sottile linea orizzontale e gli hanno dato una larghezza variabile, creando spade, laser e barriere. La "magia" dell'Atari 2600 non risiede tanto nella potenza del suo hardware, quanto nella sua vulnerabilità al controllo del software. Il TIA non era un processore grafico rigido; era un set di strumenti grezzi che un programmatore abile poteva "suonare" come uno strumento musicale, inventando nuove melodie (tecniche) ad ogni frame. Ogni gioco innovativo, da *Pitfall!* a *River Raid*, era una testimonianza non di ciò che l'hardware poteva fare, ma di ciò che l'ingegno di un programmatore poteva costringerlo a fare. È questa la vera eredità dell'Atari 2600: la dimostrazione che i limiti, quando affrontati con creatività, non sono muri, ma trampolini di lancio per l'inventiva!

8.5 – La Spada dell'Eroe

Mettiamo in pratica la creazione di un oggetto orizzontale. In questo esempio, il nostro eroe (un semplice quadrato) potrà brandire una "spada" premendo il pulsante di fuoco. Questo codice infatti crea una "spada" orizzontale di 8 pixel che appare quando si preme il pulsante di fuoco.

```
  rem La Spada dell'Eroe
  set romsize 2k

  dim hero_x = a
  dim hero_y = b

  rem --- Inizializzazione ---
  hero_x = 80
  hero_y = 50

  player0:
  %11100111
  %01100110
  %01100110
  %00111100
  %11111111
  %00011000
  %00011000
end


main loop
  rem --- Logica di Movimento ---
```

```
 if joy0left then hero_x = hero_x - 1
 if joy0right then hero x = hero x + 1
 if joy0up then hero_y = hero_y - 1
 if joy0down then hero y = hero y + 1

 rem --- Logica della Spada ---
                 ; se giocatore preme fuoco:
                 ; Allinea la spada verticalmente al centro dell'eroe
                 ; Posiziona la spada a destra dell'eroe
                 ; Altezza minima, la trasforma in una linea
                 ; Larghezza di 8 pixel (M=3, P=0 - vedi Appendice B)
 if joy0fire then missile0y = hero_y - 5: missile0x = hero_x + 8 : missile0height = 0 : NUSIZ0 =
$30
                 ; se giocatore non preme fuoco:
                 ; Nascondi la spada fuori dallo schermo
 if !joy0fire then missile0y = 200

 rem --- Disegno ---
 player0x = hero_x
 player0y = hero y

 COLUP0 = $1E ; Colore dell'eroe e della spada (giallo)
 COLUBK = $04 ; Sfondo grigio

 drawscreen
 goto main_loop
```

Premi **F5**. Muovi il tuo quadrato sullo schermo. Ora, tenendo premuta la barra spaziatrice, vedrai apparire una linea gialla orizzontale accanto a esso. Hai creato la tua prima spada! Combinando *missileXheight* e *NUSIZx*, puoi creare proiettili e oggetti di forme diverse, superando di gran lunga l'idea di un semplice "missile". Hai appena sbloccato un altro potente strumento del tuo arsenale di ingegnere Atari.

Hai appena imparato a controllare la posizione di *player0, missile0* e *ball*. Potresti pensare che per allinearli perfettamente basti assegnare loro le stesse coordinate. Ad esempio:

player0x = 50 : player0y = 50

missile0x = 50 : missile0y = 50

ballx = 50 : bally = 50

Se provi questo codice, noterai qualcosa di molto strano: orizzontalmente (x) gli oggetti saranno allineati, ma verticalmente (y) appariranno tutti a diverse altezze! Benvenuto in una delle peculiarità più complesse dell'Atari 2600!

L'Asse X (Orizzontale): Semplice e Prevedibile

Fortunatamente, sull'asse X non ci sono sorprese. La coordinata x si riferisce sempre al **pixel più a sinistra** di ogni oggetto. Impostare lo stesso valore di x per *player0, missile0* e *ball* li allineerà perfettamente sul loro bordo sinistro.

L'Asse Y (Verticale): Il Dominio del Kernel

Qui le cose si complicano. Il problema è che l'origine verticale (il "punto zero") **non è la stessa per ogni tipo di oggetto**. Questa differenza non è causata dall'hardware (il TIA), ma dal **kernel di Batari Basic**. Per ottimizzare il disegno dello schermo, il kernel introduce dei piccoli slittamenti (offset) verticali diversi per ogni classe di oggetto.

Non esiste una formula magica unica (*y_missile = y_player + N*) che funzioni in ogni situazione per allineare gli oggetti. L'offset esatto può variare leggermente a seconda delle opzioni del kernel che usi, dell'altezza dello sprite e di altri fattori di ottimizzazione. L'unico modo affidabile per allineare perfettamente gli oggetti sull'asse Y è **sperimentare e trovare l'offset giusto per il tuo gioco**. Inizia con lo stesso valore di y e poi aggiusta finché il risultato non ti soddisfa (ovvero gli oggetti appaiono dove desideri).

Cos'è il **Kernel** di Batari Basic?

Batari Basic è un "traduttore" (compilatore) che trasforma il nostro codice in linguaggio macchina per l'Atari 2600, il vero linguaggio che la CPU 6507 è in grado di eseguire. Ma non è tutto. Quando compili il tuo gioco, Batari Basic fa qualcosa di molto intelligente: inietta nel tuo file di gioco una porzione di codice pre-scritto, estremamente ottimizzato, chiamato Kernel. Pensa al Kernel come al motore grafico e sonoro del tuo gioco. È una complessa routine che si occupa dei compiti più difficili e ripetitivi. Il suo lavoro principale è uno dei più ardui della programmazione Atari: disegnare lo schermo (*drawscreen*).

Cosa Fa Esattamente il **Kernel Standard**?

Quando nel nostro *main_loop* chiamiamo *drawscreen*, in realtà stiamo dicendo al Kernel: "Prendi il comando!". A quel punto, il Kernel si assume la responsabilità di:

1. Sincronizzarsi con il Televisore: Gestisce il "Racing the Beam", assicurandosi che ogni riga venga disegnata al momento giusto.

2. Disegnare tutti gli Oggetti: Legge le posizioni e i dati grafici di *player0, player1, missile0, missile1, ball e playfield* e li disegna sullo schermo, riga per riga.

3. Creare le Basi per il Suono e l'Input: Si assicura che il TIA e il RIOT siano pronti a ricevere i nostri comandi.

In pratica, il Kernel è il nostro assistente instancabile che si occupa di tutta la "bassa manovalanza" hardware, permettendoci di concentrarci sulla logica del gioco usando comandi semplici. Senza il Kernel, dovremmo gestire manualmente ogni singola scanline del televisore, un compito incredibilmente complesso.

Questo incredibile aiuto, però, ha un costo: lo spazio. Il codice del Kernel Standard occupa una porzione significativa della nostra preziosa memoria ROM.

Esistono anche altri Kernel specializzati (come il DPC+ o i Multisprite Kernel), ognuno con i propri compromessi tra funzionalità e spazio occupato. Per questo manuale, ci concentreremo sul Kernel Standard, il perfetto punto di partenza per ogni esploratore dell'Atari 2600.

Mai dimenticarsi dei Registri Volatili!

Per come funziona il kernel, alcuni registri sono volatili ovvero **devi ricordarti di reimpostarli ad ogni frame nel *main loop*** perchè la *drawscreen* li azzererà. Ecco la lista dei registri grafici volatili più comuni che devono essere sempre reimpostati all'interno del *main loop*, **prima di ogni drawscreen**:

REFP0, REFP1 (Riflessione degli Sprite): Specchiano orizzontalmente player0 e player1.

NUSIZ0, NUSIZ1 (Dimensione e Copie degli Sprite/Missili): Controllano la larghezza dei missili e il numero di copie o la larghezza (doppia, quadrupla) dei player.

COLUP0, COLUP1, COLUPF (Colori dei Player, Missili, Palla e Playfield): Definiscono il colore degli oggetti mobili e dello sfondo.

PF0, PF1, PF2 (Dati del Playfield - per colonne fisse): Usati per disegnare colonne verticali fisse (utili per mascherare artefatti o creare barre laterali).

Troverai molte altre informazioni sui registri nell'appendice B!

8.6 – Progetto Guidato: Tiro al Bersaglio

È il momento di mettere insieme tutto quello che abbiamo imparato in questo capitolo per creare un nuovo mini-gioco completo.
Il giocatore (player0) si muove solo orizzontalmente in basso. Premendo fuoco, spara un proiettile (*missile0*) verso l'alto. Un bersaglio (*ball*) cade dall'alto in posizioni casuali. Se il proiettile colpisce il bersaglio, il punteggio aumenta. Se il bersaglio raggiunge il fondo, il gioco finisce.

La Variabile score

Batari Basic ci offre una variabile speciale chiamata score. È un contatore a 6 cifre visualizzato permanentemente in fondo allo schermo. A differenza delle normali variabili (0-255), può gestire numeri fino a 999.999. Per aggiungere punti si usa l'aritmetica standard, come *score = score + 1*. Approfondiremo il suo funzionamento e le opzioni di personalizzazione nel **Capitolo 15**.

La Casualità con rand

Il comando *rand* genera un numero casuale da 0 a 255. È lo strumento perfetto per aggiungere imprevedibilità ai nostri giochi, come far apparire i nemici in posizioni diverse. Per ottenere una vera casualità tra una partita e l'altra, è necessario però inizializzare il generatore di numeri casuali, una tecnica che esploreremo nell'**Appendice C**.

Ecco il codice completo per il nostro tiro al bersaglio.

```
  rem Progetto: Tiro al Bersaglio
  set romsize 2k

  rem --- Alias delle Variabili ---
  dim player_x = a
  dim missile_y = b
  dim target_x = c
  dim target_y = d
  dim game_over = e
  dim timer_caduta = f

  rem --- Inizializzazione del Gioco ---
  gosub reset_game

main_loop
  rem se il gioco non è finito, continua con la logica di gioco
  if game_over = 0 then goto main_loop2

  rem Se il gioco è finito, attendi l'input per riavviare
  if joy0fire then gosub reset_game
  goto draw_frame ; Salta la logica di gioco
```

```
main_loop2
  rem --- Logica di Gioco ---

  rem 1. Movimento del Giocatore
  if joy0left && player_x > 10 then player_x = player_x - 1
  if joy0right && player_x < 150 then player_x = player_x + 1

  rem 2. Logica dello Sparo
  rem Se il pulsante è premuto E non c'è già un missile attivo (missile_y > 0)
  if joy0fire && missile y = 0 then missile y = 85 : missile0x = player x + 4
                                    ; Posizione di partenza del missile
                                    ; Allinea il missile al centro del giocatore

  rem 3. Movimento del Missile
  if missile y > 0 then missile y = missile y - 3 : if missile y < 10 then missile y = 0
                      ; Muovi il missile verso l'alto
                      ; Se raggiunge la cima, disattivalo

  rem 4. Movimento del Bersaglio
  timer caduta = timer caduta + 1 ; ogni frame incrementa il timer di caduta
  if timer_caduta = 3 then timer_caduta = 0 : target_y = target_y + 1
                          ; Fai cadere il bersaglio solo quando timer caduta è uguale a 3
                          ; questo rallenta la caduta

  if target y > 90 then game over = 1 ; Se il bersaglio tocca il fondo il gioco finisce.

  rem 5. Controllo Collisioni
  rem La funzione collision() controlla se missile0 e ball si toccano
  if collision(missile0, ball) then score = score + 1 : gosub reset_target : missile_y = 0

                                  ; Aumenta il punteggio
                                  ; Fai riapparire il bersaglio in un nuovo punto
                                  ; Disattiva il missile per poter sparare di nuovo
draw_frame
  rem --- Sezione di Disegno ---

  rem Disegna il giocatore
  player0x = player_x
  player0y = 88

  player0: ; Una semplice forma a "torretta"
  %11111111
  %01111110
  %00111100
end

  rem Posiziona il missile (se attivo) oppure nascondilo
  if missile_y > 0 then missile0y = missile_y
  if missile_y = 0 then missile0y = 200
  missile0height = 8 ; Un missile alto e sottile

  rem Disegna il bersaglio (la ball)
  ballx = target_x
  bally = target_y
  ballheight = 4
  CTRLPF = 32 ; Rende la palla larga 4 pixel, per farla quadrata

  rem Imposta i colori
  if game_over = 1 then COLUBK = $44 ; Sfondo rosso in game over
  if game_over = 0 then COLUBK = $08 ; Sfondo grigio durante il gioco

  COLUP0 = $9E ; azzurro per giocatore e missile
  COLUPF = $1E ; Giallo per il bersaglio (la ball condivide il colore del playfield)
  scorecolor = $1E ; Colore giallo per il testo dello score

  drawscreen
  goto main_loop

  rem ========= SUBROUTINES =========

reset_game
```

```
  rem Questa subroutine inizializza o ripristina lo stato del gioco
  player_x = 80
  missile_y = 0
  game_over = 0
  score = 0
  gosub reset_target
  return

reset_target
  rem Riposiziona il bersaglio in un nuovo punto casuale in alto
  target_x = rand/2 + 20 ; Usa rand per la posizione X, con un offset
  target_y = 1
  timer_caduta = 0
  return
```

Analizziamo per bene il codice.

- **set romsize 2k**: Come sempre, diciamo al compilatore di preparare una cartuccia da 2 kilobyte.
- **dim ...**: Usiamo dim per dare nomi significativi alle nostre variabili. Questo rende il codice molto più facile da leggere. *player_x* controllerà la posizione orizzontale del giocatore, *missile_y* quella del proiettile, e così via. *game_over* sarà il nostro flag di stato principale, e *timer_caduta* ci servirà per rallentare il bersaglio.
- **gosub reset_game**: All'avvio del programma, chiamiamo subito la subroutine *reset_game*. Questa routine, che vedremo più avanti, si occupa di impostare tutti i valori iniziali (posizione del giocatore, punteggio a zero, ecc.), preparando il campo di gioco per la prima partita.
- Il ***main_loop*** è molto semplice e funge da "smistatore". Controlla la variabile *game_over*. Se è 0, significa che stiamo giocando, quindi salta (goto) al ciclo di gioco vero e proprio, etichettato *main_loop2*.
  Se *game_over* è 1, significa che la partita è finita. Il programma rimane in attesa. Se il giocatore preme il pulsante di fuoco (*joy0fire*), chiama di nuovo reset_game per riavviare la partita. In questo stato, salta direttamente alla sezione di disegno (*goto draw_frame*) per mantenere lo schermo attivo ma "congelare" il gioco.
- **Movimento del Giocatore:** Il codice legge il joystick. Se viene premuto sinistra o destra, e il giocatore non ha raggiunto i bordi dello schermo (delimitati da 10 e 150), la sua posizione *player_x* viene aggiornata.
- **Logica dello Sparo:** Questa è una riga molto densa. Controlla due condizioni contemporaneamente (&&):
  - Il giocatore sta premendo il pulsante di fuoco (joy0fire)?
  - Non c'è già un missile attivo sullo schermo (*missile_y = 0*)? Se entrambe le condizioni sono vere, "attiva" un nuovo missile impostando *missile_y* a 85 (la sua posizione di partenza verticale) e allinea la sua posizione orizzontale (*missile0x*) al centro del giocatore. Il fatto di controllare *missile_y = 0* ci impedisce di sparare raffiche infinite di missili.
- **Movimento del Missile:** Se un missile è attivo (*missile_y > 0*), la sua posizione verticale viene decrementata di 3 ad ogni frame, facendolo muovere verso l'alto. Se il missile raggiunge la cima dello schermo (*missile_y < 10*), la sua variabile *missile_y* viene resettata a 0, "disattivandolo" e permettendo al giocatore di sparare di nuovo.
- **Movimento del Bersaglio:** Qui usiamo un timer per rallentare la caduta. timer_caduta viene incrementato a ogni frame. Solo quando raggiunge il valore 3, il bersaglio scende di un pixel (*target_y = target_y + 1*) e il timer viene azzerato. In pratica, il bersaglio si

muove solo un frame ogni tre, apparendo più lento. Se il bersaglio raggiunge il fondo (*target_y > 90*), la variabile game_over viene impostata a 1, terminando la partita.

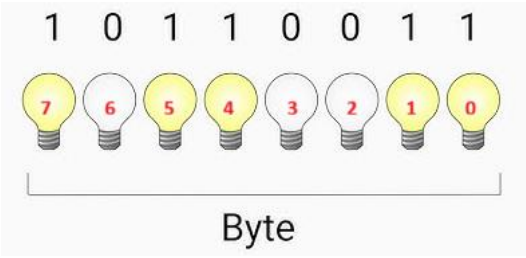
Controllo Collisioni: La funzione *collision()* controlla se il *missile0* e la *ball* (il nostro bersaglio) si stanno toccando. Se sì, esegue tre azioni in sequenza:
1. *score = score + 1* : Aumenta il punteggio.
2. *gosub reset_target*: Chiama la subroutine che riposiziona il bersaglio in un nuovo punto casuale.
3. *missile_y = 0*: Disattiva il missile, permettendo al giocatore di sparare di nuovo.

- Il codice posiziona ogni oggetto (*player0x, missile0y, ballx,* ecc.) usando i valori delle variabili calcolate nella sezione logica.
- Imposta i colori in base allo stato del gioco: lo sfondo *COLUBK* diventa rosso quando *game_over* è 1.
- Infine, *drawscreen* disegna tutto e *goto main_loop* fa ripartire il ciclo.
- ***reset_game***: Questa routine viene chiamata all'inizio e al riavvio. Imposta tutte le variabili ai loro valori di partenza (punteggio a zero, giocatore al centro, ecc.) e poi chiama *reset_target* per posizionare il primo bersaglio.
- ***reset_target***: Questa è la routine che rende il gioco imprevedibile. Usa il comando *rand* per generare un numero casuale, che viene usato per calcolare una nuova posizione *target_x* per il bersaglio. Reimposta anche la posizione *target_y* in cima allo schermo e azzera il timer di caduta.

8.7 – Usare i bit-flag

Finora, per memorizzare uno stato semplice come "il giocatore ha la chiave?" abbiamo usato un'intera variabile (un byte completo, che può contenere 256 valori diversi, da 0 a 255) per rispondere a una domanda che ha solo due risposte: sì o no. È come usare un intero foglio di carta per scrivere una singola spunta. Nella programmazione Atari, dove ogni byte è un tesoro, questo è un lusso che non sempre possiamo permetterci.
Esiste una tecnica da programmatori esperti per ottimizzare la memoria: i **bit-flag**.

Una singola variabile (un byte) è composta da 8 bit. Ogni bit può essere visto come un piccolo interruttore indipendente, che può essere "acceso" (valore 1) o "spento" (valore 0). Invece di usare una variabile per un solo stato, possiamo usarne una per memorizzarne fino a 8!



Nell'immagine sopra, vediamo il *byte* composto dai suoi 8 bit. Il bit "0" è quello più a destra e nell'esempio è accesso e quindi è 1. Il bit "6" è il penultimo a sinistra ed è spento, cioè 0.

In Batari Basic per leggere o scrivere un singolo bit di una variabile, si usa questa sintassi:

variabile{numero_bit}

Immaginiamo di voler tenere traccia di più stati per il nostro eroe: il possesso della spada e il possesso dello scudo. Senza i bit-flag, avremmo bisogno di due variabili separate (un grande spreco!):

```
  dim has_sword = a
  dim has_shield = b
```

Con i bit-flag, possiamo usare una sola variabile, che chiameremo *hero_flags*:

```
  dim hero_flags = a

  rem Inizializzazione: l'eroe non ha nulla
  hero_flags = 0

main loop
   rem ... logica di gioco ...

   rem L'eroe ha trovato la spada!
   hero_flags{0} = 1  ; Accendi il bit 0 per indicare che ha la spada


   rem ... logica di gioco ...

   rem L'eroe ha trovato lo scudo!
   hero_flags{1} = 1  ; Accendi il bit 1 per indicare che ha lo scudo


   rem ... logica di gioco ...

   rem Possiamo attaccare solo se abbiamo la spada
   if joy0fire && hero_flags{0} then gosub attack_routine
```

In questo esempio, abbiamo usato una singola variabile (a) per gestire due stati completamente diversi, semplicemente usando separatamente i suoi bit 0 e 1.

Il Test con if

Quando controlli un bit-flag in una condizione *if*, ricorda la regola che abbiamo visto per il joystick: non usare il segno di uguale!

Il test *if variabile{bit}* è già di per sé una domanda “questo bit è uguale a 1?”.

if hero_flags{0} then ... ← **Corretto!** (Significa “SE il bit 0 è 1…”)

if hero_flags{0} = 1 then ... ← **Errato!**

Per controllare se un bit è 0, usa l’operatore di negazione !:

if !hero_flags{0} then ... (Significa “SE il bit 0 NON è 1 (cioè è 0)…”)

Perché un Byte va da 0 a 255?

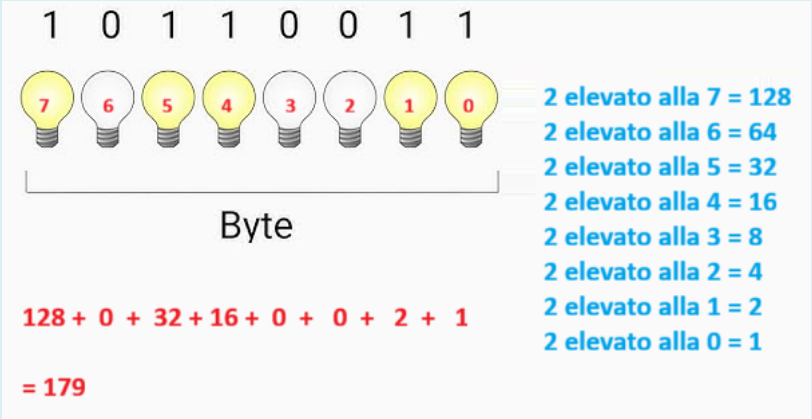
Hai notato che tutte le variabili (a...z) e i registri del TIA possono contenere solo numeri da 0 a 255?

La quantità fondamentale di informazione che il tuo Atari 2600 (e qualsiasi computer) elabora è il Byte. Un byte è un gruppo di 8 Bit.

Cos'è un Bit? Un Bit (*Binary Digit*) è un singolo interruttore elettronico che può essere solo su due stati: Acceso (1) o Spento (0).

Ogni bit all'interno del byte ha un valore fisso, che è una **potenza del 2**, proprio come le cifre nelle nostre normali decine e centinaia.

Quando un bit è Acceso (1), il suo valore viene contato. Quando è Spento (0), il suo valore viene ignorato.



Quindi, guardando all'esempio qui sopra, se un registro o una variabile ha il valore 179, significa che i suoi bit 7,5,4,1 e 0 sono "accesi".

Padroneggiare i bit-flag è una delle abilità chiave per "spremere ogni byte" e creare giochi complessi con risorse minime. Molti dei giochi classici e degli esempi nella libreria finale di questo manuale ne fanno un uso intensivo.

Capitolo 9 – Padroneggiare il Playfield

Finora abbiamo trattato il Playfield come una struttura statica, un bassorilievo digitale definito una volta per tutte all'inizio del gioco. Ma i mondi più interessanti sono quelli che cambiano, che reagiscono alle azioni del giocatore, che si muovono. In questo capitolo, sbloccheremo il vero potenziale del Playfield.

9.1 – Leggere il Mondo: Il Comando pfread

Come fa il nostro programma a sapere se un punto specifico del Playfield è un "mattone" (X) o uno spazio vuoto (.)? Possiamo interrogare la memoria della console usando il comando *pfread*. *pfread(x, y)* è una domanda che restituisce "vero" se il blocco del Playfield alla coordinata (x, y) è acceso, e "falso" se è spento. La coordinata **x** va da **0 a 31** (da sinistra a destra). La coordinata **y** va da **0 a 10** (dall'alto in basso).
Questo comando è fondamentale per creare una logica di collisione con i muri veramente solida, come abbiamo anticipato con la tecnica "Salva e Ripristina". Invece di far "rimbalzare" il giocatore, possiamo impedirgli del tutto di entrare in un muro.

pfread può essere utilizzato solo all'interno di espressioni *if* come valore vero o falso, similmente a *collision* o *joy0fire*. Inoltre per le coordinate tra parentesi vanno utilizzate solo variabili "da sole" oppure numeri interi. La stessa regola si applica a *pfpixel* che vedremo tra poco.

Convertire Coordinate Sprite in Coordinate Playfield

Per usare *pfread*( ) con il tuo player, devi "tradurre" le coordinate dello sprite in coordinate del Playfield. La formula base per trovare il blocco *(pf_x, pf_y)* su cui si trova l'angolo in alto a sinistra di player0(o player1) è:

pf_x = (player0x - 16) / 4

pf_y = player0y / 8

Tuttavia è come sempre necessario poi lavorare su eventuali offset dipendenti dalle caratteristiche dello sprite. Padroneggiare questa conversione è il segreto per creare interazioni precise con lo scenario.

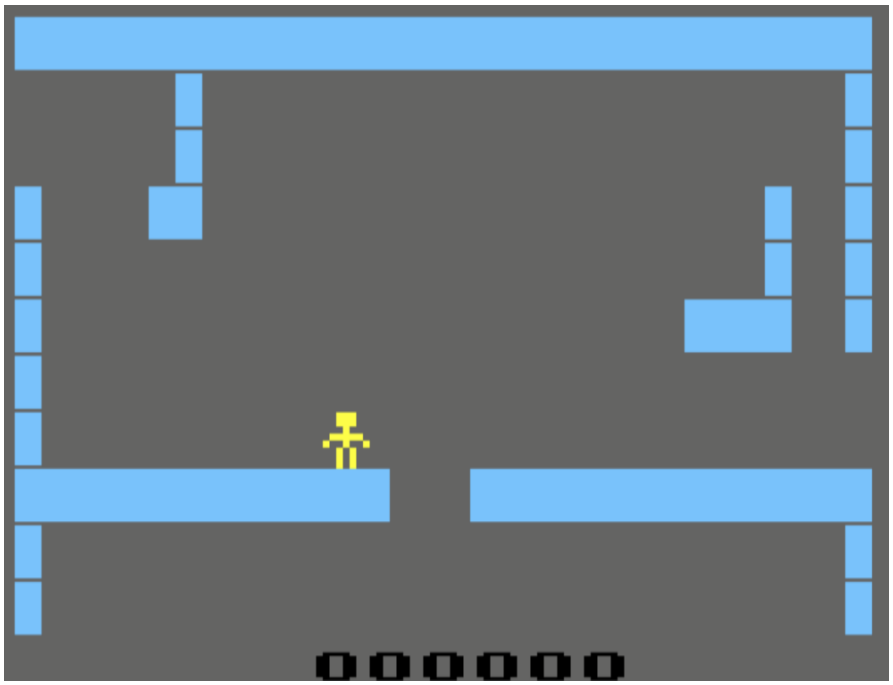
9.2 – Missione: Costruire e Distruggere con pfpixel

Il comando *pfread* ci permette di leggere, ma *pfpixel* ci permette di scrivere. Con questo comando, possiamo accendere (on) o spegnere (off) o invertire (flip) un singolo "mattone" del Playfield durante il gioco:

pfpixel x y on
pfpixel x y off
pfpixel x y flip

Questo apre le porte a un'infinità di meccaniche di gioco dinamiche: muri che possono essere distrutti, ponti che possono essere costruiti, porte che si aprono.
Creiamo un gioco in cui il giocatore deve far comparire un ponte (premendo fuoco) per attraversare un baratro. Ogni volta che si preme fuoco, compare un nuovo "blocco" su cui si può procedere.

```
  Rem Il Ponte Magico
  set romsize 2k

  dim player x = a
  dim player_y = b
  dim bricks = c
  dim floor = d
  dim retain = e

  rem --- Inizializzazione ---
  player_x = 20
  player_y = 64
  bricks = 5 ; Il giocatore ha 5 mattoni

  player0:
  %0010100
  %0010100
  %0010100
  %1001001
  %0111110
  %0001000
  %0011100
  %0011100
end

  playfield:
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  ......X.......................X
  ......X.......................X
  X....XX.....................X..X
  X..........................X..X
  X.......................XXXX..X
  X.............................
  X.............................
  XXXXXXXXXXXX.....XXXXXXXXXXXXXXXX
  X.............................X
  X.............................X
end

main_loop
  if joy0left && player_x > 20 then player_x = player_x - 1

  rem determina se il giocatore è sul ponte
  t = (player_x + 5 - 16) / 4 ; +5 perchè il piede destro finisce al pixel 5

  if joy0right && player x < 130 && pfread(t,8) then player x = player x + 1

  rem Resetta il flag di ritenzione del fire
  if !joy0fire then retain = 0

  rem Se il giocatore preme fuoco
  if joy0fire && bricks > 0 && retain = 0 then goto build_bridge
  goto continue_game

build_bridge
    t = 17 - bricks
    pfpixel t 8 on ; ...costruisce un pezzo di ponte!
    bricks = bricks - 1
    retain = 1

continue_game
  player0x = player_x
  player0y = player y

  COLUP0 = $1E ; Colore player0
  COLUBK = $04 ; Sfondo grigio
  COLUPF = $9E ; Colore playfield

  drawscreen
  goto main_loop
```



Il Problema del "Fuoco a Ripetizione" e la Tecnica del "Retain"

Se avessimo usato solo *if joy0fire && bricks>0 then ….* senza *&& retain = 0* avresti visto comparire tutti e 5 i blocchi del ponte in un istante: la pressione del fire viene infatti letta a ogni frame, 60 volte al secondo!

Come possiamo dire al programma di "eseguire l'azione una sola volta, anche se il pulsante rimane premuto, e attendere che venga rilasciato e premuto di nuovo"? La soluzione è proprio una tecnica classica chiamata "ritenzione dell'input" (o input debouncing).

L'idea è semplice: usiamo una variabile *retain* (un "flag") per ricordare se l'azione legata al pulsante è già stata eseguita in seguito all'ultima pressione.

Nel *main loop*, controlliamo se il pulsante di fuoco non è premuto *(if !joy0fire ...)* e in tal caso azzeriamo la nostra variabile di ritenzione. Questo "resetta" la possibilità di eseguire di nuovo l'azione.

Quando verifichiamo se il giocatore vuole sparare, aggiungiamo una condizione: la nostra variabile di ritenzione deve essere a zero.

Infine, appena l'azione viene eseguita, impostiamo subito la variabile di ritenzione a 1. Questo "blocca" la possibilità di eseguire di nuovo l'azione nei frame successivi, anche se il giocatore continua a tenere premuto il pulsante.

Disegno Veloce: *pfhline* e *pfvline*

Disegnare un labirinto complesso con *pfpixel* sarebbe lento e dispendioso. Per questo esistono due comandi più potenti che disegnano intere linee di mattoni in un colpo solo.

pfhline x y l on ; disegna una linea orizzontale di l blocchi (off = cancella)

pfvline x y a on ; disegna una linea verticale di a blocchi (off = cancella)

Sono perfetti per generare labirinti o strutture complesse all'inizio di un livello, risparmiando preziosa memoria ROM!

9.3 – Mondi in Movimento: Lo Scrolling con pfscroll

E se volessimo che fosse il mondo a muoversi, invece del giocatore? Il comando pfscroll ci permette di far scorrere l'intero Playfield in una delle quattro direzioni:

pfscroll up
pfscroll down
pfscroll left
pfscroll right

Questa è la tecnica fondamentale per tutti i giochi a scorrimento, come gli sparatutto verticali o i giochi di corse automobilistiche. Tuttavia **richiede di riscrivere, dopo lo scroll, la zona del playfield che è rimasta vuota**.

Per creare ad esempio l'illusione di una strada che si muove verso il basso, potremmo scrivere questo codice:

```
main
  ; ... logica di gioco ...

  rem Fa scorrere la strada
  pfscroll down

  drawscreen
  goto main
```

Lo Scrolling è Costoso!

Lo scorrimento orizzontale (*pfscroll left/right*) è una delle operazioni più "pesanti" in Batari Basic e consuma moltissimi cicli CPU. Usalo con cautela e assicurati che la logica del tuo *main loop* sia molto snella per evitare lo screen roll (ne parleremo nel capitolo 13). Lo scorrimento verticale (*pfscroll up/down*) è molto più leggero.

9.4 – Movimento su Griglia per Labirinti Giocabili

Abbiamo imparato a disegnare labirinti, ma come ci si muove al loro interno senza "incastrarsi" nei muri? I giochi classici come Pac-Man risolvono questo problema con una tecnica chiamata **Movimento su Griglia**.

L'idea è che il giocatore può cambiare direzione solo in punti specifici ("incroci") di una griglia invisibile. Il programma "ricorda" la direzione desiderata dal giocatore, ma la applica solo quando raggiunge un incrocio valido, ovvero delle coordinate ritenute valide per cambiare direzione.

Questo codice permette di cambiare direzione orizzontale solo su certe righe, e verticale solo su certe colonne.

```
  rem Movimento su Griglia
  set romsize 2k

  dim allow_h = a
  dim allow_v = b
  dim current_dir = c
  dim desired_dir = d

  player0x=77 : player0y=48 ; posizione iniziale
```

```
  playfield:
  ................................
  ................................
  ................................
  ................................
  ..XXXXXXXXXXXXXXXXXXXXXXXXXXXX..
  ..X..........................X..
  ..XXXXXXXXXXXXX..XXXXXXXXXXXXX..
  ..X..........................X..
  ..XXXXXXXXXXXXXXXXXXXXXXXXXXXX..
  ................................
  ................................
end

 player0:
  %00111100
  %01111110
  %11111111
  %11100000
  %11111111
  %01011110
  %00111100
end

main loop
  gosub check_grid_position
  gosub handle grid input
  gosub apply_grid_movement

  COLUP0 = $1E ; Giallo per il giocatore
  COLUPF = $9E ; Azzurro per lo sfondo

  drawscreen
  goto main_loop

check_grid_position
   allow h = 0 : allow v = 0
   rem Puoi cambiare direzione orizzontale solo sulle righe 48 o 64
   if player0y = 48 || player0y = 64 then allow h = 1
   rem Puoi cambiare direzione verticale solo sulla colonna 77
   if player0x = 77 then allow_v = 1
   return

handle_grid_input
  if joy0up then desired dir = 1
  if joy0down then desired_dir = 2
  if joy0left then desired_dir = 3
  if joy0right then desired_dir = 4
  return

apply_grid_movement
  rem Se sei su un incrocio orizzontale, puoi cambiare direzione orizzontale
  if allow_h && desired_dir = 3 then current_dir = desired_dir
  if allow_h && desired_dir = 4 then current_dir = desired_dir
  rem Se sei su un incrocio verticale, puoi cambiare direzione verticale
  if allow v && desired dir = 1 then current dir = desired dir
  if allow_v && desired_dir = 2 then current_dir = desired_dir

  rem Muovi sempre nella direzione corrente
  if current_dir = 1 then player0y = player0y - 1
  if current_dir = 2 then player0y = player0y + 1
  if current dir = 3 then player0x = player0x - 1
  if current_dir = 4 then player0x = player0x + 1

  rem Limita i movimenti all'interno della griglia
  if player0y < 48 then player0y = 48
  if player0y > 64 then player0y = 64
  if player0x < 29 then player0x = 29
  if player0x > 125 then player0x = 125

  return
```

Questa tecnica, combinata con *pfpixel* e *pfread*, ti permette di creare labirinti complessi e perfettamente giocabili, dando al giocatore la sensazione di un movimento fluido e controllato all'interno di passaggi stretti.

Il Ciclo for...next – Potente ma Pericoloso

Batari Basic offre il classico ciclo *for...next* per eseguire un blocco di codice un numero specifico di volte. La sua sintassi è familiare a chiunque abbia mai usato un linguaggio BASIC.

```
for variabile = valore1 to valore2 step valore3
rem ... blocco di codice da ripetere ...
next
```

- variabile è una qualsiasi variabile (a...z).

- valore1, valore2, valore3 possono essere numeri o altre variabili.

- step è opzionale; se omesso, il passo è +1. Puoi usare uno step negativo per contare all'indietro.

```
rem Conta da 1 a 10
for x = 1 to 10
rem ... fai qualcosa ...
next


rem Conta all'indietro da 50 a 0
for y = 50 to 0 step -1
rem ... fai qualcosa ...
next
```

Sebbene sembri comodo, l'uso di *for...next* in Batari Basic è generalmente sconsigliato per la logica di gioco principale, a causa di due comportamenti molto particolari e potenzialmente pericolosi.

In Batari Basic, **il comando *next* non si preoccupa di a quale for appartiene**. Quando incontra *next*, il compilatore semplicemente cerca all'indietro il primo comando *for* che trova nel codice e salta lì, indipendentemente dal flusso del programma. Questo può portare a risultati disastrosi.

Un ciclo *for...next* monopolizza la CPU finché non è completato. Se inserisci un *for...next* lungo nel tuo main_loop, l'intero gioco si "congelerà". Non potrai leggere l'input, muovere altri oggetti o riprodurre suoni fino alla fine del ciclo.

Usa i cicli *for...next* con molta cautela, principalmente per compiti di inizializzazione che avvengono una sola volta (come disegnare un labirinto all'inizio di un livello). Evitali quasi sempre all'interno del tuo main_loop.

Capitolo 10 – Mondi a Schermate Multiple e Kernel Potenziati

Finora le nostre avventure si sono svolte in un'unica stanza. Il nostro eroe è nato, si è mosso e ha interagito all'interno dei confini di un singolo schermo. Ma le grandi avventure richiedono grandi mondi da esplorare: castelli con decine di stanze, giungle intricate, labirinti sotterranei. In questo capitolo, impareremo due dei trucchi più affascinanti della programmazione Atari: come creare l'illusione di un mondo vasto e interconnesso.

10.1 – Creare Mondi a Schermate Multiple

Come abbiamo visto, l'Atari 2600 fatica persino a disegnare un singolo schermo. Come poteva allora un gioco come *Adventure* o *Pitfall!* avere centinaia di stanze diverse?
La risposta è semplice e geniale: **non le mostrava tutte insieme**. La console teneva in memoria solo la stanza in cui si trovava il giocatore. Quando l'eroe usciva da un lato dello schermo, il programma cancellava tutto, caricava la grafica della stanza successiva e riposizionava l'eroe sul lato opposto. Per il giocatore, l'effetto era quello di un passaggio fluido da un'area all'altra di un mondo enorme.

Per tenere traccia di dove si trova il giocatore, usiamo una singola variabile, il nostro "GPS" interno, che chiameremo *room*. Ogni valore di *room* corrisponderà a una stanza diversa sulla nostra mappa immaginaria.

La logica di gioco cambierà radicalmente. Invece di disegnare sempre lo stesso playfield, useremo una struttura a "centralino" per decidere quale stanza disegnare, basandoci sul valore di room.

10.2 – Le Due Stanze

Vediamo come funziona la transizione in un piccolo mondo a due stanze. Il nostro obiettivo è creare un passaggio segreto: quando il giocatore supera un certo limite a destra, il valore di room cambia e il giocatore viene riposizionato sul lato opposto del nuovo schermo.

```
  rem Le Due Stanze
  set romsize 2k

  dim player x = a
  dim player y = b
  dim room = c

  room = 1 ; si parte dalla stanza 1
  player x = 50

  player0:
  %0010100
  %0010100
  %0010100
  %1001001
  %0111110
  %0001000
  %0011100
  %0011100
end


main_loop
  gosub handle_movement
  gosub handle_room_transition
  gosub draw_current_room
  goto main_loop

  rem ========= SUBROUTINES =========
```

```
handle movement
  if joy0left then player_x = player_x - 1
  if joy0right then player x = player x + 1

  if player_x > 136 then player_x = 136
  if player_x < 16 then player_x = 16
  return

handle room transition
  rem Se esci a destra dalla stanza 1, vai alla stanza 2
  if room = 1 && player_x > 134 then room = 2 : player_x = 18

  rem Se esci a sinistra dalla stanza 2, torna alla stanza 1
  if room = 2 && player x < 18 then room = 1 : player x = 134
  return

draw_current_room
  player0x = player_x
  player0y = 64
  COLUP0 = $08

  rem Centralino grafico
  if room = 1 then gosub draw_room1
  if room = 2 then gosub draw room2

  drawscreen
  return

draw_room1
  COLUBK = $86 ; Sfondo blu
  COLUPF = $1E
  playfield:
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  X.....X.........................X
  X.....X.........................X
  X.....X.....................X..X
  X...........................X..X
  X.........................XXXX..X
  X..............................
  X..............................
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  ................................
  ................................
end
  return

draw_room2
  COLUBK = $36 ; Sfondo rosso
  COLUPF = $1E
  playfield:
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  X.....X.........................X
  X.....X.........................X
  X...XXX.........................X
  X......................X.......X
  X......................XXXX....X
  ...............................X
  ...............................X
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  ................................
  ................................
end
  return
```

Premi **F5**. Ti troverai in una stanza blu con un’apertura a destra. Muoviti verso destra. Non appena il tuo personaggio uscirà dallo schermo, **BAM!** Ti ritroverai in una nuova stanza rossa, entrando dal lato sinistro. Hai appena creato un mondo più grande del singolo schermo!

10.3 – I Segreti del Kernel: Grafica Multicolore

Finora, i nostri eroi e i nostri mondi hanno avuto un aspetto un po' monocromatico. *player0* è di un colore, *player1* di un altro, e il playfield di un altro ancora. Ma come facevano giochi come *Pitfall!* ad avere un protagonista con la maglietta di un colore e i pantaloni di un altro?
In batari basic la risposta non si trova in un comando, ma in un accordo speciale che possiamo fare con il "motore" del nostro gioco: il **Kernel**. Possiamo chiedere al kernel di usare delle versioni modificate di se stesso, sbloccando nuove capacità grafiche. Ma attenzione, tutto questo ha un prezzo! Queste "versioni" si attivano con la direttiva *set kernel_options*.

player0 e *player1* possono avere un solo colore, definito da *COLUP0, COLUP1*. Possiamo però chiedere al kernel di usare il tempo che normalmente dedicherebbe al *missile0 e missile1* per cambiare il colore di *player0* e *player1* riga per riga mentre li disegna.

Basta aggiungere:

set kernel_options playercolors player1color

all'inizio del programma. Ora, oltre ai blocchi *player0: player1:*, puoi definire due nuovi blocchi *player0color: player1color:* dove specifichi un colore esadecimale per ogni riga degli sprite. **Il Prezzo da Pagare:** Il kernel non ha più tempo per gestire *missile0 e missile1*. **Perdi completamente l'uso di *missile0* e *missile1***!

Il nostro Playfield può avere un solo colore, definito da *COLUPF*. Possiamo chiedere al kernel di cambiare il colore del Playfield per ogni riga orizzontale che disegna. Basta Aggiungere:

set kernel_options pfcolors

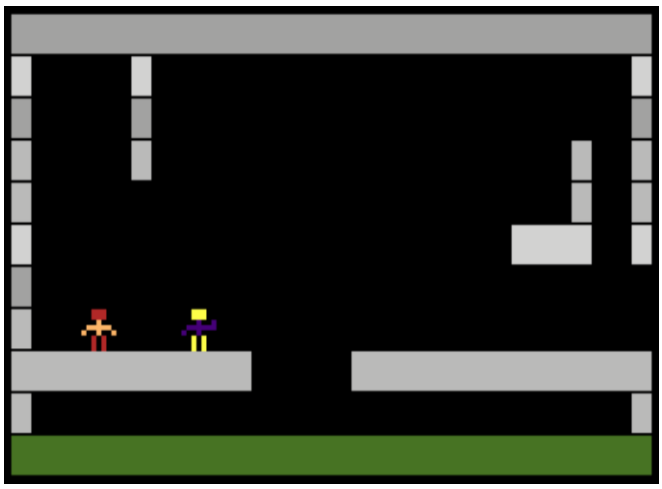
e ora puoi definire un blocco *pfcolors*: dove elenchi una sequenza di colori, uno per ogni riga del tuo playfield. Questo trucco non ha "prezzi da pagare".

Ecco un codice di esempio che utilizza *player0*, *player1* e *playfield* multicolore.

```
rem Player0,Player1,playfield multicolor
set kernel_options playercolors player1colors pfcolors
set romsize 2k


COLUBK = $00 ; Sfondo nero


player0:
%0010100 ; ultima riga player 0
%0010100
%0010100
%1001001
%0111110
%0001000
%0011100
%0011100 ; prima riga player 0
```

```
end

  player1: ; ultima riga player1
  %0010100
  %0010100
  %0010100
  %1001000
  %0111111
  %0001001
  %0011100
  %0011100 ; prima riga player1
end

  player0color:
  $44 ; colore ultima riga player0
  $44
  $44
  $3E
  $3E
  $3E
  $44
  $44 ; colore prima riga player 0
end

  player1color:
  $1E ; colore ultima riga player1
  $1E
  $1E
  $60
  $60
  $60
  $1E
  $1E ; colore prima riga player1
end

  playfield:
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  ; riga 0 playfield
  X.....X.....................X
  X.....X.....................X
  X.....X.....................X..X
  X.........................X..X
  X.......................XXXX..X
  X.............................
```

```
  X.............................
  XXXXXXXXXXXX.....XXXXXXXXXXXXXXX
  X.............................X
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  ; riga 10 playfield
end

  pfcolors:
  $08 ; colore riga 0 playfield
  $0C ; colore riga 1 playfield
  $08 ; ...
  $0A
  $0A
  $0C
  $08
  $0A
  $0A
  $0A
  $D4 ; colore riga 10 playfield
end

  player0x = 30
  player0y = 64

  player1x = 50
  player1y = 64

main_loop
  COLUPF = $08 ; colore riga 0 playfield
  drawscreen
  goto main_loop
```



Le Opzioni del Kernel

Le *kernel_options* sono potenti, ma non tutte le combinazioni sono possibili. Il Kernel Standard supporta solo alcune configurazioni specifiche. Se provi a usare una combinazione non valida, il compilatore ti darà un errore.

Ecco le combinazioni valide più comuni per ottenere personaggi e sfondi multicolore:

set kernel_options player1colors

Effetto: Solo *player1* è multicolore.

Costo: Perdi completamente l'uso di *missile1*.

set kernel_options playercolors player1colors

Effetto: Sia *player0* che *player1* sono multicolore.

Costo: Perdi completamente l'uso di entrambi i *missili, missile0 e missile1*.

set kernel_options pfcolors

Effetto: Il playfield è multicolore.

Costo: nullo!

set kernel_options playercolors player1colors pfcolors

Effetto: *player0, player1* e il playfield sono multicolore.

Costo: Perdi completamente l'uso di entrambi i missili, *missile0* e *missile1*.

Variare le Altezze delle righe del playfield: *pfheights*

Di default, ogni riga del Playfield ha la stessa altezza. Ma con l'opzione del kernel ***pfheights***, possiamo specificare un'altezza diversa per ogni riga. Questo permette di creare sfondi con un aspetto molto più organico e meno "a blocchi".

Come Funziona:

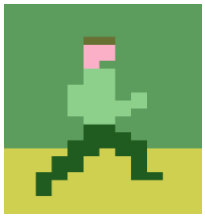
Aggiungi l'opzione all'inizio del programma: *set kernel_options pfheights*

Definisci un blocco *pfheights*: dove specifichi l'altezza in pixel di ogni riga, di cui la prima deve essere per forza di 8 pixel e la somma totale deve essere 88, ad esempio:

```
pfheights:
8
8
15
1
8
8
8
8
8
8
```

```
  8

end
```

Questa opzione può essere usata in combinazione con *pfcolors* ma in tal caso devi definire entrambi una volta sola al di fuori del *main_loop*.



L'Eroe Arcobaleno: Prendi un personaggio che hai disegnato e prova a dargli colori diversi per la testa, il corpo e le gambe usando playercolors e il blocco *player0color*: . Ricorda che non potrai più usare *missile0*!

L'Orizzonte Digitale: Prendi uno degli sfondi che hai creato e trasformalo in un paesaggio con un cielo, un orizzonte e un terreno usando pfcolors. Prova a creare una gradazione di blu per il cielo per dare un senso di profondità.

Capitolo 11 – L'Illusione della Fluidità: Movimento Sub-Pixel e Fisica

Finora, i nostri personaggi si sono mossi di un pixel intero alla volta, creando un movimento un po' "scattoso". Ma nei migliori classici dell'Atari 2600 i personaggi sembrano muoversi e scorrere in modo fluido. Come facevano?
La risposta è una delle "magie" più importanti del game design: simulavano i numeri decimali. In questo capitolo, impareremo questa tecnica, chiamata **aritmetica a virgola fissa**, che trasformerà i movimenti dei nostri eroi in animazioni fluide e professionali.

11.1 – Precisione decimale

Immagina di voler muovere un oggetto molto lentamente. La soluzione è separare la posizione "reale" del personaggio (memorizzata con precisione decimale) dalla sua posizione "visibile" sullo schermo (che può essere solo intera). L'**aritmetica a virgola fissa** (*fixed point*) ci permette di fare esattamente questo.

11.2 – L'Aritmetica a Virgola Fissa (8.8) in Batari Basic

Per usare questa tecnica, dobbiamo includere una libreria speciale all'inizio del nostro programma:

```
include fixed_point_math.asm
```

Questo ci permette di definire **variabili chiamate 8.8**, perchè usano due variabili byte (a..z): una per contenere la parte intera del numero decimale (8 bit) e una per la parte frazionaria (8 bit). Le dichiariamo con dim usando una sintassi speciale:

```
dim v1 = a.b  ; 'a' è la parte intera, 'b' la parte frazionaria
```

Le variabili 8.8 si usano tipicamente per le coordinate degli oggetti. Ad esempio:

```
dim hero_x_fixed = a.b
```

In questo esempio: la variabile *a* conterrà il numero di pixel (coordinata x) interi (da 0 a 255); la variabile *b* conterrà la frazione di pixel (dove *b=128* rappresenta 0.5, *b=64* rappresenta 0.25, e così via. Ovvero, se *b* è maggiore di 0, la parte frazionaria del numero è 128 diviso *b*).

Facciamo degli esempi di assegnazione e somma:

```
  hero_x_fixed = 60.5 ; in automatico, batari basic assegna a=60 e b=128

  hero_x_fixed = hero_x_fixed + 0.5 ; hero_x_fixed diventa 61 (in automatico, batari basic fa il
calcolo e assegna a=61 b=0)

  hero_x_fixed = hero_x_fixed + 0.25 ; hero_x_fixed diventa 61.25 (in automatico, batari basic fa
 il calcolo e assegna a=61 b=64)
```

*S*e ora scriviamo:

```
  x = hero_x_fixed ; x, essendo una variabile da 0 a 255, "prende" solo la parte intera,ovvero 61
```

Allo stesso modo:

```
  player0x = hero_x_fixed ; player0x diventa 61 (la parte frazionaria è "invisibile")
```

Vediamo ora un esempio completo:

```
  dim hero_x_fixed = a.b
  hero_x_fixed = 80
main_loop
  if joy0right then hero_x_fixed = hero_x_fixed + 0.5
  player0x = hero_x_fixed ; Assegna SOLO la parte intera a player0x
  drawscreen
  goto main_loop
```

Cosa succede frame per frame?

Frame 1: *hero_x_fixed* parte da 80.0, *player0x* è 80.

Frame 2: Supponiamo il giocatore muova il joystick a "destra". *hero_x_fixed* diventa 80.5. La parte intera è ancora 80, quindi *player0x* rimane 80. Lo sprite non si è mosso!

Frame 3: Ancora destra. *hero_x_fixed* diventa 81.0. La parte intera ora è 81, quindi *player0x* diventa 81. Lo sprite si è mosso di un pixel!

Lo sprite si è mosso di un pixel solo dopo due frame. L'effetto visivo è un movimento più liscio, alla metà della velocità.

11.3 – Platform Hero – Fisica Realistica con Salto e Gravità

Il movimento sub-pixel non serve solo per lo spostamento orizzontale. È la chiave per creare una fisica di base realistica, come il salto e la gravità. Un personaggio che salta non si muove a velocità costante: accelera verso l'alto, rallenta, si ferma per un istante e poi riaccelera verso il basso.
In questo esempio, creeremo un motore fisico completo per un personaggio platform, implementando movimento orizzontale, salto e gravità. Useremo:

- **Posizioni a Virgola Fissa (*hero_x_fixed, hero_y_fixed*):** Terranno traccia della posizione "reale" del personaggio.
- **Velocità Verticale (*y_velocity*):** Una variabile a virgola fissa che rappresenta la velocità di salita/discesa.
- **Gravità:** Un piccolo valore che aggiungeremo a *y_velocity* ad ogni frame, tirando costantemente il personaggio verso il basso.
- **Salto:** Un forte valore negativo che assegneremo a *y_velocity* quando il giocatore salta, spingendolo verso l'alto.
- **Controllo a Terra (on_ground):** Un flag per capire se permettere al giocatore di saltare o no (solo se sta toccando il suolo).

```
  rem Platform Hero - Fisica con Virgola Fissa
  set romsize 2k
  include fixed_point_math.asm

  dim hero_x_fixed = a.b      ; Posizione X a virgola fissa
  dim hero_y_fixed = c.d      ; Posizione Y a virgola fissa
  dim y_velocity = e.f        ; Velocità Y a virgola fissa
```

```
  dim on_ground = g           ; 1 = a terra, 0 = in aria

  rem --- Inizializzazione ---
  hero_x_fixed = 80
  hero_y_fixed = 64
  y_velocity = 0.0

  player0:
  %0010100
  %0010100
  %0010100
  %1001001
  %0111110
  %0001000
  %0011100
  %0011100
end

  playfield:
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  X.....X.......................X
  X.....X.......................X
  X.....X.......................X
  X.............................X
  X.............................X
  X.............................X
  X.............................X
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  ................................
  ................................
end

  COLUP0 = $1E
  COLUBK = $86
  COLUPF = $1E

main_loop
  gosub handle_input
  gosub handle_physics
  gosub move_hero

  drawscreen
  goto main_loop
```

```
  rem ========= SUBROUTINES =========

handle_input
  rem Movimento orizzontale
  if joy0left && hero_x_fixed > 16 then hero_x_fixed = hero_x_fixed - 1
  if joy0right && hero_x_fixed < 134 then hero_x_fixed = hero_x_fixed + 1

  rem Salto: si può saltare solo se si è a terra
  if joy0fire && on_ground then y_velocity = -4 ; impulso iniziale verso l'alto
  return

handle_physics
  rem 1. Applica la gravità (tira sempre verso il basso)
  y_velocity = y_velocity + 0.3

  rem 2. Applica il movimento verticale (velocità e posizione)
  hero_y_fixed = hero_y_fixed + y_velocity

  rem 3. Controlla se il giocatore è atterrato
  if hero_y_fixed < 64 then on_ground = 0 ; In aria
  if hero y fixed >= 64 then hero y fixed = 64 : y velocity = 0.0 : on ground = 1 ; Impedisce di
sprofondare, ferma la caduta
  return

move_hero
  player0x = hero_x_fixed ; Assegna SOLO la parte intera alla posizione visibile
  player0y = hero_y_fixed
  return
```

Come Funziona: Premi **F5**. Usa il joystick (tasti freccia) per muovere il personaggio a destra e a sinistra. Premi il pulsante di fuoco (barra spaziatrice): l'eroe eseguirà un salto perfetto, con una curva parabolica realistica, e atterrerà dolcemente. Noterai che non potrai saltare di nuovo finché non avrà toccato terra. Hai appena creato un motore fisico da *platform*.

Capitolo 12 – Il Cruscotto del Gioco: Punteggi, Vite e Barre di Stato

Ogni grande avventura ha bisogno di un cruscotto. Come fa un esploratore a sapere quanti tesori ha raccolto, quante vite gli sono rimaste o quanta energia ha il suo scudo? Queste informazioni vitali vengono mostrate attraverso l'**HUD** (*Heads-Up Display*), l'interfaccia grafica che si sovrappone all'azione.
In questo capitolo, impareremo a costruire il cruscotto del nostro gioco, usando gli strumenti che Batari Basic ci mette a disposizione: il classico punteggio a sei cifre e le versatili barre di stato.

12.1 – Il Punteggio Tradizionale: Il Comando score

Il modo più classico per mostrare i punti è usare la variabile speciale **score**. È una variabile fissa a **6 cifre**, visualizzata permanentemente nella parte inferiore dello schermo. A differenza delle normali variabili (0-255), score può gestire numeri da **0 a 999999**.
Funziona con uno speciale formato numerico chiamato **BCD** (Binary-Coded Decimal). Per ora, ti basta sapere che puoi **solo** aggiungere o sottrarre **solo** valori interi usando l'aritmetica standard di somma e sottrazione (es. *score = score + 10*).

Attenzione! *score* non è una variabile normale e non la puoi utilizzare negli *if* o per fare calcoli dentro a espressioni aritmetiche o usarla con altre variabili! Ad esempio:

score = a ; **NON FUNZIONA!**

score = score + b ; **NON FUNZIONA!**

if score > 100 then gosub vittoria ; **NON FUNZIONA!**

In sintesi, dovrai usare altre variabili per "tener conto" di eventuali bonus o eventi nel tuo gioco che vorresti far dipendere dallo *score*.

Le tecniche per controllare *score* sono abbastanza complesse e richiedono molti calcoli e tempo di CPU. Parleremo nell'appendice C di una delle possibili tecniche.

Per far apparire il punteggio, devi fare due cose nel tuo main loop:

1. **Impostare un colore:** Usa il registro scorecolor. Se non lo imposti, lo score sarà nero e invisibile.

2. **Assegnare un valore:** Dai un valore iniziale alla variabile score.

```
  rem Attivare lo Score
  set romsize 2k

  score = 0

main loop
  scorecolor = $1E ; Colore giallo per il punteggio

  if joy0fire then score = score + 100

  drawscreen
  goto main_loop
```

Premi **F5**. Vedrai "000000" in fondo allo schermo. Premi fuoco e lo vedrai aumentare. Hai appena creato il tuo primo contatore di punti!

12.2 – Oltre i Numeri: Le Barre di Stato pfscore

A volte i numeri non bastano. Potresti voler mostrare le vite come icone o l'energia come una barra che si svuota. Per questo, Batari Basic offre le **pfscore bars**. Sono due aree grafiche a 8 blocchi, situate a sinistra e a destra dello score, che puoi controllare.
Per attivarle, devi usare il comando *const pfscore = 1* all'inizio del programma. Ora hai accesso a tre nuove variabili:

- **pfscorecolor**: Imposta il colore di entrambe le barre.
- **pfscore1**: Controlla la barra di sinistra
- **pfscore2**: Controlla la barra di destra

Ogni barra è composta da 8 bit. Impostando un bit a 1 accendi il blocco corrispondente. Il modo più intuitivo per controllarle è usare i numeri binari (%).

12.3 – Barra della Vita e Contatore Vite

Vediamo come usare le barre di stato in un gioco per creare un HUD completo.
Useremo la barra sinistra (*pfscore1*) per mostrare fino a 3 vite come puntini.
Useremo la barra destra (*pfscore2*) come una barra della salute che si svuota.

```
  rem HUD Completo
  set romsize 2k

  const pfscore = 1 ; attiva barre laterali

  dim lives_bar = a
  dim health_bar = b

  dim retainleft = c
  dim retainright = d

  rem --- Inizializzazione ---
  lives_bar =  %00010101   ; 3 vite (i bit 0, 2, 4 sono accesi)
  health_bar = %11111111   ; Salute piena (tutti i bit accesi)

main_loop
  rem --- Logica di Gioco (Simulata) ---
  rem Se premi sinistra, perdi salute
  if !joy0left then retainleft = 0
  if joy0left && retainleft = 0 then health_bar = health_bar / 2 : retainleft = 1

  rem Se premi destra, perdi una vita
  if !joy0right then retainright = 0
  if joy0right && retainright = 0 then lives_bar = lives_bar / 4 : retainright = 1

  rem Se premi fire ripristina valori iniziali
  if joy0fire then  lives_bar = %00010101 : health_bar = %11111111 : score = score + 1

  rem --- Disegno HUD ---
  scorecolor = $1E ; colore giallo
  pfscorecolor = $86 ; colore blu
  pfscore1 = lives_bar
  pfscore2 = health_bar

  drawscreen
  goto main_loop
```

Premi **F5**. Ora hai un HUD funzionale! Premi sinistra (joy0left) per vedere la barra della salute diminuire e destra (joy0right) per vedere le vite sparire una a una.

La Magia della Divisione Binaria

Ti sei chiesto perché usiamo / 2 e / 4? È un trucco geniale che sfrutta la matematica binaria.

Barra della Salute (/ 2): Dividere un numero per 2, in binario, è equivalente a "spostare" tutti i suoi bit di una posizione verso destra (shift a destra). Il bit più a destra "cade" e viene perso, e a sinistra entra uno 0. Applicato alla nostra barra %11111111, questo la svuota gradualmente, un pezzetto alla volta: %01111111, %00111111, e così via (attenzione: per pfscore2 il bit più a destra è quello visualizzato più a sinistra!).

Barra delle Vite (/ 4): Dividere per 4 equivale a fare uno shift a destra di due posizioni. Nel nostro schema %00010101, dove le vite sono i bit 0, 2 e 4, questo "salto" di due posizioni spegne un puntino alla volta in modo netto.

12.4 – Un'Alternativa alle Vite: Il Sistema di Danni

Finora abbiamo parlato di "vite": perdi una vita, il gioco si resetta. Ma molti giochi, specialmente quelli di corse o di combattimento, usano un sistema diverso: i **punti danno**. Invece di avere un numero discreto di tentativi, il giocatore ha un'unica "barra della vita" (o un contatore invisibile) che si riempie o svuota a ogni colpo. Il gioco termina solo quando i danni raggiungono una soglia. Questa tecnica crea un feeling di gioco diverso, più orientato alla sopravvivenza. Realizzarla è molto semplice. Invece di un contatore di vite che scende, usiamo un contatore di danni che sale.

```
dim damage counter = c
max_damage = 60

; ... nel main loop, dopo drawscreen ...
if collision(player0, enemy) then damage counter = damage counter + 1


rem Controlla se il gioco è finito
if damage_counter >= max_damage then goto game_over
```

L'Effetto Sfumato (scorefade)

Vuoi dare al tuo punteggio un aspetto più professionale e tridimensionale, tipico di molti giochi classici? Batari Basic offre un effetto speciale chiamato *scorefade*. Se attivato, aggiunge una sottile ombreggiatura ai numeri dello score, dando loro un senso di profondità. Basta aggiungere *const scorefade = 1* all'inizio del tuo programma.

```
rem Esempio di Score Sfumato

set romsize 2k

const scorefade = 1
```

```
main_loop
  scorecolor = $9C ; Viola
  score = 123456
  drawscreen
  goto main_loop
```

Confrontando il risultato con e senza scorefade, noterai che i numeri appaiono meno "piatti" e più integrati con lo sfondo.

L'Effetto Arcobaleno: un Classico Atari

La funzione *scorefade* ha un "effetto collaterale" molto amato dai programmatori dell'epoca. Se, invece di usare un colore fisso, incrementi costantemente la variabile *scorecolor* ad ogni frame, otterrai il classico effetto arcobaleno, un trucco iconico dell'era Atari!

```
  set romsize 2k
  const scorefade = 1
  dim color_timer = a


main_loop
  color_timer = color_timer + 1
  scorecolor = color_timer
  score = 123456
  drawscreen
  goto main_loop
```

Questo codice farà ciclare i colori del punteggio attraverso l'intera tavolozza, creando un effetto psichedelico e vibrante, spesso usato nelle schermate dei titoli o per celebrare un record.

Attenzione! Se nel tuo gioco hai attivato le barre di stato con *const pfscore = 1*, **l'effetto scorefade non è disponibile**. Devi scegliere quale delle due funzionalità grafiche usare per il tuo HUD, non possono coesistere.

Capitolo 13 – Ottimizzazione e Debug Avanzato: La Caccia ai "Bug"

Cosa succede quando qualcosa nel nostro programma va storto? Quando lo schermo inizia a tremare, un colore lampeggia in modo strano, o il gioco semplicemente si blocca? Benvenuto nel mondo del **debugging**, l'arte investigativa di trovare e correggere gli errori, o "bug", nel nostro programma.
Inoltre, dobbiamo assicurarci che il nostro gioco non solo funzioni, ma funzioni *bene*. Deve essere veloce e reattivo. Questo processo si chiama **ottimizzazione**. In questo capitolo, indosseremo il cappello da detective e impareremo a rispettare la legge più importante di tutte: la corsa contro il raggio.

13.1 – Il Nemico Numero Uno: Lo "Screen Roll"

Il problema più temuto da ogni programmatore Atari è il famigerato **"screen roll"** (scorrimento dello schermo) o **"jitter"** (tremolio). Lo schermo trema, sfarfalla o inizia a scorrere verticalmente senza sosta.
Questo è quasi sempre un **problema di tempo**. Significa che la logica nel tuo main loop (il codice tra drawscreen e goto main) sta impiegando più del suo "budget" di cicli CPU (circa 2700). La CPU è così impegnata a fare calcoli che arriva in ritardo all'appuntamento con il raggio del televisore, e la sincronizzazione dell'immagine salta.
La soluzione è **ottimizzare**!

13.2 – Rimanere nel Budget: Strategie di Ottimizzazione

Cosa fare se il gioco è troppo lento? Non devi per forza eliminare delle funzionalità. Spesso basta distribuire il carico di lavoro in modo più intelligente.

Strategia 1: Sposta il Lavoro nel VBlank

Questa è la tecnica di ottimizzazione più importante. Chiediti: *"Questa operazione deve essere eseguita per forza in questo esatto frame?"*

- Il movimento del giocatore? **Sì**, deve essere istantaneo.
- Decidere la prossima mossa di un nemico che si trova dall'altra parte dello schermo? **Forse no.**

Tutta la logica che non è "urgente" può essere spostata in una sezione speciale alla fine del tuo programma, chiamata **vblank**. Il **Vertical Blank** è quel breve momento in cui il raggio del televisore è "spento" e sta tornando in cima allo schermo. Durante questo intervallo, la CPU ha a disposizione circa 1675 cicli extra per eseguire logica "pesante" senza interferire con il disegno.

```
  ; ...

  ; ... il tuo main loop finisce qui ...
  goto main_loop

vblank
  rem Sposta qui la logica "pesante" e non urgente
  gosub update_enemy_ai_logic
  return
```

Questo libera immediatamente cicli preziosi nel tuo main loop, dove la velocità è più critica.

Strategia 2: L'Alternanza dei Frame

Invece di aggiornare l'IA di tutti i nemici 60 volte al secondo, perché non aggiornarne metà in un frame e l'altra metà nel frame successivo? L'occhio umano non noterà quasi mai la differenza, ma il carico di lavoro sulla CPU per ogni singolo frame sarà dimezzato!

```
dim frame_counter = h

; ... nel main loop ...
frame_counter = frame_counter + 1

rem Aggiorna il Nemico 1 solo nei frame "pari"
if frame_counter{0} then gosub update_enemy1_ai

rem Aggiorna il Nemico 2 solo nei frame "dispari"
if !frame_counter{0} then gosub update_enemy2_ai
```

Qui usiamo il bit 0 di *frame_counter* per distinguere tra frame pari (i numeri pari hanno sempre il bit 0 = 0) e dispari (i numeri dispari hanno sempre il bit 0 = 1), alternando l'aggiornamento dei nemici.

13.3 – La Lente d'Ingrandimento del Detective: Il Debug Visivo

A volte il gioco non trema, ma si comporta in modo strano. Uno sprite scompare, un colore è sbagliato, un punteggio non si aggiorna. Come facciamo a sapere cosa c'è dentro una variabile *mentre* il gioco sta girando?
Non abbiamo un debugger sofisticato, ma possiamo usare la grafica stessa per "visualizzare" i dati!

Trucco 1: Visualizzare un Valore con i Colori Questo è il trucco più semplice e veloce. Se vuoi controllare il valore della variabile *a*, assegnalo temporaneamente a un registro colore.

```
rem DEBUG: Mostra il valore di 'a' come colore di sfondo
COLUBK = a
```

Ora, mentre giochi, il colore dello sfondo cambierà in base al valore di *a*. Se il colore cambia come ti aspetti, la variabile sta funzionando. Se rimane fisso o cambia in modo strano, hai trovato un problema!

Trucco 2: Usare lo score come Monitor Se il tuo gioco usa il punteggio, puoi usarlo come "monitor" di debug temporaneo, per capire se il codice "passa" nel punto giusto.

```
rem DEBUG: Mostra il valore 999 nello score
score = 999
```

Trucco 3: Usare i Suoni A volte vuoi sapere se una certa parte del codice viene eseguita. Associa un suono a quell'evento!

```
; DEBUG: Suono di collisione

if collision(player0, enemy) then gosub handle_hit: AUDV0 = 10 : AUDC0 = 12 : AUDF0 = 10
```

Se senti il "beep" nel momento sbagliato (o non lo senti affatto), sai che la tua logica di collisione ha un problema.

Capitolo 14 – E Adesso?

Congratulazioni!
Hai completato il tuo viaggio guidato attraverso le basi di Batari Basic. Ma questo non è un punto di arrivo. È un punto di partenza. Hai appena aperto una porta su un universo vasto e affascinante. Questo manuale ti ha fornito le fondamenta, ma il mondo della programmazione Atari è molto più grande. Abbiamo volutamente mantenuto il nostro viaggio focalizzato sul Kernel Standard e su giochi contenuti in 4K di ROM, per darti le basi più solide possibili.

Ora, cosa c'è oltre? Cosa puoi fare adesso con le tue nuove abilità? Questo capitolo è la tua mappa per le prossime avventure nel mondo della programmazione retro.

14.1 - Diventa un Maestro di Batari Basic
Ecco un assaggio delle tecniche più avanzate che i maestri del codice usano per trasformare un buon gioco in un capolavoro. Per approfondirle, le risorse della community sono il posto migliore dove cercare.

- **Funzioni (function) e Macro (macro):** Oltre a *gosub*, esistono modi ancora più potenti per organizzare il codice. Le **funzioni** sono come subroutine che possono "restituire" un risultato, mentre le **macro** ti permettono di creare i tuoi comandi personalizzati, rendendo il codice incredibilmente pulito.

- **Matematica a 16 bit e BCD (**, //, dec):** Per calcoli più complessi, come gestire numeri più grandi di 255 o lavorare con il punteggio in modo sicuro, esistono operatori speciali per la matematica a 16 bit e per l'aritmetica BCD (dec).

- **Spremere Ogni Bit: Le Variabili Nybble:** I maestri sanno come stipare due contatori (con valori da 0 a 15) in un singolo byte, trattandolo come due metà da 4 bit (un "nybble"). È un'arte di ottimizzazione estrema.

- **Variabili 4.4**: Oltre alle variabili 8.8 (8 bit per la parte intera, 8 per quella frazionaria), la libreria *fixed_point_math.asm* ci offre un altro strumento utile per situazioni specifiche: le variabili a virgola fissa 4.4, che permettono di risparmiare preziosa memoria RAM.

14.2 - Guardare "Sotto il Cofano": Piegare l'Hardware
Le vere magie avvengono quando si inizia a "parlare" all'hardware in modi che i creatori originali non avevano previsto.

- **Espandere la Memoria: il Bankswitching:** il bankswitching è il trucco che permette di creare cartucce da 8K, 16K o addirittura 32K. Un codice ben strutturato può "delegare" il lavoro a banchi di memoria ROM diversi, creando un'architettura a staffetta per gestire giochi enormi.

- **Grafica da Maestri: Kernel Alternativi e Chip Speciali:** Il nostro manuale si è basato sul Kernel Standard, ma è solo l'inizio. Esistono kernel specializzati come il **Multisprite**

Kernel (per mostrare più di 2 sprite sulla stessa riga) o il potentissimo **DPC+ Kernel** (per grafica ad alta risoluzione e fino a 10 sprite multicolore).

- **Il Potere Assoluto: Assembly:** Per il controllo totale, i programmatori scendono al "livello del metallo" e scrivono in **Assembly 6507**, il linguaggio nativo della CPU. Con Batari Basic, puoi inserire piccole porzioni di codice assembly per ottenere la massima velocità o creare effetti grafici impossibili altrimenti.
- **Trucchi da Hacker: Rilevare Controller Extra:** La community ha scoperto segreti hardware incredibili, come la possibilità di rilevare se un controller **Sega Genesis** è collegato e usare i suoi pulsanti extra per aggiungere più azioni al tuo gioco!

14.3 - Unisciti alla Community: Non Sei Solo!

Una delle cose più belle dello sviluppare per Atari 2600 oggi è che non sei solo. Esiste una community globale incredibilmente attiva, amichevole e pronta ad aiutare altri appassionati. Le risposte a tutte le tue future domande si trovano qui.

- **AtariAge Forums:** È il cuore pulsante della community. Nelle sezioni dedicate allo sviluppo per 2600 e al Batari Basic, puoi fare domande, condividere i tuoi progressi, trovare tutorial e scoprire i nuovi, incredibili giochi realizzati da altri.
- **La Pagina di Batari Basic di Random Terrain:** Questa è l'enciclopedia definitiva, piena di documentazione, esempi e guide approfondite.
- **GitHub:** Il codice sorgente del compilatore Batari Basic e di molti giochi homebrew è disponibile qui. Leggere il codice di altri sviluppatori è uno dei modi migliori per imparare.

Non aver paura di fare domande. La community apprezza i nuovi arrivati ed è sempre felice di condividere la propria conoscenza ed esperienza.

14.4 - Giocare sulla TV di Casa: L'Esperienza Autentica

Testare i giochi sull'emulatore Stella è fantastico, ma niente batte la sensazione di giocare alla propria creazione su un vero televisore. Oggi, questo è più facile che mai.

- **L'Atari 2600+:** Di recente, Atari ha rilasciato una nuova console moderna compatibile con le vecchie cartucce, ma con un'uscita HDMI per qualsiasi TV.
- **La Cartuccia Magica (Flash Cart):** Con una "cartuccia magica" moderna come la famosa **Harmony Cartridge**, puoi giocare i tuoi giochi su una console originale o moderna. Il processo è semplice:
    1. Compili il tuo gioco con Batari Basic (premendo F5) per creare il file .bin.
    2. Copi questo file .bin su una scheda SD.

3. Inserisci la scheda SD nella Harmony Cartridge.
4. Inserisci la Harmony Cartridge nella tua console.
5. Accendi e giochi la tua creazione sul grande schermo!

La memoria "sporca" nel vero hardware

Fai attenzione che a differenza degli emulatori, **in un vero Atari 2600 il valore iniziale delle variabili non è 0** ma è sconosciuto. Per essere sicuro che i tuoi programmi funzioneranno anche su vero hardware, azzera manualmente il valore di tutte le variabili questo codice:

```
a = 0 : b = 0 : c = 0 : d = 0 : e = 0 : f = 0 : g = 0 : h = 0 : i = 0
j = 0 : k = 0 : l = 0 : m = 0 : n = 0 : o = 0 : p = 0 : q = 0 : r = 0
s = 0 : t = 0 : u = 0 : v = 0 : w = 0 : x = 0 : y = 0 : z = 0
```

14.5 – Programmi da provare e appendici

Nella prossima parte di questo manuale troverai diversi listati di giochi che utilizzano tutte le tecniche che hai visto. Ora sei perfettamente in grado di comprenderne il funzionamento! Inoltre alla fine del manuale troverai appendici con riassunti, ulteriori informazioni e tecniche avanzate.

Buon divertimento!

Parte 3: Giochi da provare

Cartucce originali dell'Atari 2600 (Immagine: mitchelaneous.com)

Ogni programma che segue è un gioco da provare, una "cartuccia digitale" che mette in pratica le tecniche che hai imparato.

Considera ogni listato come un progetto da "smontare". Leggi l'introduzione, analizza le tecniche utilizzate e vai a ripassare i capitoli corrispondenti se hai qualche dubbio. Poi, tuffati nel codice. Non aver paura di modificare, sperimentare e "rompere" le cose! Questo è il modo migliore per trasformare la teoria in vera abilità.

I listati completi di questi giochi si trovano subito dopo le scheda di presentazione.

Ricordatevi di aggiungere una riga vuota alla fine del programma se fate copia-incolla dei listati!

1. Simple Pong (1 vs. CPU)

- **Il Gioco:** La versione più pura del classico che ha dato inizio a tutto. Controlli la racchetta destra, il computer controlla quella sinistra. Un punto di partenza eccellente per capire la fisica di base e l'IA.

- **Tecniche Principali Utilizzate:**

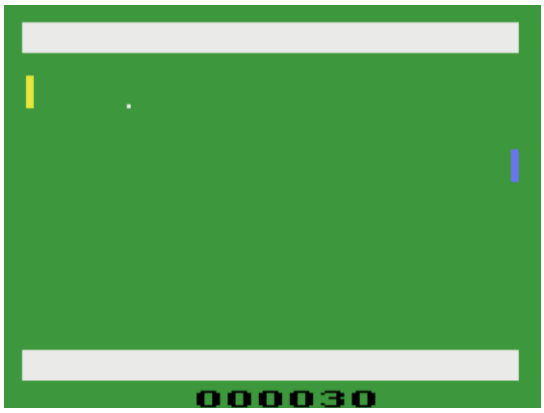
  - **Fisica di Base:** Il movimento e il rimbalzo della palla sono gestiti invertendo le variabili di velocità (vedi **Capitolo 8**).

  - **IA Semplice:** La racchetta del computer segue ciecamente la palla *(player1y = bally)*, una forma basilare di intelligenza artificiale (vedi **Capitolo 3**).

  - **Struttura a Subroutine:** La logica per i punti e le collisioni è organizzata in subroutine pulite (vedi **Capitolo 5**).

  - **Clamping:** Le racchette sono bloccate all'interno del campo da gioco (vedi **Capitolo 3**).

  –



2. Advanced Pong (Pong con Ostacoli – 1 vs 1)

- **Il Gioco:** Una variante di Pong più dinamica. Il giocatore può muoversi in 4 direzioni e il campo contiene ostacoli statici che influenzano la traiettoria della palla, aggiungendo imprevedibilità.

- **Tecniche Principali Utilizzate:**

  - **Movimento a 4 Direzioni:** Il giocatore non è più vincolato all'asse verticale, aggiungendo strategia (vedi **Capitolo 3**).

  - **Interazione con il Playfield:** La palla ora può collidere con gli ostacoli del playfield (*collision(ball, playfield)* -vedi **Capitolo 4**).

  - **Fisica di Rimbalzo Avanzata:** Quando la palla colpisce un ostacolo, vengono invertite *entrambe* le componenti della sua velocità.

3. Dynamic Pong (Racchetta che si Accorcia – 1 vs CPU)

- **Il Gioco:** Questa versione introduce una meccanica di difficoltà crescente. Ogni volta che il giocatore perde una vita, la sua racchetta diventa più corta.

- **Tecniche Principali Utilizzate:**

  - **Grafica Dinamica dello Sprite:** L'altezza dello sprite *player0* non è fissa. Il programma usa subroutine per ridefinire la grafica dello sprite in tempo reale, in base alle vite (vedi **Capitolo 6**).

  - **Indicatore di Stato Visivo:** La dimensione della racchetta stessa funge da HUD, comunicando istantaneamente al giocatore quante vite gli rimangono (vedi **Capitolo 14**).

4. Killer Acorn (Ghianda Assassina)

- **Il Gioco:** Un classico sparatutto in arena fissa. Sei una ghianda, un nemico ti insegue. Spara per guadagnare punti, evita di essere toccato per non perdere vite.

- **Tecniche Principali Utilizzate:**

  - **Animazione a Frame Multipli:** Il nemico è animato con quattro sprite diversi, gestiti da timer software (vedi **Capitolo 6**).

  - **IA di Inseguimento:** Il nemico si muove attivamente verso il giocatore.

  - **Gestione Proiettile Singolo:** Un trucco comune per gestire lo sparo, usando la posizione del missile come un flag (vedi **Capitolo 5**).

  - **Uso di rand:** La casualità viene usata per rendere imprevedibile la riapparizione del nemico (vedi **Capitolo 12** e **Appendice C**).

5. Simple Soccer (1 vs 1)

- **Il Gioco:** Un semplice gioco di calcio/hockey per due giocatori, che introduce la grafica multicolore e la gestione del possesso palla.

  Basato su “Fifa 1977” di https://8bitworkshop.com/

- **Tecniche Principali Utilizzate:**

  – **Opzione del Kernel *player1colors*:** Questa opzione viene usata per dare a player1 un aspetto multicolore, sacrificando l’uso di missile1 (vedi **Capitolo 10**).

  – **Gestione del Possesso Palla:** Una variabile (p) funge da flag per determinare quale giocatore controlla la palla.

  – **Logica di Salvataggio Posizione:** Per gestire le collisioni con i muri in modo robusto (vedi **Capitolo 4**).

6. The Watch (Il Guardiano del Castello)

- **Il Gioco:** Un gioco complesso che combina difesa, costruzione e combattimento. Il giocatore deve ricostruire un muro raccogliendo mattoni e difendersi da un mostro.

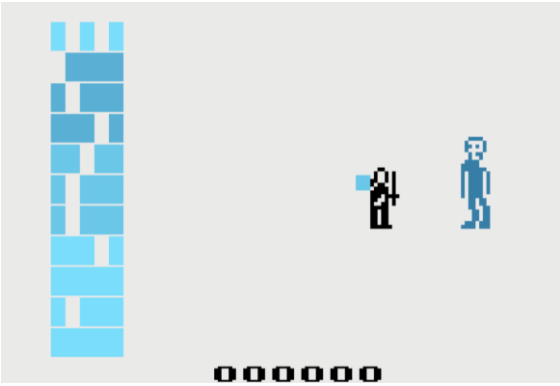
- **Tecniche Principali Utilizzate:**

  – **Opzioni del Kernel Avanzate:** *pfcolors* e *pfheights* sono usate per creare uno sfondo ricco di dettagli (vedi **Capitolo 10**).

  – **IA con Difficoltà Crescente:** La velocità e la resistenza del nemico aumentano con il progredire dei livelli.

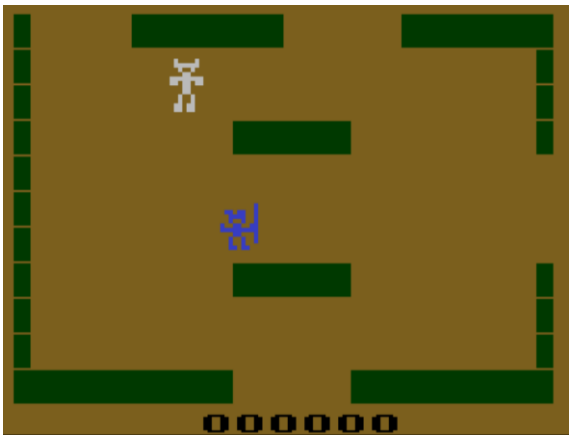
  – **Interazione Dinamica con il Playfield:** Il giocatore modifica il playfield in tempo reale con *pfpixel* e *pfread* (vedi **Capitolo 9**).

  – **Aritmetica BCD per lo Score:** Il punteggio viene gestito in modo sicuro (vedi **Capitolo 12** e **Appendice C**).



7. Minotaur (schermate multiple)

- **Il Gioco:** Un'avventura a schermate multiple in cui il giocatore esplora un labirinto, raccoglie oggetti e combatte un boss.
- **Tecniche Principali Utilizzate:**
  - **Esplorazione a Schermate Multiple:** Il cuore del gioco, gestito dalla variabile *room* (vedi **Capitolo 10**).
  - **Sistema di Inventario:** I bit-flag (*haslance, hasshield*) tengono traccia degli oggetti raccolti (vedi **Capitolo 8**).
  - **IA di Pattugliamento:** Il Minotauro si muove lungo un percorso predefinito.



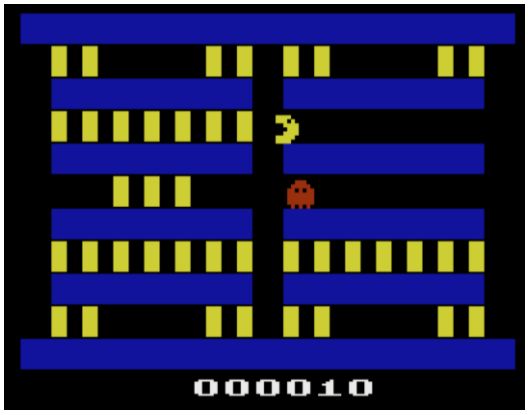
8. Snappy

- **Il Gioco:** Un platform basato sul tempismo, eccellente esempio di come usare una Macchina a Stati per gestire logiche di gioco complesse.

  Basato su: Snappy - an Atari 2600 game by Sebastian Mihai (2012)
- **Tecniche Principali Utilizzate:**
  - **Macchina a Stati Complessa:** La variabile gamestate è il cervello del gioco, controllando ogni singola fase dell'azione (vedi **Capitolo 7**).
  - **Animazione basata su Timer:** L'oscillazione della liana è un esempio di animazione del playfield (vedi **Capitoli 6** e **9**).
  - **Generazione del Seme Casuale (randseed):** Tecnica avanzata per rendere casuale la posizione di partenza (vedi **Capitolo 12** e **Appendice C**).

9. Gnamm (movimenti su griglia)

- **Il Gioco:** Una dimostrazione di come ricreare meccaniche complesse su un hardware limitato.
- **Tecniche Principali Utilizzate:**
  - **Uso Intensivo dei Bit-Flag:** La variabile *b* è un "pannello di controllo" che gestisce quasi tutta la logica del gioco (vedi **Capitolo 8**).
  - **Movimento su Griglia:** Il movimento è vincolato a "incroci" specifici (vedi **Capitolo 9**).
  - **Opzione del Kernel *pfcolors*:** Usata per dare al labirinto il suo aspetto bicolore (vedi **Capitolo 10**).



10. Highway Racer (corse in Autostrada con aritmetica a virgola fissa)

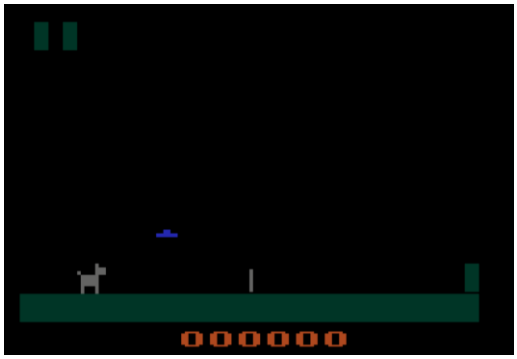
- **Il Gioco:** Dimostra come usare l'aritmetica a virgola fissa per creare un senso di velocità e movimento.
- **Tecniche Principali Utilizzate:**
  - **Aritmetica a Virgola Fissa:** Il cuore del gioco. Le variabili 8.8 sono usate per un'accelerazione e uno scorrimento fluidi (vedi **Capitolo 11**).
  - **Scrolling Verticale del Playfield:** Il comando *pfscroll down* crea l'illusione della strada che si muove (vedi **Capitolo 9**).
  - **Sistema di Danni:** Invece di vite, usa un contatore di "danni" (vedi **Capitolo 14**).

11. Disc Dog (uso di rand)

- **Il Gioco:** Un gioco unico e originale ispirato allo sport del "disc dog". Controlli un cane che deve prendere al volo un frisbee (player1) lanciato da un lanciatore fuori campo. Il

cane può correre e saltare. Se il frisbee cade a terra, perdi una vita. Il gioco è a tempo e diventa progressivamente più difficile.

- **Tecniche Principali Utilizzate:**
  - **IA dell'Oggetto:** Il frisbee non si muove in linea retta, ma segue una traiettoria parabolica simulata, cambiando velocità e altezza in modo casuale, rendendo ogni lancio imprevedibile (vedi **Capitolo 8** e **Appendice C** per *rand*).
  - **Animazione Dinamica:** La grafica del cane (player0) cambia in base alla direzione e all'azione (corsa vs. fermo) (vedi **Capitolo 6**).
  - **Interazione Complessa:** Il gioco gestisce più stati: il cane che corre, che salta, che prende il disco e che lo riporta al padrone (il blocco verticale sui lati).
  - **Manipolazione del Playfield:** Il punteggio delle vite e il timer non usano le variabili *score* o *pfscore*, ma vengono disegnati "manualmente" sullo sfondo usando *pfpixel* (vedi **Capitolo 9**).



Simple Pong

```
rem ****************************************************************
rem * Simple Pong   (1 Giocatore vs. CPU)                          *
rem *                                                              *
rem * DESCRIZIONE DEL GIOCO:                                       *
rem * Questa è una versione classica del gioco Pong per un         *
rem * giocatore. L'utente controlla la racchetta destra (player0)  *
rem * muovendola verticalmente per respingere una palla (ball).    *
rem * La racchetta sinistra (player1) è controllata dal computer   *
rem * e segue semplicemente la posizione verticale della palla.    *
rem * L'obiettivo è segnare punti facendo passare la palla oltre la *
rem * racchetta del computer. Si perde una vita se la palla supera  *
rem * la propria racchetta.                                        *
rem *                                                              *
rem * TECNICHE DI PROGRAMMAZIONE UTILIZZATE:                       *
rem * - Fisica di Base: Il movimento della palla è gestito da due  *
rem *   variabili di velocità (`ballxvelocity`, `ballyvelocity`).  *
rem *   Quando la palla colpisce un muro o una racchetta, la sua   *
rem *   velocità viene invertita (`velocita = 0 - velocita`) per   *
rem *   simulare un rimbalzo.                                      *
rem * - Intelligenza Artificiale (IA) Semplice: La racchetta del   *
rem *   computer non ha una vera logica, ma si limita a "inseguire" *
rem *   la palla. La sua coordinata Y (`player1y`) viene           *
rem *   semplicemente impostata uguale a quella della palla (`bally`)*
rem *   ad ogni frame, rendendola imbattibile a meno che la palla  *
rem *   non venga "spinta" via velocemente dopo una collisione.    *
rem * - Gestione delle Collisioni e Subroutine: Il comando         *
rem *   `collision()` viene usato per rilevare i contatti. La      *
rem *   logica di gestione degli eventi (punto segnato, vita persa, *
rem *   collisione) è organizzata in subroutine (`gosub…return`),  *
rem *   mantenendo il `main_loop` pulito e leggibile.             *
rem * - Clamping: Vengono usati dei controlli `if` per "bloccare"  *
rem *   (clamping) la posizione delle racchette, impedendo loro di *
rem *   uscire dai limiti superiore e inferiore del campo da gioco. *
rem ****************************************************************

rem --- Direttive del Compilatore ---
set romsize 4k

rem --- Sezione Definizioni Variabili (Alias) ---
rem Crea un alias per la velocità orizzontale della palla.
dim ballxvelocity = a
```

```
 rem Crea un alias per la velocità verticale della palla.
 dim ballyvelocity = b
 rem 'q' è un flag per gestire il primo avvio del gioco.
 q=0

 rem --- Impostazioni Iniziali Grafica ---
 rem Imposta il colore dello sfondo (verde).
 COLUBK = 198
 rem Imposta il colore del playfield (bianco per i bordi).
 COLUPF = 14
 rem Definisce la grafica del campo da gioco (bordi superiore e inferiore).
 playfield:
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  ................................
  ................................
  ................................
  ................................
  ................................
  ................................
  ................................
  ................................
  ................................
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 rem Definisce la grafica della racchetta del giocatore (player0).
 player0:
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
end
 rem Definisce la grafica della racchetta del computer (player1).
 player1:
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
```

```
  %00011000
  %00011000
end

 rem --- Stato 1: Inizializzazione Partita / Round ---
startNewGame
 rem Questa etichetta viene chiamata per iniziare una nuova partita o un nuovo round.
 rem Imposta la posizione iniziale della racchetta del giocatore.
 player0x = 140
 player0y = 49
 rem Imposta la posizione iniziale della racchetta del computer.
 player1x = 15
 player1y = 49
 rem Imposta la posizione iniziale della palla al centro dello schermo.
 ballx = 80
 bally = 45
 rem Imposta la velocità iniziale della palla.
 ballxvelocity = 1
 ballyvelocity = 1
 rem Inizializza il contatore delle vite (non usato nel codice ma presente).
 l=3
 rem Imposta i colori delle racchette. Questi registri sono volatili.
 COLUP0 = 140
 COLUP1 = 28

 rem Controlla se non è la prima partita in assoluto.
 if q=1 then goto gameLoop
 rem Se è la prima partita, imposta il flag e vai alla schermata 'premi fuoco'.
 q=1
firstgame
 COLUP0 = 140
 COLUP1 = 28
 drawscreen
 if joy0fire then goto gameLoop
 goto firstgame

 rem --- Ciclo di Gioco Principale ---
gameLoop
 rem Reimposta i registri TIA volatili ad ogni frame.
 COLUP0 = 140
 COLUP1 = 28
 rem Disegna il fotogramma corrente.
 drawscreen
```

```
rem --- Gestione Input Giocatore ---
if joy0up then player0y = player0y-1
if joy0down then player0y = player0y+1
rem Clamping: impedisce alla racchetta del giocatore di uscire dallo schermo.
if player0y < 16 then player0y = 16
if player0y > 79 then player0y = 79

rem --- IA del Computer ---
rem La racchetta del computer segue perfettamente la posizione Y della palla.
player1y = bally
rem Clamping: impedisce anche alla racchetta del computer di uscire.
if player1y < 16 then player1y = 16
if player1y > 79 then player1y = 79

rem --- Fisica della Palla ---
rem Aggiorna la posizione della palla in base alla sua velocità.
ballx = ballx + ballxvelocity
bally = bally + ballyvelocity

rem Fa rimbalzare la palla sui bordi superiore e inferiore.
if bally < 9 then ballyvelocity = 0 - ballyvelocity
if bally > 77 then ballyvelocity = 0 - ballyvelocity

rem --- Gestione Collisioni ---
rem Se la palla colpisce la racchetta del giocatore, chiama la subroutine di collisione.
if collision(player0, ball) then gosub playercollision
rem Se la palla colpisce la racchetta del computer, chiama la sua subroutine.
if collision(player1, ball) then gosub computercollision

rem --- Gestione Punti e Vite ---
rem Se la palla supera la racchetta del giocatore, chiama la subroutine 'vita persa'.
if ballx > 150 then gosub playerlostlife
rem Se la palla supera la racchetta del computer, chiama la subroutine 'punto segnato'.
if ballx < 5 then gosub playerscores

rem --- Condizione di Fine Partita ---
rem Se le vite sono esaurite, vai alla schermata di Game Over.
if l < 1 then goto gameover

rem Ripete il ciclo di gioco.
goto gameLoop

rem --- Sezione delle Subroutine ---
```

```
playercollision
 rem Gestisce la collisione tra la palla e il giocatore.
 rem Inverte la velocità orizzontale della palla.
 ballxvelocity = 0 - ballxvelocity
 rem Sposta la palla di qualche pixel per evitare collisioni multiple nello stesso frame.
 ballx = ballx + ballxvelocity*5
 bally = bally + ballyvelocity*5
 return

computercollision
 rem Gestisce la collisione tra la palla e il computer.
 rem Inverte la velocità orizzontale della palla.
 ballxvelocity = 0 - ballxvelocity
 rem Sposta la palla per evitare collisioni multiple.
 ballx = ballx + ballxvelocity*5
 bally = bally + ballyvelocity*5
 return

playerscores
 rem Gestisce l'evento in cui il giocatore segna un punto.
 rem Resetta la posizione della palla al centro.
 ballx = 80
 bally = 45
 rem Resetta la velocità della palla.
 ballxvelocity = 1
 ballyvelocity = 1
 rem Incrementa lo score del giocatore.
 score = score + 10
 return

playerlostlife
 rem Gestisce l'evento in cui il giocatore perde una vita.
 rem Decrementa il contatore delle vite.
 l = l - 1
 rem Resetta la posizione della palla.
 ballx = 80
 bally = 45
 return

 rem --- Stato Finale: Game Over ---
gameover
 rem attende fire per ricominciare.
 rem fai sparire la palla
```

```
bally = 110
COLUP0 = 140
COLUP1 = 28
drawscreen
if joy0fire then goto startNewGame
goto gameover
```

Advanced Pong

```
rem ****************************************************************
rem * Advanced Pong  (1 Giocatore vs. CPU con Ostacoli)             *
rem *                                                              *
rem * DESCRIZIONE DEL GIOCO:                                       *
rem * Questa è una variante del classico Pong, con delle aggiunte  *
rem * per renderlo più dinamico. Il giocatore controlla liberamente *
rem * la sua racchetta (player0) in quattro direzioni all'interno  *
rem * del campo. Il campo da gioco contiene ostacoli statici       *
rem * (playfield) contro cui la palla può rimbalzare. La racchetta *
rem * del computer (player1) continua a seguire la palla solo      *
rem * verticalmente. L'obiettivo e la gestione di punti/vite sono  *
rem * identici alla versione base.                                 *
rem *                                                              *
rem * TECNICHE DI PROGRAMMAZIONE UTILIZZATE:                       *
rem * - Movimento a 4 Direzioni: A differenza del Pong classico, il *
rem *   giocatore può muovere la sua racchetta sia in orizzontale  *
rem *   che in verticale, aggiungendo un elemento strategico.      *
rem * - Interazione con il Playfield: Il campo da gioco non è più  *
rem *   solo un bordo, ma contiene ostacoli. Il gioco utilizza     *
rem *   `collision(ball, playfield)` per rilevare quando la palla  *
rem *   colpisce questi ostacoli.                                  *
rem * - Fisica di Rimbalzo Avanzata: Quando la palla colpisce un   *
rem *   ostacolo del playfield, vengono invertite entrambe le sue  *
rem *   componenti di velocità (`ballxvelocity` e `ballyvelocity`), *
rem *   simulando un rimbalzo più complesso rispetto a quello sui  *
rem *   bordi.                                                     *
rem * - Definizione di Sprite Complessi: Gli sprite per le         *
rem *   racchette sono più grandi e dettagliati rispetto a una     *
rem *   semplice linea, utilizzando più righe di dati binari.      *
rem * - Tutte le altre tecniche (IA, gestione collisioni, clamping, *
rem *   subroutine) sono simili alla versione base di Pong.        *
rem ****************************************************************

rem --- Direttive del Compilatore ---
set romsize 4k

rem --- Sezione Definizioni Variabili (Alias) ---
rem Crea un alias per la velocità orizzontale della palla.
dim ballxvelocity = a
rem Crea un alias per la velocità verticale della palla.
dim ballyvelocity = b
```

```
 rem 'q' è un flag per gestire il primo avvio del gioco.
 q=0

 rem --- Impostazioni Iniziali Grafica ---

 rem Imposta il colore dello sfondo.
 COLUBK = $81
 rem Imposta il colore del playfield (bordi e ostacoli).
 COLUPF = 68

 rem Definisce la grafica del campo da gioco con ostacoli interni.
 rem Nota: 'x' minuscolo viene trattato come 'X' maiuscolo.
 playfield:
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  X..............................
  X............X........X.........
  ................................
  ................................
  ................................
  ............................X...
  ......X.........................
  ................................X
  ...X......X........X...........X
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end

 rem Definisce la grafica della racchetta del giocatore (player0).
 player0:
  %11111111
  %00011000
  %00011000
  %00011000
  %00111100
  %00111100
  %00111100
  %00011000
  %00011000
  %00011000
  %11111111
end
 rem Definisce la grafica della racchetta del computer (player1).
 player1:
  %01111000
```

```
  %00011000
  %00011000
  %00011100
  %00011111
  %00011100
  %00011000
  %00011000
  %01111000
end

 rem --- Stato 1: Inizializzazione Partita / Round ---
startNewGame
 rem Questa etichetta viene chiamata per iniziare una nuova partita o un nuovo round.
 rem Imposta la posizione iniziale della racchetta del giocatore.
 player0x = 135
 player0y = 45
 rem Imposta la posizione iniziale della racchetta del computer.
 player1x = 20
 player1y = 45
 rem Imposta la posizione iniziale della palla al centro.
 ballx = 80
 bally = 45
 rem Imposta la velocità iniziale della palla.
 ballxvelocity = 1
 ballyvelocity = 1
 rem Inizializza il contatore delle vite.
 l=3
 rem Imposta i colori delle racchette. Questi registri sono volatili.
 COLUP0 = 140
 COLUP1 = 28

 rem Controlla se è la prima partita in assoluto.
 if q=1 then goto gameLoop
 rem Se è la prima partita, imposta il flag e vai alla schermata 'premi fuoco'.
 q=1
firstgame
 rem Imposta i colori e attende l'input del giocatore.
 COLUP0 = 140
 COLUP1 = 28
 drawscreen
 if joy0fire then goto gameLoop
 goto firstgame
```

```
 rem --- Ciclo di Gioco Principale ---
gameLoop
 rem Reimposta i registri TIA volatili ad ogni frame.
 COLUP0 = 140
 COLUP1 = 28
 rem Disegna il fotogramma corrente.
 drawscreen

 rem --- Gestione Input Giocatore (4 Direzioni) ---
 if joy0up then player0y = player0y-1
 if joy0down then player0y = player0y+1
 if joy0left then player0x = player0x-1
 if joy0right then player0x = player0x+1
 rem Clamping Orizzontale: impedisce alla racchetta di uscire lateralmente.
 if player0x < 20 then player0x = 20
 if player0x > 140 then player0x = 140
 rem Clamping Verticale: impedisce di uscire dall'alto e dal basso.
 if player0y < 16 then player0y = 16
 if player0y > 79 then player0y = 79

 rem --- IA del Computer ---
 rem L'IA è la stessa: la racchetta del computer segue la palla verticalmente.
 player1y = bally
 rem Clamping per la racchetta del computer.
 if player1y < 16 then player1y = 16
 if player1y > 79 then player1y = 79

 rem --- Fisica della Palla ---
 rem Aggiorna la posizione della palla.
 ballx = ballx + ballxvelocity
 bally = bally + ballyvelocity
 rem Fa rimbalzare la palla sui bordi superiore e inferiore del campo.
 if bally < 9 then ballyvelocity = 0 - ballyvelocity
 if bally > 77 then ballyvelocity = 0 - ballyvelocity

 rem --- Gestione Collisioni ---
 rem Se la palla colpisce la racchetta del giocatore…
 if collision(player0, ball) then gosub playercollision
 rem Se la palla colpisce la racchetta del computer…
 if collision(player1, ball) then gosub computercollision
 rem Se la palla colpisce gli ostacoli del playfield…
 if collision(ball,playfield) then gosub ballplayfieldcollision
```

```
 rem --- Gestione Punti e Vite ---
 if ballx > 154 then gosub playerlostlife
 if ballx < 5 then gosub playerscores

 rem --- Condizione di Fine Partita ---
 if l < 1 then goto gameover

 rem Ripete il ciclo di gioco.
 goto gameLoop

 rem --- Sezione delle Subroutine ---
playercollision
 rem Gestisce la collisione palla-giocatore.
 rem Inverte la velocità orizzontale.
 ballxvelocity = 0 - ballxvelocity
 rem Sposta la palla per evitare collisioni multiple.
 ballx = ballx + ballxvelocity*3
 bally = bally + ballyvelocity*3
 return

computercollision
 rem Gestisce la collisione palla-computer.
 rem Inverte la velocità orizzontale.
 ballxvelocity = 0 - ballxvelocity
 rem Sposta la palla per evitare collisioni multiple.
 ballx = ballx + ballxvelocity*3
 bally = bally + ballyvelocity*3
 return

playerscores
 rem Gestisce il punto segnato dal giocatore.
 rem Resetta palla e velocità.
 ballx = 80
 bally = 45
 ballxvelocity = 1
 ballyvelocity = 1
 rem Aumenta il punteggio.
 score = score + 10
 return

playerlostlife
 rem Gestisce la vita persa dal giocatore.
 rem Decrementa le vite.
```

```
 l = l - 1
 rem Resetta la palla.
 ballx = 80
 bally = 45
 return

ballplayfieldcollision
 rem Gestisce la collisione della palla con gli ostacoli.
 rem Inverte entrambe le componenti della velocità per un rimbalzo diagonale.
 ballxvelocity = 0 - ballxvelocity
 ballyvelocity = 0 - ballyvelocity
 rem Sposta la palla per evitare che rimanga "incastrata" nell'ostacolo.
 ballx = ballx + ballxvelocity*3
 bally = bally + ballyvelocity*3
 return

 rem --- Stato Finale: Game Over ---
gameover
 rem Mostra la schermata finale e attende l'input per ricominciare.
 COLUP0 = 140
 COLUP1 = 28
 bally = 110
 drawscreen
 if joy0fire then goto startNewGame
 goto gameover
```

Dynamic Pong

```
rem ******************************************************************
rem * Dynamic Pong  (Racchetta che si Accorcia)                      *
rem *                                                                *
rem * DESCRIZIONE DEL GIOCO:                                         *
rem * Questa variante di Pong introduce una meccanica di difficoltà  *
rem * crescente. Il giocatore controlla la racchetta destra (player0)*
rem * e affronta una racchetta controllata dal computer (player1).   *
rem * La caratteristica distintiva di questa versione è che la       *
rem * racchetta del giocatore si accorcia ogni volta che perde una   *
rem * vita, rendendo il gioco progressivamente più difficile.        *
rem *                                                                *
rem * TECNICHE DI PROGRAMMAZIONE UTILIZZATE:                         *
rem * - Grafica Dinamica dello Sprite: La dimensione (altezza) dello *
rem *   sprite del giocatore (player0) non è fissa. Il programma     *
rem *   utilizza una logica `if` nel `main_loop` per controllare il  *
rem *   numero di vite rimanenti (`l`). In base a questo valore,     *
rem *   viene chiamata una diversa subroutine (`pll3`, `pll2`, `pll1`)*
rem *   che ridefinisce la grafica di `player0:` con altezze diverse.*
rem * - Organizzazione del Codice con Subroutine: Le diverse        *
rem *   definizioni grafiche dello sprite sono incapsulate in        *
rem *   subroutine separate. Questo mantiene il `main_loop` pulito e *
rem *   rende chiara la logica di selezione dello sprite.            *
rem * - Indicatore di Stato Visivo: La dimensione della racchetta    *
rem *   funge da indicatore visivo immediato per il giocatore del    *
rem *   numero di vite rimaste, integrando l'HUD (Heads-Up Display)  *
rem *   direttamente nell'elemento di gioco principale.              *
rem * - Le altre tecniche (fisica della palla, IA, collisioni, etc.) *
rem *   sono identiche alla versione base del Pong.                  *
rem ******************************************************************

rem --- Direttive del Compilatore ---
set romsize 4k

rem --- Sezione Definizioni Variabili (Alias) ---
rem Crea un alias per la velocità orizzontale della palla.
dim ballxvelocity = a
rem Crea un alias per la velocità verticale della palla.
dim ballyvelocity = b
rem 'q' è un flag per gestire il primo avvio del gioco.
q=0
```

```
 rem --- Impostazioni Iniziali Grafica ---
 rem Imposta il colore dello sfondo (verde).
 COLUBK = 198
 rem Imposta il colore del playfield (bianco per i bordi).
 COLUPF = 14
 rem Definisce la grafica del campo da gioco (bordi superiore e inferiore).
 playfield:
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  ................................
  ................................
  ................................
  ................................
  ................................
  ................................
  ................................
  ................................
  ................................
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 rem Definisce la grafica della racchetta del computer (player1).
 player1:
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
end

 rem --- Stato 1: Inizializzazione Partita / Round ---
startNewGame
 rem Questa etichetta viene chiamata per iniziare una nuova partita o un nuovo round.
 rem Imposta la posizione iniziale della racchetta del giocatore.
 player0x = 140
 player0y = 48
 rem Imposta la posizione iniziale della racchetta del computer.
 player1x = 15
 player1y = 45
 rem Imposta la posizione iniziale della palla al centro.
 ballx = 80
 bally = 41
```

```
 rem Imposta la velocità iniziale della palla.
 ballxvelocity = 1
 ballyvelocity = 1
 rem Inizializza il contatore delle vite.
 l=3
 rem Imposta i colori delle racchette. Questi registri sono volatili.
 COLUP0 = 140
 COLUP1 = 28
 rem Imposta la grafica iniziale della racchetta del giocatore (grandezza massima).
 gosub pll3

 rem Controlla se è la prima partita in assoluto.
 if q=1 then goto gameLoop
 rem Se è la prima partita, imposta il flag e vai alla schermata 'premi fuoco'.
 q=1
firstgame
  COLUP0 = 140
  COLUP1 = 28
  drawscreen
  if joy0fire then goto gameLoop
  goto firstgame

 rem --- Ciclo di Gioco Principale ---
gameLoop
 rem Reimposta i registri TIA volatili ad ogni frame.
 COLUP0 = 140
 COLUP1 = 28

 rem Grafica Dinamica: seleziona la dimensione della racchetta in base alle vite.
 if l=3 then gosub pll3
 if l=2 then gosub pll2
 if l=1 then gosub pll1

 rem Disegna il fotogramma corrente.
 drawscreen

 rem --- Gestione Input Giocatore ---
 if joy0up then player0y = player0y-1
 if joy0down then player0y = player0y+1
 rem Clamping: impedisce alla racchetta del giocatore di uscire dallo schermo.
 if player0y < 16 then player0y = 16
 if player0y > 79 then player0y = 79
```

```
 rem --- IA del Computer ---
 rem La racchetta del computer segue la posizione Y della palla.
 player1y = bally
 rem Clamping per la racchetta del computer.
 if player1y < 16 then player1y = 16
 if player1y > 79 then player1y = 79

 rem --- Fisica della Palla ---
 rem Aggiorna la posizione della palla.
 ballx = ballx + ballxvelocity
 bally = bally + ballyvelocity
 rem Fa rimbalzare la palla sui bordi.
 if bally < 9 then ballyvelocity = 0 - ballyvelocity
 if bally > 77 then ballyvelocity = 0 - ballyvelocity

 rem --- Gestione Collisioni ---
 if collision(player0, ball) then gosub playercollision
 if collision(player1, ball) then gosub computercollision

 rem --- Gestione Punti e Vite ---
 if ballx > 150 then gosub playerlostlife
 if ballx < 5 then gosub playerscores

 rem --- Condizione di Fine Partita ---
 if l < 1 then goto gameover

 rem Ripete il ciclo di gioco.
 goto gameLoop

 rem --- Sezione delle Subroutine di Gioco ---
playercollision
 rem Gestisce la collisione palla-giocatore.
 ballxvelocity = 0 - ballxvelocity
 ballx = ballx + ballxvelocity*5
 bally = bally + ballyvelocity*5
 return
computercollision
 rem Gestisce la collisione palla-computer.
 ballxvelocity = 0 - ballxvelocity
 ballx = ballx + ballxvelocity*5
 bally = bally + ballyvelocity*5
 return
playerscores
```

```
 rem Gestisce il punto segnato dal giocatore.
 rem Resetta la palla e la sua velocità.
 ballx = 80
 bally = 45
 ballxvelocity = 1
 ballyvelocity = 1
 rem Incrementa lo score.
 score = score + 10
 return
playerlostlife
 rem Gestisce la vita persa.
 rem Decrementa il contatore delle vite.
 l = l - 1
 rem Resetta la palla.
 ballx = 80
 bally = 45
 return

 rem --- Stato Finale: Game Over ---
gameover
 rem Mostra l'ultimo stato e attende l'input per ricominciare.
 COLUP0 = 140
 COLUP1 = 28
 bally = 110
 drawscreen
 if joy0fire then goto startNewGame
 goto gameover

 rem --- Subroutine Grafiche: Dimensioni Racchetta Giocatore ---
pll3
 rem Racchetta a grandezza massima (16 pixel di altezza) quando le vite sono 3.
 player0:
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
```

```
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
end
  return
pll2
 rem Racchetta a grandezza media (8 pixel di altezza) quando le vite sono 2.
 player0:
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
  %00011000
end
  return
pll1
 rem Racchetta a grandezza minima (4 pixel di altezza) quando la vita è 1.
 player0:
  %00011000
  %00011000
  %00011000
  %00011000
end
  return
```

Killer Acorn

```
 rem ****************************************************************
 rem * Killer Acorn (Ghianda Assassina)                             *
 rem *                                                              *
 rem * DESCRIZIONE DEL GIOCO:                                       *
 rem * Il giocatore controlla una ghianda (player0) in un'arena     *
 rem * chiusa. Un nemico (player1) insegue costantemente il         *
 rem * giocatore. Il giocatore può sparare un proiettile (missile0) *
 rem * per colpire il nemico, guadagnando punti e facendolo        *
 rem * riapparire in una posizione casuale. Se il nemico tocca il  *
 rem * giocatore, si perde una vita e punti. Il gioco termina      *
 rem * quando le vite si esauriscono.                             *
 rem *                                                              *
 rem * TECNICHE DI PROGRAMMAZIONE UTILIZZATE:                      *
 rem * - Intelligenza Artificiale (IA) Semplice: Il nemico (player1)  *
 rem *   implementa una logica di inseguimento ("chasing logic")    *
 rem *   basata su semplici confronti tra le sue coordinate e quelle  *
 rem *   del giocatore, muovendosi di un pixel alla volta verso di  *
 rem *   lui.                                                       *
 rem * - Animazione a Frame Multipli: L'animazione del nemico è     *
 rem *   realizzata alternando quattro definizioni grafiche         *
 rem *   diverse (`player1:`). Un contatore (`v`) rallenta          *
 rem *   l'animazione per renderla visibile, mentre un secondo      *
 rem *   contatore (`w`) tiene traccia del frame corrente da        *
 rem *   visualizzare.                                              *
 rem * - Gestione Proiettile Singolo: Il gioco permette di avere un   *
 rem *   solo proiettile attivo alla volta. La sua posizione verticale*
 rem *   (`missile0y`) viene usata come flag: un valore alto (>240)   *
 rem *   indica che il proiettile è "inattivo" e se ne può sparare  *
 rem *   un altro.                                                  *
 rem * - Gestione Vite e Punteggio: Il gioco utilizza variabili     *
 rem *   standard per le vite (`a`) e lo score (`score`).           *
 rem * - Uso di `rand`: Il comando `rand` viene usato per far       *
 rem *   riapparire il nemico in una posizione orizzontale casuale  *
 rem *   dopo essere stato colpito.                                 *
 rem ****************************************************************

   set romsize 4k

 rem --- Stato 1: Schermata Titolo ---
opening
 rem Definisce la grafica statica del titolo.
```

```
 playfield:
 ............................
 ....X..X.XXX..X...X...XXX.XX....
 ....X.X...X...X...X...X...X.X...
 ....XX....X...X...X...XX..XX....
 ....X.X...X...X...X...X...X.X...
 ....X..X.XXX..XXX.XXX.XXX.X..X..
 ............................
 .......XXX.XXX.XXX.XXX.X.X......
 .......X.X.X...X.X.X.X.X.X......
 .......XXX.X...X.X.XX..XXX......
 .......X.X.XXX.XXX.X.X.X.X......
end
title
 rem Imposta i colori per il titolo.
 COLUBK = $60
 COLUPF = 212
 rem Disegna lo schermo e attende l'input del giocatore per iniziare.
 drawscreen
 if joy0fire || joy1fire then goto skiptitle
 goto title

skiptitle
 rem --- Inizializzazione Partita ---
 rem Imposta i colori di gioco: sfondo blu, playfield (muri) neri.
 COLUPF = 0
 COLUBK = 212
 rem Imposta la posizione iniziale del giocatore e del nemico.
 player0x = 50
 player0y = 70
 player1x = 20
 player1y = 8
 rem Imposta un valore iniziale per lo score e il suo colore.
 score = 103
 scorecolor = 1
 rem Imposta le proprietà del missile: altezza e posizione iniziale fuori schermo.
 missile0height=1
 missile0y=255
 rem Imposta la dimensione dello sprite del giocatore a larghezza doppia.
 NUSIZ0 = 2
 rem Inizializza il contatore delle vite.
 a = 3
 rem Definisce la grafica del playfield di gioco (l'arena).
```

```
 playfield:
 X.............................X
 X.............................X
 X.............................X
 X.............................X
 X.............................X
 X.............................X
 X.............................X
 X.............................X
 X.............................X
 X.............................X
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end

 rem --- Ciclo di Gioco Principale ---
main
 rem Incrementa il contatore 'v' per rallentare l'animazione del nemico.
 v = v + 1

 rem Logica di animazione: se 'v' raggiunge la soglia (7), cambia il frame
 rem dello sprite del nemico in base al valore di 'w'.
 if v = 7 && w = 0 then goto ax
 if v = 7 && w = 1 then goto bx
 if v = 7 && w = 2 then goto cx
 if v = 7 && w = 3 then goto dx
 goto nextstep

 rem --- Subroutine di Animazione Nemico ---
 rem Queste quattro sezioni (ax, bx, cx, dx) definiscono i 4 frame di
 rem animazione per player1. Ognuna reimposta il contatore 'v' e
 rem aggiorna il contatore di frame 'w'.
ax
 v = 0
 w = 1
 player1:
 %00001000
 %01101000
 %00101000
 %01010000
 %01011110
 %01110000
 %00011000
 %00011000
```

```
end
 goto nextstep
bx
 v = 0
 w = 2
 player1:
 %00100000
 %01110000
 %00101000
 %01010000
 %01011110
 %01110000
 %00011000
 %00011000
end
 goto nextstep
cx
 v = 0
 w = 3
 player1:
 %00011000
 %00011000
 %00101000
 %01010000
 %01011110
 %01110000
 %00011000
 %00011000
end
 goto nextstep
dx
 v = 0
 w = 0
 player1:
 %00100000
 %01110000
 %00101000
 %01010000
 %01011110
 %01110000
 %00011000
 %00011000
end
```

```
 goto nextstep

nextstep
 rem Definizione grafica dello sprite del giocatore (la ghianda).
 player0:
 %00111100
 %01011010
 %00100100
 %00111100
 %00011000
 %00011000
 %00010000
 %00010000
end

 rem --- Logica del Proiettile ---
checkfire
 rem Controlla se un missile è già attivo (missile0y <= 240).
 if missile0y>240 then goto skip
 rem Se è attivo, lo muove verso l'alto.
 missile0y = missile0y - 2
 goto draw

skip
 rem Se non ci sono missili attivi, controlla se il giocatore preme 'fuoco'.
 rem Se sì, crea un nuovo missile alla posizione del giocatore.
 if joy0fire then missile0y=player0y-2:missile0x=player0x+4

draw
 rem Disegna il fotogramma corrente.
 drawscreen

 rem --- Logica di Movimento e Limiti ---
 rem Clamping: impedisce al giocatore di uscire dai bordi dello schermo.
 if player0x < 18 then player0x = 18
 if player0x > 136 then player0x = 136
 if player0y < 8 then player0y = 8
 if player0y > 80 then player0y = 80

 rem IA Semplice: il nemico (player1) insegue il giocatore (player0).
 if player1y < player0y then player1y = player1y + 1
 if player1y > player0y then player1y = player1y - 1
 if player1x < player0x then player1x = player1x + 1
```

```
 if player1x > player0x then player1x = player1x - 1

 rem --- Gestione Collisioni ---
 rem Rileva la collisione tra il missile e il nemico.
 if collision(missile0,player1) then goto point
 rem Rileva la collisione tra il nemico e il giocatore.
 if collision(player0,player1) then goto dead

 rem --- Gestione Input Giocatore ---
 if joy0up then player0y = player0y-1
 if joy0down then player0y = player0y+1
 if joy0left then player0x = player0x-1
 if joy0right then player0x = player0x +1

 rem Ripete il ciclo di gioco.
 goto main

 rem --- Subroutine di Evento: Nemico Colpito ---
point
 rem Incrementa lo score.
 score=score+100
 rem Fa riapparire il nemico in una nuova posizione casuale in cima allo schermo.
 player1x=rand/2
 player1y=0
 rem Disattiva il missile.
 missile0y=255
 rem Torna al ciclo principale.
 goto main

 rem --- Subroutine di Evento: Giocatore Colpito ---
dead
 rem Decrementa lo score.
 score=score-1
 rem Fa riapparire il nemico in una nuova posizione casuale.
 player1x=rand/2
 player1y=0
 rem Disattiva il missile.
 missile0y=255
 rem Decrementa il contatore delle vite.
 a=a-1
 rem Se le vite sono esaurite, passa alla schermata di Game Over.
 if a = 0 then goto resetfire
 rem Altrimenti, torna al ciclo principale.
```

```
 goto main


 rem --- Stato Finale: Game Over / Riavvio ---
resetfire
 rem Nasconde il giocatore.
 player0y=200
 rem Usa un flag temporaneo 'f' per rilevare una singola pressione del fuoco.
 f = 0
 if joy0fire || joy1fire then f = 1
 rem Se il giocatore non preme fuoco, rimane in un loop che mostra la schermata titolo.
 if f = 0 then goto opening
 rem Se preme fuoco, esce dal loop e riavvia.
 drawscreen
 goto resetfire
```

Simple Soccer

```
rem ****************************************************************
rem * Simple Soccer  (2 Giocatori)                                 *
rem *                                                              *
rem * DESCRIZIONE DEL GIOCO:                                       *
rem * Un gioco di calcio/hockey a due giocatori. Ogni giocatore    *
rem * controlla il proprio personaggio (player0 e player1) in      *
rem * quattro direzioni. I giocatori possono "dribblare" la palla  *
rem * (ball) tenendola vicina a sé e tirare premendo il pulsante   *
rem * di fuoco. L'obiettivo è segnare nella porta avversaria. Le   *
rem * porte sono le aree aperte ai lati del campo. Il gioco        *
rem * gestisce il possesso palla e il tiro.                       *
rem *                                                              *
rem * TECNICHE DI PROGRAMMAZIONE UTILIZZATE:                      *
rem * - Kernel Option `player1colors`: Questa opzione avanzata del *
rem *   kernel viene usata per dare a `player1` un aspetto         *
rem *   multicolore, permettendo di definire un colore diverso per *
rem *   ogni linea dello sprite. Questo sacrifica l'uso del        *
rem *   missile1, ma arricchisce notevolmente la grafica.          *
rem * - Gestione del Possesso Palla: Una variabile (`p`) funge da  *
rem *   flag per determinare quale giocatore ha il possesso della  *
rem *   palla. Quando la palla non è in fase di tiro, la sua       *
rem *   posizione viene costantemente aggiornata per "attaccarsi"  *
rem *   al giocatore in possesso.                                  *
rem * - Gestione del Tiro: Una variabile (`z`) funge da flag di    *
rem *   stato per il tiro. Se `z` è 0, la palla è in possesso. Se è 1,*
rem *   la palla è stata tirata da player0 e si muove da sola. Se è 2,*
rem *   la palla è stata tirata da player1.                        *
rem * - Logica di Salvataggio Posizione: Per gestire le collisioni *
rem *   con i muri, il programma salva le coordinate "valide" dei  *
rem *   giocatori all'inizio di ogni frame (`e, f, g, h`). Se viene *
rem *   rilevata una collisione, le coordinate vengono ripristinate *
rem *   a quelle precedenti, impedendo al giocatore di passare     *
rem *   attraverso i muri.                                         *
rem * - Gioco a Due Giocatori: Il codice legge l'input da entrambi i *
rem *   joystick (`joy0` e `joy1`), permettendo a due persone di   *
rem *   giocare contemporaneamente.                                *
rem ****************************************************************

rem --- Direttive del Compilatore ---
rem Abilita l'opzione del kernel per avere uno sprite player1 multicolore.
set romsize 4k
```

```
 set kernel_options player1colors

 rem --- Definizioni Grafiche Iniziali ---
 rem Definisce il campo da gioco con le due porte laterali.
 playfield:
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 X....X..................X....X
 X............................X
 X............................X
 X............................X
 X............................X
 X............................X
 X............................X
 X............................X
 X....X..................X....X
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 rem Definisce la grafica per il giocatore 1 (player0).
 player0:
 %00100010
 %00010100
 %00001000
 %00111110
 %00001000
 %00011100
 %00011100
 %00011100
end
 rem Definisce la grafica per il giocatore 2 (player1).
 player1:
 %01000100
 %00101000
 %00010000
 %01111100
 %00010000
 %00111000
 %00111000
 %00111000
end

 rem --- Impostazioni di Gioco Iniziali ---
 rem Imposta il colore di sfondo iniziale.
 COLUBK = $0F
```

```
 rem Imposta la larghezza del missile0 (non usato per sparare ma il registro esiste).
 NUSIZ0 = $30
 rem Azzera lo score e imposta il colore del testo.
 score = 00000
 scorecolor = $08

 rem --- Inizializzazione Variabili ---
 rem Le variabili non hanno alias 'dim', ma rappresentano:
 rem a, b: coordinate x, y del giocatore 0.
 a = 75
 b = 75
 rem c, d: coordinate x, y del giocatore 1.
 c = 75
 d = 25
 rem z: stato della palla (0=in possesso, 1=tiro p0, 2=tiro p1).
 z = 0
 rem p: possesso palla (0=p0, 1=p1).
 p = 0

 rem Imposta le posizioni iniziali degli oggetti di gioco.
 player0x = a : player0y = b
 player1x = c : player1y = d
 ballx = x : bally = y

 rem --- Ciclo di Gioco Principale ---
main
 rem Imposta i registri TIA volatili ad ogni frame.
 COLUP1 = $80
 COLUP0 = $40
 COLUBK = $C4
 COLUPF = $0E
 rem Salva le coordinate correnti dei giocatori prima di ogni movimento.
 e = a
 f = b
 g = c
 h = d

 rem Tabella colori per lo sprite multicolore player1.
 player1color:
    $38;
    $3A;
    $F4;
    $F6;
```

```
    $0C;
    $1A;
    $D8;
    $D2;
end
 rem Disegna il fotogramma corrente.
 drawscreen

 rem --- Gestione Input Giocatori ---
 rem Legge l'input dal joystick 0 per muovere il giocatore 0.
 if joy0left then a = a - 1
 if joy0up then b = b - 1
 if joy0down then b = b + 1
 if joy0right then a = a + 1
 rem Legge l'input dal joystick 1 per muovere il giocatore 1.
 if joy1left then c = c - 1
 if joy1up then d = d -1
 if joy1down then d = d + 1
 if joy1right then c = c + 1

 rem --- Logica della Palla (Possesso e Tiro) ---
 rem Se il giocatore in possesso preme 'fuoco', imposta lo stato 'tiro'.
 if p = 0 && joy0fire then z = 1
 if p = 1 && joy1fire then z = 2
 rem Se la palla è in possesso (z=0), la "attacca" al giocatore corretto.
 if z = 0 && p = 0 then ballx = a + 5 : bally = b - 10
 if z = 0 && p = 1 then ballx = c + 4 : bally = d + 2
 rem Se la palla è in stato 'tiro', la muove in verticale.
 if z = 1 then bally = bally - 1
 if z = 2 then bally = bally + 1

 rem Aggiorna le coordinate finali degli sprite.
 player0x = a : player0y = b
 player1x = c : player1y = d

 rem --- Gestione Reset e Collisioni ---
 rem Controlla se il pulsante di reset è stato premuto.
 if switchreset then goto hardReset

 rem Se la palla tocca un giocatore, quel giocatore ne ottiene il possesso.
 if collision(ball, player0) then goto save0
 if collision(ball, player1) then goto save1
 rem Se un giocatore tocca i muri...
```

```
 if collision(player0, playfield) then goto player0HitWall
 if collision(player1, playfield) then goto player1HitWall
 rem Impedisce ai giocatori di entrare nelle porte avversarie.
 if player0y < 30 then goto player0HitWall
 if player1y > 66 then goto player1HitWall
 rem Se la palla tocca il playfield (muri o porte)...
 if collision(ball, playfield) then goto shoot

 rem Ripete il ciclo di gioco.
 goto main

 rem --- Subroutine di Gestione Collisioni Muri ---
player0HitWall
 rem Ripristina la posizione del giocatore 0 a quella valida precedente.
 a = e
 b = f
 goto main
player1HitWall
 rem Ripristina la posizione del giocatore 1 a quella valida precedente.
 c = g
 d = h
 goto main

 rem --- Subroutine di Gestione Possesso Palla ---
save0
 rem Il giocatore 0 ora ha il possesso.
 p = 0
 rem La palla smette di essere in 'tiro'.
 z = 0
 goto main
save1
 rem Il giocatore 1 ora ha il possesso.
 p = 1
 rem La palla smette di essere in 'tiro'.
 z = 0
 goto main

 rem --- Subroutine di Gestione Tiro in Porta ---
shoot
 rem Controlla se la palla ha colpito un muro laterale o una delle porte.
 rem Le porte si trovano tra x=41 e x=119.
 if ballx > 41 && ballx < 119 then goto hit
 rem Se ha colpito un muro laterale, resetta le posizioni.
```

```
 goto reset
hit
 rem Controlla in quale metà del campo si trova la palla per determinare chi ha segnato.
 if bally < 50 then goto player0Score
 if bally > 50 then goto player1Score
 goto reset
player0Score
 rem Il giocatore 0 ha segnato.
 score = score + 1
 rem La palla passa al giocatore 1.
 p = 1
 goto reset
player1Score
 rem Il giocatore 1 ha segnato.
 score = score + 1000
 rem La palla passa al giocatore 0.
 p = 0
 goto reset

 rem --- Subroutine di Reset Posizioni ---
reset
 rem Riporta i giocatori alle posizioni iniziali.
 a = 75
 b = 75
 c = 75
 d = 25
 rem Resetta lo stato della palla a 'in possesso'.
 z = 0
 goto main

 rem --- Subroutine di Hard Reset ---
hardReset
 rem Azzera completamente lo score.
 score = 000000
 rem Restituisce il possesso iniziale al giocatore 0.
 p = 0
 rem Chiama la subroutine di reset delle posizioni.
 goto reset
```

The Watch

```
rem ****************************************************************
rem * The Watch  (Il Guardiano del Castello)                       *
rem *                                                              *
rem * DESCRIZIONE DEL GIOCO:                                       *
rem * Il giocatore controlla un cavaliere (il Ranger, player0) che *
rem * deve difendere un muro da un mostro (il Wright, player1). Il *
rem * mostro insegue il giocatore e tenta di distruggerlo. Il      *
rem * giocatore può attaccare con la sua spada (missile0) per      *
rem * respingere e infine sconfiggere il mostro. Il giocatore deve *
rem * anche raccogliere mattoni (ball) e portarli sul muro per     *
rem * ricostruirlo. Il gioco diventa progressivamente più difficile, *
rem * con il mostro che diventa più veloce e resistente.           *
rem *                                                              *
rem * TECNICHE DI PROGRAMMAZIONE UTILIZZATE:                       *
rem * - Kernel Options Avanzate: `pfcolors` e `pfheights` vengono  *
rem *   usate per creare un playfield multicolore e con blocchi di *
rem *   altezze diverse, permettendo una grafica di sfondo più ricca.*
rem * - IA con Difficoltà Crescente: La velocità del nemico è      *
rem *   controllata da un timer (`d`). Ad ogni livello completato, *
rem *   il valore di `d` diminuisce, rendendo il nemico più veloce. *
rem *   Anche la sua resistenza (`a`, colpi per ucciderlo) aumenta. *
rem * - Interazione Dinamica con il Playfield: Il giocatore può    *
rem *   modificare il playfield in tempo reale. Raccoglie un mattone *
rem *   (ball) e, quando tocca il muro, usa `pfpixel ... on` per   *
rem *   "disegnare" un nuovo blocco, ricostruendo la fortificazione. *
rem *   Il comando `pfread` viene usato per verificare se un blocco *
rem *   è già stato posato.                                        *
rem * - Animazione a Frame Multipli: Il giocatore ha un'animazione *
rem *   di camminata (frame1, frame2) e una di attacco (dosword).  *
rem * - Gestione del Punteggio BCD: Viene usato l'approccio        *
rem *   consigliato con alias ai singoli byte dello score (`_sc1`, *
rem *   `_sc2`, `_sc3`) per controllare in modo sicuro condizioni  *
rem *   come il Game Over (score < 0).                            *
rem * - Effetti Sonori Temporizzati: Gli effetti sonori per colpi e *
rem *   vittorie sono gestiti tramite loop che si ripetono per un  *
rem *   numero fisso di frame, creando suoni di breve durata.      *
rem ****************************************************************

rem --- Sezione Definizioni Variabili (Commenti originali mantenuti) ---
rem a: numero di colpi per uccidere i wright
rem b: posizione orizzontale del ranger (p0_x)
```

```
 rem c: posizione verticale del ranger (p0_y)
 rem d: velocita' con cui il wright insegue il disertore (in funzione di t)
 rem e: suono dell'uccisione di Wright
 rem f: timer per l'animazione del player0 che cammina
 rem g: suono wright colpito
 rem h: distanza del wright dopo essere stato pugnalato (colpo)
 rem i: possesso mattone (ball)
 rem j-u: flag per le sezioni del muro ricostruite
 rem t: timer per velocita' wright inseguimento del ranger (vedi d)
 rem v: movimento player1 (wright) orizzontalmente
 rem w: movimento player1 (wright) in verticale
 rem y: suono del castello costruito

 rem --- Direttive del Compilatore ---
 set romsize 4k
 set kernel_options pfcolors pfheights

 rem --- Stato 0: Inizializzazione Globale e Schermata Titolo ---
init
 rem Imposta i colori iniziali.
 COLUP1 = $A6
 COLUBK = $0E

 rem Definizione grafica iniziale del nemico (Wright).
 player1:
 %11110111
 %01110111
 %00011011
 %00011011
 %00011011
 %00110011
 %00110110
 %00110110
 %10111100
 %10111101
 %10111101
 %10111101
 %10111101
 %10111101
 %10111101
 %11111110
 %11111000
 %01111000
```

```
 %00100100
 %00101100
 %01000010
 %01101010
 %01000010
 %00111100
end
 rem Posiziona il nemico fuori dallo schermo.
 player1x = 0 : player1y = 200

 rem Definisce le altezze per ogni linea del playfield.
 pfheights:
 8
 8
 8
 8
 8
 8
 8
 8
 8
 8
 8
end
 rem Definisce i colori per ogni linea del playfield.
 pfcolors:
 $00
 $AA
 $AA
 $AA
 $AC
 $AC
 $AC
 $AE
 $AE
 $AE
 $AE
end
 rem Posiziona palla e giocatore fuori schermo.
 ballx = 0
 bally = 200
 player0x = 0 : player0y = 200
```

```
 rem --- Alias per le variabili di gioco principali ---
 dim p0_x = b
 dim p0_y = c
 v=152
 w=40
 rem Alias per i byte dello score (gestione BCD).
 dim _sc1 = score
 dim _sc2 = score+1
 dim _sc3 = score+2

 rem Inizializzazione parametri di difficoltà.
 a = 4
 d = 30
 h = 20

 rem Grafica della schermata titolo.
 playfield:
 .XXXX.....................X.X
 XX........XXX.X.X.XXX.....XXXXX
 .XX........X..X.X.X.........XX
 ..XX.......X..XXX.XXX.......X..
 ..X.X......X..X.X.X........X.X.
 ..XX.......X..X.X.XXX.......X.X
 .X...........................XX
 XX....X...X.XXX.XXX.XXX.X.X...X
 .XX...X.X.X.X.X..X..X...XXX...X
 X.XX...X.X..XXX..X..X...X.X..XX
 XXXX...X.X..X.X..X..XXX.X.X...X
end
 rem Silenzia i canali audio.
 AUDV1 = 0
 AUDC1 = 0
 AUDF1 = 0

firstscreen
 rem Loop della schermata titolo: attende la pressione di 'fuoco'.
 drawscreen
 if joy0fire then goto preloop
 goto firstscreen

 rem --- Stato 1: Preparazione del Livello ---
preloop
 rem Posiziona il giocatore, il nemico e il mattone per l'inizio del round.
```

```
 player0x = 56 : player0y = 96
 player1x = 80 : player1y = 60
 ballx = 81
 bally = 78
 rem Definisce la grafica del muro da ricostruire.
 playfield:
 X.X.X.........................
 .XXXX.........................
 X.XXX.........................
 XXX.X.........................
 XX.XX.........................
 X.XXX.........................
 X.XXX.........................
 XXX.X.........................
 .XXXX.........................
 X.XXX.........................
 XXXXX.........................
end
 rem Resetta la posizione del nemico e i flag di costruzione del muro.
 v= 152 : w = 40
 j = 0 : k = 0 : l = 0 : o = 0 : p = 0 : q = 0 : r = 0 : s = 0 : u = 0
 drawscreen

 rem --- Ciclo di Gioco Principale ---
loop
 rem Controlla la pressione del tasto Reset della console.
 if switchreset then goto init

 rem Se tutte le sezioni del muro sono costruite, passa allo stato di 'livello completato'.
 if j = 1 && k = 1 && l = 1 && o = 1 && p = 1 && q = 1 && r = 1 && s = 1 && u = 1 then goto castl
ecompleted

 rem Impostazioni dei registri TIA per il gioco.
 ballheight = 3
 CTRLPF = $21
 player1x=v
 player1y=w

 rem --- Logica di Animazione e Timer ---
 rem Incrementa i timer per l'animazione e la velocità del nemico.
 f=f+1
 t=t+1
 if t>30 then t=0
```

```
 rem Controlla la condizione di Game Over (score andato in negativo).
 if _sc1 = $99 && _sc2 = $99 && _sc3 <= $99 then score = score +1 : goto firstscreen


 rem --- Gestione Movimento e Animazione Giocatore ---
 rem Azzera il contatore di animazione.
 if f = 20 then f = 0
 rem Seleziona il frame di animazione in base al timer 'f'.
 if f < 10 then gosub frame1
 if f > 10 && f < 20 then gosub frame2
 if f > 10 && f < 20 && !joy0left && !joy0right && !joy0up && !joy0down then gosub frame1


 rem Gestione input per movimento. Usa l'aritmetica a complemento a due per il movimento.
 p0_x = 0
 if joy0left && !joy0fire then REFP0 = 8 : p0_x = 255 : player0x = player0x + p0_x : if i = 1 the
n ballx = player0x - 3 : bally = player0y - 11
 if joy0right && !joy0fire then REFP0 = 0 : p0_x = 1 : player0x = player0x + p0_x : if i = 1 then
 ballx = player0x - 3 : bally = player0y - 11
 p0_y = 0
 if joy0up then p0 y = 255 : player0y = player0y + p0 y : if i = 1 then ballx = player0x - 3 : ba
lly = player0y - 11
 if joy0down then p0 y = 1 : player0y = player0y + p0 y : if i = 1 then ballx = player0x - 3 : ba
lly = player0y - 11


 rem --- Gestione Attacco Giocatore ---
 rem Imposta la dimensione orizzontale della spada.
 NUSIZ0 = $30
 rem Se preme fuoco, mostra la spada e perde il possesso del mattone.
 if joy0fire then missile0x = player0x + 9 : missile0y = player0y - 7 : i = 0 : gosub dosword els
e missile0x = 0 : missile0y = 0


 rem Clamping: impedisce a giocatori e oggetti di uscire dallo schermo.
 if player0x < 38 then player0x = 38
 if player0x > 124 then player0x = 124
 if player0y < 17 then player0y = 17
 if player0y > 89 then player0y = 89
 if ballx < 37 then ballx = 37
 if bally < 11 then bally = 11
 if bally > 78 then bally = 78
 if player1x > 152 then player1x = 152


 rem Imposta i colori di gioco.
 COLUBK = $0E
 COLUP1 = $A6
 rem Disegna il fotogramma.
```

```
 drawscreen


 rem --- Gestione delle Collisioni ---
 rem Collisione spada-nemico: se il nemico ha ancora vita, lo respinge.
 if collision(missile0,player1) && a <> 0 then v = v + h : a = a - 1: goto strikewright
 rem Se il nemico non ha più vita, lo sconfigge.
 if collision(missile0,player1) && a = 0 then v = 152 : score = score + 1 : goto killwright
 rem Collisione giocatore-nemico: resetta posizione, perde punti.
 if collision(player0,player1) then player0x = 56 : player0y = 96 : i = 0 : score = score -1 : if
d = 30 then v = v + 5
 rem Collisione giocatore-mattone: prende possesso del mattone.
 if collision(ball,player0) then ballx = player0x - 3 : bally = player0y - 11 : i = 1
 rem Collisione mattone-muro: posa il mattone.
 if collision(ball, playfield) then gosub putoncastle


 rem --- IA del Nemico ---
 rem Muove il nemico verso il giocatore solo se il timer 't' supera la soglia 'd'.
 if t<d then goto skipmovement
 if v < player0x then v=v+1
 if v > player0x then v=v-1 : AUDV1 = 4
 if w < player0y then w=w+1
 if w > player0y then w=w-1 : AUDV1 = 4
skipmovement
 rem Ripete il ciclo di gioco.
 goto loop


 rem --- Subroutine per Costruire il Castello ---
putoncastle
 rem Controlla la posizione del mattone e, se lo spazio è libero (!pfread), disegna un nuovo pixe
l.
 if bally > 11 && bally < 17 && !pfread(0,1) then pfpixel 0 1 on : i = 0 : ballx = 81 : bally = 3
9 : j = 1
 if bally >= 17 && bally < 25 && !pfread(1,2) then pfpixel 1 2 on : i = 0 : ballx = 81 : bally =
75 : k = 1
 if bally >= 25 && bally < 35 && !pfread(3,3) then pfpixel 3 3 on : i = 0 : ballx = 81 : bally =
11 : l = 1
 if bally >= 35 && bally < 41 && !pfread(2,4) then pfpixel 2 4 on : i = 0 : ballx = 81 : bally =
59 : o = 1
 if bally >= 41 && bally < 49 && !pfread(1,5) then pfpixel 1 5 on : i = 0 : ballx = 81 : bally =
19 : p = 1
 if bally >= 49 && bally < 56 && !pfread(1,6) then pfpixel 1 6 on : i = 0 : ballx = 81 : bally =
53 : q = 1
 if bally >= 56 && bally < 65 && !pfread(3,7) then pfpixel 3 7 on : i = 0 : ballx = 81 : bally =
27 : r = 1
 if bally >= 65 && bally < 73 && !pfread(0,8) then pfpixel 0 8 on : i = 0 : ballx = 81 : bally =
19 : s = 1
```

```
 if bally >= 73 && bally < 78 && !pfread(1,9) then pfpixel 1 9 on : i = 0 : ballx = 81 : bally =
65 : u = 1
 return


 rem --- Subroutine Grafiche Giocatore ---
frame1
 rem Primo frame dell'animazione di camminata.
 player0:
 %11111100
 %11011000
 %11011000
 %11011000
 %11011000
 %11111000
 %10011010
 %11001010
 %11101111
 %11111010
 %01111010
 %10110010
 %11001010
 %01001010
 %01001010
 %00110000
end
 return
frame2
 rem Secondo frame dell'animazione di camminata.
 player0:
 %11101100
 %11001110
 %11001100
 %11011100
 %11011000
 %11111000
 %10011010
 %11001010
 %11101111
 %11111010
 %01111010
 %10110010
 %11001010
 %01001010
```

```
 %01001010
 %00110000
end
 return
dosword
 rem Frame per l'animazione di attacco con la spada.
 player0:
 %11111100
 %11011000
 %11011000
 %11011000
 %11011000
 %11111001
 %10011001
 %11001111
 %11101001
 %11111001
 %01111000
 %10110000
 %11001000
 %01001000
 %01001000
 %00110000
end
 return

 rem --- Subroutine Audio ed Eventi ---
killwright
 rem Suono per la sconfitta del nemico.
 AUDV1 = 4
 AUDC1 = 7
 AUDF1 = e
 e = e + 1
 drawscreen
 if e < 10 then killwright
 e = 0
 AUDV1 = 0 : AUDC1 = 0 : AUDF1 = 0
 score = score + 1
 rem Aggiorna la resistenza del nemico per il prossimo livello.
 if d <= 30 then a = 4
 if d <= 20 then a = 2
 if d <= 10 then a = 0
 goto loop
```

```
strikewright
 rem Suono per il nemico colpito.
 AUDV1 = 4
 AUDC1 = 7
 AUDF1 = 2
 g = g + 1
 drawscreen
 if g < 5 then strikewright
 g = 0
 AUDV1 = 0 : AUDC1 = 0 : AUDF1 = 0
 goto loop
castlecompleted
 rem Suono per il completamento del muro.
 AUDV1 = y
 AUDC1 = 4
 AUDF1 = y
 y = y + 1
 drawscreen
 if y < 64 then goto castlecompleted
 rem Aumenta la difficoltà per il livello successivo.
 y = 0
 d = d - 1
 score = score + 2
 if d <= 30 then a = 4
 if d <= 20 then a = 2 : h = 40
 if d <= 10 then a = 1 : h = 60
 if d = 0 then d = 1
 AUDV1 = 0 : AUDC1 = 0 : AUDF1 = 0
 goto preloop
```

Minotaur

```
rem ****************************************************************
rem * Minotaur                                                     *
rem *                                                              *
rem * DESCRIZIONE DEL GIOCO:                                       *
rem * Il giocatore controlla un eroe (player0) in un labirinto a   *
rem * schermate multiple. Lo scopo è esplorare, raccogliere oggetti *
rem * (lancia, scudo), sconfiggere un Minotauro (player1) e        *
rem * raccogliere monete per aumentare il punteggio. Il mondo è    *
rem * composto da diverse stanze interconnesse.                   *
rem *                                                              *
rem * TECNICHE DI PROGRAMMAZIONE UTILIZZATE:                       *
rem * - Esplorazione a Schermate Multiple: Il gioco gestisce il    *
rem *   passaggio tra diverse stanze (`room`). Quando il giocatore  *
rem *   raggiunge un bordo dello schermo, la variabile `room` viene *
rem *   aggiornata, viene chiamata la subroutine della nuova stanza *
rem *   e il giocatore viene riposizionato sul lato opposto.        *
rem * - Sistema di Inventario e Stati: Variabili come `haslance`,  *
rem *   `hasshield` e `hascoin` fungono da flag per tenere traccia  *
rem *   degli oggetti raccolti e degli stati del giocatore (es. può *
rem *   attaccare, è protetto, ha raccolto la moneta).             *
rem * - Logica di Combattimento: Il giocatore può attaccare con una *
rem *   lancia (missile0), la cui forma e direzione cambiano in     *
rem *   base all'orientamento del giocatore. Il Minotauro ha una    *
rem *   sorta di "punti vita" (`minotauro`) e cambia aspetto quando *
rem *   viene colpito.                                             *
rem * - IA di Pattugliamento: Il Minotauro non insegue direttamente *
rem *   il giocatore, ma si muove lungo un percorso predefinito,    *
rem *   pattugliando l'area.                                       *
rem * - Gestione di Oggetti Multipli con Sprite: Il gioco usa gli   *
rem *   oggetti TIA in modo creativo. `player0` è l'eroe, `player1` è*
rem *   il Minotauro O la moneta, `missile0` è la lancia e `ball` è *
rem *   lo scudo. Il codice gestisce quale oggetto visualizzare in  *
rem *   base allo stato del gioco.                                 *
rem * - Logica di Collisione Avanzata: Il movimento del giocatore   *
rem *   viene bloccato dalle pareti del `playfield` ripristinando la *
rem *   sua posizione precedente in caso di collisione.            *
rem ****************************************************************

set romsize 4k
set smartbranching on
```

```
 rem --- Sezione Definizioni Variabili (Alias) ---
 rem Flag per bloccare il movimento dopo una collisione con i muri.
 dim nodown = a
 dim noup = b
 dim noleft = c
 dim noright = d
 rem Numero della stanza corrente.
 dim room = e
 rem Stato della lancia (0=non posseduta, 1=in mano, 2=lanciata, 3=a terra).
 dim haslance = f
 rem Flag per il possesso dello scudo (0=no, 1=sì).
 dim hasshield = g
 rem Flag per indicare se la moneta nella stanza è stata raccolta.
 dim hascoin = h
 rem Variabile temporanea per i numeri casuali.
 dim randnumber = i
 rem Valore della moneta corrente (1, 5, o 32).
 dim coinvalue = j
 rem Direzione del giocatore (1=su, 2=destra, 3=giù, 4=sinistra).
 dim compass = k
 rem "Punti vita" del Minotauro (2=sano, 1=ferito, 0=morto).
 dim minotauro = m
 rem Contatore per l'invulnerabilità temporanea del giocatore dopo essere stato colpito.
 dim hitted = n

init
 rem --- Inizializzazione Globale ---
 score = 0
 COLUBK = $F4

 rem --- Inizializzazione Variabili di Stato ---
 haslance = 0
 hasshield = 0
 hascoin = 0
 coinvalue = 1
 hitted = 0
 room = 0
 nodown = 0
 noup = 0
 noleft = 0
 noright = 0

 rem --- Impostazioni Iniziali Oggetti di Gioco ---
```

```
 player0x = 24
 player0y = 76
 missile0height = 8
 missile0x = 83
 missile0y = 48
 ballheight = 4
 rem CTRLPF=$21 imposta la palla (scudo) dietro al playfield.
 CTRLPF = $21
 ballx = 0
 bally = 0

 rem per inizializzare le collisioni
 drawscreen

 rem --- Ciclo di Gioco Principale ---
mainloop
 rem Carica la stanza 1 se il gioco è appena iniziato (room=0).
 if room = 0 then gosub room1 : gosub moverderecha : gosub minoheridados : COLUP0 = $86

 rem Imposta la dimensione dello sprite del nemico.
 NUSIZ1 = $10
 rem Imposta il colore del nemico/moneta. Cambia colore in base al valore della moneta.
 COLUP1 = $4A
 if coinvalue = 5 then COLUP1 = $0A
 if coinvalue = 32 then COLUP1 = $1E

 rem Gestisce l'invulnerabilità del giocatore: se è stato colpito, lampeggia.
 if hitted > 0 then hitted = hitted - 1 : COLUP0 = $40 else COLUP0 = $86

 rem --- Logica di Raccolta Oggetti ---
 rem Raccoglie la lancia (missile0).
 if collision(missile0,player0) && haslance = 0 then haslance = 1 : NUSIZ0 = $00 : missile0height
 = 8
 rem Raccoglie la lancia dopo averla lanciata.
 if collision(missile0,player0) && haslance = 3 then haslance = 1 : NUSIZ0 = $00 : missile0height
 = 8
 rem Raccoglie lo scudo (ball).
 if collision(ball,player0) && hasshield = 0 then hasshield = 1

 rem --- Logica delle Collisioni Principali ---
 rem Collisione con Minotauro: se ha lo scudo, lo perde. Se non ce l'ha, muore (non implementato)
.
 if minotauro > 0 && hitted = 0 && collision(player1,player0) && hasshield = 1 then hasshield = 0
 : ballx = 0 : bally = 0 : gosub hit
 if minotauro > 0 && hitted = 0 && collision(player1,player0) && hasshield = 0 then goto gameover
```

```
 rem Collisione con moneta (quando il minotauro è morto).
 if minotauro = 0 && collision(player1,player0) && hascoin = 0 then hascoin = 1 : score = score + coinvalue : gosub colocarmoneda


 rem --- Gestione Input Giocatore (Movimento e Attacco) ---
 if joy0left && !joy0right && !joy0up && !joy0down && noleft = 0 then gosub moverizquierda
 if !joy0left && joy0right && !joy0up && !joy0down && noright = 0 then gosub moverderecha
 if !joy0left && !joy0right && joy0up && !joy0down && noup = 0 then gosub moverarriba
 if !joy0left && !joy0right && !joy0up && joy0down && nodown = 0 then gosub moverabajo
 if joy0fire && haslance = 1 then haslance = 2


 rem --- Logica della Lancia ---
 rem Muove la lancia se è stata lanciata.
 if haslance = 2 && !collision(playfield,missile0) then gosub moverlanza
 rem Ferma la lancia se colpisce un muro.
 if haslance = 2 && collision(playfield,missile0) && compass = 2 then haslance = 3 : missile0x = missile0x - 0
 if haslance = 2 && collision(playfield,missile0) then haslance = 3
 rem Ferisce il Minotauro se lo colpisce.
 if haslance = 2 && collision(player1,missile0) then haslance = 3 : minotauro = minotauro - 1


 rem --- Logica del Nemico ---
 rem Aggiorna la posizione e l'aspetto del Minotauro.
 if minotauro > 0 then gosub moverenemigo else gosub minomuerto
 if minotauro = 1 then gosub minoheridauno


 rem --- Logica Grafica della Lancia ---
 rem Cambia la forma della lancia (orizzontale/verticale) in base alla direzione.
 if compass = 1 && haslance = 3 then NUSIZ0 = $00 : missile0height = 8
 if compass = 2 && haslance = 3 then NUSIZ0 = $30 : missile0height = 0
 if compass = 3 && haslance = 3 then NUSIZ0 = $00 : missile0height = 8
 if compass = 4 && haslance = 3 then NUSIZ0 = $30 : missile0height = 0


 rem --- Gestione Posizione Oggetti Nelle Stanze ---
 rem Posiziona/nasconde la lancia e lo scudo a seconda della stanza in cui si trova il giocatore.
 if room = 1 && haslance = 0 then missile0x = 83 : missile0y = 48
 if room <> 1 && haslance = 0 then missile0x = 0 : missile0y = 0
 if room = 2 && hasshield = 0 then ballx = 83 : bally = 45
 if room <> 2 && hasshield = 0 then ballx = 0 : bally = 0


 rem --- Logica di Transizione tra le Stanze ---
 rem Controlla se il giocatore ha raggiunto un bordo per cambiare stanza.
 if room = 1 && player0x > 145 then gosub room2 : player0x = 22
 if room = 2 && player0x < 5 then gosub room1 : player0x = 140
```

```
 if room = 1 && player0y < 10 then gosub room3 : player0y = 80
 if room = 3 && player0y > 85 then gosub room1 : player0y = 10
 if room = 3 && player0x > 145 then gosub room4 : player0x = 22
 if room = 4 && player0x < 5 then gosub room3 : player0x = 140
 if room = 2 && player0y < 5 then gosub room4 : player0y = 80
 if room = 4 && player0y > 85 then gosub room2 : player0y = 10
 if room = 3 && player0y < 10 then gosub room5 : player0y = 80
 if room = 5 && player0y > 85 then gosub room3 : player0y = 10
 if room = 4 && player0x > 145 then gosub room6 : player0x = 22
 if room = 6 && player0x < 5 then gosub room4 : player0x = 140
 if room = 6 && player0x > 145 then gosub room7 : player0x = 22
 if room = 7 && player0x < 5 then gosub room6 : player0x = 140
 if room = 7 && player0x > 145 then gosub room9 : player0x = 22
 if room = 9 && player0x < 5 then gosub room7 : player0x = 140
 if room = 8 && player0y < 5 then gosub room7 : player0y = 80
 if room = 7 && player0y > 85 then gosub room8 : player0y = 10

 rem Disegna il fotogramma e ripete il ciclo.
 drawscreen
 goto mainloop

gameover
  if joy0fire then goto gameover
gameover2
  COLUBK = $08
  drawscreen
  if joy0fire then goto init
  goto gameover2

 rem --- Subroutine di Movimento e Collisione con i Muri ---
moverizquierda
 player0:
 %01101100
 %00100100
 %00100100
 %00011000
 %11111111
 %10011001
 %00100100
 %00111100
 %00110110
end
 rem Imposta la direzione e controlla la collisione con il playfield.
```

```
 if haslance = 1 then compass = 4
 if collision(playfield,player0) then gosub x001 else gosub x002
 rem Aggiorna la posizione di lancia e scudo per "attaccarli" al giocatore.
 if haslance = 1 then missile0x = player0x : missile0y = player0y - 2
 if hasshield = 1 then ballx = player0x + 7 : bally = player0y - 3
 return
x001
 rem Se c'è collisione, spinge indietro il giocatore.
 player0x = player0x + 1 : noright = 0 : noleft = 1 : noup = 0
 nodown = 0
 return
x002
 rem Se non c'è collisione, esegue il movimento.
 player0x = player0x - 1 : noright = 0 : noleft = 0
 noup = 0 : nodown = 0
 return
x003
 player0x = player0x - 1 : noright = 1 : noleft = 0
 noup = 0 : nodown = 0
 return
x004
 player0x = player0x + 1: noright = 0 : noleft = 0 : noup = 0 : nodown = 0
 return
x005
 player0y = player0y + 1 : noright = 0 : noleft = 0 : noup = 1 : nodown = 0
 return
x006
 player0y = player0y - 1: noright = 0 : noleft = 0 : noup = 0 : nodown = 0
 return
x007
 player0y = player0y - 1 : noright = 0 : noleft = 0 : noup = 0 : nodown = 1
 return
x008
 player0y = player0y + 1: noright = 0 : noleft = 0 : noup = 0 : nodown = 0
 return
moverderecha
 player0:
 %00110110
 %00100100
 %00100100
 %00011000
 %11111111
 %10011001
```

```
 %00100100
 %00111100
 %01101100
end
 if haslance = 1 then compass = 2
 if collision(playfield,player0) then gosub x003 else gosub x004
 if haslance = 1 then missile0x = player0x + 9 : missile0y = player0y - 2
 if hasshield = 1 then ballx = player0x - 1 : bally = player0y - 3
 return
moverarriba
 player0:
 %01100110
 %00100100
 %00100100
 %00011000
 %11111111
 %10011001
 %00100100
 %00111100
 %00100100
end
 if haslance = 1 then compass = 1
 if collision(playfield,player0) then gosub x005 else gosub x006
 if haslance = 1 then missile0x = player0x : missile0y = player0y - 2
 if hasshield = 1 then ballx = player0x + 7 : bally = player0y - 3
 return
moverabajo
 player0:
 %01100110
 %00100100
 %00100100
 %10011001
 %11111111
 %00011000
 %00100100
 %00111100
 %00100100
end
 if haslance = 1 then compass = 3
 if collision(playfield,player0) then gosub x007 else gosub x008
 if haslance = 1 then missile0x = player0x : missile0y = player0y - 2
 if hasshield = 1 then ballx = player0x + 7 : bally = player0y - 3
 return
```

```
 rem --- Subroutine di Gioco ---
hit
 rem Attiva l'invulnerabilità temporanea del giocatore.
 if hitted = 0 then hitted = 100
 return
colocarmoneda
 rem Posiziona una moneta in un punto casuale della stanza.
 gosub minoheridados
 if hascoin = 1 then player1x = 0 : player1y = 0
 randnumber = rand
 if hascoin = 0 && randnumber <= 153 then coinvalue = 1
 if hascoin = 0 && randnumber > 153 && randnumber <= 204 then coinvalue = 5
 if hascoin = 0 && randnumber > 204 && randnumber <= 255 then coinvalue = 32
 randnumber = rand
 if hascoin = 0 && randnumber <= 64 then player1x = 28 : player1y = 22
 if hascoin = 0 && randnumber > 65 && randnumber <= 128 then player1x = 118 : player1y = 22
 if hascoin = 0 && randnumber > 129 && randnumber <= 192 then player1x = 28 : player1y = 77
 if hascoin = 0 && randnumber > 193 && randnumber <= 255 then player1x = 118 : player1y = 77
 return
moverlanza
 rem Muove la lancia in base alla direzione del giocatore.
 if compass = 1 then NUSIZ0 = $00 : missile0height = 8 : missile0y = missile0y - 2
 if compass = 2 then NUSIZ0 = $30 : missile0height = 0 : missile0x = missile0x + 2
 if compass = 3 then NUSIZ0 = $00 : missile0height = 8 : missile0y = missile0y + 2
 if compass = 4 then NUSIZ0 = $30 : missile0height = 0 : missile0x = missile0x - 2
 return
moverenemigo
 rem IA di pattugliamento: il Minotauro si muove lungo un percorso rettangolare.
 if minotauro > 0 && player1y = 22 && player1x < 118 then player1x = player1x + 1
 if minotauro > 0 && player1y = 77 && player1x > 28 then player1x = player1x - 1
 if minotauro > 0 && player1x = 118 && player1y < 77 then player1y = player1y + 1
 if minotauro > 0 && player1x = 28 && player1y > 22 then player1y = player1y - 1
 return


 rem --- Subroutine Grafiche Nemico ---
minoheridados
 rem Sprite del Minotauro sano.
 player1:
 %01101100
 %01101100
 %00100100
```

```
 %00100100
 %00011000
 %00011000
 %11011011
 %11111111
 %00011000
 %00100100
 %01111110
 %01000010
end
 minotauro = 2
 return
minoheridauno
 rem Sprite del Minotauro ferito.
 player1:
 %01101100
 %01101100
 %00100100
 %00100100
 %00011000
 %00011000
 %11011011
 %11111111
 %00011000
 %00100100
 %00111100
end
 return
minomuerto
 rem Sprite del Minotauro sconfitto (ora appare la moneta al suo posto).
 player1:
 %00010000
 %00001000
 %11101011
 %00011101
 %00011101
 %11101011
 %00001000
 %00010000
end
 return

 rem --- Subroutine di Definizione delle Stanze ---
```

```
 rem Ognuna di queste subroutine definisce la grafica di una stanza
 rem e la logica per la generazione delle monete al suo interno.
room1
 room = 1
 hascoin = 0
 COLUPF = $C0
 playfield:
 XXXXXXXXXXXXX.......XXXXXXXXXXXX
 X............................X
 X............................X
 X...........XXXXXXX...........X
 X................X............
 X................X............
 X................X............
 X...........XXXXXXX...........X
 X............................X
 X............................X
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 if haslance = 1 then gosub colocarmoneda else hascoin = 1 : gosub colocarmoneda
 if haslance = 3 then haslance = 0 : NUSIZ0 = $00 : missile0height = 8
 return
room2
 room = 2
 hascoin = 0
 playfield:
 XXXXXXXXXXXXX.......XXXXXXXXXXXX
 X............................X
 X............................X
 X...........XXXXXXX...........X
 ............X................X
 ............X................X
 ............X................X
 X...........XXXXXXX...........X
 X............................X
 X............................X
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 if hasshield = 1 then gosub colocarmoneda else hascoin = 1 : gosub colocarmoneda
 if haslance = 3 then haslance = 0
 return
room3
 room = 3
```

```
 hascoin = 0
 playfield:
 X......XXXXXXXXX.......XXXXXXXXX
 X.............................X
 X.............................X
 X...........XXXXXXX...........X
 X.............................
 X.............................
 X.............................
 X...........XXXXXXX...........X
 X.............................X
 X.............................X
 XXXXXXXXXXXXX.......XXXXXXXXXXX
end
 gosub colocarmoneda
 if haslance = 3 then haslance = 0
 return
room4
 room = 4
 hascoin = 0
 playfield:
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 X.............................X
 X.............................X
 X...........X.......X.........X
 ............X.......X...........
 ............X.......X...........
 ............X.......X...........
 X...........X.......X.........X
 X.............................X
 X.............................X
 XXXXXXXXXXXXX.......XXXXXXXXXXX
end
 gosub colocarmoneda
 if haslance = 3 then haslance = 0
 return
room5
 room = 5
 hascoin = 0
 playfield:
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 X.............................X
 X.............................X
```

```
 X.............................X
 X......XXXXXXXXXXXXXXXXXXXXXXXX
 X......X......................X
 X......X......................X
 X......X......................X
 X......X......................X
 X......X......................X
 X......XXXXXXXXX.......XXXXXXXXX
end
 gosub colocarmoneda
 if haslance = 3 then haslance = 0
 return
room6
 room = 6
 hascoin = 1
 playfield:
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXXXXXXXXXXXXXXXXXXXX........
 XXXXXXXXXXXXXXXXXXXXXXX........
 XXXXXXXXXXXXXXXXXXXXXXX........
 .........XXXXXXXXXXXXXX.......X
 .........XXXXXXXXXXXXXX.......X
 .........XXXXXXXXXXXXXX.......X
 X.............................X
 X.............................X
 X.............................X
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 gosub colocarmoneda
 if haslance = 3 then haslance = 0
 return
room7
 room = 7
 hascoin = 1
 playfield:
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 ...................XXXXXXXXXXXX
 ...................XXXXXXXXXXXX
 ...................XXXXXXXXXXXX
 XXXXXXXXXXXXX.................X
 XXXXXXXXXXXXX.................X
 XXXXXXXXXXXXX.................X
 XXXXXXXXXXXXX.......XXXXXXXXXXX
```

```
 XXXXXXXXXXXXX.......X..........
 XXXXXXXXXXXXX.......X..........
 XXXXXXXXXXXXX.......X.........X
end
 gosub colocarmoneda
 if haslance = 3 then haslance = 0
 return
room8
 room = 8
 hascoin = 0
 playfield:
 XXXXXXXXXXXXX.......XX.........X
 X...................XX.........X
 X...................XX.........X
 X...................XX.........X
 X...................XX.........X
 X...................XX.........X
 X.......XXXXXXXXXXXXXX.........X
 X..............................X
 X..............................X
 X..............................X
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 gosub colocarmoneda
 if haslance = 3 then haslance = 0
 return
room9
 room = 9
 hascoin = 0
 playfield:
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 X..............................X
 X..............................X
 X............XXXXXXX...........X
 X............X.................X
 X............X.................X
 X............X.................X
 X............XXXXXXX...........X
 ...............................X
 ...............................X
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 gosub colocarmoneda
```

```
if haslance = 3 then haslance = 0
return
```

Snappy

```
rem ******************************************************************
rem * Snappy                                                         *
rem *                                                                *
rem * DESCRIZIONE DEL GIOCO:                                         *
rem * Il giocatore controlla un esploratore (player0) che deve       *
rem * attraversare una voragine usando una liana (playfield). Il     *
rem * tempismo è cruciale: se l'esploratore salta al momento         *
rem * sbagliato, cadrà nella voragine dove un coccodrillo           *
rem * (Snappy, player1) lo attende.                                 *
rem *                                                                *
rem * TECNICHE DI PROGRAMMAZIONE UTILIZZATE:                        *
rem * - Macchina a Stati (State Machine): Il cuore del programma.    *
rem *   La variabile `gamestate` controlla lo stato attuale del      *
rem *   giocatore (in caduta, in attesa, in corsa, sulla liana,      *
rem *   etc.). Il `main_loop` delega la logica a una subroutine      *
rem *   "centralino" (`handlestate`) che esegue solo il codice       *
rem *   relativo allo stato corrente, mantenendo il programma        *
rem *   organizzato ed efficiente.                                   *
rem * - Animazione basata su Timer: L'animazione della liana, del    *
rem *   giocatore e del coccodrillo è gestita da contatori           *
rem *   (`frame`, `playerframe`, `snappyframe`) che vengono          *
rem *   incrementati a ogni ciclo. Questo permette di alternare      *
rem *   le definizioni grafiche per creare l'illusione del           *
rem *   movimento.                                                   *
rem * - Gestione Audio con Subroutine: Gli effetti sonori sono       *
rem *   incapsulati in piccole subroutine (`playvinesound`,          *
rem *   `playdeathsound`, etc.) e attivati in punti specifici del    *
rem *   codice per sincronizzarli con l'azione.                      *
rem * - Generazione del Seme Casuale (Seed): Nella schermata del     *
rem *   titolo, un contatore (`randseed`) viene incrementato.        *
rem *   Questo valore viene poi usato per inizializzare il           *
rem *   generatore di numeri casuali `rand`, garantendo che la       *
rem *   posizione iniziale del giocatore cambi a ogni partita.       *
rem ******************************************************************

rem --- Direttive del Compilatore ---
set romsize 4k

rem --- Sezione Definizioni Variabili (Alias) ---
rem  Contatore per l'animazione della liana (0-119). Determina la posizione della liana.
dim frame = d
```

```
 rem  Contatore per l'animazione del giocatore. Decide quale frame di animazione dell'eroe mostra
re.
 dim playerframe = e
 rem  Gestore dello stato di gioco. La variabile "cervello" che controlla la logica corrente.
 dim gamestate = f
 rem  Contatore per l'animazione del coccodrillo 'Snappy'.
 dim snappyframe = g
 rem  Variabile per l'effetto cambio colore nella schermata del titolo.
 dim introcolour = h
 rem  Contatore per l'animazione di Snappy che mangia il giocatore.
 dim snappyeatingframe = i
 rem  Contatore delle vite del giocatore.
 dim life = j
 rem  Contatore usato per generare un seme casuale per il comando 'rand'.
 dim randseed = k

 rem --- Inizializzazione delle Variabili di Gioco (eseguita una sola volta) ---
 rem  Imposta la posizione iniziale della liana.
 frame = 30
 rem  Azzera il contatore per il seme casuale.
 randseed = 0
 rem  Inizia con il primo frame di animazione del giocatore.
 playerframe = 0

 rem --- Definizione degli Stati della Macchina a Stati (gamestate) ---
 rem 1 = Il giocatore sta cadendo con il paracadute.
 rem 2 = Il giocatore è a terra e in attesa dell'input.
 rem 3 = Il giocatore sta correndo verso la voragine.
 rem 4 = Il giocatore è aggrappato alla liana.
 rem 5 = Il giocatore corre verso il traguardo dopo la liana.
 rem 6 = Il giocatore è al sicuro (stato iniziale o dopo aver completato un round).
 rem 7 = Il giocatore sta cadendo nella voragine (morte).
 rem 8 = Il coccodrillo 'Snappy' sta mangiando il giocatore.
 rem  Inizia il gioco nello stato 'sicuro'.
 gamestate = 6

 rem --- Impostazioni Iniziali HUD ---
 rem  Imposta il colore per il punteggio.
 scorecolor = 22
 rem  Imposta il colore iniziale per l'effetto del titolo.
 introcolour = 0

 rem Salta direttamente al ciclo di gioco principale.
```

```
 goto main

 rem ===========================================================
 rem ====== Sottoprogramma: Schermata Titolo e Attesa =========
 rem ===========================================================
showintro
 rem Definisce la grafica statica del titolo (testo "SNAPPY").
 playfield:
   ................................
   ...XXX..........................
   ..X...X.........................
   ..X.............................
   ...XXX..XX...XXX..XX..XX..X..X..
   ......X.X.X.X..X..X.X.X.X.X..X..
   ..X...X.X.X.X..X..X.X.X.X.X..X..
   ...XXX..X.X..XX.X.XX..XX...XXX..
   ..................X...X......X..
   ..................X...X....XX...
   ................................
end

 rem --- Logica della Schermata Titolo ---
 rem Incrementa la variabile per creare un effetto di cambio colore arcobaleno.
 introcolour = introcolour + 1
 COLUPF = introcolour
 rem Imposta i registri TIA: sfondo e sprite neri per nasconderli.
 COLUBK = 0
 COLUP0 = 0
 COLUP1 = 0
 rem Nasconde fisicamente gli sprite posizionandoli fuori dallo schermo.
 player0x = 0
 player0y = 0
 player1x = 0
 player1y = 0
 rem Incrementa il seme per il generatore di numeri casuali mentre il giocatore attende.
 randseed = randseed + 1
 rem Silenzia entrambi i canali audio.
 gosub stopvoiceone
 gosub stopvoicezero
 rem Disegna lo schermo e attende l'input del giocatore.
 drawscreen
 rem  Se il giocatore preme fuoco, inizia il gioco.
 if joy0fire then goto initialize
```

```
 rem  Altrimenti, continua a mostrare la schermata del titolo.
 goto showintro


 rem ================================================
 rem ====== Sottoprogramma: Inizializzazione Partita ======
 rem ================================================
initialize
 rem Azzera il punteggio e imposta le vite.
 score = 0
 life = 10
 rem Inizializza il generatore di numeri casuali con il seme raccolto durante la schermata del ti
tolo.
 if randseed = 0 then rand = 1 else rand = randseed
 rem Torna al ciclo di gioco principale per iniziare la partita.
 goto main


 rem ================================================
 rem ====== Sottoprogramma: Animazione di Snappy =======
 rem ================================================
animatesnappy
 rem Alterna due sprite per il coccodrillo in base al contatore 'snappyframe' per creare un'anima
zione a 2 frame.
 if snappyframe = 0 then player1:
   %00111100
   %00111100
   %00111100
   %00111100
   %01100111
   %01100110
   %01100110
   %11000011
   %11000011
   %11000011
   %10000001
end
 if snappyframe = 10 then player1:
   %00111100
   %00111100
   %00111100
   %00111100
   %00111110
   %00111100
   %00111100
   %00111100
```

```
   %00011100
   %00011100
   %00011000
end
 return

 rem ================================================
 rem ====== Sottoprogramma: Animazione della Liana ======
 rem ================================================
swingvine
 rem Questa lunga catena di 'if' disegna una diversa grafica del playfield in base al valore del contatore 'frame', simulando l'oscillazione della liana.
 if frame = 0 then playfield:
   ..................XXXXXXXXX.....
   ......................X.........
   ......................X.........
   ......................X.........
   ......................X.........
   ......................X.........
   ..............................
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 if frame = 10 || frame = 110 then playfield:
   ..................XXXXXXXXX.....
   ......................X.........
   ......................X.........
   ......................X.........
   ......................X.........
   ......................X.........
   ..............................
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 if frame = 20 || frame = 100 then playfield:
   ..................XXXXXXXXX.....
   ......................X.........
   ......................X.........
   ......................X.........
```

```
   ..................X..........
   ..................X..........
   ............................
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 if frame = 30 || frame = 90 then playfield:
   ................XXXXXXXXX.....
   ..................X.........
   ..................X.........
   ..................X.........
   ..................X.........
   ..................X.........
   ............................
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 if frame = 40 || frame = 80 then playfield:
   ................XXXXXXXXX.....
   ..................X.........
   ..................X.........
   ..................X.........
   ..................X.........
   ..................X.........
   ............................
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 if frame = 50 || frame = 70 then playfield:
   ................XXXXXXXXX.....
   ..................X.........
   ..................X.........
   ..................X.........
   ..................X.........
   ..................X.........
   ............................
   XXXXXXXXXXXXXXXXXX.........XXXXX
```

```
   XXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 if frame = 60 then playfield:
   ..................XXXXXXXXX.....
   ......................X.........
   .......................X........
   ........................X.......
   .........................X......
   ...........................X.....
   ..............................
   XXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXX.........XXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
 rem Sincronizza l'audio con l'animazione della liana.
 if frame = 0 then gosub playvinesound
 if frame = 60 then gosub playvinesecondsound
 rem Spegne il suono dopo pochi frame per creare un effetto breve.
 if frame = 3 then gosub stopvoicezero
 if frame = 63 then gosub stopvoicezero
 return

 rem ===================================
 rem ==== Sottoprogrammi Audio =========
 rem ===================================
playvinesound
 AUDV0=4:AUDC0=12:AUDF0=28
 return

playvinesecondsound
 AUDV0=4:AUDC0=12:AUDF0=20
 return

playdeathsound
 AUDV1=4:AUDC1=14:AUDF1=20
 return

playvictorysound
 AUDV1=4:AUDC1=4:AUDF1=10
 return
```

```
stopvoicezero
 AUDV0=0
 return

stopvoiceone
 AUDV1=0
 return

 rem ================================================
 rem ====== Sottoprogrammi Grafici: Disegno del Giocatore ======
 rem ================================================
drawplayer
 rem Alterna due sprite per l'animazione di corsa del giocatore in base al timer 'playerframe'.
 if playerframe = 0 then player0:
        %00110110
        %00100100
        %00100100
        %00011000
        %01111110
        %00011000
        %00111100
        %00111100
end
 if playerframe = 10 then player0:
        %00011100
        %00011000
        %00011000
        %00011010
        %00111100
        %01011000
        %00111100
        %00111100
end
 return

drawplayerparachute
 rem Sprite speciale per il giocatore quando si lancia col paracadute.
 player0:
        %00110110
        %00100100
        %00100100
        %00011000
```

```
        %00111100
        %01011010
        %01111110
        %01111110
        %00100100
        %00100100
        %01000010
        %10000001
        %11111111
        %11111111
        %01111110
        %00111100
end
 return

drawplayerbeingeaten
 rem Sprite del giocatore mentre viene mangiato.
 player0:
        %00000000
        %00000000
        %00000000
        %00000000
        %00011110
        %00011000
        %00111100
        %00111100
end
 return

 rem ================================
 rem ====== CICLO DI GIOCO PRINCIPALE ======
 rem ================================
main
 rem Controlla se il gioco e' finito (Game Over). Se si, torna alla schermata del titolo.
 if life = 0 then goto showintro

 rem Aggiorna la grafica della liana (playfield).
 gosub swingvine

 rem Aggiorna la grafica di Snappy (player1).
 gosub animatesnappy
```

```
 rem Seleziona lo sprite corretto per il giocatore (player0) in base allo stato attuale del gioco
.
 rem  Se sta cadendo, disegna il paracadute.
 if gamestate = 1 then gosub drawplayerparachute
 rem  Se viene mangiato, disegna l'animazione corrispondente.
 if gamestate = 8 then gosub drawplayerbeingeaten
 rem  In tutti gli altri casi, disegna l'animazione di corsa.
 if gamestate <> 1 && gamestate <> 8 then gosub drawplayer

 rem Incrementa i contatori per le animazioni.
 frame=frame+1
 playerframe=playerframe+1
 snappyframe=snappyframe+1

 rem Accelera l'animazione di Snappy quando sta mangiando il giocatore.
 if gamestate = 8 then snappyframe=snappyframe+1

 rem Azzera i contatori quando raggiungono il loro limite per creare un loop.
 if playerframe >= 20 then playerframe=0
 if frame>=120 then frame=0
 if snappyframe >= 20 then snappyframe=0

 rem Esegue la logica dello stato di gioco corrente ("Centralino" della Macchina a Stati).
 gosub handlestate

 rem Imposta i registri TIA volatili ad ogni frame.
 rem  Colore di Snappy
 COLUP1 = 206
 rem  Colore del Giocatore
 COLUP0 = 28
 rem  Colore dello Sfondo (cielo/acqua)
 COLUBK = 192
 rem  Colore del Playfield (liana/terreno)
 COLUPF = 88

 rem Comando che dice al TIA di disegnare l'intero frame.
 drawscreen

 rem Torna all'inizio del ciclo di gioco.
 goto main

 rem ============================================================
 rem = Sottoprogramma: Centralino della Macchina a Stati       =
```

```
 rem = Esegue la subroutine corretta in base al valore di 'gamestate' =
 rem ========================================================
handlestate
 if gamestate = 6 then gosub createplayer : return
 if gamestate = 1 then gosub dropplayer : return
 if gamestate = 2 then gosub playerwaiting : return
 if gamestate = 3 then gosub playerrunning : return
 if gamestate = 4 then gosub playeronvine : return
 if gamestate = 7 then gosub playerdying : return
 if gamestate = 5 then gosub playerruntosafety : return
 if gamestate = 8 then gosub snappyeating : return
 return

snappyeating
 rem Gestisce l'animazione di Snappy che mangia il giocatore.
 snappyeatingframe = snappyeatingframe + 1
 playerframe=0
 rem Dopo 60 frame, il round finisce e il giocatore perde una vita.
 if snappyeatingframe = 60 then gosub stopvoiceone : gamestate = 6 : life=life-1
 return

playerdying
 rem Il giocatore cade verso il basso nella voragine.
 player0y = player0y + 1
 rem Quando raggiunge il fondo, passa allo stato 'snappyeating'.
 if player0y = 74 then gamestate = 8 : gosub playdeathsound
 return

playerruntosafety
 rem Il giocatore corre verso il bordo destro dello schermo per completare il livello.
 player0x = player0x + 1
 rem Se ha raggiunto la salvezza alla fine dello schermo, il round è vinto.
 if player0x = 139 then gamestate = 6 : score = score + 1 : gosub stopvoiceone : life=life-1
 return

playeronvine
 rem Aggiorna la posizione orizzontale del giocatore per seguire l'oscillazione della liana.
 if frame = 20 then player0x = 98
 if frame = 30 then player0x = 106
 if frame = 40 then player0x = 111
 if frame = 50 then player0x = 116
 if frame = 60 then player0x = 121
```

```
 rem Quando la liana raggiunge l'altro lato, il giocatore si stacca.
 if frame = 60 then gamestate = 5 : gosub playvictorysound

 return

playerrunning
 rem Il giocatore corre verso la voragine.
 player0x = player0x + 1
 rem Controlla se il giocatore afferra la liana quando raggiunge il bordo della voragine.
 if player0x = 90 then gosub didplayercatchvine
 return

didplayercatchvine
 rem Controlla se la liana ('frame') è nella posizione giusta per essere afferrata. Se non lo è,
il giocatore cade.
 if frame <= 10 || frame >= 115 then gamestate = 4 else gamestate = 7 : player1x = 90
 return

playerwaiting
 rem Blocca l'animazione del giocatore in attesa dell'input.
 playerframe = 0

 rem Se il giocatore preme 'fuoco', inizia a correre.
 if joy0fire then gamestate = 3
 return

dropplayer
 rem Il giocatore scende con il paracadute.
 player0y = player0y + 1
 rem Quando raggiunge il terreno, passa allo stato di attesa.
 if player0y = 56 then gamestate = 2

 rem Blocca l'animazione durante la discesa per mostrare il giocatore fermo.
 playerframe = 0
 return

createplayer
 rem Inizia un nuovo round. Genera il giocatore in una posizione casuale.
 player0x = 20 + (rand / 4)
 player0y = 10
 rem Imposta lo stato iniziale su 'dropplayer' (caduta col paracadute).
 gamestate = 1
 playerframe = 0
```

```
rem Resetta la posizione di Snappy in fondo alla voragine.
player1x = 102
player1y = 80
snappyeatingframe = 0
return
```

Gnamm

```
rem ****************************************************************
rem * Gnamm                                                        *
rem *                                                              *
rem * DESCRIZIONE DEL GIOCO:                                       *
rem * Una versione semplificata del classico vidogame. Il giocatore  *
rem * (player0) si muove in un labirinto statico, mangiando palline. *
rem * Un fantasma (player1) insegue il giocatore attraverso il       *
rem * labirinto. Il gioco include la musica di inizio, i suoni per   *
rem * le palline e una sequenza di animazione per la morte del       *
rem * giocatore.                                                   *
rem *                                                              *
rem * TECNICHE DI PROGRAMMAZIONE UTILIZZATE:                       *
rem * - IA di Inseguimento Semplice: Il fantasma non si muove a      *
rem * caso, ma cerca attivamente di raggiungere il giocatore.        *
rem * Ad ogni incrocio, decide se dare priorità al movimento         *
rem * orizzontale o verticale per ridurre la distanza dal suo        *
rem * bersaglio, pur rispettando i vincoli del labirinto.            *
rem * - Uso Intensivo dei Bit-Flag: Quasi tutta la logica di stato   *
rem * del gioco è gestita tramite singoli bit della variabile b,     *
rem * controllando movimento, suoni, animazioni e logica.            *
rem * - Movimento su Griglia: Il movimento del giocatore è vincolato *
rem * a una griglia invisibile. I bit-flag `b{0}` e `b{1}`           *
rem * verificano se il giocatore è a un "incrocio" e può cambiare    *
rem * direzione.                                                   *
rem * - Animazione a Frame Multipli per Direzione: Il giocatore ha   *
rem * set di animazioni diversi per ogni direzione di movimento      *
rem * (su, giù, sinistra/destra), creando un effetto più realistico. *
rem * - Interazione Dinamica con il Playfield: Le palline da         *
rem * mangiare sono parte del `playfield`. Il comando `pfpixel`      *
rem * viene usato per "cancellarle" dinamicamente.                   *
rem * - Kernel Option `pfcolors`: Questa opzione viene usata per     *
rem * dare al labirinto un aspetto bicolore.                         *
rem ****************************************************************

rem --- Direttive del Compilatore ---
set romsize 4k
set kernel_options pfcolors
set smartbranching on

rem --- Alias delle Variabili ---
dim player_x = player0x
```

```
 dim player_y = player0y
 dim ghost_x = player1x
 dim ghost_y = player1y

 dim player_dir = c
 dim ghost_dir = r
 dim ghost_can_h = t
 dim ghost_can_v = u
 dim tmp1 = v
 dim tmp2 = w
 dim framecounter = z

 rem bit-flags:
 rem b{0} = giocatore a incrocio orizzontale
 rem b{1} = giocatore a incrocio verticale
 rem b{2} = toggle movimento giocatore (per rallentare)
 rem b{3} = direzione orizzontale giocatore (0=destra,1=sinistra)
 rem b{4} = sequenza di fine livello
 rem b{5} = suono "waka-waka" attivo
 rem b{6} = animazione di morte attiva
 rem b{7} = sequenza di inizio partita attiva

 rem Imposta il flag b{7} per indicare che siamo nella sequenza di inizio.
 b{7}=1

 rem --- Stato 0: Inizio Partita / Animazione Iniziale ---
beginning
 g=0

 rem Definisce la grafica del labirinto.
 playfield:
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  ..X.X.......X.X..X.X.......X.X..
  ..XXXXXXXXXXXXX..XXXXXXXXXXXXX..
  ..X.X.X.X.X.X.X..X.X.X.X.X.X.X..
  ..XXXXXXXXXXXXX..XXXXXXXXXXXXX..
  ......X.X.X..........X.X.X......
  ..XXXXXXXXXXXXX..XXXXXXXXXXXXX..
  ..X.X.X.X.X.X.X..X.X.X.X.X.X.X..
  ..XXXXXXXXXXXXX..XXXXXXXXXXXXX..
  ..X.X.......X.X..X.X.......X.X..
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
```

```
 rem Definisce i colori per ogni linea del playfield.
 pfcolors:
 130
 26
 130
 26
 130
 26
 130
 26
 130
 26
 130
 130
end

 rem Imposta i colori, la posizione iniziale del giocatore e attiva il suono di inizio.
 COLUPF=h : COLUP0=26 : scorecolor=14
 player_x=77 : player_y=48 : h=130
 if b{7} then AUDV0=8 : AUDV1=8 : AUDC0=4 : AUDC1=4
 b{6}=0

 drawscreen

 rem Definizione grafica del giocatore
 player0:
  %00111100
  %01111110
  %11111111
  %11100000
  %11111111
  %01011110
  %00111100
end
 rem Definizione grafica del nemico (Fantasma)
 player1:
  %01010100
  %11111110
  %11111110
  %11111110
  %11010110
  %01111100
  %00111000
```

```
end


 rem Posiziona il nemico e imposta il suo colore.
 ghost_x=77 : ghost_y=80
 COLUP1 = $32


noise
 rem Gestisce la musica di inizio partita.
 if b{7} then a=a+1
 if !b{7} then c=0 : goto main
 if a>16 then a=0 : c=c+1
 if c=1 then AUDF0=4 : AUDF1=18
 if c=2 then AUDF0=12 : AUDF1=14
 if c=3 then AUDF0=4 : AUDF1=18
 if c=4 then AUDF0=8 : AUDF1=14
 if c=5 then k=0 : c=0 : goto anim
 goto beginning


 rem --- Ciclo di Gioco Principale ---
main
 rem Disattiva il flag della sequenza di inizio.
 b{7}=0


 rem Esegui la logica del fantasma
 gosub update_ghost_ai


 rem Imposta il volume del suono (k è il contatore del suono "waka-waka").
 AUDV0=k : AUDV1=0
 rem Controlla la collisione con le palline del playfield.
 if collision(player0,playfield) && g<50 then k=9 : b{5}=1 : score=score+1 : g=g+1 : pfpixel e f
off
 if !collision(player0,playfield) && b{5} then b{5}=1
 rem Gestisce il suono "waka-waka" quando si mangia.
 if b{5} then k=k-1 : AUDC0=9 : AUDF0=9
 if b{5} && k<1 then k=0 : b{5}=0
 rem Controlla se tutte le palline sono state mangiate (g=70).
 if collision(player0,playfield) && g=50 then g=0 : b{5}=0 : k=0 : b{4}=1 : c=0 : a=0 : b{3}=0 :
AUDV0=0
 if b{4} then j=j+1


 rem Gestisce l'animazione e il suono della morte del giocatore.
 if b{6} then AUDV0=k : AUDF0=m : AUDC0=4 : n=n+1
 if n=4 then k=k-1 : n=0 : o=o+1 : p=p+1
 if n=3 then m=m-1
```

```
if o>5 then k=0
if o>6 then o=0 : k=15
if b{6} && m=2 then o=0 : p=0 : k=0
if b{6} && m=2 then b{6}=0 : n=0 : c=0 : q{0}=1

rem --- Logica di Movimento su Griglia (Giocatore) ---
b{0}=0 : b{1}=0
if player_x=17 then b{1}=1
if player_x=77 then b{1}=1
if player_x=137 then b{1}=1
if player_y=16 then b{0}=1
if player_y=32 then b{0}=1
if player_y=48 then b{0}=1
if player_y=64 then b{0}=1
if player_y=80 then b{0}=1

rem Imposta i colori volatili.
COLUP0=26 : scorecolor=14
COLUP1 = $32

drawscreen

rem Se il flag q{0} è attivo, entra in pausa dopo la morte.
if q{0} then goto pause
rem Se il flag b{6} è attivo, continua l'animazione di morte.
if b{6} then goto anim_death

rem Gestisce l'animazione di movimento del giocatore.
if !b{4} then a=a+1
if a>20 then a=0
rem Gestisce la sequenza di fine livello.
if b{4} && j=20 then j=0 : i=i+1
if b{4} && i=5 then i=0 : b{4}=0 : AUDV0=0 : goto beginning
if b{4} then goto main

rem --- Gestione Collisione Giocatore-Fantasma ---
if collision(player0,player1) && !b{6} && !b{5} then k=15 : m=31 : b{6}=1 : c=0

rem --- Gestione Input Giocatore ---
if joy0left && b{0} then player_dir=1
if joy0right && b{0} then player_dir=2
if joy0up && b{1} then player_dir=3
if joy0down && b{1} then player_dir=4
```

```
 rem salta un frame se il giocatore ha mangiato una pallina
 if k = 8 then goto chomp_delay
 rem Muove il giocatore nella direzione `c` memorizzata.
 if b{2} then b{2}=0 else b{2}=1
 if b{2} && player_dir=1 && player_x>17 then player_x=player_x-1 : b{3}=1
 if b{2} && player_dir=2 && player_x<137 then player_x=player_x+1 : b{3}=0
 if b{2} && player_dir=3 && player_y>16 then player_y=player_y-1
 if b{2} && player_dir=4 && player_y<80 then player_y=player_y+1

chomp_delay
 rem Specchia lo sprite (REFP0) in base alla direzione orizzontale.
 if b{3} then REFP0=8 else REFP0=0

 rem Calcola la coordinata del playfield sotto il giocatore per cancellare le palline.
 if b{3} then e=(player_x-17)/4
 if !b{3} then e=(player_x-10)/4
 f=(player_y-1)/8

 goto anim

pause
 rem Pausa dopo la morte, prima di ricominciare.
 r=r+1 : player_x=77 : player_y=48
 if r>60 then r=0 : b{7}=1 : q{0}=0 : AUDF0=4 : AUDF1=18 : a=0 : c=1 : score = 0 : goto beginning
 goto anim

 rem ========= SUBROUTINE IA FANTASMA =========
update_ghost_ai
 rem  Non muovere il fantasma se il giocatore è morto
 if b{6} || q{0} then return
 rem  Non muovere il fantasma durante lo schema di fine livello
 if b{4} then return

 rem --- Logica di Movimento su Griglia per il Fantasma ---
 ghost_can_h=0 : ghost_can_v=0
 if ghost_x=17 then ghost_can_v=1
 if ghost_x=77 then ghost_can_v=1
 if ghost_x=137 then ghost_can_v=1
 if ghost_y=16 then ghost_can_h=1
 if ghost_y=32 then ghost_can_h=1
 if ghost_y=48 then ghost_can_h=1
 if ghost_y=64 then ghost_can_h=1
```

```
 if ghost_y=80 then ghost_can_h=1

 rem --- Logica Decisionale del Fantasma ---
 rem Se il fantasma è a un incrocio (può cambiare direzione)
 if ghost_can_h || ghost_can_v then goto update_ghost_ai2
 goto update_ghost_ai3

update_ghost_ai2
 rem Logica di Inseguimento: scegli la direzione migliore

 rem Priorità al movimento orizzontale se è la distanza maggiore
 if player_x > ghost_x then tmp1 =  player_x - ghost_x
 if player_x <= ghost_x then tmp1 =  ghost_x - player_x
 if player_y > ghost_y then tmp2 =  player_y - ghost_y
 if player_y <= ghost_y then tmp2 =  ghost_y - player_y

 if tmp1 > tmp2 then goto update_ghost_ai2c

 rem Altrimenti, priorità al movimento verticale
 if ghost_y < player_y && ghost_can_v && ghost_dir <> 1 then ghost_dir = 2
 if ghost_y > player_y && ghost_can_v && ghost_dir <> 2 then ghost_dir = 1
 if ghost_x < player_x && ghost_can_h && ghost_dir <> 3 then ghost_dir = 4
 if ghost_x > player_x && ghost_can_h && ghost_dir <> 4 then ghost_dir = 3
 goto update_ghost_ai3

update_ghost_ai2c
 if ghost_x < player_x && ghost_can_h && ghost_dir <> 3 then ghost_dir = 4
 if ghost_x > player_x && ghost_can_h && ghost_dir <> 4 then ghost_dir = 3
 if ghost_y < player_y && ghost_can_v && ghost_dir <> 1 then ghost_dir = 2
 if ghost_y > player_y && ghost_can_v && ghost_dir <> 2 then ghost_dir = 1

update_ghost_ai3
 framecounter = framecounter + 1
 rem  Muovi il fantasma solo ogni due frame
 if framecounter{0} then return

 rem --- Muove il Fantasma nella sua direzione corrente ---
 if ghost_dir = 1 && ghost_can_v then ghost_y = ghost_y - 1
 if ghost_dir = 2 && ghost_can_v then ghost_y = ghost_y + 1
 if ghost_dir = 3 && ghost_can_h then ghost_x = ghost_x - 1
 if ghost_dir = 4 && ghost_can_h then ghost_x = ghost_x + 1

 return
```

```
 rem --- Centralino Animazioni ---
anim
 rem Seleziona il set di animazioni corretto in base alla direzione di movimento.
 if c<3 then goto anim_lr
 if c=3 then goto anim_up
 if c=4 then goto anim_dn

anim_lr
 rem Animazione per il movimento orizzontale (bocca che si apre e chiude).
 if a<5 then goto frame_1
 if a>4 && a<10 then goto frame_2
 if a>9 && a<15 then goto frame_3
 if a>14 then goto frame_2
anim_up
 rem Animazione per il movimento verso l'alto.
 if a<5 then goto frame_1_up
 if a>4 && a<10 then goto frame_2_up
 if a>9 && a<15 then goto frame_3_up
 if a>14 then goto frame_2_up
anim_dn
 rem Animazione per il movimento verso il basso.
 if a<5 then goto frame_1_down
 if a>4 && a<10 then goto frame_2_down
 if a>9 && a<15 then goto frame_3_down
 if a>14 then goto frame_2_down

anim_death
 rem Seleziona il frame per l'animazione di morte.
 if p<4 then goto death_frame_1
 if p>3 && p<8 then goto death_frame_2
 if p>7 && p<12 then goto death_frame_3
 if p>11 && p<16 then goto death_frame_4
 if p>15 && p<20 then goto death_frame_5
 if p>19 && p<24 then goto death_frame_6
 if p>23 then goto death_frame_7

 rem --- Subroutine Grafiche: Animazione di Morte ---
death_frame_1
 player0:
  %0011100
  %0111110
  %1100011
```

```
  %1100011
  %1100011
  %0110110
  %0010100
end
 goto main
death_frame_2
 player0:
  %0011100
  %0011100
  %0110110
  %0110110
  %1110111
  %1110111
  %1100011
end
 goto main
death_frame_3
 player0:
  %0011100
  %0110110
  %1110111
  %1100011
  %0000000
  %0000000
  %0000000
end
 goto main
death_frame_4
 player0:
  %1111111
  %0111110
  %0000000
  %0000000
  %0000000
  %0000000
  %0000000
end
 goto main
death_frame_5
 player0:
  %0000000
  %0000000
```

```
  %0010100
  %0001000
  %0010100
  %0000000
  %0000000
end
 goto main
death_frame_6
 player0:
  %1000001
  %0100010
  %0010100
  %0000000
  %0010100
  %0100010
  %1000001
end
 goto main
death_frame_7
 player0:
  %0010100
  %0000000
  %0100010
  %0000000
  %0100010
  %0000000
  %0010100
end
 goto main

 rem --- Subroutine Grafiche: Animazione Movimento ---
frame_1_down
 player0:
  %0110110
  %1110111
  %1110111
  %1110111
  %1111111
  %0111010
  %0011100
end
 goto main
frame_2_down
```

```
 player0:
  %0100010
  %1100011
  %1110111
  %1110111
  %1111111
  %0111010
  %0011100
end
 goto main
frame_3_down
 player0:
  %0100010
  %1100011
  %1100011
  %1110111
  %1111111
  %0111010
  %0011100
end
 goto main
frame_1_up
 player0:
  %0011100
  %0111010
  %1111111
  %1110111
  %1110111
  %1110111
  %0110110
end
 goto main
frame_2_up
 player0:
  %0011100
  %0111010
  %1111111
  %1110111
  %1110111
  %1100011
  %0100010
end
 goto main
```

```
frame_3_up
 player0:
  %0011100
  %0111010
  %1111111
  %1110111
  %1100011
  %1100011
  %0100010
end
 goto main

frame_1
 player0:
  %00111100
  %01111110
  %11111111
  %11100000
  %11111111
  %01011110
  %00111100
end
 goto main

frame_2
 player0:
  %00111100
  %01111110
  %11111100
  %11100000
  %11111100
  %01011110
  %00111100
end
 goto main

frame_3
 player0:
  %00111100
  %01111100
  %11110000
  %11100000
  %11110000
```

```
  %01011100
  %00111100
end
 goto main
```

Highway Racer

```
rem ****************************************************************
rem * Highway Racer  (Corse in Autostrada)                         *
rem *                                                              *
rem * DESCRIZIONE DEL GIOCO:                                       *
rem * Un gioco di corse con visuale dall'alto e scrolling verticale. *
rem * Il giocatore controlla un'auto (player0) e deve evitare le     *
rem * auto nemiche (player1) e i posti di blocco (ball) che          *
rem * appaiono sulla strada. Il giocatore può sparare (missile0) per *
rem * distruggere le auto nemiche e ottenere punti. Anche le auto    *
rem * nemiche possono sparare (missile1). La velocità aumenta        *
rem * progressivamente. Il gioco termina quando i "danni" subiti,    *
rem * rappresentati da un contatore, raggiungono una soglia.         *
rem *                                                              *
rem * TECNICHE DI PROGRAMMAZIONE UTILIZZATE:                       *
rem * - Aritmetica a Virgola Fissa (Fixed-Point Math): Il gioco      *
rem *   include `fixed_point_math.asm` per utilizzare variabili 8.8. *
rem *   `scroll` (per lo scrolling della strada e nemici) e `mis`    *
rem *   (per il proiettile nemico) usano questa tecnica per ottenere *
rem *   un movimento e un'accelerazione fluidi e graduali.           *
rem * - Scrolling Verticale del Playfield: Il comando `pfscroll down`*
rem *   viene usato per creare l'illusione del movimento continuo    *
rem *   della strada.                                               *
rem * - IA con Comportamento Casuale: Le auto nemiche non si         *
rem *   limitano a scorrere, ma possono muoversi lateralmente e      *
rem *   "sbandare" in modo casuale grazie all'uso del comando `rand`.*
rem *   Anche i posti di blocco appaiono in posizioni casuali.       *
rem * - Gestione Dinamica degli Sprite: Le auto nemiche cambiano     *
rem *   aspetto (`CarCreate`) e colore in base al livello di         *
rem *   difficoltà. L'auto del giocatore cambia forma quando sterza. *
rem * - Sistema di "Punti Vita"/Danno: Invece di vite discrete, il   *
rem *   gioco usa un contatore di danni (`c`). Ogni collisione       *
rem *   incrementa il contatore. Al raggiungimento di una soglia,    *
rem *   si attiva la sequenza di Game Over.                         *
rem * - Oggetti Multipli e Power-up: Il gioco gestisce 4 oggetti     *
rem *   mobili contemporaneamente: l'auto del giocatore (player0),   *
rem *   l'auto nemica (player1), il proiettile del giocatore         *
rem *   (missile0) e il proiettile nemico (missile1). C'è anche una  *
rem *   meccanica di power-up (invincibilità temporanea).            *
rem ****************************************************************

set romsize 4k
```

```
init
 rem Include la libreria per la matematica a virgola fissa (8.8).
 include fixed_point_math.asm

 rem --- Sezione Definizioni Variabili (Alias) ---
 rem 'scroll' (m.n) è la variabile 8.8 per la posizione Y dell'auto nemica.
 dim scroll=m.n
 m=0 : n=0
 scroll=1.0
 rem 'mis' (k.j) è la variabile 8.8 per la posizione Y del missile nemico.
 dim mis=k.j
 j=0 : k=0
 mis=1.0

 rem --- Mappa delle Variabili (Commenti originali mantenuti) ---
 rem a: Posizione X dell'auto nemica per calcoli
 rem b: Posizione X del posto di blocco (ball)
 rem c: Contatore dei danni subiti dal giocatore
 rem d: Flag per il movimento laterale dell'auto nemica (0=dritto, 1=dx, 2=sx)
 rem f: Contatore generico
 rem h: Flag per power-up attivo
 rem i: Flag per la schermata titolo
 rem o: Contatore generico
 rem p: Posizione Y del giocatore
 rem q: Timer per il proiettile del giocatore
 rem r: Timer per la pausa (es. game over)
 rem t: Contatore di scrolling (aumenta la difficoltà)
 rem u: Posizione X dell'auto nemica
 rem w: Flag per il colore (power-up)
 rem x: Posizione X del giocatore
 rem y: Contatore per il ciclo dei colori
 rem z: Flag per nemico colpito
 a=0 : b=82 : c=0 : d=0 : f=0 : h=0 : i=0 : o=0
 p=85 : q=0 : r=0 : t=0 : u=94 : W=0 : x=75 : y=16 : z=0

 rem --- Impostazioni Iniziali Oggetti e Colori ---
 missile0x=0:missile0y=0
 missile1x=0:missile1y=0
 COLUP0=0
 COLUP1=208
 COLUPF=160
 COLUBK=0
```

```
 CTRLPF=$35
 scorecolor=246
 AUDV0=0
 AUDV1=0

 rem --- Stato 0: Schermata Titolo ---
intro
 rem Fa scorrere lo sfondo per un effetto dinamico.
 g=g+1
 if g=2 then pfscroll down :g=0
 rem Cicla i colori per un effetto "attract mode".
 y=y+1
 if y<17 then y=16
 if y>29 then y=16

 rem Disegna i bordi della strada per la schermata titolo.
 pfpixel 7 1 on : pfpixel 25 1 on
 pfpixel 7 2 on : pfpixel 25 2 on
 pfpixel 7 3 on : pfpixel 25 3 on
 pfpixel 7 4 on : pfpixel 25 4 on
 pfpixel 7 5 on : pfpixel 25 5 on
 pfpixel 7 6 on : pfpixel 25 6 on
 pfpixel 7 7 on : pfpixel 25 7 on
 pfpixel 7 8 on : pfpixel 25 8 on
 pfpixel 7 9 on : pfpixel 25 9 on
 drawscreen

 rem Attende l'input del giocatore o il reset per iniziare il gioco.
 if joy0down then i=1 : score=0 : g=0:i=0:goto intro2
 if switchreset then i=1 : score=0 : g=0:i=0:goto intro2
 if joy0fire then i=1: score=0 : g=0:i=0:goto intro2
 goto intro

 rem --- Inizializzazione della Partita ---
intro2
 rem Posiziona le auto per l'inizio.
 player1x=68:player1y=82
 player0x=88:player0y=83
 rem Genera la prima auto nemica.
 gosub MakeNewCar1
 rem Azzera i contatori di gioco.
 g=0
 e=0
```

```
 missile0x=0:missile0y=0


 rem --- Ciclo di Gioco Principale ---
main
 y=y+1
 if y>250 then y=1
 if c<1 then c=1


 rem Attiva il suono del motore del giocatore se non sta accelerando/frenando.
 if joy0up then goto audskip
 if joy0down then goto audskip
 AUDF0=18:AUDC0=14:AUDV0=10
 rem Aumenta la difficoltà massima dopo un certo tempo.
 if t>240 then t=31


audskip
 rem Resetta i flag quando l'auto nemica esce dallo schermo.
 if scroll > 90 then o=0
 if scroll > 96 then w=0 : h=0
 if scroll > 96 then u=94


 rem --- Logica dello Scrolling Verticale (Virgola Fissa) ---
 rem 'scroll' è la posizione Y dell'auto nemica.
 e=e+1
 if e=2 && t < 10 then e=0:goto skipscroll
 rem Incrementa la posizione verticale. Se esce dallo schermo, la resetta e aumenta la difficoltà
 ('t').
 if scroll < 97 then scroll=scroll+1.0 else scroll=0.0 : t=t+1
 rem Aumenta la velocità di scrolling ai livelli più alti.
 if scroll < 97 && t > 35 then scroll=scroll+0.9 : mis=mis+0.9


skipscroll
 rem Controlla se è necessario creare una nuova auto nemica.
 gosub CarCreate


 rem --- Logica IA e Generazione Ostacoli ---
 v=rand
 rem Ai livelli più alti, l'auto nemica può "sbandare" casualmente.
 if t > 30 then skipmv
 if t < 8 then goto skipmv
 if t > 20 && v < 35 then u=u+1


skipmv
```

```
 rem Genera casualmente un posto di blocco (ball) in una delle 8 posizioni.
 if v=2 && scroll > 86 then b = 75
 if v=234 && scroll > 86 then b = 105
 if v=112 && scroll > 86 then b = 89
 if v=50 && scroll > 86 then b = 81
 if v=188 && scroll > 86 then b = 115
 if v=166 && scroll > 86 then b = 79
 if v=132 && scroll > 86 then b = 95
 if v=176 && scroll > 86 then b = 111

 rem Fa muovere l'auto nemica lateralmente in modo casuale.
 if v < 10 then u=u+1
 if v > 245 then u=u-2

 rem Cambia il colore dell'auto nemica in base al livello di difficoltà.
 if t>20 && t<36 then COLUP1=104
 if t>35 then COLUP1=68

 rem Se l'auto nemica ha colpito un bordo, la fa muovere nella direzione opposta.
 if d=1 then u=u+1
 if d=2 then u=u-1

skiplr
 rem Se un'auto nemica colpita da un missile esce dallo schermo, resetta il flag 'z'.
 if scroll > 96 then z=0
 rem Se il flag 'z' è 0, l'auto nemica è visibile (NUSIZ1=1).
 if z=0 then NUSIZ1=$01
 rem Aggiorna la posizione dell'auto nemica usando la variabile a virgola fissa.
 player1y=scroll : player1x=u

skiplrm
 rem Seleziona lo sprite del giocatore in base alla sterzata.
 if joy0left then goto TurnCar1
 if joy0right then goto TurnCar2
 player0:
 %01111110
 %01000010
 %00111100
 %10100101
 %11100111
 %10111101
 %00111100
 %10011001
```

```
 %11111111
 %10011001
end
turned
 rem --- Gestione Grafica e Scrolling ---
 rem Fa scorrere la strada.
 if joy0down then skipscrl
 pfscroll down
skipscrl
 g=g+1
 if g=2 then pfscroll down:g=0

skipsc
 rem Imposta il colore del giocatore, che cambia in base ai danni subiti ('c').
 if c < 11 then COLUP0=128
 if c > 10 && c < 21 then COLUP0=60
 if c > 20 && c < 31 then COLUP0=30
 if c > 30 && c < 41 then COLUP0=64
 if c > 40 && c < 61 then COLUP0=y
 player0x=x : player0y=p

 rem Disegna i bordi della strada usando pfpixel.
 pfpixel 7 1 on : pfpixel 25 1 on
 pfpixel 7 3 on : pfpixel 25 3 on
 pfpixel 7 5 on : pfpixel 25 5 on
 pfpixel 7 7 on : pfpixel 25 7 on
 pfpixel 7 9 on : pfpixel 25 9 on
 drawscreen

 rem Posiziona il posto di blocco (ball).
 ballx=b : bally=scroll+15 : ballheight=2

 rem --- Gestione Input Giocatore (Movimento) ---
 if joy0left then x=x-1
 if joy0right then x=x+1
 rem Accelerare e frenare modifica la velocità di scrolling e il punteggio.
 if joy0up then scroll=scroll+0.3 : mis=mis+0.5 : score=score+10 : AUDF0=12:AUDC0=14:AUDV0=10
 if joy0down && scroll >=1 then scroll=scroll-0.3 : mis=mis-0.5 : score=score-10: AUDF0=24:AUDC0=
14:AUDV0=10

 rem --- Logica dei Proiettili ---
 rem Gestisce il proiettile del giocatore.
 if joy0fire && q<1 then AUDF1=8:AUDC1=1:AUDV1=15 : goto playerfires
```

```
 if q>0 then q=q-2 : missile0y=q
 if q>50 then AUDV1=0
 rem Gestisce il proiettile dell'auto nemica.
 if scroll >35 && t<10 then goto fireskip
 if scroll >50 && t<36 then goto fireskip
 if scroll >60 && t>35 then goto fireskip
 if h=1 then goto fireskip
 if scroll<=1 then mis=1.0 : AUDF1=13:AUDC1=1:AUDV1=9
 if mis > 18 then AUDV1=0
 missile1y=mis : missile1x=u : missile1height=6
 mis=mis+2.0
 goto pskip
fireskip
 missile1y=0: missile1x=0

pskip
 rem --- Logica delle Collisioni ---
 rem Collisione giocatore-bordo strada.
 if collision(playfield,player0) && x > 75 then x=x-2
 if collision(playfield,player0) && x < 75 then x=x+2
 rem Collisione nemico-bordo strada (cambia la sua direzione).
 if collision(player1,playfield) && u > 100 then d=2
 if collision(player1,playfield) && u < 80 then d=1
 rem Rallenta il giocatore se tocca il bordo.
 if collision(player0,playfield) && scroll >=1 then scroll=scroll-0.5 : mis=mis-0.5
 rem Collisione giocatore-nemico: aumenta i danni. Se troppi, game over.
 if collision(player1,player0) && w=1 then gosub addhitpoints
 if w=1 then goto damageskip
 if collision(player1,player0) then c=c+1 : if c=60 then r=120: goto thisisit
 if collision(player1,player0) && scroll >=1 then scroll=scroll-1.5 : mis=mis-1.5
 rem Collisione missile del giocatore-nemico.
 if collision(player1,missile0) then missile0y=1 : goto carhit
 rem Collisione missile nemico-giocatore.
 if collision(player0,missile1) && x > 75 then c=c+1 : x=x-2 : if c=60 then r=160: goto thisisit
 if collision(player0,missile1) && x < 75 then c=c+1 : x=x+2 : if c=60 then r=160: goto thisisit
damageskip
 rem Collisione giocatore-posto di blocco.
 if collision(player0,ball) && u > 90 then x=x-6
 if collision(player0,ball) && u < 90 then x=x+6

 rem Aumenta il punteggio per il tempo sopravvissuto.
 score=score+20
 goto main
```

```
 rem --- Sezione delle Subroutine ---
playerfires
 rem Crea il proiettile del giocatore.
 if !switchleftb then missile0x=x+10 else missile0x=x+4
 q=80 : missile0y=75
 missile0height=6
 goto main
thisisit
 rem Prepara la schermata di Game Over.
 AUDV0=0
 goto eog
carhit
 rem Gestisce l'evento di un'auto nemica colpita.
 a=u
 z=1 : score = score + 1000 : missile0y=0 : q=0
 a=u+8
 if o<1 then goto skipblank
 if o>0 then goto blankcar
skipblank
 rem Fa scomparire l'auto nemica e può generare un power-up.
 if missile0x < a then NUSIZ1=0 : o=o+1 : u=u+16
 if missile0x > a then NUSIZ1=0 : o=o+1
199 l=rand:if l>215 then 199
 l=l/8:l=l+1
 if joy0up then goto main
 if l < 10 && scroll > 1 then gosub powerup : w=1
 goto main
eog
 rem Sequenza di animazione e suono per il Game Over.
 if r<1 then r=88:goto eog2
 r=r-1
 gosub explode
 COLUPF=r
 AUDF0=160-r:AUDC0=1:AUDV0=6
 drawscreen
 goto eog
eog2
 rem Loop finale che attende il riavvio del gioco.
 COLUP0=68
 AUDV0=0
 r=r-1
 e=e+1
```

```
 if e=2 then pfscroll up:e=0
 player0y=r
 player1y=0
 missile0y=0
 missile1y=0
 COLUPF=160
 drawscreen
 if r<1 then pfclear:goto init
 ballx=0:bally=0
 scroll=0
 t=0
 goto eog2
CarCreate
 rem Seleziona casualmente uno dei 5 modelli di auto nemiche da creare.
 if scroll<96 then return
200 l=rand:if l>215 then 200
 l=l/8:l=l+1
 if l>1 && l<6 then gosub MakeNewCar1 : return
 if l>5 && l<11 then gosub MakeNewCar2 : return
 if l>10 && l<16 then gosub MakeNewCar3 : return
 if l>15 && l<21 then gosub MakeNewCar4 : return
 if l>20 && l<28 then gosub MakeNewCar5 : return
 return

 rem --- Subroutine Grafiche ---
MakeNewCar1
 player1:
 %10011001
 %11111111
 %10011001
 %00011000
 %10111101
 %11111111
 %10011001
 %00111100
end
 return
MakeNewCar2
 player1:
 %10111101
 %11111111
 %10111101
 %00100100
```

```
 %11100111
 %10111101
 %01000010
 %01111110
end
 return
MakeNewCar3
 player1:
 %10011001
 %11111111
 %10011001
 %00011000
 %11011011
 %11111111
 %11011011
 %00100100
end
 return
MakeNewCar4
 player1:
 %10011001
 %11111111
 %10011001
 %00011000
 %00011000
 %10011001
 %11111111
 %10011001
end
 return
MakeNewCar5
 player1:
 %10011001
 %11111111
 %10011001
 %00111100
 %10100101
 %11111111
 %10011001
 %00100100
end
 return
TurnCar1
```

```
 rem Sprite del giocatore che sterza a sinistra.
 player0:
 %01111110
 %01000010
 %00111100
 %10100101
 %11100111
 %10111101
 %00111100
 %01011001
 %11111111
 %10011010
end
 goto turned
TurnCar2
 rem Sprite del giocatore che sterza a destra.
 player0:
 %01111110
 %01000010
 %00111100
 %10100101
 %11100111
 %10111101
 %00111100
 %10011010
 %11111111
 %01011001
end
 goto turned
powerup
 rem Attiva il power-up e cambia lo sprite nemico in quello del power-up.
 h=1
 missile0y=0: q=0
 player1:
 %11111111
 %10000001
 %10011001
 %10011001
 %10111101
 %10111101
 %10011001
 %10011001
 %10000001
```

```
 %11111111
end
 return
addhitpoints
 rem Se il power-up è attivo, il giocatore non subisce danni.
 if h=1 then c=c-1
 return
blankcar
 rem Fa scomparire l'auto nemica dopo essere stata colpita.
 u=94
 player1:
 %00000000
end
 goto main
explode
 rem Sprite per l'animazione di esplosione del giocatore.
 COLUPF=70
 COLUP0=64
 player0:
 %01111110
 %01000010
 %00111100
 %10100101
 %11110111
 %00111000
 %10001100
 %10011001
 %01101110
 %00000000
end
 return
 player0:
 %10000001
 %00100100
 %00000000
 %00011001
 %10011000
 %00000000
 %00100100
 %10000001
End
```

Disc Dog

```
 rem ************************************************************
 rem * Disc Dog                                                 *
 rem *                                                          *
 rem * DESCRIZIONE DEL GIOCO:                                   *
 rem * Un gioco ispirato allo sport del "disc dog". Il giocatore      *
 rem * controlla un cane (player0) che deve prendere al volo un frisbee*
 rem * (player1) prima che cada a terra. Il gioco è a tempo e si      *
 rem * perdono vite se il frisbee non viene preso.                    *
 rem *                                                          *
 rem * TECNICHE DI PROGRAMMAZIONE UTILIZZATE:                   *
 rem * - IA dell'Oggetto: Il frisbee segue una traiettoria complessa  *
 rem *   e la sua velocità cambia casualmente.                    *
 rem * - Animazione Dinamica: Lo sprite del cane cambia in base alla  *
 rem *   direzione del movimento.                                 *
 rem * - Manipolazione del Playfield: Vite e timer sono disegnati     *
 rem *   manualmente sullo sfondo con `pfpixel`.                  *
 rem * - Gestione degli Stati: Il codice usa variabili per tracciare  *
 rem *   lo stato del cane (salto, corsa) e del disco (preso, in volo).*
 rem ************************************************************

 rem  Etichetta di inizio gioco, usata per il riavvio completo.
begin

 rem --- Definizioni Grafiche Iniziali ---
 rem  Disegna 3 blocchi per le 3 vite iniziali
 rem  Marcatori di bordo per il campo
 rem  Linea di terra
 playfield:
 ................................
 .X.X.X..........................
 ................................
 ................................
 ................................
 ................................
 ................................
 ................................
 ................................
 ...............................X
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
end
```

```
 rem  Grafica iniziale del cane (fermo)
 player0:
 %01000100
 %01000100
 %01111100
 %01111100
 %01111100
 %10000111
 %00000111
 %00000100
end

 rem  Grafica del frisbee
 player1:
 %01111110
 %00011000
end

 rem --- Impostazioni Iniziali dei Registri ---
 rem  Colore del playfield (es. l'erba)
 COLUPF = 176
 rem  Colore del punteggio (non usato ma impostato)
 scorecolor = 52
 score = 0

 rem  Posizione iniziale del cane
 player0x = 21
 player0y = 80

 rem  Posizione iniziale del frisbee
 player1x = 138
 player1y = 65

 rem  Posizione iniziale dell'ombra del cane
 missile0x = 82
 missile0y = 79
 missile0height = 5

 rem --- Alias delle Variabili (mappatura su a-z) ---
 rem  Stato del salto del cane (0=a terra, 1=sale, 2=scende)
 dim perrosalto = a
 rem  Direzione del frisbee (1=da dx a sx, 2=da sx a dx)
 dim discodireccion = b
```

```
 rem  Stato del frisbee (1=in volo, 2=preso dal cane)
 dim discocogido = c
 rem  Punteggio (non usato, ma c'è la variabile)
 dim puntos = d
 rem  Direzione in cui è rivolto il cane (1=dx, 2=sx)
 dim perrodireccion = e
 rem  Velocità orizzontale del frisbee
 dim discovelocidad = f
 rem  Variabile temporanea per valori casuali
 dim aleatorio = g
 rem  Altezza massima dell'arco del frisbee
 dim discoaltura = h
 rem  Contatore delle vite del cane
 dim perrovidas = i
 rem  Timer di gioco (conto alla rovescia disegnato su schermo)
 dim cuentaatras = j
 rem  Coordinata X in cui il frisbee inizia a scendere
 dim discoalturasube = k
 rem  Coordinata X in cui il frisbee inizia a salire
 dim discoalturabaja = l
 rem  Timer per rallentare il movimento verticale del frisbee
 dim discoalturapaso = m

 rem --- Inizializzazione delle Variabili di Gioco ---
 puntos = 0
 discocogido = 1
 discodireccion = 1
 perrodireccion = 1
 discovelocidad = 2
 perrovidas = 3
 cuentaatras = 0
 rem  Calcola un'altezza casuale per il lancio
 discoaltura = (rand & 10) + 1
 discoalturasube = 18 + discoaltura
 discoalturabaja = 138 - discoaltura
 discoalturapaso = 4

 rem  Ciclo di gioco principale.
mainloop

 rem  Silenzia il canale audio 0 all'inizio di ogni frame.
 AUDV0 = 0
 rem  Colore del cane (player0)
```

```
COLUP0 = 4
rem  Colore del frisbee (player1)
COLUP1 = 132

rem --- Logica di Game Over ---
rem  Se le vite sono finite, cambia colore e pulisce lo schermo.
if perrovidas = 0 then COLUPF = 52 : gosub limpiarpantalla
rem  Disegna la scritta "GAME"
if perrovidas = 0 then gosub game
rem  Disegna la scritta "OVER"
if perrovidas = 0 then gosub over
rem  Se il gioco è finito e si preme fuoco, riavvia tutto.
if perrovidas = 0 && joy0fire then goto begin

rem --- Lettura Input Giocatore ---
if joy0left && player0x > 21 && perrovidas > 0 then gosub moverizquierda
if joy0right && player0x < 133 && perrovidas > 0 then gosub moverderecha
rem  Inizia la sequenza di salto
if joy0up && perrosalto = 0 && perrovidas > 0 then perrosalto = 1

rem --- Aggiornamento Timer Visivo ( disegna una barra che si riempie ) ---
if cuentaatras = 25 then pfpixel 21 1 on : missile0height = 6
if cuentaatras = 50 then pfpixel 22 1 on
if cuentaatras = 75 then pfpixel 23 1 on : missile0height = 7
if cuentaatras = 100 then pfpixel 24 1 on
if cuentaatras = 125 then pfpixel 25 1 on : missile0height = 8
if cuentaatras = 150 then pfpixel 26 1 on
if cuentaatras = 175 then pfpixel 27 1 on : missile0height = 9
if cuentaatras = 200 then pfpixel 28 1 on
if cuentaatras = 225 then pfpixel 29 1 on : missile0height = 10
if cuentaatras = 250 then pfpixel 30 1 on

rem  Se il timer arriva alla fine, il gioco termina.
if cuentaatras = 250 then perrovidas = 0

rem --- Logica di Stato del Frisbee ---
rem  Se il cane ha il disco, lo tiene; altrimenti, muovi il disco.
if discocogido = 2 then gosub cogerdisco else gosub moverdisco

rem --- Logica del Salto del Cane ---
rem  Se sta saltando (fase di salita)
if perrosalto = 1 then gosub saltarsubida
rem  Se sta saltando (fase di discesa)
```

```
 if perrosalto = 2 then gosub saltarbajada


 rem --- Controllo Collisioni ---
 rem  Se il cane tocca il frisbee, lo prende.
 if collision(player0,player1) then discocogido = 2


 rem Se il cane con il frisbee raggiunge uno dei bordi, lancia di nuovo.
 if player0x = 21 && collision(playfield,player1) && discocogido = 2 then gosub lanzardisco2
 if player0x = 133 && collision(playfield,player1) && discocogido = 2 then gosub lanzardisco1


 rem Se il cane tocca la sua ombra (missile0), perde una vita (è una meccanica di gioco per aggiu
ngere difficoltà).
 if collision(player0,missile0) then AUDV0 = 15 : AUDC0 = 6 : AUDF0 = 4 : player0x = 21 : perrovi
das = perrovidas - 1


 rem --- Aggiornamento Vite Visive ---
 rem  Spegne un blocco-vita se ne rimangono 2.
 if perrovidas = 2 then pfpixel 5 1 off
 rem  Spegne un altro blocco-vita se ne rimane 1.
 if perrovidas = 1 then pfpixel 3 1 off
 rem  Spegne l'ultimo blocco-vita.
 if perrovidas = 0 then pfpixel 1 1 off


 drawscreen
 goto mainloop


 rem  Subroutine per lanciare il frisbee da destra verso sinistra.
lanzardisco1
 rem  Pulisce vecchi pixel del playfield
 pfpixel 31 9 off
 pfpixel 0 9 on
 rem  Suono di lancio
 AUDV0 = 5 : AUDC0 = 12 : AUDF0 = 4
 rem  Imposta lo stato del disco a "in volo".
 discocogido = 1
 player1y = 65
 player1x = 18
 rem  Imposta la direzione del disco.
 discodireccion = 2
 rem  Calcola una velocità casuale.
 aleatorio = (rand & 3) + 1
 if aleatorio = 4 then discovelocidad = 4
 if aleatorio = 3 then discovelocidad = 2
 if aleatorio = 2 then discovelocidad = 1
```

```
 if aleatorio = 1 then discovelocidad = 1
 score = score + 100
 puntos = puntos + 100
 cuentaatras = cuentaatras + 1
 rem  Calcola una nuova traiettoria casuale.
 discoaltura = (rand & 10) + 1
 discoalturasube =  138 - (discoaltura * 4)
 discoalturabaja = 18 + (discoaltura * 4)
 discoalturapaso = 4
 return

 rem  Subroutine per lanciare il frisbee da sinistra verso destra.
lanzardisco2
 rem  Pulisce vecchi pixel del playfield
 pfpixel 31 9 on
 pfpixel 0 9 off
 rem  Suono di lancio
 AUDV0 = 5 : AUDC0 = 12 : AUDF0 = 4
 discocogido = 1
 player1y = 65
 player1x = 138
 discodireccion = 1
 rem  Calcola una velocità casuale.
 aleatorio = (rand & 3) + 1
 if aleatorio = 4 then discovelocidad = 4
 if aleatorio = 3 then discovelocidad = 2
 if aleatorio = 2 then discovelocidad = 1
 if aleatorio = 1 then discovelocidad = 1
 score = score + 100
 puntos = puntos + 100
 cuentaatras = cuentaatras + 1
 rem  Calcola una nuova traiettoria casuale.
 discoaltura = (rand & 10) + 1
 discoalturabaja = 18 + (discoaltura * 4)
 discoalturasube = 138 - (discoaltura * 4)
 discoalturapaso = 4
 return

 rem  Cambia la grafica del cane per il movimento a sinistra.
moverizquierda
 player0:
 %00100010
 %00100010
```

```
 %00111110
 %00111110
 %00111111
 %11100000
 %11100000
 %00100000
end
 rem  Imposta la direzione del cane.
 perrodireccion = 2
 rem  Muove il cane.
 player0x = player0x - 1
 return

 rem  Cambia la grafica del cane per il movimento a destra.
moverderecha
 player0:
 %01000100
 %01000100
 %01111100
 %01111100
 %01111100
 %10000111
 %00000111
 %00000100
end
 player0x = player0x + 1
 perrodireccion = 1
 return

 rem  Fase di salita del salto.
saltarsubida
 player0y = player0y - 1
 rem  Se raggiunge l'apice, passa alla fase di discesa.
 if player0y = 62 then perrosalto = 2
 rem  Meccanica di penalità
 if puntos >= 10 && perrosalto = 2 then score = score - 10 : puntos = puntos - 10
 return

 rem  Fase di discesa del salto.
saltarbajada
 player0y = player0y + 1
 rem  Se tocca terra, fine del salto.
 if player0y = 80 then perrosalto = 0
```

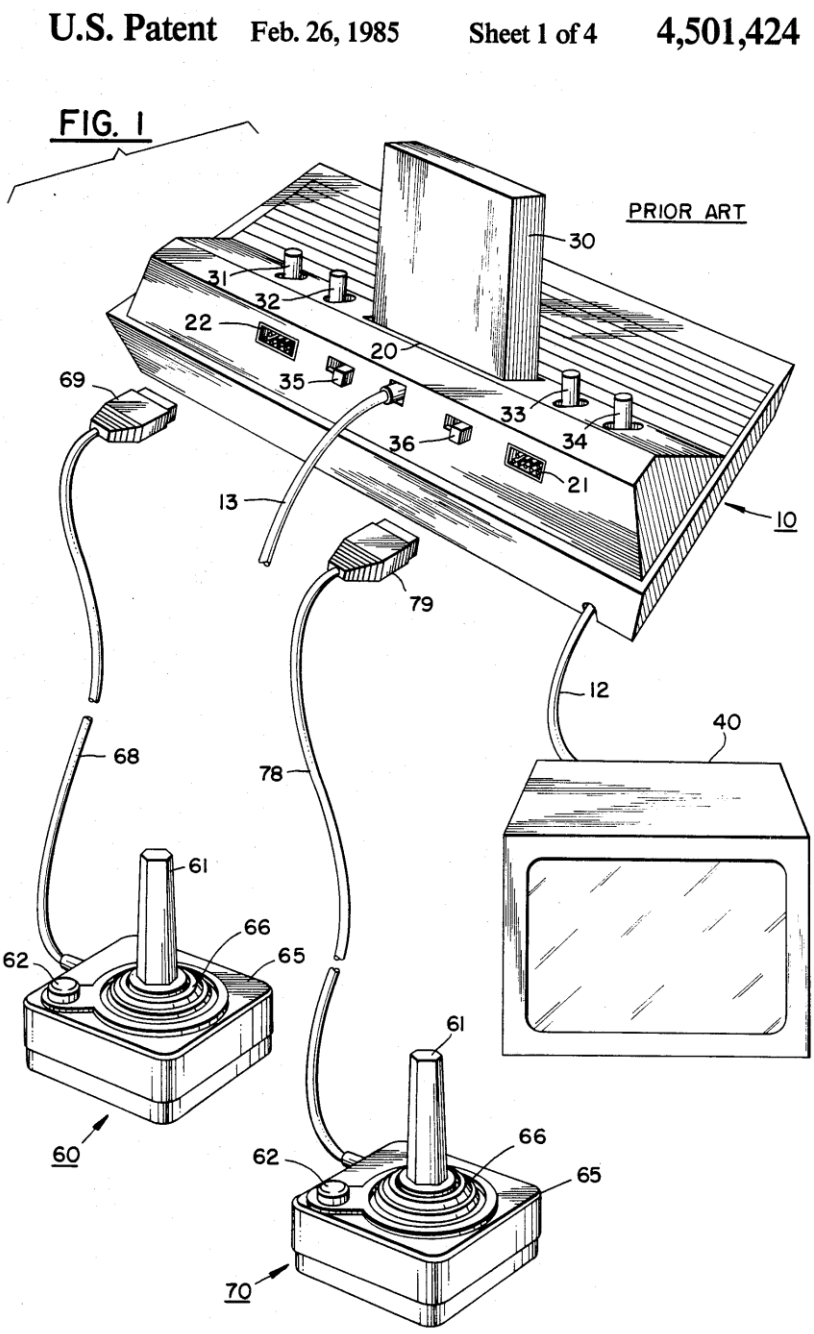
```
 return


 rem  Logica di movimento del frisbee in volo.
moverdisco
 rem  Rallenta il movimento verticale
 if discoalturapaso > 1 then discoalturapaso = discoalturapaso - 1


 rem --- Simulazione della traiettoria parabolica del frisbee ---
 if discoalturapaso = 1 && player1x <= discoalturabaja && discovelocidad = 1 && discodireccion =
1 then player1y = player1y + 1 : discoalturapaso = 4
 if discoalturapaso = 1 && player1x >= discoalturasube && discovelocidad = 1 && discodireccion =
1 then player1y = player1y - 1 : discoalturapaso = 4


 if discoalturapaso = 1 && player1x >= discoalturasube && discovelocidad = 1 && discodireccion =
2 then player1y = player1y + 1 : discoalturapaso = 4
 if discoalturapaso = 1 && player1x <= discoalturabaja && discovelocidad = 1 && discodireccion =
2 then player1y = player1y - 1: discoalturapaso = 4


 rem --- Movimento orizzontale e inversione ai bordi ---
 if player1x <= 138 && discodireccion = 1 then player1x = player1x - discovelocidad
 if player1x >= 18 && discodireccion = 2 then player1x = player1x + discovelocidad
 rem  Se tocca il bordo, inverte e aggiorna il timer.
 if player1x <= 18 then discodireccion = 2 : cuentaatras = cuentaatras + 1 : player1y = 65
 if player1x >= 138 then discodireccion = 1 : cuentaatras = cuentaatras + 1 : player1y = 65


 AUDV0 = 0
 return


 rem  Logica per quando il cane ha preso il frisbee.
cogerdisco
 rem  Il frisbee segue il cane.
 if perrodireccion = 1 then player1x = player0x + 6
 if perrodireccion = 2 then player1x = player0x - 6
 player1y = player0y - 5
 return


 rem  Pulisce i pixel usati per l'HUD.
limpiarpantalla
 pfpixel 5 1 off
 pfpixel 3 1 off
 pfpixel 1 1 off
 pfpixel 22 1 off
 pfpixel 23 1 off
 pfpixel 24 1 off
 pfpixel 25 1 off
```

```
 pfpixel 26 1 off
 pfpixel 27 1 off
 pfpixel 28 1 off
 pfpixel 29 1 off
 pfpixel 30 1 off
 pfpixel 31 1 off
 drawscreen
 return

 rem  Disegna "GAME" sul playfield.
game
 pfpixel 6 0 on
 pfpixel 7 0 on
 pfpixel 8 0 on
 pfpixel 11 0 on
 pfpixel 12 0 on
 pfpixel 13 0 on
 pfpixel 15 0 on
 pfpixel 19 0 on
 pfpixel 21 0 on
 pfpixel 22 0 on
 pfpixel 23 0 on
 pfpixel 24 0 on

 pfpixel 5 1 on
 pfpixel 10 1 on
 pfpixel 13 1 on
 pfpixel 15 1 on
 pfpixel 16 1 on
 pfpixel 18 1 on
 pfpixel 19 1 on
 pfpixel 21 1 on

 pfpixel 5 2 on
 pfpixel 7 2 on
 pfpixel 8 2 on
 pfpixel 10 2 on
 pfpixel 13 2 on
 pfpixel 15 2 on
 pfpixel 17 2 on
 pfpixel 19 2 on
 pfpixel 21 2 on
 pfpixel 22 2 on
```

```
 pfpixel 5 3 on
 pfpixel 8 3 on
 pfpixel 10 3 on
 pfpixel 11 3 on
 pfpixel 12 3 on
 pfpixel 13 3 on
 pfpixel 15 3 on
 pfpixel 19 3 on
 pfpixel 21 3 on

 pfpixel 6 4 on
 pfpixel 7 4 on
 pfpixel 10 4 on
 pfpixel 13 4 on
 pfpixel 15 4 on
 pfpixel 19 4 on
 pfpixel 21 4 on
 pfpixel 22 4 on
 pfpixel 23 4 on
 pfpixel 24 4 on
 drawscreen
 return

 rem  Disegna "OVER" sul playfield.
over
 pfpixel 6 6 on
 pfpixel 7 6 on
 pfpixel 10 6 on
 pfpixel 14 6 on
 pfpixel 16 6 on
 pfpixel 17 6 on
 pfpixel 18 6 on
 pfpixel 19 6 on
 pfpixel 21 6 on
 pfpixel 22 6 on
 pfpixel 23 6 on
 pfpixel 24 6 on

 pfpixel 5 7 on
 pfpixel 8 7 on
 pfpixel 10 7 on
 pfpixel 14 7 on
```

```
pfpixel 16 7 on
pfpixel 21 7 on
pfpixel 24 7 on

pfpixel 5 8 on
pfpixel 8 8 on
pfpixel 10 8 on
pfpixel 14 8 on
pfpixel 16 8 on
pfpixel 17 8 on
pfpixel 21 8 on
pfpixel 22 8 on
pfpixel 23 8 on

pfpixel 5 9 on
pfpixel 8 9 on
pfpixel 11 9 on
pfpixel 13 9 on
pfpixel 16 9 on
pfpixel 21 9 on
pfpixel 23 9 on

pfpixel 4 10 off
pfpixel 5 10 off
pfpixel 8 10 off
pfpixel 9 10 off
pfpixel 10 10 off
pfpixel 11 10 off
pfpixel 13 10 off
pfpixel 14 10 off
pfpixel 15 10 off
pfpixel 20 10 off
pfpixel 22 10 off
pfpixel 23 10 off
pfpixel 25 10 off
drawscreen
return
```

Parte 4: Appendici



Schema tecnico Atari 2600 — circuito e controller, 1983 (George C. Stone & Stuart E. Ross), dominio pubblico.

Appendice A: I Pilastri del Codice – Sintassi e Operatori

Questa appendice è il tuo "cheat sheet" fondamentale. Contiene le regole d'oro della sintassi e i concetti base per "parlare" direttamente con la macchina. Quando hai un dubbio su come strutturare il codice, su cosa sia un $ o su come funzionano gli operatori, questa è la prima pagina da aprire.

1. Struttura del Codice e Indentazione

La posizione di una riga di codice ne determina la funzione. Un errore di indentazione è la causa più comune di problemi di compilazione.

Elemento	Posizione	Esempio
Etichetta (Label)	**Colonna 0** (nessuno spazio prima)	*main_loop:*
end	**Colonna 0** (nessuno spazio prima)	*end*
Istruzione/Comando	**Indentata** (almeno uno spazio)	*player0x = 80*
Dati Binari	**Indentati** (almeno uno spazio)	*%11111111*
Commento (rem o ;)	**Indentato** (almeno uno spazio)	*rem Questo è un commento*

Batari Basic è molto pignolo sulla spaziatura. Se il compilatore ti dà un errore "Illegal token", la prima cosa da controllare è sempre l'indentazione. Assicurati che le etichette e gli *end* siano in colonna 0 e che tutto il resto sia indentato.

2. Binario ed esadecimale

Noi contiamo in base 10, ma i computer "pensano" in modi diversi. Per programmare l'Atari, ne useremo principalmente due.

- **Binario (%): è** il linguaggio fondamentale della macchina. Un **bit** è un singolo interruttore: 0 (spento) o 1 (acceso). Un **byte** è un gruppo di 8 bit. Usiamo il prefisso % per scrivere in binario, specialmente per la grafica. *Esempio: %01100110* è un byte con i bit 1, 2, 5 e 6 accesi. Il bit 0 è quello "più a destra", il bit 7 è quello "più a sinistra".
- **Esadecimale ($): s**crivere lunghi numeri binari è scomodo. Per questo, i programmatori usano un sistema in base 16. Usa 16 "cifre": i numeri da 0 a 9, più le lettere da A a F per rappresentare i valori da 10 a 15. Usiamo il prefisso $ per i numeri esadecimali. È un modo compatto per scrivere i valori dei registri, specialmente per i colori. *Esempio: COLUBK = $8E*

Come si converte un numero esadecimale come $8E?
Un numero esadecimale a due cifre, come $XY, è semplicemente una somma. La prima cifra (X) va moltiplicata per 16, e la seconda (Y) va sommata al risultato. Ricorda che le lettere A,B,C,D,E,F corrispondono ai numeri 10,11,12,13,14,15.
Prendiamo $8E: 1. La prima cifra è 8. 2. La seconda cifra è E, che in decimale vale 14. 3. La formula è: (8 * 16) + 14 = 142.
Quindi, *COLUBK = $8E* è la stessa cosa di *COLUBK = 142*

Con questo metodo, puoi decifrare velocemente alcuni valori chiave:
$00 = (0 * 16) + 0 = **0** (il valore minimo di un byte).
$FF = (15 * 16) + 15 = 240 + 15 = **255** (il valore massimo di un byte).

Come si "vede" un numero decimale in binario?
Il processo inverso, da decimale a binario, è ancora più utile per la programmazione grafica. Si tratta di trovare quali potenze del 2, sommate insieme, danno il tuo numero.
Immagina di voler rappresentare il numero **166** in binario. Parti dalla potenza del 2 più alta (128) e scendi, chiedendoti: "Ci sta?".

- Il **128** ci sta in 166? **Sì**. Restano 166 - 128 = 38. → Bit 7 = **1**
- Il **64** ci sta in 38? No. → Bit 6 = **0**
- Il **32** ci sta in 38? **Sì**. Restano 38 - 32 = 6. → Bit 5 = **1**
- Il **16** ci sta in 6? No. → Bit 4 = **0**
- L'**8** ci sta in 6? No. → Bit 3 = **0**
- Il **4** ci sta in 6? **Sì**. Restano 6 - 4 = 2. → Bit 2 = **1**
- Il **2** ci sta in 2? **Sì**. Restano 2 - 2 = 0. → Bit 1 = **1**
- L'**1** ci sta in 0? No. → Bit 0 = **0**

Mettendo insieme i bit da 7 a 0, otteniamo: %10100110.
Questa tecnica ti permette di "pensare in binario" e di capire immediatamente quali bit (e quindi quali pixel in uno sprite) sono accesi o spenti in un dato valore numerico.

3. Operatori Matematici e Logici

Operatore	Nome	Esempio	Descrizione
+	Addizione	a = 5 + 3	Somma due numeri.
-	Sottrazione	a = 5 – 3	Sottrae un numero da un altro.
*****	Moltiplicazione	a = 5 * 3	Moltiplica due numeri.
/	Divisione Intera	a = 5 / 2	Divide due numeri e **scarta il resto** (il risultato è 2).
()	Parentesi	a = (5 + 3) * 2	Forza l'ordine delle operazioni. Le espressioni tra parentesi vengono calcolate per prime.
&&	AND Logico (E)	if a > 0 && b > 0	Restituisce "vero" solo se **entrambe** le condizioni sono vere.
||	OR Logico (OPPURE)	if a > 0 || b > 0	Restituisce "vero" se **almeno una** delle condizioni è vera.
!	NOT Logico (NON)	if !joy0fire	Inverte il valore di una condizione (vero diventa falso, falso diventa vero).

4. Operatori Bitwise

Questi operatori ti permettono di manipolare i singoli bit all'interno di un byte. Sono strumenti avanzati per ottimizzazioni estreme.

Operatore	Nome	Esempio	Descrizione
&	AND Bitwise	a = a & %00001111	Mantiene solo i bit che sono 1 in entrambi i valori ("maschera").
|	OR Bitwise	a = a | %00000001	"Accende" un bit (lo imposta a 1) senza modificare gli altri.
^	XOR Bitwise	a = a ^ %00000001	Inverte lo stato di un bit (da 0 a 1 o viceversa, "flip").
<<	Shift a Sinistra	a = a << 1	Sposta tutti i bit a sinistra. Equivale a moltiplicare per 2. **Molto veloce!** Se ad esempio vuoi moltiplicare per 5 (4+1) velocemente, puoi fare: a = (a << 2) + a
>>	Shift a Destra	a = a >> 1	Sposta tutti i bit a destra. Equivale a dividere per 2. **Molto veloce!**

Appendice B: Il Cruscotto dell'Atari – Guida ai Registri e alle Variabili Speciali

Questa appendice è il tuo manuale tecnico per "parlare" direttamente con l'hardware dell'Atari 2600. Conoscerli ti darà il pieno controllo.

1. Gerarchia di Visibilità degli Oggetti (Ordine di Disegno)

Sull'Atari 2600, gli oggetti vengono disegnati su strati fissi, come fogli di acetato trasparenti impilati uno sull'altro. L'ordine di base, dal più lontano al più vicino, è:

Sfondo (COLUBK) → Playfield (playfield) / Palla (ball) → Player 1 / Missile 1 → Player 0 / Missile 0

Questo significa che, di default, player0 apparirà sempre sopra a player1. Questo ordine può essere alterato con il registro CTRLPF.

2. Tabella Completa dei Registri e Variabili Speciali

Questa tabella è il tuo riferimento rapido per le parole chiave che controllano la grafica e l'audio. Ricordati che **volatile** significa che dopo un *drawscreen* il suo valore è azzerato e che quindi dobbiamo ripristinarlo ad ogni frame (nel main_loop).

Nome Registro/Variabile	Scopo Breve	Capitolo	Volatile?	Note / Valori Comuni
Colori (Registri TIA)				
COLUBK	Colore di Sfondo	2	No	$00 (nero) - $FE (bianco). Valori esadecimali.
COLUPF	Colore Playfield e Palla	4	**Sì**	Valori esadecimali. Va reimpostato ad ogni frame.
COLUP0	Colore Player 0 e Missile 0	2	**Sì**	Valori esadecimali. Va reimpostato ad ogni frame.
COLUP1	Colore Player 1 e Missile 1	7	**Sì**	Valori esadecimali. Va reimpostato ad ogni frame.
Posizioni (Registri TIA)				
player0x, player0y	Posizione Sprite Player 0	2	No	x: 0-159 (circa), y: 0-95 (circa).
player1x, player1y	Posizione Sprite Player 1	7	No	x: 0-159 (circa), y: 0-95 (circa).
missile0x, missile0y	Posizione Missile 0	8	No	x: 0-159 (circa), y: 0-95 (circa).
missile1x, missile1y	Posizione Missile 1	8	No	x: 0-159 (circa), y: 0-95 (circa).

Nome Registro/Variabile	Scopo Breve	Capitolo	Volatile?	Note / Valori Comuni
ballx, bally	Posizione Palla	7	No	x: 0-159 (circa), y: 0-95 (circa).
Controllo Grafico (Registri TIA)				
REFP0, REFP1	Riflessione Orizzontale Sprite	3	**Sì**	0 (normale), 8 (specchiato).
NUSIZ0, NUSIZ1	Dimensione/Copie Sprite/Missili	8	**Sì**	Sintassi $MP. M (missile): 0-3 (1-8px). P (player): 5 (doppio), 7 (quadruplo).
CTRLPF	Controllo Palla e Priorità	4, 8	No	Somma di valori: bit 2=4 (priorità), bit 4-5 (16, 32, 48) per larghezza palla.
missile0height, missile1height	Altezza Missili (in pixel)	8	No	Valori 1-8. 1 è usato per oggetti orizzontali.
ballheight	Altezza Palla (in pixel)	8	No	Valori 1-8.
Input Joystick 0 (Comandi RIOT)				
joy0fire	Lettura pulsante fuoco Giocatore 1	3	N/A	Restituisce vero/falso in un if.
joy0up	Lettura direzione SU Giocatore 1	3	N/A	Restituisce vero/falso in un if.
joy0down	Lettura direzione GIÙ Giocatore 1	3	N/A	Restituisce vero/falso in un if.
joy0left	Lettura direzione SINISTRA Giocatore 1	3	N/A	Restituisce vero/falso in un if.
joy0right	Lettura direzione DESTRA Giocatore 1	3	N/A	Restituisce vero/falso in un if.
Input Joystick 1 (Comandi RIOT)				
joy1fire	Lettura pulsante fuoco Giocatore 2	3	N/A	Restituisce vero/falso in un if.
joy1up	Lettura direzione SU Giocatore 2	3	N/A	Restituisce vero/falso in un if.
joy1down	Lettura direzione GIÙ Giocatore 2	3	N/A	Restituisce vero/falso in un if.
joy1left	Lettura direzione SINISTRA Giocatore 2	3	N/A	Restituisce vero/falso in un if.

Nome Registro/Variabile	Scopo Breve	Capitolo	Volatile?	Note / Valori Comuni
joy1right	Lettura direzione DESTRA Giocatore 2	3	N/A	Restituisce vero/falso in un if.
Input Interruttori Console (Comandi RIOT)				
switchreset	Lettura interruttore GAME RESET	3	N/A	Restituisce vero se premuto.
Switchselect	Lettura interruttore GAME SELECT	3	N/A	Restituisce vero se premuto.
Switchbw	Lettura interruttore COLOR/B&W	3	N/A	Restituisce vero se in posizione B&W.
Switchleftb	Lettura interruttore DIFF. SINISTRO	3	N/A	Restituisce vero se in posizione B (beginner).
switchrightb	Lettura interruttore DIFF. DESTRO	3	N/A	Restituisce vero se in posizione B (beginner).
Audio (Registri TIA)				
AUDV0, AUDV1	Volume Canali 0 e 1	5	No	0 (silenzio) - 15 (massimo).
AUDC0, AUDC1	Timbro (tipo di suono)	5	No	0-15. Vedi Appendice D per la tabella.
AUDF0, AUDF1	Frequenza (intonazione)	5	No	0 (acuto) - 31 (grave).
HUD (Punteggio - Variabili Speciali)				
score	Variabile punteggio a 6 cifre	14	No	Formato BCD da 0 a 999999.
scorecolor	Colore del testo dello score	14	No	Valori esadecimali.
const noscore = 1	Nasconde lo score	14	N/A	Costante da definire a inizio codice.
const scorefade = 1	Attiva effetto sfumato	14	N/A	Costante da definire a inizio codice.
set pfscore on	Attiva le barre di stato	14	N/A	Direttiva da inserire a inizio codice.
pfscorecolor	Colore delle barre di stato	14	No	Valori esadecimali.
pfscore1, pfscore2	Dati binari per le barre	14	No	Valori in binario (es. %11110000).

Nota: N/A = non attinente

3. Moltiplicare gli Oggetti: Trucchi con NUSIZ e CTRLPF

Hai imparato a disegnare e muovere i tuoi cinque oggetti grafici. Ma se osservi i giochi classici, vedrai cose che sembrano impossibili: racchette da tennis larghe, proiettili che sono più spessi di un normale missile, o addirittura più copie dello stesso giocatore sullo schermo. Come è possibile?
La risposta non sta nel creare nuovi oggetti, ma nell'alterare quelli esistenti usando due dei registri di controllo più potenti del TIA: **NUSIZ** e **CTRLPF**. In questa appendice, impareremo a moltiplicare, allargare e allungare i nostri attori digitali.

I registri *NUSIZ0* (per *player0/missile0*) e *NUSIZ1* (per *player1/missile1*) sono speciali. Ognuno è un singolo byte, ma il TIA lo interpreta come due metà separate (due "nybble" da 4 bit) che controllano due cose diverse:

- **I bit "a destra":** Controllano il **Player** (*player0* o *player1*).
- **I bit "a sinistra":** Controllano il **Missile** (*missile0* o *missile1*).

Per impostarli, usiamo un singolo numero esadecimale $MP, dove M è il valore per il Missile e P è il valore per il Player.

Controllare i Missili (la parte M)

La parte M del registro controlla semplicemente la **larghezza** del missile.

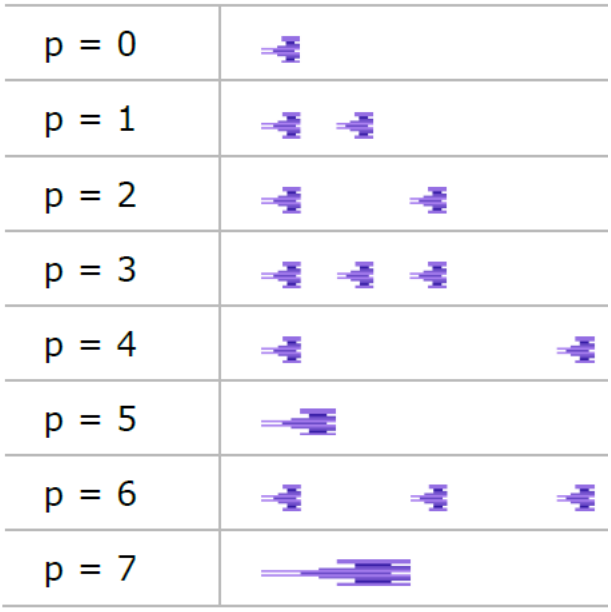
Valore di M	Impostazione	Larghezza Missile
0	NUSIZx = $0_	1 pixel (default)
1	NUSIZx = $1_	2 pixel
2	NUSIZx = $2_	4 pixel
3	NUSIZx = $3_	8 pixel

Questo è il trucco che abbiamo usato per creare la "spada" orizzontale! Con *NUSIZ0 = $30*, abbiamo impostato la larghezza di *missile0* a 8 pixel.

Controllare i Player (la parte P)

La parte P è molto più interessante. Permette di **moltiplicare** o **allargare** lo sprite del giocatore.

Valore di P	Impostazione	Effetto sul Player
0	NUSIZx = $_0	1 copia, larghezza normale (default)
1	NUSIZx = $_1	2 copie, vicine
2	NUSIZx = $_2	2 copie, a media distanza
3	NUSIZx = $_3	3 copie, vicine
4	NUSIZx = $_4	2 copie, a lunga distanza
5	NUSIZx = $_5	**Doppia larghezza (x2)**
6	NUSIZx = $_6	3 copie, a media distanza
7	NUSIZx = $_7	**Quadrupla larghezza (x4)**



Ricorda, *NUSIZ0* e *NUSIZ1* sono **volatili**! Devono essere reimpostati ad ogni frame nel main_loop se vuoi che il loro effetto sia persistente. Quando imposti un valore, ad esempio *NUSIZ0 = $35*, stai impostando **contemporaneamente** la larghezza del missile (M=3) e l'effetto sul player (P=5).

E per la ball? Non ha un registro NUSIZ dedicato. La sua larghezza è controllata da due bit all'interno del registro **CTRLPF**, lo stesso che usiamo per la priorità.
A differenza di NUSIZ, CTRLPF **non è volatile**. Di solito lo si imposta una volta all'inizio del gioco.

Valore per CTRLPF	Larghezza Palla
0	1 pixel (default)
16	2 pixel
32	4 pixel
48	8 pixel

Poiché CTRLPF controlla più cose, i suoi valori vanno combinati. Se vuoi una palla larga 4 pixel (32) e vuoi che il Playfield abbia la priorità sugli sprite (4), imposterai CTRLPF alla loro somma: CTRLPF = 32 + 4 ; Risultato: 36

Padroneggiare NUSIZ e CTRLPF ti permette di superare i limiti grafici apparenti della console. Puoi creare sprite imponenti, effetti visivi interessanti e oggetti che si adattano meglio alle necessità del tuo gioco, trasformando i 5 oggetti base in un arsenale grafico molto più versatile.

Appendice C: Ricette di Codice Avanzate

Questa appendice contiene "ricette" di codice complete e funzionanti per alcune delle tecniche di programmazione più potenti e utili.

1. Il Centralino Veloce: on...gosub e on...goto

Hai una macchina a stati con molti stati diversi (es. diversi tipi di nemici) e una lunga catena di *if* sta rallentando il tuo main loop?
La soluzione è usare le istruzioni *on...gosub o on...goto* per creare un "centralino" velocissimo che smista l'esecuzione alla subroutine o all'etichetta corretta in base al valore di una variabile.

2. on...gosub

Questa versione è ideale quando ogni blocco di codice deve terminare con un return per tornare al punto di chiamata. Attenzione: dopo *on* segue solo una variabile, non puoi usare espressioni come "x+1" o "x+y".

```
  dim enemy_type = a ; 0=Goomba, 1=Koopa, 2=Beetle

main_loop
  ; ... logica del gioco ...

  on enemy_type gosub goomba_ai, koopa_ai, beetle_ai ; in base al valore di enemy_type il program
ma va ad una subroutine diversa

  ; ... resto del main loop ...
  drawscreen
  goto main_loop

goomba_ai ; arrivo qui se enemy_type è 0
  COLUBK = $D8 : return
koopa_ai  ; arrivo qui se enemy_type è 1
  COLUBK = $9E : return
beetle_ai ; arrivo qui se enemy_type è 2
  COLUBK = $88 : return
```

3. on...goto

Questa versione è utile quando ogni blocco di codice deve poi proseguire verso una parte comune del programma, usando goto invece di return.

```
  dim enemy type = a ; 0=Goomba, 1=Koopa, 2=Beetle

main_loop
  ; ... logica del gioco ...

  rem --- Centralino IA ---
  on enemy_type goto goomba_ai, koopa_ai, beetle_ai

continue_logic
  ; ... logica comune che prosegue dopo la scelta del nemico ...

  drawscreen
  goto main_loop

goomba_ai
  COLUBK = $D8 : goto continue logic
koopa_ai
  COLUBK = $9E : goto continue logic
beetle_ai
  COLUBK = $88 : goto continue_logic
```

4. Gestione dei numeri casuali

Il comando *rand* produce una sequenza di numeri che, se non diversamente specificato, è sempre la stessa a ogni avvio del gioco. Una possibile soluzione è usare il tempo che il giocatore passa nella schermata del titolo come "seme" (seed) per il generatore di numeri casuali. Infatti per "cambiare" la generazione della sequenza di numeri casuali basta assegnare a *rand* un numero diverso ogni volta.

```
  dim randseed = k
  dim gamestate = f ; 1=Titolo, 2=Gioco

  gamestate = 1

main_loop
  if gamestate = 1 then gosub state_title
  if gamestate = 2 then gosub state gameplay
  drawscreen
  goto main loop

state_title
  ; ...logica della schermata del titolo...

  rem Il contatore 'randseed' aumenta finché siamo nel titolo
  randseed = randseed + 1
  if !joy0fire then goto state_title

  rem Usa il contatore come seme. Per forzare il seme, si scrive rand = seme
  if randseed = 0 then rand = 1 ; Il seme 0 non è valido, quindi usiamo 1 in questo caso.
  if randseed <> 0 then rand = randseed
  gamestate = 2
  return

state gameplay
  ; ...logica di inizializzazione del gioco...
  player0x = rand ; Questa posizione sarà diversa a ogni partita!
  ; ...
```

randseed è un contatore che aumenta finchè il giocatore non preme fuoco e poiché questo momento è "casuale", lo sarà anche *randseed* il cui valore viene poi assegnato a *rand* che da lì in poi genererà un sequenza casuale diversa.

5. Range casuali

Per generare numeri casuali all'interno di un intervallo specifico, la chiave è l'operatore bitwise **& (AND)**. È una tecnica estremamente veloce ed efficiente.

Codice	Range del Risultato
a = (rand & 1)	0 o 1
a = (rand & 3)	da 0 a 3
a = (rand & 7)	da 0 a 7
a = (rand & 15)	da 0 a 15
a = (rand & 31)	da 0 a 31
a = (rand & 63)	da 0 a 63
a = (rand & 127)	da 0 a 127

Per ottenere un range che parte da 1, basta aggiungere 1 al risultato (es. *a = (rand & 3) + 1* per un range da 1 a 4).

6. Posizionamento Casuale e Intelligente degli Sprite

Invece di generare un numero e poi controllarlo con un if, possiamo usare la tecnica dei range per generare direttamente un numero nell'intervallo desiderato.
Esempio: Posizionare un nemico tra le coordinate X=21 e X=131

1. **Offset di Partenza:** Il nostro numero minimo è 21.
2. **Ampiezza del Range:** 131 - 21 = 110. Dobbiamo generare un numero casuale da 0 a 110.
3. **Scomposizione in Potenze di 2:** 110 è 64 + 32 + 8 + 4 + 2. Per mascherare i bit corrispondenti, useremo i valori 63 (per i primi 6 bit), 31 (5 bit), 15 (4 bit) e 1 (1 bit). La combinazione più efficiente è scomporre 110 come 63 + 31 + 15 + 1.
4. **Costruzione della Formula:**

   player1x = (rand & 63) + (rand & 31) + (rand & 15) + (rand & 1) + 21

7. Generare -1 o +1 Casualmente

Per decidere casualmente una direzione (positiva o negativa), si può usare un trucco con il complemento a due.

```
rem Genera un valore casuale che sarà -1 oppure +1
a = 255 + (rand & 2)
```

1. **(rand & 2)** può dare solo 0 o 2.
2. Se il risultato è 0: a = 255 + 0 → a = 255 (che per la CPU equivale a **-1**).
3. Se il risultato è 2: a = 255 + 2 → a = 257. A causa dell'overflow (superamento del limite 255), il risultato diventa 1.

Questa tecnica è perfetta per invertire casualmente una velocità: *velocita_x = velocita_x * a*

8. Le Variabili temp: La Memoria "Usa e Getta"

A volte, all'interno di una singola subroutine, hai bisogno di una variabile "di servizio" solo per un breve calcolo, ma hai già usato tutte le lettere a-z per dati importanti del gioco.
In tal caso puoi usare le **variabili temporanee**. Batari Basic mette a disposizione 6 variabili speciali chiamate *temp1, temp2, temp3, temp4, temp5 e temp6.*
Le variabili temp sono estremamente **volatili**. Il loro contenuto viene **cancellato dopo ogni drawscreen** e può essere sovrascritto da molti comandi interni di Batari Basic (specialmente calcoli complessi). Usa le variabili temp solo per calcoli molto brevi all'interno di una singola subroutine e non fare mai affidamento sul fatto che il loro valore si mantenga tra un ciclo e l'altro del *main_loop*. Esiste anche una variabile *temp7*, ma è riservata al meccanismo di bankswitching. **Non usarla mai!**

9.Gli Array data: Archivi di Informazioni nella ROM

A volte hai bisogno di conservare una lista di valori che non cambiano mai, come le posizioni di partenza dei nemici, una sequenza di colori o i dati di un livello.

In questo caso puoi usare un **array data**. Un array data è una tabella memorizzata nella ROM (memoria di sola lettura) da cui puoi leggere qualsiasi elemento in qualsiasi momento, usando la sua posizione (indice). Per creare un array data, si usa una sintassi a blocco:

data <nome_array> ... end

Per leggere un valore, si usa la sintassi

<nome_array>[indice]

dove l'indice parte da 0.

Questo esempio crea una tabella di 7 colori e la usa per far ciclare il colore dello sfondo ogni volta che si preme il pulsante di fuoco.

```
  rem Ciclo Colori con Array data
  set romsize 2k
  set smartbranching on

  dim color_index = a

main_loop
  if joy0fire then color_index = color_index + 1

  rem Se l'indice supera la dimensione dell'array, lo azzera
  if color_index > 6 then color_index = 0

  rem Leggi il colore dall'array e assegnalo allo sfondo
  COLUBK = palette_colors[color_index]

  drawscreen
  goto main_loop

  rem --- Definisci la tua tavolozza di colori in un array data ---
  data palette_colors
  $1E, $48, $86, $9C, $D4, $EA, $34
end
```

Ad ogni pressione del fuoco, *color_index* viene incrementato. Il nuovo valore dell'indice viene usato per "pescare" un colore dall'array *palette_colors*, che viene poi assegnato a *COLUBK*. Essendo memorizzati in ROM, non puoi modificare i valori di un array data durante il gioco (es. *palette_colors[0] = $FF* non funzionerà). Inoltre un singolo array data non può contenere più di 256 valori. Infine, se cerchi di leggere un indice che non esiste (es. *palette_colors[10]*), il programma non darà errore, ma leggerà "spazzatura" dalla memoria, con risultati imprevedibili.

10. Le “Comb Lines” e la Maschera Nera

A volte possono apparire delle sottili linee nere frastagliate a sinistra: le **“Comb Lines”**. La CPU è troppo impegnata a riposizionare gli sprite e “perde la corsa contro il raggio” per un istante. Una possibile soluzione è disegnare una colonna verticale nera sopra di esse usando il registro **PF0** per nasconderle, i cui primi 4 bit permettono di riempire le 4 colonne più a sinistra dello schermo.

```
main_loop
  COLUBK = $1E ; Sfondo giallo
  COLUPF = $00 ; Playfield nero
  PF0 = %11110000 ; riempi le 4 colonne a sinistra
  ; ... logica del gioco ...
  drawscreen
  goto main_loop
```

Attenzione: se usi PF0 con un playfield multicolore (pfcolors), il pixel più in basso della barra PF0 potrebbe assumere il colore sbagliato. Per risolvere, se il tuo playfield ha 11 righe, definisci **12** colori nel blocco pfcolors:, assicurandoti che il dodicesimo colore sia **identico** all’undicesimo.

Un esempio di comb lines

11. Eliminare le Linee Nere del Playfield con no_blank_lines

Per ottenere uno sfondo solido e continuo, puoi eliminare le linee di separazione tra le righe del Playfield usando un’opzione speciale del kernel.

```
set kernel_options no_blank_lines
```

Attenzione: **perdi completamente l’uso di missile0**. Questa opzione è compatibile con poche altre opzioni del kernel. Inoltre, se usi *no_blank_lines* insieme a *pfcolors*, noterai che ogni riga del Playfield avrà una sottile “glassa” in cima, colorata con il colore della riga *precedente*. Puoi sfruttare questo problema a tuo vantaggio per creare sfondi dall’aspetto quasi ad “alta risoluzione”, con linee sottili per disegnare griglie o altri dettagli.

12. Aritmetica BCD e score

La variabile score è speciale: usa un formato numerico chiamato BCD (*Binary-Coded Decimal*). Batari Basic offre un comando apposito per l'aritmetica BCD: *dec*.

Usa *dec* per aggiungere o sottrarre valori allo *score*. I numeri che aggiungi devono però essere in formato esadecimale, ma "pensati" come decimali, e non possono superare $99. Ad esempio, per aggiungere 15 punti, bisogna scrivere:

```
dec score = score + $15
```

Usando *dec* puoi anche sommare il contenuto di una variabile, scrivendone anche per essa i valori in esadecimale:

```
p = $15
dec score = score + p
```

score è internamente composto da tre byte di memoria: *sc1*, *sc2* e *sc3*, che possono rappresentare ognuno un numero da 0 a 99.

Quando *sc3* supera $99 ritorna a $00 e viene incrementato *sc2*. Quando *sc2* supera $99 ritorna a $00 e viene incrementato *sc1*. Quando *sc1, sc2, sc3* vanno sotto zero, diventano $99!

Ognuno di questi byte contiene due cifre decimali in formato BCD (Binary-Coded Decimal), ovvero ogni cifra da 0 a 9 è codificata in 4 bit:

sc1: Cifra delle Centinaia di Migliaia + Cifra Decine di Migliaia

sc2: Cifra delle Migliaia + Cifra delle Centinaia

sc3: Cifra delle Decine + Cifra delle Unità

Si può agire separatamente su *sc1, sc2, sc3* con questo codice:

```
dim _sc1 = score  ; Crea Alias per il byte più significativo dello score.
dim _sc2 = score+1; Crea Alias per il byte centrale.
dim _sc3 = score+2; Crea Alias per il byte meno significativo.
```

Essendo normali variabili byte, possiamo poi usare if per controllare un certo stato del punteggio. Ad esempio:

```
if _sc1 = $00 && _sc2 = $00 && _sc3 < $10 then ... ; il punteggio è minore di 10
if _sc1 = $99 && _sc2 = $99 && _sc3 <= $99 then ... ; il punteggio è andato sotto 0
```

Appendice D: La Sala del Compositore – Guida ai Suoni e alle Note

Benvenuto nella sala del compositore! In questa sezione troverai tutto ciò che ti serve per dare una voce ai tuoi giochi. L'Atari 2600 ha un sistema sonoro semplice ma sorprendentemente versatile, capace di creare dai *beep* iconici di *Space Invaders* ai complessi rombi di motore di *River Raid*.

1.I Registri del Suono (le Manopole del Sintetizzatore)

Il chip TIA ha due canali audio indipendenti (Canale 0 e Canale 1). Per ogni canale, devi regolare tre "manopole" (registri) per produrre un suono. Tutti i registri audio sono **persistenti**: una volta impostati, continueranno a produrre suono finché non li modificherai o non azzererai il volume.

Registro	Scopo	Range Valori	Note
AUDV0 / AUDV1	**Volume**	0 - 15	Controlla la potenza del suono. 0 è silenzio, 15 è il volume massimo. **È l'unico modo per spegnere un suono.**
AUDC0 / AUDC1	**Timbro**	0 - 15	Controlla la "voce" o la "texture" del suono. Ogni valore seleziona un tipo di suono diverso.
AUDF0 / AUDF1	**Frequenza**	0 - 31	Controlla l'intonazione (la nota). **Attenzione:** valori **bassi** = suoni **acuti**; valori **alti** = suoni **gravi**.

2.La Scelta dello Strumento (il Registro AUDC)

Il registro AUDC è il cuore creativo del suono Atari. Ogni valore seleziona un "timbro" o "strumento" diverso. Scegliere lo strumento giusto è il primo passo per comporre la tua melodia o il tuo effetto sonoro.

Timbro (AUDC)	Strumento / Descrizione	Uso Tipico
4, 5, 12, 13	**Tono Puro (Flauto)**: Pulito e rotondo. I valori 12 e 13 raggiungono note più gravi.	Melodie, suoni di raccolta oggetti, effetti positivi. Il più musicale tra i timbri.
6, 10	**Tono Intermedio**: Un suono a metà tra il puro e il ronzante.	Suoni di avviso, allarmi non troppo aggressivi .
7, 9	**Tono "Ancia"**: Aspro, brillante e penetrante.	Motori, allarmi acuti, suoni aggressivi.
1	**Tono "Buzzy"**: Ronzante, distorto e molto elettronico.	Laser, spari, suoni stridenti e fantascientifici.
3	**Tono "UFO"**: Fluttuante, modulato, quasi un lamento.	Sirene, effetti speciali, suoni alieni.
2, 14, 15	**Rombi e Distorsioni**: Suoni complessi con bassi profondi che si trasformano in rombi.	Motori potenti, suoni cupi, impatti pesanti.
8	**Rumore Bianco**: Un fruscio puro, simile al suono di una TV non sintonizzata.	Esplosioni, spari, vento, onde del mare.
0, 11	Silenzio	-

3.La Partitura: Tavola Completa delle Note (il Registro AUDF)

Il registro AUDF controlla l'intonazione. Ricorda sempre la regola d'oro: **più basso è il valore, più acuta è la nota**. A causa del modo in cui l'hardware genera i suoni, non tutte le note sono perfettamente intonate. La tabella seguente elenca le note più "pure" e utilizzabili per ogni strumento, coprendo quasi 5 ottave.

Come leggere la tabella: Cerca la nota desiderata. La colonna "Valore AUDF" ti dà il numero da usare. La colonna "Strumenti Migliori" indica con quali timbri (AUDC) quella nota suona più intonata.

Ottava	Nota	Valore AUDF	Strumenti Migliori (AUDC)
Ottava 1 (la più alta)	Do	3	1
	Do#	0, 1	1, 2, 3
	Fa	2	1, 2, 3
Ottava 2	Do	7, 15	1, 7, 9
	Do#	6, 14	1, 6, 10
	Re	13	1, 6, 10
	Mi	11, 12	2, 3, 6, 10, 12, 13
	Fa	11, 29	1, 12, 13
	Fa#	10	3, 6, 10
	Sol	9, 19, 20	3, 7, 9
	Sol#	9, 19	1, 3, 4, 5, 6, 10
	La	8, 18	4, 5
	La#	8, 17	1, 2, 3, 4, 5, 12, 13
	Si	16	1, 2, 3, 4, 5, 12, 13
Ottava 3 (centrale)	Do	29	4, 5, 12, 13
	Do#	28	6, 10
	Re	27	2, 3, 6, 10
	Re#	26	12, 13
	Mi	25	2, 3
	Fa	23, 29	12, 13
	Fa#	22	2, 3, 12, 13
	Sol	21	6, 10, 12, 13
	Sol#	20, 24	12, 13
	La	18	4, 5, 6, 10, 12, 13
	La#	17, 22	12, 13
	Si	16, 21, 30	12, 13
Ottava 4 (bassa)	Do	29	12, 13
	Re	27	12, 13
	Mi	25, 31	12, 13
	Fa	23, 29	12, 13
	Fa#	22, 27	12, 13
	Sol	20, 26	12, 13
	Sol#	19, 24	12, 13
	La	18, 23	12, 13
	La#	17, 22	12, 13

Ottava	Nota	Valore AUDF	Strumenti Migliori (AUDC)
	Si	16, 21, 30, 31	12, 13
Ottava 5 (la più grave)	Do	29	12, 13
	Re	27	12, 13
	Mi	31	12, 13
	Fa	29	12, 13
	Sol	26	12, 13

Come puoi vedere, le note non sono distribuite in modo uniforme. A volte, la stessa nota si ottiene con valori AUDF diversi, e alcune note semplicemente non esistono per certi timbri. Comporre per l'Atari 2600 è come suonare un vecchio organo: bisogna conoscere lo strumento e adattare la melodia alle sue peculiarità, scegliendo le note e i timbri che funzionano meglio insieme.

4.Il Motore Musicale: Creare Melodie con sdata

I "Sound Timer" sono perfetti per effetti sonori brevi, ma come si fa a creare una colonna sonora complessa? La soluzione è costruire un **motore musicale**, una piccola macchina software che legge una "partitura" dalla memoria e la suona.
Le normali tabelle *data* sono come array e sono limitate a circa 256 byte. Per una canzone, non bastano.
sdata (*Sequential Data*) crea un **flusso di dati** che può essere letto solo in sequenza, uno dopo l'altro, come leggere le parole di un libro. Questo permette di creare tabelle di dati grandi quanto l'intera memoria ROM. Ecco come funziona:

sdata music_data = x

sdata: Dichiara l'inizio di una tabella di dati sequenziali.
music_data: È il nome che diamo alla nostra "playlist".
= x: Questo è il pezzo cruciale. Stiamo dicendo a Batari Basic di usare la variabile x come **puntatore** (si può usare qualsiasi variabile a..z)

y = sread(music_data)

sread: È il comando per **leggere il prossimo dato** dalla tabella.
Ogni volta che chiami sread(), lui legge il valore a cui punta x, lo assegna a y, e poi **incrementa automaticamente x**, spostando il puntatore al valore successivo della tabella di valori, pronto per la prossima lettura.

Nell'esempio che segue la subroutine *music_setup* serve a riposizionare questo "segnalibro" all'inizio del "libro" ogni volta che la canzone finisce.
Attenzione: i dati di *sdata* si possono solo leggere in sequenza a differenza dell'array data da cui puoi leggere qualsiasi elemento in qualsiasi momento, usando la sua posizione (indice).

```
rem Motore Musicale con sdata
set romsize 2k

dim music_note_duration = a
dim mus1 = b
dim mus2 = c
```

```
  dim mus3 = d

  rem --- Inizializzazione ---
  gosub music setup

main_loop
  gosub music_play
  drawscreen
  goto main_loop

music_play
  music_note_duration = music_note_duration - 1
  if music_note_duration > 0 then return ; Se la nota non è finita, esci

  rem --- La nota e' finita, leggi la prossima dalla tabella ---
  rem Formato dati: Volume, Timbro, Frequenza, Durata

  mus1 = sread(music_data) ; Leggi il Volume
  rem Se il volume è 255, la canzone è finita. Riavvolgi.
  if mus1 = 255 then gosub music setup : return

  mus2 = sread(music data) ; Leggi il Timbro
  mus3 = sread(music_data) ; Leggi la Frequenza
  music_note_duration = sread(music_data) ; Leggi la Durata

  rem Imposta i registri audio per suonare la nuova nota
  AUDV0 = mus1
  AUDC0 = mus2
  AUDF0 = mus3
  return

music_setup
  rem --- TABELLA DATI MUSICALE ---
  rem Scegli la variabile x come puntatore alla tabella musicale
  rem Vol, Timbro, Freq, Durata
  sdata music_data = x
  12, 4, 28, 15 ; Do
  12, 4, 25, 15 ; Re
  12, 4, 22, 15 ; Mi
  12, 4, 28, 30 ; Do (lunga)
  0, 0, 0, 15 ; Pausa

  12, 4, 28, 15 ; Do
  12, 4, 25, 15 ; Re
  12, 4, 22, 15 ; Mi
  12, 4, 28, 30 ; Do (lunga)
  0, 0, 0, 15 ; Pausa

  12, 4, 22, 15 ; Mi
  12, 4, 21, 15 ; Fa
  12, 4, 18, 30 ; Sol (lunga)
  0, 0, 0, 15 ; Pausa

  12, 4, 22, 15 ; Mi
  12, 4, 21, 15 ; Fa
  12, 4, 18, 30 ; Sol (lunga)
  0, 0, 0, 15 ; Pausa

  12, 4, 18, 15 ; Sol
  12, 4, 16, 15 ; La
  12, 4, 18, 15 ; Sol
  12, 4, 21, 15 ; Fa
  12, 4, 22, 15 ; Mi
  12, 4, 28, 30 ; Do (lunga)
  0, 0, 0, 15 ; Pausa

  12, 4, 18, 15 ; Sol
  12, 4, 16, 15 ; La
  12, 4, 18, 15 ; Sol
  12, 4, 21, 15 ; Fa
  12, 4, 22, 15 ; Mi
```

```
  12, 4, 28, 30 ; Do (lunga)
  0, 0, 0, 15 ; Pausa

  12, 4, 28, 15 ; Do
  12, 4, 30, 15 ; Sol (basso)
  12, 4, 28, 30 ; Do (lunga)
  0, 0, 0, 15 ; Pausa

  12, 4, 28, 15 ; Do
  12, 4, 30, 15 ; Sol (basso)
  12, 4, 28, 30 ; Do (lunga)
  0, 0, 0, 15 ; Pausa

  255           ; Marcatore di fine canzone
end

  music_note_duration = 1 ; Inizia subito la prima nota
  return
```

Nota che non ci sono commenti tra *sdata* e *end*. **Non inserire MAI un commento rem o ; in una riga di un blocco sdata.** Se lo fai, il compilatore interpreterà la parola "rem" o il punto e virgola come un dato numerico, "inquinando" la tua partitura musicale e causando errori imprevedibili o suoni distorti.
Sentirai "Fra Martino Campanaro" suonare in loop. La subroutine *music_play* si occupa di tutto: tiene il tempo, legge i dati usando sread e il puntatore *x*, e imposta i registri.

Appendice E: Cicli e Kernel

Questa appendice è il tuo riferimento tecnico per una delle risorse più limitate e cruciali della console: il **tempo della CPU**. Consultala ogni volta che hai bisogno di ottimizzare il tuo codice, quando il tuo gioco "trema". Cosa fare se il gioco è troppo lento? Non devi per forza eliminare delle funzionalità. Spesso basta distribuire il carico di lavoro in modo più intelligente.

1.Il Budget di un Frame e la Tabella dei Cicli CPU

Ogni frame dura circa 16.67 millisecondi. Durante questo tempo, la CPU 6507 può eseguire un numero limitato di "cicli". Il tuo codice deve rientrare in questo budget per evitare problemi grafici.

Divisione del Tempo in un Frame:

Fase	Durata (Cicli CPU approx.)	Scopo Principale
VBlank	~1675 cicli	Esecuzione del blocco *vblank*. Ideale per logica "pesante" (IA complessa).
Disegno Visibile	(gestito dal Kernel)	Il Kernel disegna lo schermo. Il tuo codice non viene eseguito qui.
Overscan	~2700 cicli	Esecuzione del main_loop. Ideale per logica "urgente" (input, movimento).

Regola Fondamentale: La quantità di codice eseguita tra un drawscreen e il successivo deve richiedere **molto meno di 2700 cicli** per evitare lo *screen roll*.

Tabella dei Costi delle Operazioni Comuni

Operazione	Cicli CPU (stima)	Livello di Costo	Note
Assegnazione (a = 5)	4 - 6	Molto Basso	Veloce e sicura.
if (semplice)	8 - 12	Basso	
if collision(...)	14 - 18	Basso	Leggermente più costoso.
gosub / return	20 - 24	Medio-Basso	Ha un piccolo overhead.
pfscroll up/down	~30	Medio	
Moltiplicazione / Divisione	50 - 100+	Alto	Evitare nei loop stretti. Ricordati che puoi moltiplicare e dividere per 2 usando gli operatori di bit shift << e >> (appendice A)
pfscroll left/right	80 - 100+	Molto Alto	Estremamente costoso, da usare con cautela.

2.Sfruttare il "Tempo Morto" – Spostare il Lavoro nel VBlank

Questa è la tecnica di ottimizzazione più importante. Per capirla, dobbiamo tornare per un istante a come funziona un vecchio televisore.

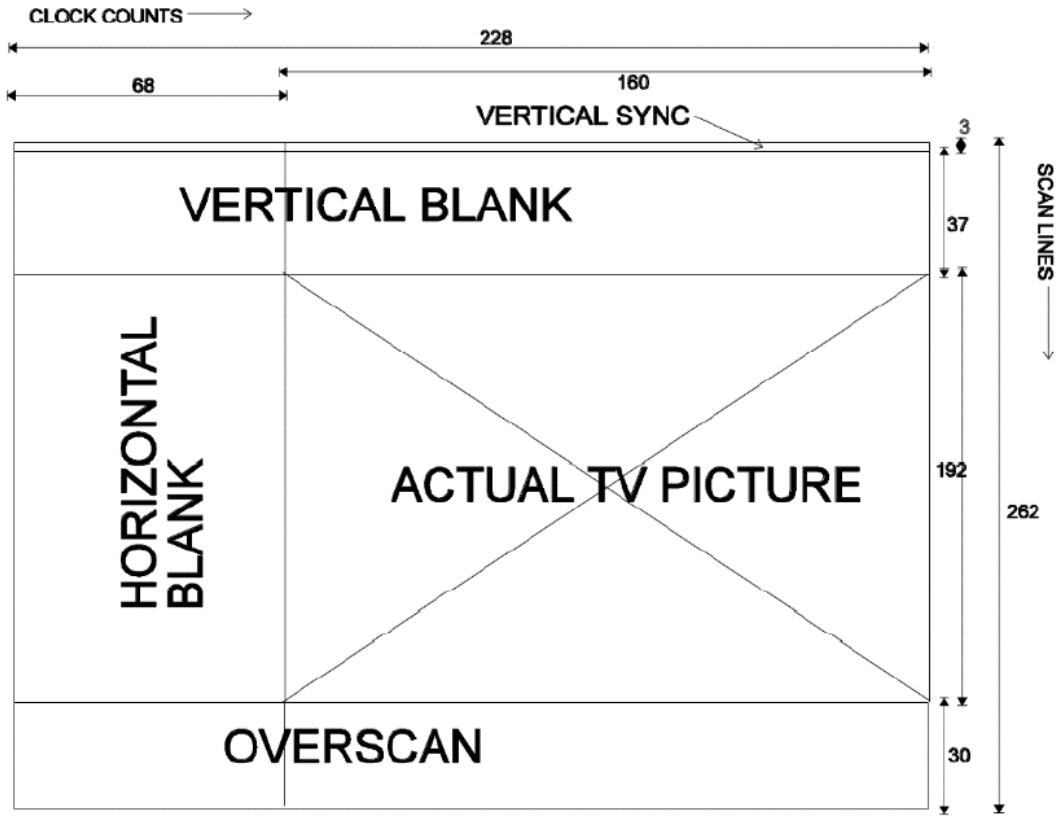
Cos'è il Vertical Blank (VBlank)? Come abbiamo visto, un televisore disegna un'immagine (un *frame*) tracciando righe orizzontali dall'alto verso il basso. Una volta arrivato in fondo, il pennello elettronico deve "tornare indietro" fino all'angolo in alto a sinistra per iniziare a disegnare il frame successivo. Durante questo breve viaggio di ritorno, il raggio viene **spento**. Questo intervallo di tempo in cui lo schermo è "buio" si chiama **Vertical Blank (VBlank)**.

Anche se dura solo pochi millisecondi, per la velocissima CPU dell'Atari 2600 questo è un tempo prezioso. Il kernel standard di Batari Basic ci mette a disposizione circa **1675 cicli CPU** durante il *VBlank*, un'enorme quantità di "tempo libero" in cui possiamo eseguire calcoli senza interferire con il delicato processo di disegno.

L'Overscan e il VBlank in Batari Basic

Il codice del nostro main_loop viene eseguito in un'altra fase, chiamata **Overscan**, che avviene subito *dopo* che il frame è stato disegnato. *L'Overscan* è il momento ideale per la logica "urgente" (leggere il joystick, muovere il giocatore), perché le sue conseguenze saranno visibili nel frame immediatamente successivo.

Il *VBlank,* invece, avviene *prima* del disegno. È quindi perfetto per tutta la logica "pesante" e non urgente, i cui risultati possono aspettare un frame per essere visualizzati.



Composizione frame video

Come si Usa vblank in Batari Basic? Per eseguire del codice durante il VBlank, è sufficiente creare un blocco speciale nel tuo programma, delimitato dall'etichetta *vblank* (indentata!) e dal comando return.

```
main loop
  rem ... qui va solo la logica "leggera" e urgente ...
  drawscreen
  goto main_loop

  vblank
  rem Qui va la logica "pesante" e non urgente
  gosub update complex enemy ai
  gosub calculate_scores
  return
```

Il codice nel blocco vblank verrà eseguito **automaticamente ad ogni frame**, subito prima che *drawscreen* inizi il suo lavoro. Non devi chiamarlo con gosub; il kernel lo fa per te.
Prova questo codice. Anche se nel main_loop cerchiamo di impostare lo sfondo a verde**, il comando nel vblank lo cambia a rosso** perché in realtà è eseguito un attimo prima di *drawscreen*, cioè prima che il TIA inizi a disegnare.

```
main_loop
  COLUBK = $9E ; Verde
  drawscreen
  goto main loop

  vblank
  COLUBK = $44 ; Rosso
  return
```

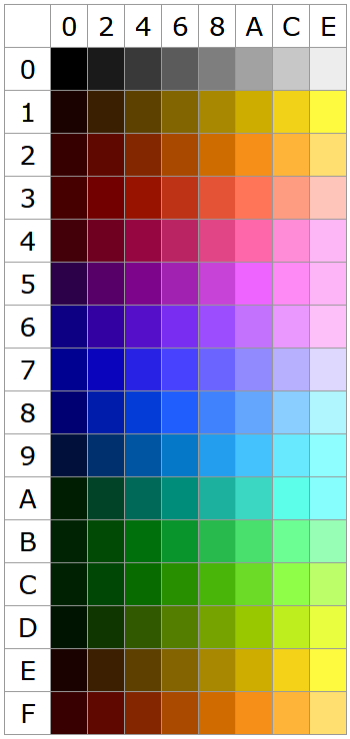
Poiché il codice nel *vblank* viene eseguito *prima* del disegno, qualsiasi modifica alle posizioni degli oggetti (*player0x, score*, ecc.) non sarà visibile fino al drawscreen **successivo**. Questo introduce un frame di ritardo. Per questo motivo, **non spostare mai nel vblank la logica che richiede una risposta immediata**, come la lettura del joystick e il movimento del giocatore. Riserva il *vblank* per calcoli che possono "permettersi" di essere aggiornati con un piccolo ritardo, come l'intelligenza artificiale di un nemico lontano o l'aggiornamento di un timer complesso.

Appendice F: Guida ai Colori e Standard TV

Il chip TIA dell'Atari 2600 può generare una gamma di colori sorprendentemente ampia per l'epoca. Conoscere la tavolozza e come i valori esadecimali corrispondono ai colori è fondamentale per dare ai tuoi giochi l'aspetto giusto e creare l'atmosfera perfetta.

1. Come Funzionano i Colori sull'Atari 2600

Un colore sull'Atari 2600 è definito da un singolo byte. Questa tabella mostra la tavolozza di 128 colori disponibile sullo standard televisivo NTSC (Nord America, Giappone). I valori sono in esadecimale. Per trovare un colore, incrocia la la riga della **Tonalità** (la prima cifra, $X-) con la colonna della **Luminosità** (la seconda cifra, $-Y). Ad esempio $1E è un bel giallo brillante.



2. NTSC vs. PAL: Gestire i Diversi Standard Televisivi

I televisori nel mondo non sono tutti uguali. I due standard principali dell'epoca erano **NTSC** e **PAL**, e avevano differenze importanti che influenzano i nostri giochi.

Frequenza di Aggiornamento video:
NTSC: ~60 frame al secondo (Hz). È lo standard usato in Nord America e Giappone.
PAL: ~50 frame al secondo (Hz). È lo standard usato in gran parte d'Europa e Australia.

Tavolozza dei Colori:
NTSC: 128 colori (la tabella sopra).
PAL: 104 colori, generalmente meno saturi e con tonalità leggermente diverse.

Il compilatore bB per default crea una ROM in formato **NTSC**. Per creare una versione per il mercato europeo (PAL), dovresti usare la direttiva *set tv pal* all'inizio del codice. Per garantire la

massima compatibilità e divertimento, il consiglio della community homebrew è quasi unanime: **sviluppa sempre per NTSC**. Un gioco NTSC (60Hz) funzionerà sulla maggior parte dei sistemi e televisori PAL moderni (spesso girando a 60Hz), mantenendo la velocità e il gameplay originali. Al contrario, un gioco PAL (50Hz) risulterà ingiocabilmente veloce e con suoni striduli sui sistemi NTSC. Attieniti allo standard NTSC per raggiungere il pubblico più vasto e garantire un'esperienza di gioco coerente.

3. Consigli Pratici per la Scelta dei Colori

La regola più importante. Assicurati che i tuoi personaggi si distinguano chiaramente dallo sfondo. Un eroe blu scuro su uno sfondo nero sarà quasi invisibile! Scegli colori con luminosità molto diverse. Per creare ombre o punti luce su un personaggio, non cambiare tonalità. Usa semplicemente una versione più scura (luminosità più bassa) o più chiara (luminosità più alta) dello stesso colore. I colori su un emulatore sono perfetti e brillanti. Su un vecchio televisore a tubo catodico (CRT), apparivano più scuri, "impastati" e con leggere sbavature. Quando scegli i colori, preferisci quelli brillanti e ad alto contrasto per garantire che siano ben visibili anche sull'hardware reale.