MONDRIAN AL QUADRATO

un inizio semplice per creare figure complesse

Ecco un approccio ad una progettualità che si sviluppa passo dopo passo, le prime istruzioni sono semplici e introducono a LibreLogo e ad alcuni paradigmi della programmazione.
Si inizia dalla linea per arrivare al quadrato, un quadrato che si aggiungerà ad altri per crearne uno suddiviso in quattro sezioni, si trasformerà in cerchio e poi in figure via via più complesse.
In LibreLogo il disegno viene creato sulla prima pagina del documento, per questo consiglio di utilizzare ***Inserisci → Interruzione di pagina*** e iniziare a scrivere il codice nella seconda pagina. Inoltre, consiglio di utilizzare la ***Vista a più pagine***, si trova nell'angolo in basso a destra, vicino allo zoom. In questo modo, soprattutto all'inizio, si può avere il codice di fianco al disegno.

Obiettivi

Corpo del testo

00. IL QUADRATO [mondrian_00.odt]

Se si vuole che il disegno inizi dal centro del foglio si può utilizzare l'istruzione

HOME

```
HOME ;(tana nel Logo originale)
```

Questa istruzione riporta la tartaruga al centro del foglio con la testa puntata verso l'alto, rappresenta uno *stato* della tartaruga, si tratta infatti di dare la posizione di partenza, e la direzione della testa verso l'alto fa parte anch'essa dello stato della tartaruga. Si tratta, infatti, della *geometria della tartaruga*, una geometria che è *soggettiva*: **posizione e direzione ne costituiscono lo stato** e le istruzioni vengono date in funzione dello stato della tartaruga stessa.

La prima istruzione relativa all'ambiente è

CLEARSCREEN

```
CLEARSCREEN ;(pulisci schermo)
```

Questa istruzione è quasi sempre presente all'inizio dei miei script, è relativa all'ambiente perché ripulisce la pagina da qualsiasi altro disegno creato precedentemente. È come per le lavagnette magiche, esempio che utilizzo spesso con i ragazzi, prima di poterci scrivere si puliscono dalle tracce lasciate dal precedente utilizzo.

HOME e **CLEARSCREEN** si possono invertire? In questo momento no. Se la tartaruga finisce il disegno in un punto differente da **HOME** e si fa ripartire l'esecuzione, **CLEARSCREEN** pulirebbe lo schermo ma quando esegue **HOME** si traccerebbe una linea dal punto dove la tartaruga si trovava alla posizione di **HOME**.
Vedremo più avanti un'altra istruzione di stato riferita alla penna, la penna

che permette di scrivere, si può alzarla o abbassarla sul foglio a seconda della necessità.

Proviamo a far muovere la tartaruga

```
FORWARD 100 ;(avanti)
```

FORWARD

La tartaruga si muove verso l'alto del foglio di 100. 100 sono i punti tipografici, si possono utilizzare altre unità di misura che devono essere specificate. Se si scrivesse 10mm si sposterebbe in avanti di 10 millimetri, 10cm si sposta di 10 centimetri, 10in o 10" si sposta di 10 pollici.

Unità di misura

Si muove verso l'alto perché, se non abbiamo messo altre istruzioni dopo **CLEARSCREEN**, la tartaruga è orientata verso l'alto e, per la geometria della tartaruga, se l'istruzione dice **FORWARD,** avanti, come in questo caso, la tartaruga si muove seguendo la direzione in cui è orientata.
Se si ripete l'esecuzione si può vedere che la tartaruga torna alla posizione di partenza e il foglio viene ripulito e di nuovo la tartaruga si sposta verso l'alto di **100**.

E ora giriamola

```
RIGHT 90 ;(destra)
```

RIGHT

In questo modo la tartaruga gira a destra di 90, 90 sono i classici gradi degli angoli, quindi ruota di 90° verso destra, ma a destra rispetto a cosa? Nella geometria soggettiva il soggetto è la tartaruga e quindi è la sua destra. in questo momento la tartaruga è orientata verso l'alto e, quindi, la sua destra e la nostra destra coincidono, ma non è sempre così.
Se si prova ora il codice si può vedere che viene tracciata una linea verso l'alto e poi la tartaruga ruota e la testa è ora verso destra, ovviamente non si muove verso destra, ruota soltanto verso destra.

Se ora vogliamo che la tartaruga si muova verso destra dovremo utilizzare ancora l'istruzione **FORWARD**

```
FORWARD 100
```

In questo modo viene tracciata un'altra linea perpendicolare alla prima della stessa lunghezza. È l'inizio di un quadrato.

Per orientare la tartaruga verso il basso e poter proseguire il disegno del quadrato serve di nuovo l'istruzione di rotazione

```
RIGHT 90
```

Ed ecco che la soggettiva si fa più chiara la destra della tartaruga è il nostro giù.

Diventa evidente ora, che se si vuole continuare, occorre di nuovo l'istruzione

```
FORWARD 100
```

Di nuovo la rotazione

Monica Umberta Oriani
www.code2play.space

```
RIGHT 90
```

E ancora una volta

```
FORWARD 100
```

Un'ultima istruzione di rotazione ci permette di riportare la tartaruga con lo stesso orientamento di partenza

```
RIGHT 90
```

Vedremo più avanti come il fatto di riportare la tartaruga nelle condizioni iniziali sia importante. Conoscere l'orientamento della tartaruga, stabilire coscientemente l'orientamento iniziale, permette di poter creare elaborazioni complesse suddividendo il disegno in piccole parti, esattamente come per un programma.

Questo è l'approccio che verrà seguito in questo testo, in parallelo, sia per il disegno e sia per la programmazione, si inizia da un disegno semplice con istruzioni semplici e si costruisce un disegno complesso con uno script man mano più complesso.

01. LA PIGRIZIA DEL PROGRAMMATORE [mondrian_01.odt]

Un programmatore a questo punto si domanda se qualcosa possa essere in qualche modo ottimizzato. Si domanda, cioè, *cosa si ripete*, se c'è un modo per rendere più "snello" il codice. Di solito è per pigrizia, la pigrizia del programmatore, scrivere troppe righe di codice costa fatica, se qualcosa si ripete allora si può risparmiare fatica, ma, soprattutto, si abbassa la percentuale di errore di scrittura e la conseguente faticosa caccia alla ricerca di bug.

Cosa notiamo? Notiamo che le istruzioni si ripetono a due a due

```
FORWARD 100
RIGHT 90
```

Queste sono le istruzioni che permettono di tracciare una riga e prepararsi nella direzione giusta per tracciare la riga seguente, ovvero si predispongono le condizioni di partenza per la nuova linea.

REPEAT

L'istruzione più semplice per poter ripetere qualcosa è per l'appunto: **REPEAT**, ripeti, questa istruzione consente di ripetere le istruzioni contenute all'interno del suo costrutto.
L'istruzione **REPEAT** richiede un parametro, **quante volte**, e un'apertura e una chiusura che vengono fatte con le parentesi quadre. Di seguito:
REPEAT *numero* [
istruzioni
]
Le parentesi quadre contengono le istruzioni che devono essere ripetute. Nel nostro caso:

```
REPEAT 4 [
    FORWARD 100
    RIGHT 90
]
```

Ma prima di proseguire occorre fare alcune considerazioni.
Le istruzioni **HOME** e **CLEARSCREEN** sono istruzioni semplici, di un solo termine, non sono necessarie altre indicazioni.
Per **FORWARD** e **RIGHT** abbiamo dovuto indicare un valore, che viene chiamato **parametro**, per indicare di quanto: di quanto andare avanti, di quanto ruotare.
Anche **REPEAT** ha bisogno di una quantità, quante volte. In più necessita di parentesi quadre per delimitare le istruzioni da eseguire più volte.

Si provi ora a sostituire le istruzioni **FORWARD 100** e **RIGHT 90** con l'istruzione **REPEAT** appena indicata e la si esegua, il codice è cambiato ma il risultato non cambia. Questa è una verifica che i programmatori fanno spesso quando cambiano delle parti di codice mentre lo ottimizzano. È una prova che verrà fatta più volte anche in questo progetto, proprio perché si sostituiranno via via istruzioni semplici con istruzioni più complesse.

02. UN QUADRATO PIÙ PICCOLO O PIÙ GRANDE [mondrian_02.odt]

E se volessimo un quadrato più piccolo? Oppure se ne volessimo uno più grande?
Occorre cambiare il parametro che definisce la lunghezza della linea, ovvero il parametro dell'istruzione **FORWARD**, ad esempio se lo volessimo piccolo la metà dovremmo scrivere

```
FORWARD 50
```

Se lo volessimo più grande della metà

```
FORWARD 150
```

Nulla impedisce di utilizzare altri valori a scelta.

La proverbiale pigrizia del programmatore ci obbliga a utilizzare un altro modo per indicare la quantità, soprattutto quando la quantità si può ripetere più volte nel codice.
Nel nostro caso si scrive ancora solo in un punto, ma vedremo che poi ci servirà più volte.
Il *più volte* aumenta la possibilità di errore, come detto sopra, il valore verrà indicato solo in un punto del codice, possibilmente all'inizio e gli verrà data un'etichetta, possibilmente significativa: verrà creata una **variabile**.

variabile

All'inizio del codice perché sicuramente si trovano prima di quando vengono utilizzate, perché dichiarare una variabile in mezzo a tante righe di codice non ne permette una sua facile individuazione se deve esserne modificato il valore iniziale. E, soprattutto, il programmatore è pigro e trovare le variabili tutte all'inizio del proprio script ne permette una veloce

Monica Umberta Oriani
www.code2play.space

individuazione e permette anche di capire con cosa si ha a che fare.
In particolare è comprensibile se le etichette sono *parlanti*, ovvero se si dà un nome significativo alle variabili. Ad esempio: ***lato*** ci fa capire che ci si riferisce al valore del lato, alla sua lunghezza, ***pluto*** non è altrettanto significativa se contiene il valore della lunghezza del lato.
Come definire, anzi, *come dichiarare una variabile*?
In LibreLogo è molto semplice basta scrivere il nome dell'etichetta seguita dal segno di uguale e il valore iniziale. Iniziale perché è una variabile e può essere modificata nel corso del programma.
Nel caso del nostro esempio abbiamo bisogno di dichiarare una variabile per il valore della lunghezza del lato:

```
lato = 100
```

Attenzione il segno di uguale non va inteso in senso matematico, ma come assegnazione di valore.
Per il nome della variabile si possono utilizzare solo caratteri alfanumerici e il trattino basso _ (l'underscore), senza spazi, e occorre iniziare con una lettera, ovvero, **`lato1`** ma non **`1lato`**. LibreLogo è *case sensitive*, ovvero **`Lato`** è diverso da **`lato`** e diverso da **`LATO`**. Un modo per scrivere nomi composti di variabili è definito *camelCase*, ogni parola inizia con la lettera maiuscola, ad esempio: **`NumeroLati`** o anche **`numeroLati`**. Un altro metodo consiste nel dividere le parole con l'underscore, cioè, **`numero_lati`**. Sono scelte individuali che aiutano a rendere il codice più leggibile, si consiglia, qualsiasi sia la preferenza, di scegliere un solo metodo per tutto lo script.
Ecco come cambia il nostro script.

```
HOME
CLEARSCREEN
```

restano invariate poi si aggiunge la dichiarazione di variabile

```
lato = 100
```

E di seguito l'istruzione per disegnare il quadrato

```
REPEAT 4 [
    FORWARD ?
    RIGHT 90
]
```

Cosa mettere al posto del punto interrogativo? La nostra variabile, quindi:

```
REPEAT 4 [
    FORWARD lato
    RIGHT 90
]
```

Logo leggerà **`lato`** e vi sostituirà il valore assegnato.
A **`lato`** è stato assegnato inizialmente **`100`**, lo stesso valore precedentemente utilizzato, sempre per la regola della verifica che cambiando il codice non cambi il risultato, si può quasi dire che è una legge della programmazione insieme alla pigrizia.

Ora si possono utilizzare altri valori per provare a vedere se funziona.

03. DIAMO UN NOME A UNA SEQUENZA DI ISTRUZIONI [mondrian_03.odt]

Se diamo un nome alle cose le cose esistono e le possiamo chiamare.
Cosa significa dare un nome ad una sequenza di istruzioni. Significa poter richiamare quella sequenza in altri punti del codice. Fa sempre parte della prima legge della pigrizia, se abbiamo bisogno di una determinata sequenza di istruzioni in più punti allora tanto vale scriverla in un punto solo e dargli un'etichetta, come per le variabili, ma in questo caso stiamo dando il nome ad una **procedura**.

procedura

La regola per dare il nome è sempre la stessa, solo caratteri alfanumerici e il trattino basso, anche in questo caso si deve iniziare solo con una lettera e non utilizzare spazi. Per i nomi composti valgono gli stessi suggerimenti, utilizzare il metodo *camelCase* o suddividere i nomi con il trattino basso. Personalmente per i nomi di variabile utilizzo l'iniziale in minuscolo e distinguo i nomi delle procedure utilizzando l'iniziale maiuscola.
Per scrivere una procedura è necessario inserire le istruzioni della stessa tra le istruzioni **TO *NomeProcedura*** e **END**. **TO *NomeProcedura*** indica "per fare" *NomeProcedura*, seguono le istruzioni e in ultimo la parola chiave **END**, fine, per indicare la chiusura della procedura.

TO … END

Inseriamo le istruzioni per disegnare il quadrato in una procedura che si chiamerà **DisegnaQuadrato** (spesso i nomi parlanti di procedure hanno un verbo che dice esattamente ciò che la procedura fa).

```
TO DisegnaQuadrato
    REPEAT 4 [
          FORWARD lato
          RIGHT 90
    ]
END
```

Ma così ancora non funziona, se proviamo a eseguire il codice in questo modo non succede nulla.
Se diamo un nome alle cose le cose esistono e le possiamo *chiamare*.
Ecco la procedura la dobbiamo *chiamare* e lo possiamo fare molto semplicemente scrivendone il nome, così:

```
DisegnaQuadrato
```

Eseguendo ora il codice, però, ci viene restituito un messaggio di errore molto chiaro «Nome sconosciuto: "lato"».
All'interno della procedura, infatti, viene utilizzata la variabile **lato**, ma ciò che è dichiarato all'esterno della procedura non può essere utilizzato all'interno (un'eccezione c'è e la vedremo più avanti).

Come fare allora?

Passeremo alla procedura un parametro proprio come si fa con le istruzioni **FORWARD** e **RIGHT**, ad esempio.
Basterà, quindi, scrivere:

Monica Umberta Oriani
www.code2play.space

```
DisegnaQuadrato lato
```

Cliccando su esegui si ottiene il nostro solito quadrato.

04. RIPETIAMO IL QUADRATO [Mondrian_04.odt]

Ora proveremo a chiamare più volte la procedura **DisegnaQuadrato** con un ciclo **REPEAT** e utilizziamo la variabile facendone "variare", per l'appunto, il valore ad ogni chiamata. La variabile **lato** assumerà un valore differente in funzione del numero di ciclo in cui si trova.
Per mettere la chiamata di funzione all'interno di un ciclo è sufficiente aprire il ciclo indicando il numero di ripetizioni, chiamare la procedura e chiudere il ciclo

```
REPEAT 10 [
    DisegnaQuadrato lato
]
```

REPCOUNT

In questo modo disegnerà 10 quadrati, ma eseguendo il codice vediamo un solo quadrato perché la lunghezza di **lato** ha sempre lo stesso valore. Utilizziamo il contatore interno del ciclo **REPEAT** per cambiare il valore di **lato** ad ogni passaggio. Il contatore si chiama **REPCOUNT** e inizia da 1. Assegniamo **110** come valore iniziale di **lato**, in questo modo possiamo sottrarre 10 ogni volta per 10 volte e avendo, quindi, come ultimo valore 10. Pertanto il quadrato più piccolo avrà 10 come valore di **lato**. Sottraendo 10 * *il valore del numero di ciclo corrente* si otterranno via via i valori di 100, 90, 80 e così via fino ad arrivare all'ultimo quadrato che avrà **lato** di valore 10.

```
REPEAT 10 [
    DisegnaQuadrato lato - (10*REPCOUNT)
]
```

05. RIPETIAMO IL QUADRATO IN MODO CASUALE [mondrian_05.odt]

E ora assegniamo un valore casuale al valore di **lato**.

```
REPEAT 10 [
    lato = RANDOM 100
    DisegnaQuadrato lato
]
```

RANDOM

In questo modo alla variabile **lato** viene assegnato un valore casuale ad ogni ciclo.
Il valore viene assegnato utilizzando la funzione **RANDOM**, scrivendo **RANDOM 100** il valore verrà scelto casualmente tra **0** e **100**. Il valore è un numero di tipo decimale e questo sarà da tenere presente quando si avrà bisogno di valori numerici interi.

HIDETURTLE
SHOWTURTLE

La tartaruga inizia a dare un po' di fastidio, non permette di vedere bene tutti i quadrati. La sua visibilità fa parte del suo stato, la possiamo nascondere alla fine con **HIDETURTLE** (nascondi tartaruga), se la nascondiamo dobbiamo renderla visibile all'inizio con **SHOWTURTLE**

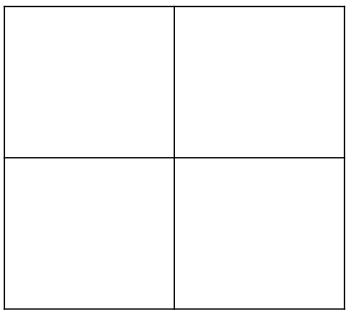
(mostra tartaruga), possiamo mettere questa istruzione subito dopo **CLEARSCREEN**.

06. QUATTRO QUADRATI PER UN SOLO QUADRATO [mondrian_06.odt]

Mettiamo la ripetizione della chiamata di **DisegnaQuadrato** in un'altra procedura e proviamo a richiamare anche questa procedura 4 volte per costruire un quadrato formato da quattro quadrati.[1]
Per prima cosa si può eliminare la variabile **lato** dato che viene via via valorizzata all'interno della funzione che ripete i quadrati.
La ripetizione del punto precedente viene messa all'interno delle istruzioni per creare una nuova procedura che si chiamerà **RipetiQuadrati**, in questo modo:

```
TO RipetiQuadrati
    REPEAT 10 [
          lato = RANDOM 100
          DisegnaQuadrato lato
    ]
END
```

E ora facciamo un passo in più, proviamo a richiamare questa procedura in modo che disegni quattro quadrati in forma di quadrato:



Per disegnare il quadrato di base siamo partiti dall'angolo in basso a sinistra, dobbiamo metterci nella soggettiva della tartaruga per creare gli altri quadrati utilizzando le stesse istruzioni del primo. Sarà sufficiente, al termine di ogni disegno, ruotare di 90 gradi verso destra per ritrovarsi nello stato di partenza del primo quadrato.

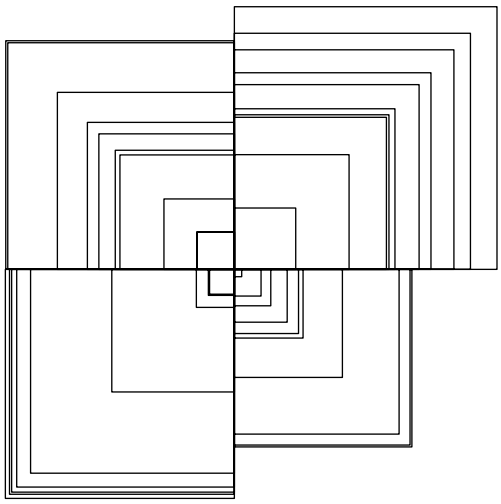
```
REPEAT 4 [
    RipetiQuadrati
    RIGHT 90
]
```

Questo è un possibile risultato, possibile perché la funzione **RANDOM** renderà i risultati simili ma non identici:
Mettiamo anche questa ripetizione in una procedura, avremo delle procedure nidificate, il termine rende bene l'idea, anche "a matrioska" potrebbe funzionare, ma il termine corretto è *procedure nidificate o*

1 La base di questo esercizio è stata tradotta da un esempio in FSMLogo tratto dal sito http://fmslogo.sourceforge.net/workshop/. FSMLogo è un altro dialetto derivante da Logo, come del resto anche LibreLogo.

Monica Umberta Oriani
www.code2play.space

annidate.



La procedura la chiameremo **DisegnaFigura**, *figura* perché questo è solo l'inizio, al prossimo passaggio cominceremo a mettere un po' di colore alla Mondrian, da cui il titolo del progetto. E alla fine ci ritroveremo con quadrati che si deformano e cerchi che fanno i quadrati.
Per ora avremo quindi:

```
TO DisegnaFigura
    REPEAT 4 [
          RipetiQuadrati
          RIGHT 90
    ]
END
```

Chiamare **DisegnaFigura** per poter eseguire il codice. È sufficiente scrivere il nome della procedura, non occorre passare nessun parametro, così:

```
DisegnaFigura
```

07. IL COLORE ENTRA IN SCENA [mondrian_07.odt]

FILLCOLOR

ANY

RGB

CMY

Tanti quadrati uno dentro l'altro invitano a dare del colore, cominceremo con la casualità per provare le nuove istruzioni di colore. Bisogna impostare il colore di riempimento e si utilizza l'istruzione **FILLCOLOR** (colore di riempimento) a cui segue il parametro del colore.
Per il parametro del colore abbiamo diversi modi a disposizione per scriverlo. Il primo che utilizzeremo è la parola chiave **ANY** (qualunque), ovvero sceglie in modo casuale un colore.
Quanti colori abbiamo a disposizione? 256*256*256 che fa 16.777.216.
Questi numeri si riferiscono al sistema RGB dei monitor, il sistema RGB si basa sui colori primari rosso (Red), verde (Green) e blu (Blue), è un sistema adattivo al contrario di quello sottrattivo. Il sistema sottrattivo, CMY, è quello dei pigmenti e si basa sui colori primari Cyan, Magenta e giallo (Yellow). Nel modello RGB, la somma dei colori costituisce il bianco e la loro assenza il nero. Ogni colore è *discretizzato* in 256 valori rappresentati

da un *byte*, un *byte* è costituito da 8 *bit*, 8 valori che possono assumere il valore di 0 o 1. Ogni colore può valere da 0 (assenza), in binario 00000000, fino a 255 (colore pieno), in binario 11111111, quindi in totale 256 valori per ogni colore. Le combinazioni possibili fanno il numero segnalato qualche riga fa.
Ancora poco e utilizzeremo questi numeri.

Per il momento utilizziamo **`ANY`** e lasciamo la scelta del colore al sistema.
Dove mettere questa nuova istruzione?
Vogliamo che *ogni* quadrato abbia un colore differente. Questo apre la strada a due possibilità: impostare il colore all'interno della procedura che disegna un solo quadrato, **`DisegnaQuadrato`**; impostare il colore all'interno della procedura che ripete la chiamata alla procedura che disegna il singolo quadrato, **`RipetiQuadrati`**. Sicuramente non in **`DisegnaFigura`**, perché in quel caso avremmo un solo colore per ogni gruppo di quadrati, un colore per ogni quadrante.
Per scegliere il posto ottimale è importante avere già una visione di dove si vuole arrivare, se si dice soltanto che ogni quadrato avrà un colore casuale è abbastanza logico e naturale pensare di mettere le indicazioni di colore in **`DisegnaQuadrato`**.
E possiamo provare questa soluzione pronti, però, a modificare tutto nel momento in cui si farà strada la curiosità di provare a osare qualcosa in più. Quando si vorrà manipolare il colore presi dal delirio di onnipotenza che causa la programmazione, creare è bello!

`FILL`

Come inserire l'istruzione colore nella procedura **`DisegnaQuadrato`**, la prima istruzione dopo la dichiarazione del nome della procedura sarà proprio l'istruzione che indica quale colore di riempimento utilizzare, l'ultima istruzione, prima della chiusura, sarà **`FILL`** (riempi), questa è l'istruzione che indica al sistema di colorare la figura appena creata. **`FILL`** non si limita solo a riempire di colore, ma permette di chiudere e colorare figure aperte.

```
TO DisegnaQuadrato lato
    FILLCOLOR ANY
    REPEAT 4 [
          FORWARD lato
          RIGHT 90
    ]
    FILL
END
```

Se si esegue il codice vediamo qualcosa che probabilmente non ci aspettavamo, il colore si sovrappone, i quadrati vengono creati con un valore di **`lato`** casuale e, quindi, vengono creati quadrati piccoli e grandi in modo casuale, sovrapponendosi. Questo fa sì che se un quadrato grande segue ad uno piccolo, il colore di quello piccolo viene nascosto da quello grande.

Possiamo giocare con questa cosa, possiamo giocare con le trasparenze, possiamo, cioè, dare al colore una trasparenza che ci permetta di

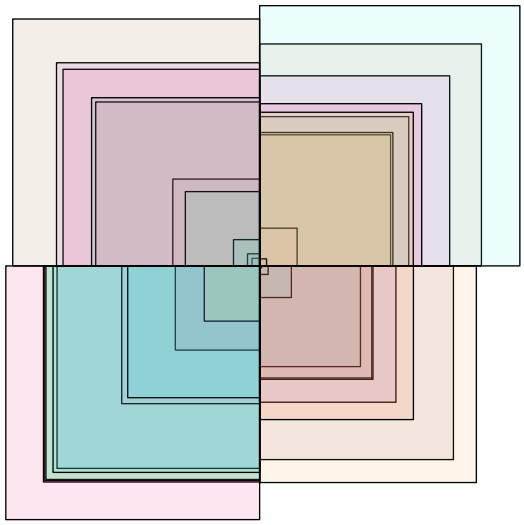
Monica Umberta Oriani
www.code2play.space

FILLTRANSPARENCY

intravedere le varie sovrapposizioni di colore.
Anche per questo abbiamo a disposizione un'istruzione apposita: **FILLTRANSPARENCY** (trasparenza di riempimento) a cui va indicato quanto trasparente, i valori possibili vanno da **0** (assenza) a **100** (massima trasparenza). Perché i valori vanno da **0** a **100**? Perché è un valore in percentuale, quindi, da 0% a 100%.
Provare ad aggiungere questa istruzione subito dopo **FILLCOLOR ANY**, così:

```
TO DisegnaQuadrato lato
    FILLCOLOR ANY
    FILLTRANSPARENCY 90
    REPEAT 4 [
          FORWARD lato
          RIGHT 90
    ]
    FILL
END
```

Ed ecco un possibile risultato



Un effetto vetrate gotiche.

08. ORDINIAMO I QUADRATI [mondrian_08.odt]

È arrivato il momento di mettere ordine nella casualità, l'altra legge dei programmatori è il delirio di onnipotenza di cui parlavo prima, quindi possiamo dare un nostro ordine alla casualità. I quadrati avranno ancora un valore per di **lato** casuale, ma faremo in modo di iniziare dal più grande per arrivare al più piccolo, in questo modo avremo una pila di quadrati con sotto il più grande. Potremo lasciare il colore puro, senza trasparenza e vedremo tutti i quadrati.
Dove agire per poter guidare il caso?
Dove creiamo i quadrati a ripetizione, dove, cioè, già viene impostato il valore di **lato**, all'interno della procedura **RipetiQuadrati**.
Per stabilire il valore da assegnare a **lato** volta per volta utilizzeremo il

contatore **REPCOUNT** che abbiamo già utilizzato nel punto *04*.
Bisogna tenere presente che

- **RANDOM** seguito da un numero significa: scegli un numero a caso tra 0 e numero indicato;
- **REPCOUNT** inizia dal valore 1

Se vogliamo iniziare dal valore più grande di **lato** dovremo sottrarre da questo valore.
Vorrei che il quadrato più grande abbia un valore massimo di 150 e voglio una differenza di 15 tra un quadrato e l'altro se non si considera la casualità. Il calcolo è presto fatto:
165 – (15*REPCOUNT). 165 perché **REPCOUNT** inizia da 1 e quindi la prima volta che si entra nel ciclo la sottrazione sarà: **165 – (15*1) = 150**, al decimo passaggio, l'ultimo, si avrà **165 – (15*10) = 15**, in questo modo avremo un ultimo quadrato piccolo.

Aggiungiamo il *caso*.
Dobbiamo togliere ogni volta un valore a caso tra 0 e 15, faremo in modo che il valore sia intero per ridurre notevolmente le possibilità del caso: **INT(RANDOM 15)**. **INT** dice di utilizzare solo la parte intera, un 0.1827365 diventa 0, un 4.8743737 diventa 4 e così via.

INT

Il valore da assegnare a **lato** sarà:
lato = ((165 – (15*REPCOUNT)) – INT(RANDOM 15))
Questa la procedura modificata:

```
TO RipetiQuadrati
    REPEAT 10 [
          lato = ((165 - (15*REPCOUNT)) - INT(RANDOM 15))
          DisegnaQuadrato lato
    ]
END
```

Per il primo quadrato potremo avere valori tra 150 e 135, il secondo tra 135 e 120, l'ultimo tra 15 e 0.

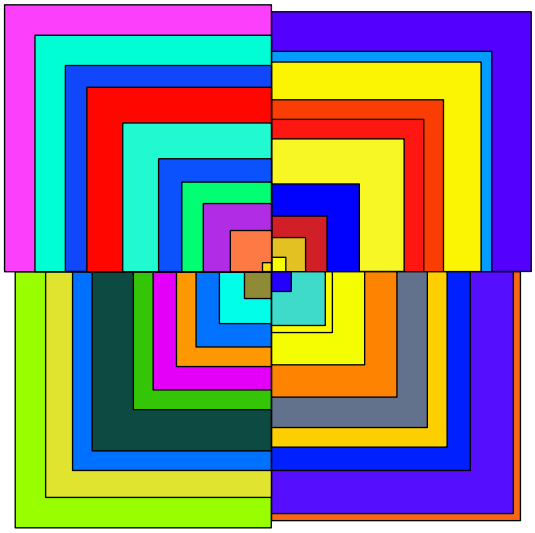
Cosa abbiamo perso e cosa abbiamo guadagnato? Un bilancio costi benefici.

Abbiamo guadagnato la possibilità di vedere tutti i quadrati, abbiamo perso la possibilità di avere un maggior range di differenza tra i quadranti, perché in questo modo avremo sempre un quadrato con lato tra 150 e 165 per ogni quadrante.

Da un grande potere derivano grandi responsabilità, siamo onnipotenti, possiamo creare, ma dobbiamo essere consci di quello che perdiamo e di quello che guadagniamo e decidere cosa va meglio per il nostro progetto.

Monica Umberta Oriani
www.code2play.space

Possiamo togliere **FILLTRANSPARENCY** se vogliamo provare con i colori pieni.



09. E MONDRIAN? [mondrian_09.odt]

Abbiamo parlato di colori alla Mondrian, ma qui stiamo usando quelli di Arlecchino. Siamo fortunati perché Mondrian usava colori puri: rosso, blu, giallo, bianco e nero.
Ho utilizzato il termine "puri" per indicare i colori e non "primari", perché il rosso e il blu sono colori primari per il sistema **RGB**, ma il giallo è secondario in questo sistema e primario in quello **CMY**, il sistema di riferimento dei colori dei pigmenti dove il rosso e il blu sono, a loro volta, secondari.
Ancora un po' di teoria dei colori, in particolare come si definiscono i valori dei colori nel sistema **RGB**.
0 assenza di colore, 255 colore pieno. Tre valori definiscono il colore, ovvero: quanto rosso, quanto verde e quanto blu occorre per ciascun colore.
Il bianco abbiamo detto che è la somma di tutti i colori, colore pieno, 255, per tutte le componenti.
Bianco **[255, 255, 255]**
Il nero è l'assenza, l'assenza ha valore 0.
Nero **[0, 0, 0]**
Il rosso è facile, basta ripensare a **R**GB, si inizia da **R**ed, rosso, quindi il primo valore vale 255, non ci saranno componenti né di verde e nemmeno di blu e, quindi, gli altri due valori della terna saranno 0 e 0.
Rosso **[255, 0, 0]**
Il giallo è composto, per fare il giallo puro, nel sistema **RG**B, occorre la stessa quantità di rosso e di verde, è un colore secondario.
Giallo **[255, 255, 0]**
Il blu è facile come il rosso, RG**B**, è l'ultimo valore.
Blu **[0, 0, 255]**

Ancora un po' di teoria, perché abbiamo una terna di valori tra parentesi

array

quadre. Abbiamo un **array**, una variabile evoluta che può contenere più valori, ogni elemento dell'array ha un **indice** che ne identifica la sua posizione all'interno dell'array. Questo significa che i valori hanno un ordine preciso, il primo elemento ha indice 0, il secondo indice 1 e così via. *Gli indici iniziano dal valore 0*, i programmatori cominciano a contare da 0.
Questo ci sarà molto utile per poter programmare variazioni di colore scegliendo su quale componente agire.

Per l'istruzione **FILLCOLOR** abbiamo detto che ci sono diversi modi per indicare il parametro del colore, finora è stata utilizzata l'indicazione **ANY** che lascia al sistema il controllo sul colore da scegliere. Ma il parametro può essere anche passato come terna delle componenti proprio come indicato sopra: **FILLCOLOR [255, 0, 0]**
Per provare basta sostituire ad **ANY** l'array **[255, 0, 0]**. I quadrati saranno tutti rossi.

L'effetto che si vorrebbe ottenere è di avere un colore per ogni quadrante: rosso, giallo, blu e bianco. Ogni quadrante avrà il colore puro nel quadrato più esterno e più scuro verso l'interno. Per avere un colore più scuro si deve abbassare il valore della componente colore, nel caso del rosso e del blu sarà sufficiente abbassare l'unica componente che ha valore di 255. Per il giallo si dovranno abbassare entrambe le componenti con valore di 255, avendo cura di avere lo stesso valore per entrambe per fare in modo che non prevalga né il rosso né il verde nella composizione del giallo. Per il bianco le componenti da abbassare sono tutte e tre e anche in questo caso le tre componenti devono avere lo stesso valore.

Un'implementazione alla volta.
Iniziamo ad assegnare un colore a ciascun quadrante.
L'istruzione **FILLCOLOR** va ancora bene nella procedura **DisegnaQuadrato**? A questo punto è meglio metterla all'interno di **RipetiQuadrati** perché il colore varierà per ogni quadrante e per ogni quadrato. Cancelliamo **FILLCOLOR** e **FILL** da **DisegnaQuadrato**.
Prepariamo all'interno della procedura **RipetiQuadrati** la nostra tavolozza di colori, proprio come un pittore prepariamo i colori che utilizzeremo per colorare i quadrati: rosso, giallo, blu e bianco.
La nostra tavolozza sarà un array di quattro elementi in cui ogni elemento è a sua volta un array di tre elementi che rappresentano le componenti di ciascun colore.
Un array è come una variabile nella sua dichiarazione, ha bisogno di un'etichetta costituita da caratteri alfanumerici, il nome scelto è proprio **tavolozza**

```
tavolozza = [[255,0,0], [255,255,0], [0,0,255],
[255,255,255]]
```

I colori sono nell'ordine dichiarato sopra: rosso, giallo, blu, bianco.
Non c'è spazio tra le parentesi quadre e il carattere che le segue o le precede. Per LibreLogo questo è molto importante, in caso di spazio, ad esempio **[255, 0, 0]**, viene restituito un messaggio di errore. Lo

Monica Umberta Oriani
www.code2play.space

spazio tra gli elementi è invece ininfluente, si può mettere o meno lo spazio dopo la virgola prima dell'elemento successivo, ad esempio si può scrivere **`[255,0,0]`** ma anche **`[255, 0, 0]`**.
Dove mettere la dichiarazione di **`tavolozza`**?
Subito dopo la dichiarazione di apertura della procedura **`RipetiQuadrati`**.

```
TO RipetiQuadrati
    tavolozza = [[255,0,0], [255,255,0], [0,0,255],
[255,255,255]]
```

In questo modo possiamo sfruttare l'indice degli elementi dell'array per assegnare il colore ad ogni quadrante. Dobbiamo sapere in quale quadrante ci troviamo.

Il quadrante viene stabilito da **`DisegnaFigura`**, è lì, infatti, che si ruota e si richiama una nuova ripetizione di quadrati. Dobbiamo passare un parametro a **`RipetiQuadrati`** che ci dica a quale numero di ciclo ci troviamo e, quindi, in quale quadrante.
Ci aiuta di nuovo **`REPCOUNT`**, il contatore di **`REPEAT`** e ci aiuta in questo modo

```
TO DisegnaFigura
    REPEAT 4 [
          RipetiQuadrati REPCOUNT-1
          RIGHT 90
    ]
END
```

Quando chiamiamo la procedura **`RipetiQuadrati`** passiamo il parametro che ci servirà come indice per l'elemento di **`tavolozza`** necessario. Gli indici degli array iniziano dal valore 0, il primo valore di **`REPCOUNT`** è 1, togliendo 1 da **`REPCOUNT`** otteniamo i valori **`0, 1, 2, 3`** per i quattro cicli, in questo modo abbiamo già l'indice che ci serve per **`tavolozza`**.
Cosa cambia in **`RipetiQuadrati`**.
Aggiungiamo alla dichiarazione il nome del parametro necessario

```
TO RipetiQuadrati iQuadrante
```

Di solito una "**`i`**" indica indice, spesso è utilizzato come variabile per i contatori, in questo caso abbiamo "i" prima della parola "Quadrante" e questo rende evidente che è l'indice per il quadrante, abbiamo un nome di parametro *parlante*.
L'istruzione **`FILLCOLOR`** la metteremo subito dopo **`tavolozza`** e utilizziamo l'indice **`iQuadrante`** per avere l'elemento corrispondente di **`tavolozza`**.
Per utilizzare un elemento di un array è sufficiente scriverne il nome seguito dalle parentesi quadre con all'interno il valore dell'indice necessario, **`nomeArray[0]`** per avere il primo elemento, **`nomeArray[1]`** per avere il secondo e così via. Nel nostro caso possiamo utilizzare il parametro che abbiamo ricevuto **`iQuadrante`** in questo modo:

```
FILLCOLOR tavolozza[iQuadrante]
```

Ricordiamoci di mettere l'istruzione **FILL** subito dopo **DisegnaQuadrato**.

```
TO RipetiQuadrati iQuadrante
    tavolozza = [[255,0,0], [255,255,0], [0,0,255],
[255,255,255]]
    FILLCOLOR tavolozza[iQuadrante]
    REPEAT 10 [
          lato = ((165 - (15*REPCOUNT)) - INT(RANDOM 15))
          DisegnaQuadrato lato
          FILL
    ]
END
```

La prima implementazione è fatta, ogni quadrante ha un colore diverso scelto da una nostra tavolozza appositamente preparata.

10. IL GRADIENTE DI COLORE [mondrian_10.odt]

Le cose si complicano ma si procede un passo per volta.
Come prima cosa non utilizziamo subito **FILLCOLOR** ma mettiamo in una variabile chiamata **colore** l'elemento **iQuadrante** di **tavolozza**.
FILLCOLOR tavolozza[iQuadrante]
diventa
colore = tavolozza[iQuadrante]
In questo modo sarà più semplice lavorare le singole componenti del colore che ci serve.

Abbiamo detto che occorre abbassare la componente che ha valore 255 per ciascun ciclo, ovvero per ciascuno dei 10 quadrati contenuti in ciascun quadrante. Il calcolo andrà fatto, pertanto, all'interno del ciclo che richiama **DisegnaQuadrato**, come nel caso del calcolo del valore di **lato**. Definiamo **delta** il nome della variabile che darà il nuovo valore della componente posta a 255 e utilizziamo ancora **REPCOUNT** per calcolare il valore in funzione del numero di ciclo in cui ci si trova. Questa volta, a differenza di **lato**, si toglie 1 a **REPCOUNT** prima di moltiplicarlo per 20, attenzione alle parentesi, sono necessarie perché vengono seguite le stesse regole della matematica. In questo modo il primo valore del primo quadrato sarà 255, colore pieno. 20 è il valore che definisce la gradualità del cambiamento di colore, più è alto il valore maggiore la differenza di colore. Se si utilizza un numero che moltiplicato per il numero di ripetizioni dà valori maggiori di 255, si ottengono effetti particolari, perché il numero non viene considerato negativo, di conseguenza si ricomincia dal colore puro, provare con 50 ad esempio.

```
delta = 255 - (20*(REPCOUNT-1))
```

Ora bisogna andare a cambiare il valore della componente di colore corrispondente a 255.
Un modo semplice è quello di vedere quale colore è, e questo lo sappiamo dal valore del quadrante, e andare a modificarne la

Monica Umberta Oriani
www.code2play.space

IF

componente dell'indice corrispondente, e anche questo è un dato che conosciamo perché abbiamo preparato noi la tavolozza.
Occorre quindi verificare **`SE iQuadrante è 0, 1, 2 o 3`**.
Possiamo utilizzare l'istruzione condizionale **`IF`** (se) facendola seguire dalla condizione da valutare. **`IF`** verifica se è vera la condizione posta, se vera si eseguono le istruzioni indicate nel costrutto.
`IF condizione [ istruzioni ]`
Questa è la sintassi, parola chiave **`IF`** seguita dalla **`condizione`** da valutare che può essere tra parentesi tonde o meno e all'interno delle parentesi quadre, le **`istruzioni`** da eseguire se la condizione è vera. In questo caso le parentesi quadre contengono un blocco di istruzioni, quindi, occorre lasciare lo spazio dopo la parentesi di apertura e prima della parentesi di chiusura.

Ricapitolando: se array non si deve lasciare alcuno spazio, se si tratta di istruzioni occorre lasciare lo spazio, in ogni caso LibreLogo segnala l'errore di spazio mancante.

Se il quadrante è il primo, se **`iQuadrante`** ha valore 0 allora occorre cambiare la componente con indice 0 di **`colore`**, siamo nel quadrante con il colore rosso **`[255,0,0]`**.
Per assegnare il nuovo valore contenuto in **`delta`** è sufficiente assegnarlo all'elemento corrispondente dell'array **`colore`**. Sempre restando nell'esempio del rosso è sufficiente scrivere **`colore[0] = delta`**.
Quadrante Rosso → cambia la componente R, indice 0

```
IF (iQuadrante = 0) [ colore[0] = delta ]
```

Quadrante Giallo → cambiano le componenti R e G, indici 0 e 1

```
IF (iQuadrante = 1) [ colore[0] = delta ]
IF (iQuadrante = 1) [ colore[1] = delta ]
```

Quadrante Blu → cambia la componente B, indice 2

```
IF (iQuadrante = 2) [ colore[2] = delta ]
```

Quadrante Bianco → cambiano tutte le componenti, indici 0, 1 e 2

```
IF (iQuadrante = 3) [ colore[0] = delta ]
IF (iQuadrante = 3) [ colore[1] = delta ]
IF (iQuadrante = 3) [ colore[2] = delta ]
```

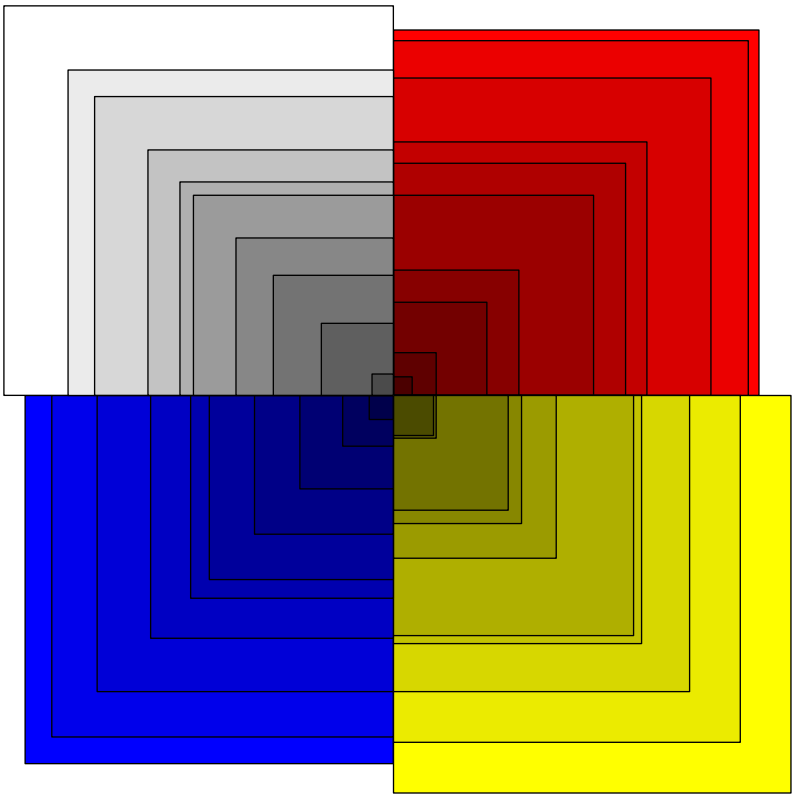
Dopo le assegnazioni di delta si può impostare il colore di riempimento

```
FILLCOLOR colore
```

Di seguito la procedura così modificata

```
TO RipetiQuadrati iQuadrante
    tavolozza = [[255,0,0], [255,255,0], [0,0,255],
[255,255,255]]
    colore = tavolozza[iQuadrante]
    REPEAT 10 [
```

```
            lato = ((165 - (15*REPCOUNT)) - INT(RANDOM 15))
            delta = 255 - (20*(REPCOUNT-1))
            IF (iQuadrante = 0) [ colore[0] = delta ]
            IF (iQuadrante = 1) [ colore[0] = delta ]
            IF (iQuadrante = 1) [ colore[1] = delta ]
            IF (iQuadrante = 2) [ colore[2] = delta ]
            IF (iQuadrante = 3) [ colore[0] = delta ]
            IF (iQuadrante = 3) [ colore[1] = delta ]
            IF (iQuadrante = 3) [ colore[2] = delta ]
            FILLCOLOR colore
            DisegnaQuadrato lato
            FILL
    ]
END
```



11. TROPPE RIGHE DI CODICE [mondrian_11.odt]

Si può fare meglio? Troppi **IF** tutti in una volta, comodi, certamente e un valido primo approccio perché si arriva al risultato che si voleva ottenere, ma c'è un'altra strada?
In informatica non c'è *LA* via, ce ne sono tante, la scelta di quale usare dipende da molti fattori, non ultimo il proprio grado di esperienza e lo scopo del codice, e, perché no, il grado di leggibilità.
Vediamo, quindi, un'altra via, che non è meglio o peggio ma permette di fare alcune riflessioni.

Monica Umberta Oriani
www.code2play.space

Il valore che deve cambiare è qualsiasi valore diverso da 0, ci troviamo all'interno del ciclo, il valore è 255 quando entra ma al secondo ciclo il valore cambia, se si valutasse solo 255 lo si avrebbe solo al primo ciclo, dopo il colore sarebbe sempre lo stesso. Diverso e non maggiore di 0 perché abbiamo visto che se il numero è negativo viene trattato come positivo (non è previsto un valore negativo per una terna di componenti colore, LibreLogo ci corregge l'errore).

Questo significa che se noi potessimo scrivere qualcosa tipo
`SE colore[qualsiasi indice] diverso da 0 allora assegna delta a colore[questo indice]`
[quasiasi indice] può assumere solo tre valori 0, 1 e 2.
Il valore del quadrante è in effetti ininfluente, quello che conta è che l'elemento 0, 1 o 2 non abbia valore 0.
Potremmo utilizzare una ripetizione per tre cicli e utilizzare un indice per 0, 1, 2 e valutare se **`colore[x] è diverso da 0`**, oppure possiamo utilizzare un altro tipo di ciclo come **`FOR i IN [valori] [ istruzioni ]`**, in questo modo possiamo utilizzare i con il valore che ci serve.

`FOR … IN`

```
FOR i IN [0,1,2] [
    IF colore[i] != 0 [ colore[i] = delta ]
]
```

`!= (diverso)`

Il punto esclamativo seguito dal segno di uguale (**`!=`**) indica diverso. "**`i`**" assume di volta in volta i valori contenuti nell'array che segue la parola chiave **`IN`**, più semplicemente "per ogni valore contenuto in array esegui le istruzioni".
Così "**`i`**" vale 0 al primo ciclo, 1 al secondo e 2 al terzo. Se il valore dell'elemento corrispondente è diverso da zero gli viene assegnato il valore di **`delta`**, altrimenti non verrà modificato nulla.
La procedura è ora così:

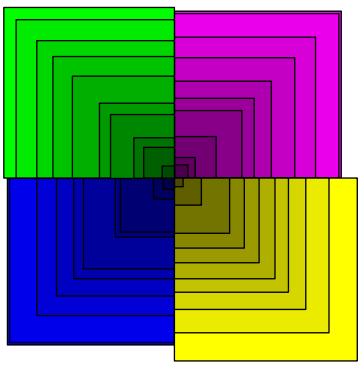
```
TO RipetiQuadrati iQuadrante
    tavolozza = [[255,0,0], [255,255,0], [0,0,255],
[255,255,255]]
    colore = tavolozza[iQuadrante]
    REPEAT 10 [
          lato = ((165 - (15*REPCOUNT)) - INT(RANDOM 15))
          delta = 255 - (20*(REPCOUNT-1))
          FOR i IN [0,1,2] [
                IF colore[i] != 0 [ colore[i] = delta ]
          ]
          FILLCOLOR colore
          DisegnaQuadrato lato
          FILL
    ]
END
```

Non si tratta solo di aver sostituito 7 righe di codice con una (qui è su tre per maggior facilità di lettura, è possibile, infatti, mettere tutto sulla stessa riga avendo l'accortezza di lasciare lo spazio dopo l'apertura di parentesi e prima della chiusura).

Si tratta di aver reso il codice più versatile.
Prima funzionava solo con quella precisa tavolozza in quel preciso ordine, ora potremmo utilizzare il magenta al posto del rosso e il verde al posto del bianco e il codice funzionerebbe lo stesso senza dover modificare altro.

```
tavolozza = [[255,0,255], [255,255,0], [0,0,255], [0,255,0]]
```

In questo caso i colori sono: magenta, giallo, blu, verde.



12. CECI N'EST PAS UN CARRÉ [Mondrian_12.odt]

Sì, nonostante la citazione omaggio a Magritte il progetto è ancora *Mondrian al quadrato*.
In effetti questo non è un quadrato e allora proviamo a renderlo meno quadrato, proviamo a stondare gli angoli.
Si tratta di sostituire i quadrati con quarti di cerchio.

CIRCLE

La soluzione più immediata è quella di utilizzare la funzione **`CIRCLE`** di LibreLogo, come dice il nome disegna un cerchio, ma può fare anche molto di più.

SQUARE

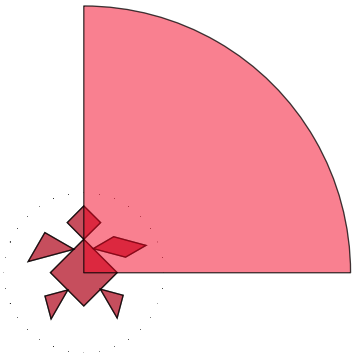
Sì, c'è anche una funzione **`SQUARE`**, ma per il momento va bene la nostra **`DisegnaQuadrato`**.

`CIRCLE` permette di fare diverse cose, si diceva:

- può disegnare cerchi o ellissi
- può disegnare una sezione di cerchio o ellisse tracciandone solo il contorno
- può disegnare una sezione di cerchio o ellisse tracciando solo il segmento della circonferenza
- è possibile indicare la porzione di cerchio o ellisse in gradi o in ore dell'orologio

In particolare è interessante l'ultimo punto, in gradi il riferimento è assoluto 0° è in alto, 90° a destra, 180° in basso e 270° a sinistra. In ore dell'orologio **`12h`** è in alto, ma questa volta il riferimento è relativo, relativo alla tartaruga, quindi è in alto, anzi, davanti alla tartaruga. Le **`3h`** sono a destra della tartaruga. È come se la tartaruga indossasse l'orologio.
Per fare un quarto di cerchio orientato come il quadrato iniziale, con la

tartaruga orientata verso l'alto, dovremo quindi utilizzare **12h** come punto di partenza e **3h** come punto di arrivo, è il quadrante in alto a destra, come il primo quadrato che abbiamo disegnato.



Possiamo, allora, creare una procedura che chiameremo, per uniformità, **DisegnaCerchio**, ben consci che si sta disegnando un quarto di cerchio.

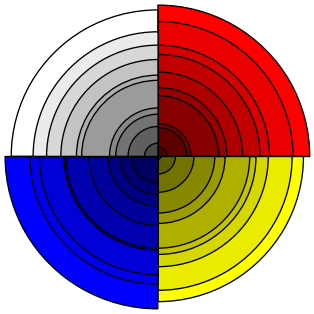
```
TO DisegnaCerchio lato
    CIRCLE [lato*2, lato*2, 12h, 3h]
END
```

Si deve passare come parametro il diametro, in questo caso dovremo moltiplicare il valore di **lato*2**, perché **lato** rappresenta il raggio.
Il disegno del cerchio, o di un segmento di cerchio, inizia dal centro. Si deve passare due volte il parametro del diametro perché con la stessa funzione è possibile anche disegnare ellissi che di diametri ne hanno due differenti.
Possiamo mettere questa procedura subito dopo **DisegnaQuadrato**.
A questo punto possiamo chiamare **DisegnaCerchio** al posto di **DisegnaQuadrato**, non eliminiamo la chiamata alla precedente procedura, la possiamo commentare. Un **commento** è una parte scritta all'interno di uno script che non viene eseguita perché preceduta da segni che dicono al sistema di non considerare quella parte di testo. In LibreLogo il commento è preceduto dal punto e virgola ("**;**").

commento

```
DisegnaCerchio lato
;DisegnaQuadrato lato
```

la prima istruzione verrà eseguita, la seconda no.
Teniamo un attimo da parte **DisegnaQuadrato** perché nel prossimo punto la utilizzeremo ancora.



Ed ecco il risultato

13. A CORRENTE ALTERNATA [mondrian_13.odt]

E se volessimo alternare quadrati con cerchi?
Ad esempio, se il quadrante è pari allora quadrato, altrimenti cerchio.
Come capire se un numero è pari o dispari, *un numero è pari se diviso per due dà resto pari a 0*. Detta così è semplice, ma come lo diciamo al sistema?

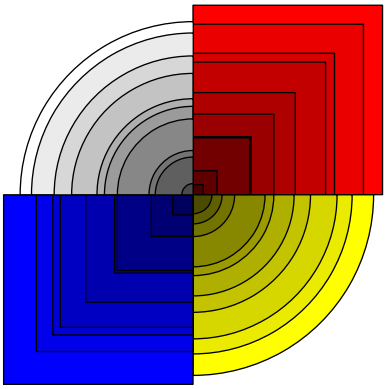
modulo %

Ci viene in soccorso l'operatore **modulo** (**%**), il modulo ci restituisce il resto di una divisione, è un operatore molto sfruttato nei codici, ….almeno nei miei.
Si utilizza scrivendo **dividendo modulo divisore**.

Dovremo valutare una condizione, ovvero, utilizzare il costrutto condizionale **IF** e in questo caso avremo delle istruzioni sia nel caso in cui la condizione è vera e sia nel caso in cui la condizione è falsa.
IF condizione [istruzioni se vera] [istruzioni se falsa]

Ci sono, quindi, due blocchi di codice racchiusi tra parentesi quadre, lo spazio è necessario anche tra la parentesi di chiusura del primo blocco e la parentesi di apertura del secondo blocco.

```
IF iQuadrante % 2 = 0 [
    DisegnaQuadrato lato
] [
    DisegnaCerchio lato
]
```



Erano quattro quadrati in forma di quadrato.

14a. DAL QUADRATO ALLA STELLA [mondrian_14a.odt]

Dal particolare all'universale. Proposito pomposo, ma si può pensare di far comporre la figura da più di quattro elementi? Sembra di no per quanto riguarda i quadrati, almeno. Ma se i quadrati si trasformassero in rombi? Una trasformazione che deve avvenire in funzione del numero di suddivisioni.
È possibile trasformare la procedura **DisegnaQuadrato** per disegnare rombi? E poter disegnare ancora quadrati? Il quadrato è l'eccezione del

Monica Umberta Oriani
www.code2play.space

rombo.

Il rombo è caratterizzato dall'avere i lati tutti uguali e gli angoli uguali a due a due.
Rivediamo abbiamo disegnato il quadrato:

```
REPEAT 4 [
    FORWARD lato
    RIGHT 90
]
```

Si ripete per quattro volte il disegno del lato e la rotazione per prepararsi alla nuova linea. La rotazione è di 90°, questo potrebbe generare confusione. Nel quadrato 90° è la misura sia dell'angolo interno e sia quello esterno. Nella geometria della tartaruga occorre tenere presente che gli angoli sono quelli esterni.
Per il rombo un modo di procedere potrebbe essere:

- tracciare un lato
- ruotare per una misura pari al complementare dell'angolo ottuso del rombo, che altro non è che la stessa dell'angolo acuto
- tracciare il secondo lato
- ruotare per una misura pari al complementare dell'angolo acuto del rombo

...e si ripetono queste operazioni una seconda volta.

```
REPEAT 2 [
    FORWARD lato
    RIGHT angolo acuto
    FORWARD lato
    RIGHT complementare dell'angolo acuto (180-angolo acuto)
]
```

Quanto vale l'angolo acuto?
Questa misura ce l'abbiamo già: basta dividere l'angolo giro per le suddivisioni volute.

```
REPEAT 2 [
    FORWARD lato
    RIGHT 360/nSuddivisioni
    FORWARD lato
    RIGHT 180-(360/nSuddivisioni)
]
```

Sostituiamo il codice all'interno di **`DisegnaQuadrato`** con queste righe. Abbiamo messo **`nSuddivisioni`** per lasciare ampia libertà su quanti rombi comporranno la figura finale, significa, quindi, che a **`DisegnaQuadrato`** dovremo passare questo parametro, oltre al parametro **`lato`**

```
TO DisegnaQuadrato lato nSuddivisioni
    REPEAT 2 [
          FORWARD lato
```

```
            RIGHT 360/nSuddivisioni
            FORWARD lato
            RIGHT 180-(360/nSuddivisioni)
        ]
END
```

Se provassimo ora ad eseguire il codice riceveremmo un messaggio di errore, ancora non viene passato il parametro **nSuddivisioni** e dobb iamo predisporre anche il resto del codice per questo cambiamento.

Un passo per volta, si è detto anche in altre occasioni.
Il primo è passare il parametro ovunque sia necessario:

- quando si richiama la funzione **DisegnaQuadrato**

```
    DisegnaQuadrato lato nSuddivisioni
```

- **DisegnaQuadrato** si trova all'interno di **RipetiQuadrati** e, quindi, anche a **RipetiQuadrati** bisognerà passare **nSuddivisioni**

```
    RipetiQuadrati REPCOUNT-1 nSuddivisioni
```

- occorrerà anche metterlo nella dichiarazione della procedura **RipetiQuadrati**

```
    TO RipetiQuadrati iQuadrante nSuddivisioni
```

- **DisegnaFigura** richiama tutte le altre e anche in questo caso dovremo aggiungerlo, nella dichiarazione

```
    TO DisegnaFigura nSuddivisioni
```

- E quando la si richiama basterà indicare il numero voluto

```
    DisegnaFigura 8
```

Mettiamo un attimo da parte **DisegnaCerchio**, ci torneremo nel prossimo punto. Restiamo sui rombi. Per mettere da parte **DisegnaCerchio** possiamo commentare dove viene richiamata, facendo attenzione a commentare anche le parti del costrutto **IF** che valutano quando va chiamata, così:

```
; IF iQuadrante % 2 = 0 [
    DisegnaQuadrato lato nSuddivisioni
; ] [
;   DisegnaCerchio lato
; ]
```

Rimane attiva solo la chiamata **DisegnaQuadrato**.

Ora possiamo provare ad eseguire il codice, vedremo che disegna solo 4 rombi invece degli 8 indicati.

DisegnaFigura ha ancora solo 4 ripetizioni e la rotazione è ancora di

90°.
Sostituiamo il valore **4** di **REPEAT** con **nSuddivisioni** e la rotazione non sarà più di 90° ma **360/nSuddivisioni**, in questo modo:

```
TO DisegnaFigura nSuddivisioni
    REPEAT nSuddivisioni [
          RipetiQuadrati REPCOUNT-1 nSuddivisioni
          RIGHT 360/nSuddivisioni
    ]
END
```

Eseguendo il codice, dopo i primi 4 rombi perfettamente disegnati, si ha un nuovo messaggio di errore: "Indice fuori dall'area". Questo è un messaggio ci comunica che abbiamo richiesto un elemento di un array con un valore di indice al di fuori di quelli possibili per quell'array.
Ci comunica anche il numero di riga in cui è presente l'errore, vediamo, così, che è la riga dove viene richiamato **colore** da **tavolozza** utilizzando l'indice di **iQuadrante**

```
colore = tavolozza[iQuadrante]
```

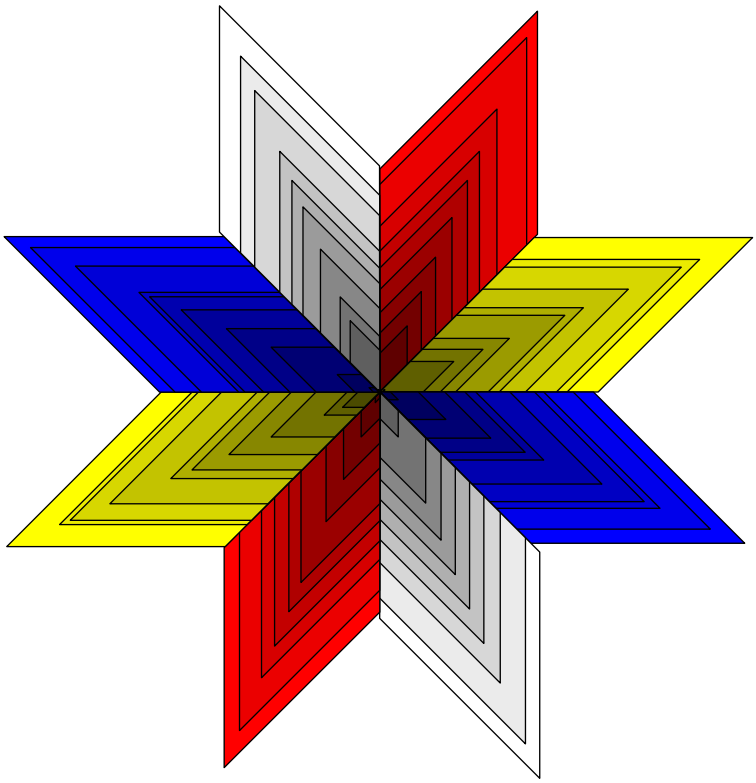
Prima, infatti, avevamo quattro quadranti e quattro colori per tavolozza, ora i quadranti-settori possono essere più numerosi dei colori.
Occorre fare in modo che il numero di indice che viene passato a **tavolozza** possa essere un valore tra 0 e 3. Torna l'operatore **modulo**, il dividendo è il valore di **iQuadrante**, il divisore il numero di elementi di **tavolozza**, in questo modo il resto della divisione può essere solo un valore presente nell'array **tavolozza**. Ad esempio: **iQuadrante** vale 6, il numero di elementi di tavolozza è 4, il resto della divisione di 6 / 4 è 2. Se il numero di elementi di **tavolozza** è 4, i valori di modulo possibili sono 0,1,2,3. Esattamente quello che si cercava. Si utilizza la funzione **len** (in minuscolo) per avere il valore del numero di elementi presenti nell'array **tavolozza**. **len** è un'istruzione di Python, che esegue anche LibreLogo, e restituisce il numero di elementi presenti in un array. La scelta di utilizzare **len(*nomeArray*)** e non il numero 4 è dettata dalla volontà di rendere il codice più duttile, è possibile così aggiungere un nuovo colore a **tavolozza** senza dover cambiare altro.

len()

L'istruzione **len(*nomeArray*)** necessita di parentesi tonde dentro le quali si scrive il nome dell'array.

```
colore = tavolozza[iQuadrante % len(tavolozza)]
```

Ed ecco il risultato



Provare a richiamare **DisegnaFigura** con il valore di **4**.

```
DisegnaFigura 4
```

Si ottiene la figura di partenza.

14b. UN ALTRO MODO PER DISEGNARE UN ROMBO [mondrian_14b.odt]

C'è almeno un altro modo per disegnare il rombo, magari lasciando le 4 ripetizioni?
In effetti quello che cambia è solo la rotazione, che si ripete in modo alternato, basterebbe quindi sapere se il numero di ciclo è il primo, il secondo, il terzo o il quarto. Per essere più precisi è sufficiente sapere se il numero di ciclo è il primo o il terzo oppure se il secondo o il quarto. Una valutazione a coppie, quindi, e osservando meglio, se il numero di ciclo è pari o dispari.
Lo abbiamo visto al punto 13 come stabilire se un numero è pari o dispari, dove abbiamo anche visto **IF condizione [*se vera*] [*se falsa*]**.
Anche in questo caso possiamo sfruttare la stessa cosa con l'aiuto del contatore **REPCOUNT** e utilizzare un valore di rotazione o l'altro a seconda del valore di ciclo corrente.

```
TO DisegnaQuadrato lato nSuddivisioni
    REPEAT 4 [
```


Monica Umberta Oriani
www.code2play.space

```
            FORWARD lato
            IF REPCOUNT % 2 = 0 [
                  RIGHT 180-(360/nSuddivisioni)
            ] [
                  RIGHT 360/nSuddivisioni
            ]
      ]
END
```

L'ordine del valore degli angoli di rotazione è al contrario rispetto al codice precedente perché **REPCOUNT** inizia da 1, pertanto al primo ciclo la condizione è falsa e ricade nel secondo blocco di istruzioni, che è proprio il primo valore che ci serve.

15. UNA STELLA CON SPICCHI [mondrian_15.odt]

E ora riattiviamo **DisegnaCerchio**. Basta togliere il punto e virgola di commento.
Aggiungiamo **nSuddivisioni** anche alla chiamata di **DisegnaCerchio**, anche gli spicchii di cerchio saranno in funzione del numero delle suddivisioni volute.

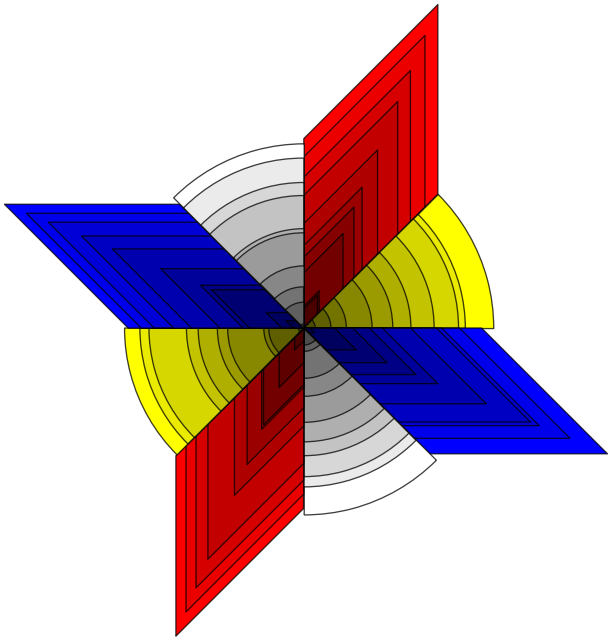
```
DisegnaCerchio lato nSuddivisioni
```

Mettiamo **nSuddivisioni** anche nella dichiarazione della procedura:

```
TO DisegnaCerchio lato nSuddivisioni
```

Nella funzione **CIRCLE** occorre cambiare l'ora di arrivo, la partenza sarà sempre **12h**, ma l'arrivo sarà **12h/nSuddivisioni**, **12** sono le ore possibili, il giro dell'orologio, e occorre aggiungere **h**, questo perché non è un semplice numero, ma un numero con l'indicazione dell'unità di misura.

```
TO DisegnaCerchio lato nSuddivisioni
    CIRCLE [lato*2, lato*2, 12h, 12h/nSuddivisioni]
END
```



E se invece di **8** ad **nSuddivisioni** si assegna il valore **4** si torna alle figure precedenti.

Monica Umberta Oriani
www.code2play.space

ISTRUZIONI

In corsivo il tipo di parametro (valore) necessario all'istruzione

; *(commento) - carattere che indica l'inizio di un commento*

!= *(diverso) - Operatore di confronto*

% *(nodulo) - Operatore matematico, restituisce il resto di una divisione → ad esempio: 9 % 2 restituisce 1, 9 % 3 restituisce 0*

ANY *- Valore casuale predefinito di colore*

CIRCLE *numero - Disegna un cerchio con diametro pari al valore indicato dal parametro*

CLEARSCREEN [CS] *- Pulisci schermo*

FILL *- Riempi*

FILLCOLOR *quale colore - Imposta colore di riempimento*

FILLTRANSPARENCY *numero - Imposta colore con percentuale di trasparenza definita dal valore del parametro*

FOR *elemento* **IN** [*lista*] **[** *istruzioni* **]** *- Esegue le istruzioni tra le parentesi quadre per ogni elemento contenuto nella lista*

FORWARD *numero* [FD] *- Avanti di quanto*

HIDETURTLE *- Nascondi tartaruga*

HOME *- Riporta la tartaruga al centro del foglio*

IF (*condizione*) [*istruzioni*] *- Esegue le istruzioni tra le parentesi quadre se la condizione è vera*

INT *- Restituisce la parte intera di un valore numerico*

len(*elemento***)** *- Restituisce la lunghezza dell'elemento indicato tra parentesi, va scritto in minuscolo perché funzione di Python. L'elemento può essere una stringa o un array.*

PENCOLOR *quale colore - Imposta il colore della penna, le linee*

RANDOM *tra quali valori - Sceglie un valore casuale nell'intervallo dato*
RANDOM 100 → valore numerico decimale tra 0 e 99
RANDOM "stringa" → lettera casuale contenuta in "stringa"

RANDOM [elemento1, elemento2, elemento4] → un elemento della lista

REPCOUNT - Contatore di ciclo

REPEAT *numero* [*istruzioni*] - *Ripeti quante volte [istruzioni da ripetere]*

RIGHT *numero* [RT] - *Destra di quanto*

SHOWTURTLE - Mostra *tartaruga*

SQUARE *numero* - *Disegna un quadrato con lato pari al valore indicato dal parametro*

TO *nomeProcedura parametri istruzioni* **END** - *Permette di assegnare un nome ad un gruppo di istruzioni, il numero dei parametri è opzionale e può anche non esserci alcun parametro. Le parole chiave TO e END non devono essere sulla stessa riga*