Grafica con Red (autore: Vittorio Albertoni)

Premessa

Questo manualetto è complementare a quello intitolato «Red» e allegato al mio articolo «Red: un linguaggio rivoluzionario» pubblicato nell'ottobre 2023 sul mio blog *www.vittal.it*.

Là sono esposte le basi del linguaggio Red e ciò che serve sapere per produrre programmi per console, cioè eseguibili su terminale.

Qui vediamo come sia possibile arricchire questi programmi con interfaccia grafica, e per questo è ovviamente innanzi tutto necessario conoscere il linguaggio con cui scriverli senza interfaccia grafica.

Vediamo anche come sia possibile produrre semplicemente disegni, e per questo potremmo anche prescindere dalla completa conoscenza del linguaggio di programmazione vero e proprio, ma lo sconsiglio.

Al richiamato manualetto, in ogni caso, rimando per quanto riguarda la predisposizione del computer (Capitolo 1) e il funzionamento di base (Capitolo 2).

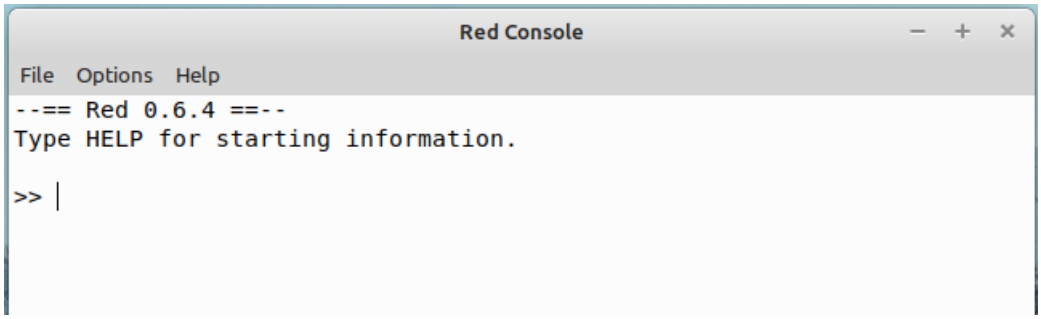
Indice

1 Il motore grafico di Red

Il sistema grafico per il linguaggio di programmazione Red si chiama View Graphic Engine, o semplicemente View.

Esso è disponibile utilizzando il file classificato GUI Red di quelli che scarichiamo per predisporre il computer, con un nome che incomincia per red-view e che consiglio di ribattezzare brevemente come `redv`.

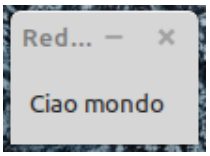
Se scriviamo a terminale il nome di questo file apriamo una console grafica, che assomiglia al terminale di sistema ma è invece un oggetto grafico di Red/View



nella quale possiamo scrivere istruzioni in linguaggio Red e vederle eseguite nella console stessa, come fossimo sul terminale di sistema aperto su Red, ma possiamo anche scrivere istruzioni nel dialetto chiamato VID con cui realizziamo oggetti grafici.

```
>> print "Ciao mondo"
Ciao mondo
>> 6 * 8
== 48
>> view [text "Ciao mondo"]
```

Nell'illustrazione, le prime due istruzioni sono in linguaggio Red e sono eseguite in console, la terza è in linguaggio VID, incorporato nel linguaggio Red, e produce questo



Se scriviamo il programma con un editor di testo per eseguirlo poi, compilato o meno, dobbiamo richiamare il motore grafico nella prima riga del file di programma con

```
Red [needs:  view]
```

Il resto del programma sarà un misto di linguaggio Red e di linguaggio VID se stiamo lavorando ad un programma con GUI o sarà quasi interamente in linguaggio Draw, il linguaggio nel linguaggio per disegnare, se stiamo lavorando ad un programma per eseguire semplici disegni.

La grafica in Red ha un suo gergo particolare nel quale gli elementi con cui si costruiscono gli oggetti grafici, che penso universalmente si chiamino widgets, qui si chiamano faces.

Le caratteristiche di questi oggetti, che in altri contesti esprimiamo attraverso l'indicazione di opzioni, in Red si indicano facendo seguire al nome del face una serie di dati, chiamati facet, dal cui tipo Red capisce che caratteristica indicano e la attribuisce al face: per esempio, se il dato è di tipo pair! ci si riferisce alla dimensione del face, se il dato è di tipo string! ci si riferisce a un testo da scrivere sul face.

Ancora a seguire, se al face indicato vogliamo abbinare l'esecuzione di qualche azione, indichiamo l'azione stessa in un blocco.

Per esempio, con

```
button 50x20 "ESCI" [quit]
```

creiamo un pulsante di dimensione 50 x 20 in pixel con sopra scritto ESCI, premendo il quale chiudiamo la finestra e abbandoniamo il programma.

Come si vede, anche nella grafica Red manifesta la propria assoluta originalità.

Si vede anche che l'istruzione, che è scritta in linguaggio VID, per essere compresa appieno richiede la conoscenza delle basi del linguaggio Red. Solo così sappiamo che 50x20 esprime un valore di tipo pair!, che una parola scritta tra doppi apici è di tipo string! e che un blocco è qualche cosa di inserito tra parentesi quadre.

2 GUI

Un programma corredato da interfaccia utente grafica tipicamente si compone, nell'ordine, dei seguenti elementi:
. creazione e settaggio del contenitore,
. definizione del layout,
. inserimento dei face con eventuale indicazione delle azioni collegate.

2.1 Contenitore

Il comando costruttore della finestra contenitore è
`view []`

Con questo comando generiamo una finestrella senza titolo, a sfondo grigio chiarissimo, al centro dello schermo, con uno spazio contenitore di 80x80 pixel che automaticamente si allarga per contenere tutti i face che vi inseriamo.

Per avere una finestra contenitore come la vogliamo noi abbiamo a disposizione una serie di strumenti.

Innanzi tutto, all'interno delle parentesi quadre che seguono il comando costruttore possiamo indicare queste caratteristiche della finestra che vogliamo:
`title` seguito da una stringa indicante il titolo della finestra,
`size` seguito dal pair! con cui indicare le dimensioni in pixel della finestra,
`backdrop` seguito dal nome inglese di un colore[1] o da una tupla RGB per indicare il colore di fondo della finestra.

Con
```
view [title "FINESTRA DI PROVA" size 300x200 backdrop green]
```

[1] I nomi di colori riconosciuti da Red sono i seguenti

otteniamo la finestra

collocata al centro dello schermo
(avremmo ottenuto lo stesso colore con `backdrop 0.255.0`).

Se vogliamo collocare la finestra in posizione diversa dal centro dello schermo dobbiamo introdurre l'opzione offset con questa sintassi:
`view/options [<caratteristiche>] [offset: <posizione>]`
dove <posizione> è un pair! che indica la posizione sullo schermo dell'angolo in alto a sinistra della finestra.

Con
`view/options [title "FINESTRA DI PROVA" size 300x200 backdrop green] [offset:  30x50]`
otteniamo la finestra di prima, collocata a 30 pixel dal bordo sinistro dello schermo e a 50 pixel dal bordo superiore (l'origine delle coordinate dello schermo è infatti nell'angolo in alto a sinistra.

2.2 Layout

All'interno del contenitore dobbiamo sistemare i face che servono per comporre la nostra GUI.

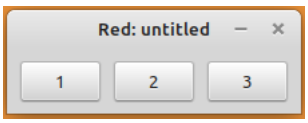
I face si inseriscono scrivendoli uno via l'altro, eventualmente seguiti dai relativi facet.

Il layout di default, che Red applica se non indichiamo altro, si chiama `across` e dispone i face uno di fianco all'altro nel contenitore partendo da sinistra.

Un semplice layout alternativo si chiama `below` e dispone i face uno sotto l'altro partendo dall'alto.

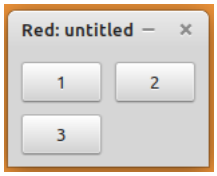
Nell'elencazione dei face possiamo inserire la parola `return` quando vogliamo che la collocazione dei face passi alla riga successiva se siamo in modalità across o alla colonna successiva se siamo in modalità below.

Esempi:
`view [across button "1" button "2" button "3"]`
o, semplicemente,
`view [button "1" button "2" button "3"]`
dispone i tre pulsanti così

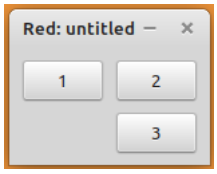


`view [below button "1" button "2" button "3"]`
dispone i tre pulsanti così

```
view [across button "1" button "2" return button "3"]
```
o, semplicemente
```
view [button "1" button "2" return button "3"]
```
dispone i tre pulsanti così




```
view [below button "1" return button "2" button "3"]
```
dispone i tre pulsanti così



Si tratta di modelli di layout così detti relativi, nei quali il posizionamento di un face avviene in conseguenza del posizionamento del face che lo precede.

Molto spesso, per costruire GUI sofisticate, conviene ricorrere al modello di layout con posizionamento assoluto, nel quale scegliamo noi dove va posizionato il face facendo precedere al suo nome la parola chiave `at` seguita da un pair! che indica il punto dove posizionare l'angolo in alto a sinistra del face.

Esempio:
```
view [at 10x10 button "1" at 10x50 button "2" at 80x30 button "3"]
```
dispone i pulsanti così

Per GUI molto sofisticate e per alleggerire il lavoro per determinare e indicare i punti di ancoraggio dei face può convenire suddividere il contenitore in più zone utilizzando il face `panel`, che vedremo subito, applicando ad ogni zona un modello di layout diverso, scegliendo il più conveniente e meno laborioso per ciò che vogliamo mettere nella zona stessa.

2.3 Face

Per utilizzare i face occorre sapere a cosa servono e quali sono i facet ai quali ogni face è sensibile: non tutti i facet, infatti, si adattano a ciascun face. Per esempio il face `button` non può essere colorato ma dobbiamo prenderlo con il colore grigio chiaro come ce lo passa il convento.

Una cosa molto importante da ricordare: il face è un oggetto e può essere assegnato ad una parola.

In simile contesto, i facet sono suoi attributi e a ciascuno di essi si accede con una notazione composta dal nome della parola cui è assegnato il face e dal nome del facet separati da una barra (`/`).

Per esempio, se è stato assegnato il face `button "OK"` alla parola `b` con
```
b:  button "OK"
```
`b/text` ritorna `OK`

I face che ci offre Red sono moltissimi.

Cominciamo da quello richiamato alla fine del precedente Capitolo e che ci consente di suddividere il contenitore in più zone:

2.3.1 panel

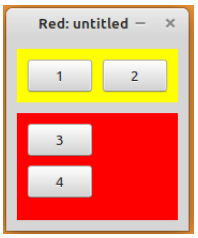
E' un face contenitore di altri face.

Gli si può dare una dimensione con un pair! e un colore di sfondo con un nome di colore o una tupla RGB e si dispongono i face al suo interno elencandoli in un blocco.

Con

```
view [below panel 150X50 yellow [button "1" button "2"] panel 150x100 255.0.0 [below button "3" button "4"]]
```

otteniamo



Possiamo inserire i panel in pagine diverse ricorrendo al face

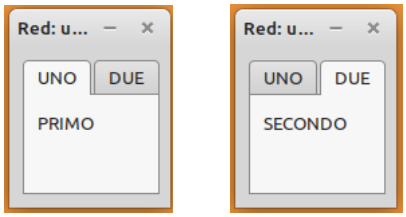
2.3.2 tab-panel

I panel non si dispongono sulla stessa pagina del contenitore ma in pagine diverse e sovrapposte, apribili cliccando sulla linguetta che le contraddistingue.

Con

```
view [tab-panel 100x100 ["UNO" [text "PRIMO"] "DUE" [text "SECONDO"]]]
```

otteniamo



dove la figura di destra si ottiene cliccando sulla linguetta DUE della figura di sinistra.

Ora vediamo i tre face che ricorrono in ogni GUI: il testo come etichetta, la finestra per l'immissione di testo e il pulsante.

2.3.3 text

E' quello che in altri linguaggi si chiama label.

Nelle GUI serve per inserire scritte destinate a istruzioni o a esposizione di risultati di elaborazioni.

Il testo si indica con una stringa e si può dare una dimensione allo sfondo con un pair! e un colore di sfondo con un nome di colore o una tupla RGB.

Per le caratteristiche del font abbiamo invece a disposizione tre istruzioni che vanno indicate dopo i facet:

. `font-name` seguita da un nome di font installato sul sistema su cui lavoriamo,
. `font-size` seguita da un numero intero indicante la dimensione del font in punti,
. `font-color` seguita da nome di un colore o da una tupla RGB ad indicare il colore del font.

Con

```
view [text "Ciao" cyan 90x30 font-name "eufm10" font-size 35 font-color red]
```

otteniamo

2.3.4 field

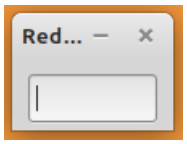
Serve per immettere dati in forma di testo da tastiera su una sola riga.

La relativa finestrella può essere dimensionata con un pair! ed ha sfondo bianco immutabile.

Con

view [field 80x25]

otteniamo



2.3.5 button

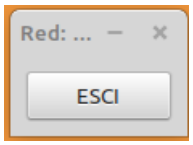
E' il classico pulsante su cui sui clicca per ottenere qualcosa.

Può essere dimensionato con un pair! e si può fare in modo che sia contraddistinto da una scritta, indicabile con una stringa, o da una icona/immagine, indicandone percorso e nome preceduti dal carattere %.

Con

view [button 90x30 "ESCI"]

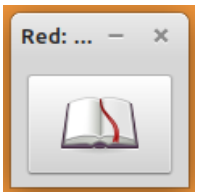
otteniamo



Con

```
view [button 90x60 %/home/vittorio/Immagini/dictionary.png]
```

otteniamo



Abbiamo poi due utilissimi face che possono servire per applicazioni d'ufficio o grafiche.

2.3.6 area

Serve per immettere testo su più righe.

Può essere dimensionata con un pair! ed ha per default sfondo bianco, che può essere modificato con un nome o con una tupla RGB.

Con

```
view [area 200x100 yellow]
```

otteniamo

2.3.7 base

Serve per creare altri face e per disegnare.

E' alla base della grafica di disegno di Red che vedremo più avanti.

Per default, con

```
view [base]
```

si presenta così

con una dimensione di 80x80 pixel e sfondo grigio.

Aggiungendo un pair! possiamo scegliere una diversa dimensione e possiamo scegliere un colore diverso indicandone il nome o attraverso una tupla RGB.

2.3.8 Altri face

Per altri face di arricchimento delle GUI, come check, radio, text-list, drop-down, drop-list, progress, slider, rimando alla documentazione ufficiale.

2.4 Eventi e azioni

Quando parliamo di eventi che hanno a che fare con una GUI alludiamo a manipolazioni effettuate attraverso la tastiera (pressione di uno o di determinati tasti) o attraverso il mouse (puntamento e/o pressione di uno dei tasti o uso della rotella del mouse).

Dalla documentazione ufficiale di Red possiamo vedere che gli eventi gestiti da questo linguaggio sono tantissimi. In questo manualetto mi limito a presentare quelli più comunemente usati:

. pressione del tasto sinistro del mouse (in Red denominato `down` o `click`),
. pressione del tasto INVIO della tastiera (in Red denominato `enter`),
. pressione di un qualsiasi tasto della tastiera (in Red denominato `change` o `key`).

Ciascuno dei face che abbiamo visto nel precedente paragrafo è predisposto per gestire un evento di default:

. pressione del tasto sinistro del mouse per il face `text`,
. pressione del tasto INVIO della tastiera per il face `field`,
. pressione del tasto sinistro del mouse per il face `button`,
. pressione di un qualsiasi tasto della tastiera per il face `area`,
. pressione del tasto sinistro del mouse per il face `base`.

Per abbinare azioni a questi eventi basta far seguire al nome del face e degli eventuali facet un blocco, che in Red si chiama `action facet`, in cui, tra parentesi quadre, si indica il codice da eseguire.

Con

```
view [button "PROVA" [print "Ciao"]]
```

apriamo una finestrella in cui compare un pulsante con la scritta PROVA e cliccando su di esso scriviamo la parola Ciao nella console di Red.

Se vogliamo che il pulsante reagisca ad un evento diverso da quello di default, appena prima del blocco per l'azione dobbiamo inserire la parola `on` seguita, separando con trattino, dal nome dell'evento voluto.

Per esempio, tra i tanti, Red prevede l'evento `over` che consiste nel passaggio del mouse sopra il face.

Con

```
view [button "PROVA" on-over [print "Ciao"]]
```
apriamo una finestrella in cui compare un pulsante con la scritta PROVA e semplicemente passando il mouse su di esso scriviamo la parola Ciao nella console di Red.

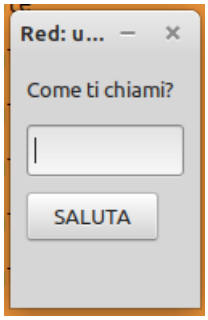
Negli esempi l'azione richiesta coinvolge la console e questo serve a dimostrare che si può fare, anche se contraddice l'uso della GUI, e soprattutto alla semplificazione degli esempi stessi.

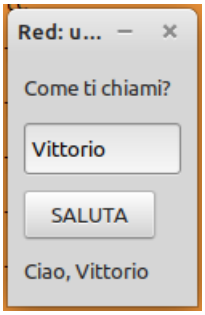
Per avere anche il risultato delle elaborazioni nella GUI si deve lavorare un po' di più sfruttando il fatto che, come ho richiamato all'inizio del paragrafo precedente, i face sono oggetti e i facet sono loro attributi.

Questo è un programmino, semplificato nell'aspetto grafico per rimanere all'essenziale, in cui viene chiesto il nome all'utente per salutarlo

```
Red [needs:  view]
view [
    below
    text "Come ti chiami?"
    i:  field
    button "SALUTA" [
        nome:  i/text
        insert o/text rejoin ["Ciao, " nome]
    ]
    o:  text ""
]
```

La figura di sinistra mostra la GUI quando il programma parte e la figura di destra mostra la GUI dopo che l'utente ha inserito il nome nell'apposita finestrella e premuto il pulsante SALUTA:





Con la sintassi minimale tipica del linguaggio Red, abbiamo la scelta del layout (`below`), la creazione di un'etichetta che invita ad inserire il nome, la creazione di un campo di inserimento testo assegnato alla variabile `i` (come input), la creazione di un pulsante con scritto SALUTA e la creazione con inizializzazione a stringa vuota di un'etichetta assegnata alla variabile `o` (come output).

Il blocco eseguibile in corrispondenza all'evento di default «pressione del tasto sinistro del mouse» sul pulsante crea una variabile nome assegnandole il testo inserito nel campo di inserimento `i` e finalmente aggiunge alla stringa vuota che costituisce il contenuto dell'etichetta `o` la concatenazione tra la stringa `"Ciao, "` e la stringa assegnata alla parola `nome`: più difficile descriverlo che farlo.

Ovviamente i nomi delle variabili possono essere scelti con qualsiasi criterio e l'indentazione che si vede nel modo con cui ho riprodotto il programma è stata adottata unicamente per rendere più chiaro il contenuto ad occhio umano.

Rammento che Red legge il programma prescindendo dagli a capo, come se fosse scritto tutto su una sola lunga riga: importante che vi sia sempre uno spazio che divide le parole.

2.5 Esempio

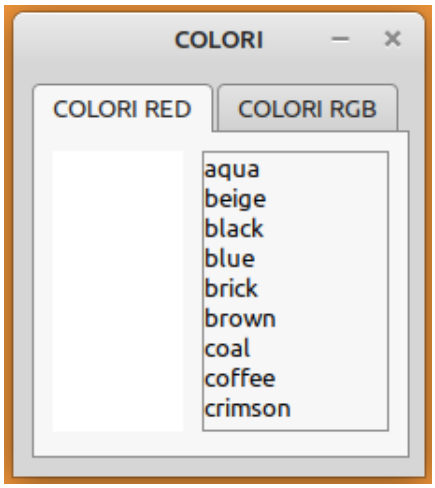
Per riepilogare un po' tutto propongo un programma non banalissimo che ci può servire, soprattutto per la scelta dei colori di tutto ciò che faremo nel prossimo Capitolo.

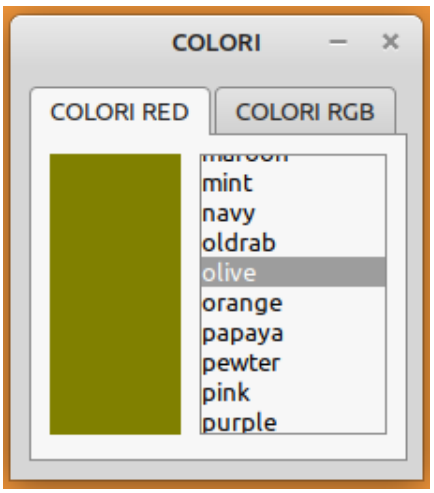
Il programma è questo:

```
Red [needs:  view]
view
[title "COLORI" tab-panel 200x200
    ["COLORI RED" [
        b1:  base 70x150 white
        t:  text-list 100x150 data ["aqua" "beige" "black" "blue" "brick" "brown" "coal"
        "coffee" "crimson" "cyan" "forest" "gold" "gray" "green" "ivory" "khaki" "leaf"
        "linen" "magenta" "maroon" "mint" "navy" "oldrab" "olive" "orange" "papaya"
        "pewter" "pink" "purple" "reblue" "rebolor" "red" "violet" "white" "yellow" " "]
        [switch t/selected [
            1 [b1/color:  aqua]
            2 [b1/color:  beige]
            3 [b1/color:  black]
            4 [b1/color:  blue]
            5 [b1/color:  brick]
            6 [b1/color:  brown]
            7 [b1/color:  coal]
            8 [b1/color:  coffee]
            9 [b1/color:  crimson]
            10 [b1/color:  cyan]
            11 [b1/color:  forest]
            12 [b1/color:  gold]
            13 [b1/color:  gray]
            14 [b1/color:  green]
            15 [b1/color:  ivory]
            16 [b1/color:  khaki]
            17 [b1/color:  leaf]
            18 [b1/color:  linen]
            19 [b1/color:  magenta]
            20 [b1/color:  maroon]
            21 [b1/color:  mint]
            22 [b1/color:  navy]
            23 [b1/color:  oldrab]
            24 [b1/color:  olive]
            25 [b1/color:  orange]
            26 [b1/color:  papaya]
            27 [b1/color:  pewter]
            28 [b1/color:  pink]
            29 [b1/color:  purple]
            30 [b1/color:  reblue]
            31 [b1/color:  rebolor]
            32 [b1/color:  red]
            33 [b1/color:  violet]
            34 [b1/color:  white]
            35 [b1/color:  yellow]
            ]
        ]
    ]
    "COLORI RGB" [
        text 60x15 "   Rosso" text 60x15 "  Verde" text 60x15 "    Blu" return
        i1:  field 55x15 i2:  field 55x15 i3:  field 55x15
        at 59x75 button "COLORE" [
            x1:  to integer!  i1/text
            x2:  to integer!  i2/text
            x3:  to integer!  i3/text
            x:  1.1.1
            x/1:  x1
            x/2:  x2
            x/3:  x3
            b2/color:  x
            ]
        at 10x115 b2:  base 184x52 white
        ]
    ]
]
```

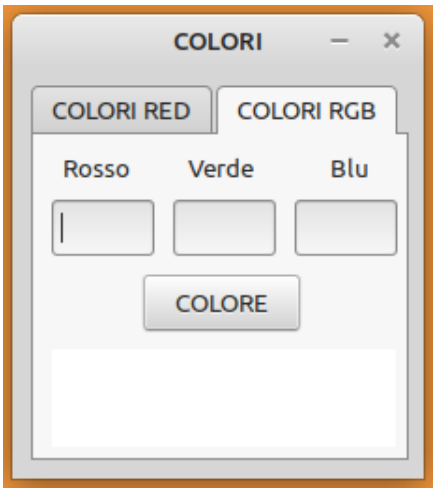
Esso serve per verificare visivamente il colore corrispondente ad una delle parole identificative del linguaggio Red oppure corrispondente ad una qualsiasi combinazione di valori nel sistema RGB, dove ogni colore corrisponde al mix di certe quantità (da 0 a 255) dei colori base Rosso Verde e Blu.

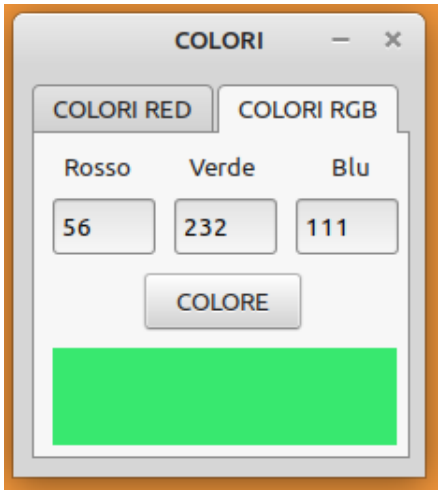
Al lancio del programma ci si presenta la GUI di sinistra, aperta sulla finestra dedicata a verificare il colore corrispondente alle 35 parole identificative del linguaggio Red.
Sulla destra abbiamo la stessa finestra dopo che abbiamo scelto nell'elenco la parola `olive`





Cliccando sulla linguetta COLORI RGB apriamo la finestra che compare sotto a sinistra, dedicata a verificare il colore corrispondente ad una combinazione RGB.
Sulla destra abbiamo la stessa finestra dopo che abbiamo inserito la combinazione 56 per il Rosso, 232 per il Verde e 111 per il Blu





Ognuna delle tre finestrelle per comporre la combinazione deve contenere un numero intero tra 0 e 255 prima di premere il pulsante COLORE per vedere il risultato della combinazione.

Lasciando al lettore l'esercizio di capire i vari passi del programma, richiamo soltanto l'attenzione sull'utilizzo del face `text-list`, che ho semplicemente nominato senza altri commenti nel paragrafo 2.3.8 tra gli Altri face, e sul meccanismo con cui è costruita la tupla per generare il colore in combinazione RGB.
Per avere un oggetto di tipo tupla è stato necessario costruirne una con valori provvisori e la giusta sintassi (`x: 1.1.1`) e poi sostituire i valori provvisori con quelli letti nelle finestrelle di immissione dati.
Per il resto si tratta di cose contenute, per la parte grafica, in questo manualetto e, per la parte di trattamento dei dati, nel manualetto `red.pdf` reperibile sul mio blog all'indirizzo *www.vittal.it*.

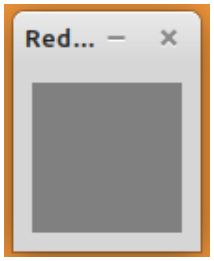
3 Disegno

Per disegnare occorre un piano su cui farlo e Red, a questo scopo, ci mette a disposizione il face base che abbiamo visto nel precedente Capitolo.

Come abbiamo appreso nel paragrafo 2.3.7 esso si si costruisce con un blocco view

```
view [base]
```

e, in mancanza di altre indicazioni, si presenta così



con una dimensione di 80x80 pixel e sfondo grigio, che ricorda il colore della lavagna.

Aggiungendo un pair! possiamo scegliere una diversa dimensione e possiamo scegliere un colore diverso indicandone il nome o attraverso una tupla RGB.

L'origine dei pixel si trova nell'angolo in alto a sinistra, le cui coordinate sono 0 e 0, che si indicano in Red con il pair! `0x0`. La prima cifra si riferisce all'asse orizzontale e la seconda cifra si riferisce all'asse verticale.
Con `10x20` indichiamo il pixel (praticamente un punto) a 10 pixel da sinistra e a 20 pixel dall'alto della lavagna.

3.1 Linee e figure

Per disegnare linee e figure facciamo seguire al face `base` il facet `draw` e un blocco che contiene le istruzioni per come costruire la linea o la figura, secondo la sintassi che indico nel seguito per ciascun tipo di linea o di figura.

Importante ricordare che il face è un oggetto e il facet è un metodo di quell'oggetto. Per cui se assegniamo alla parola a istruzioni per un disegno con

```
a:  base draw [<istruzioni>]
```

con `a/draw` accediamo al blocco `[<istruzioni>]`
Questo è molto utile per animare ciò che disegniamo.

3.1.1 line

Si costruisce con

`line` <punto> <punto>

dove i due punti, indicati con due pair!, sono i punti di partenza e di arrivo della linea.

Indicando più punti possiamo costruire una linea spezzata.

Con

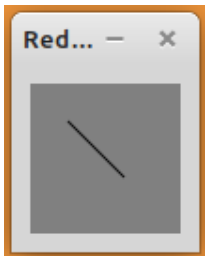
```
view [base draw [line 20x20 50x50]]
```

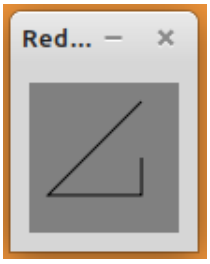
otteniamo la figura sulla sinistra.

Con

```
view [base draw [line 60x10 10x60 60x60 60x40]]
```

otteniamo la figura sulla destra.





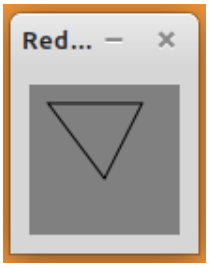
3.1.2 triangle

Si costruisce con
`triangle` <punto> <punto> <punto>
dove i tre punti, indicati con altrettanti pair!, sono i vertici del triangolo.

Con

```
view [base draw [triangle 10x10 40x50 60x10]]
```

otteniamo



3.1.3 box

Rettangoli e quadrati e si costruiscono con
`box` <punto> <punto>
dove i due punti, indicati con due pair!, corrispondono rispettivamente allo spigolo in alto a sinistra e allo spigolo in basso a destra della figura.

Possiamo aggiungere un numero intero per indicare il raggio di arrotondamento degli spigoli.

Con

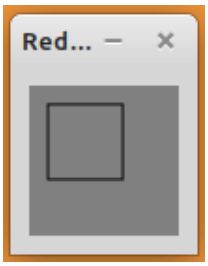
```
view [base draw [box 10x10 50x50]]
```

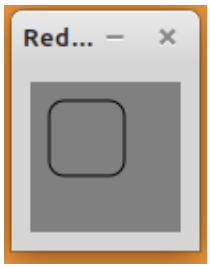
otteniamo la figura sulla sinistra.

Con

```
view [base draw [box 10x10 50x50 8]]
```

otteniamo la figura sulla destra





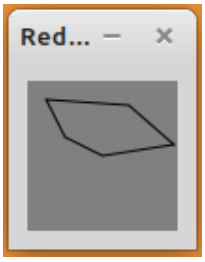
3.1.4 polygon

Si costruisce con
`polygon` <punto> <punto> <punto> `....`
dove i punti, indicati pair!, sono i vertici del poligono e la chiusura avviene automaticamente.

Con

```
view [base draw [polygon 10x10 20x30 40x40 78x34 54x13]]
```

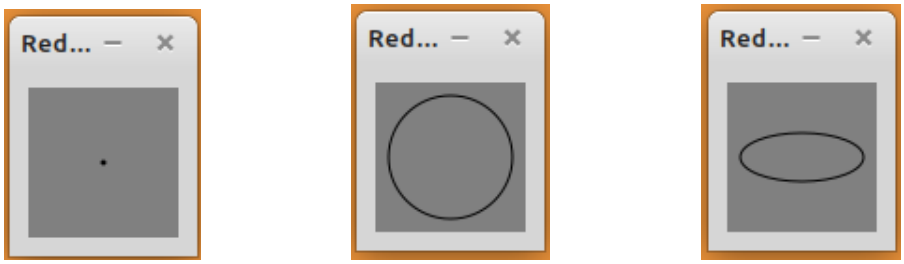
otteniamo



3.1.5 circle

Si costruisce con
`circle` <punto> <raggio>
dove <punto> è il pair! indicante il centro del cerchio e <raggio> è un numero che indica il raggio del cerchio stesso.
Indicando il raggio con il numero 1 costruiamo un punto localizzato dove indicato dal pair!
Se indichiamo due raggi costruiamo una ellisse avente come asse orizzontale il primo indicato e come asse verticale il secondo.

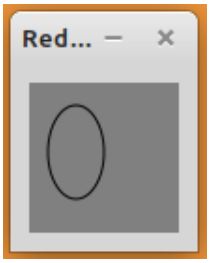
Con:
`view [base draw [circle 40x40 1]]`
otteniamo la figura di sinistra,
`view [base draw [circle 40x40 32,5]]`
otteniamo la figura centrale,
`view [base draw [circle 40x40 30 12.8]]`
otteniamo la figura di destra.



3.1.6 ellipse

Possiamo costruire una ellisse anche con
`ellipse` <punto> <punto>
dove i due punti, indicati da altrettanti pair!, sono lo spigolo in alto a sinistra e lo spigolo in basso a destra di un rettangolo immaginario in cui è inscritta l'ellisse voluta.

Con
`view [base draw [ellipse 10x12 30x50]]`
otteniamo



3.1.7 arc

Costruisce l'arco su un cerchio o ellisse immaginari con
`arc` <punto> <raggi> <inizio> <fine>
dove
<punto> è un pair! indicante il centro del cerchio o dell'ellisse,
<raggi> è un pair! il cui numero di sinistra indica il raggio orizzontale e il numero di destra indica il raggio verticale (se i due numeri sono uguali avremo un cerchio, altrimenti una ellisse,
<inizio> è un intero indicante, in gradi, l'angolo di inizio dell'arco,
<fine> è un intero indicante, in gradi, l'angolo di fine dell'arco.

Gli angoli di inizio e fine dell'arco si muovono in senso orario partendo dall'asse orizzontale a destra del centro.

Aggiungendo la parola `closed` otteniamo la chiusura dell'arco con due linee con origine nel centro.

Con

```
view [base draw [arc 40x40 30x30 90 160]]
```

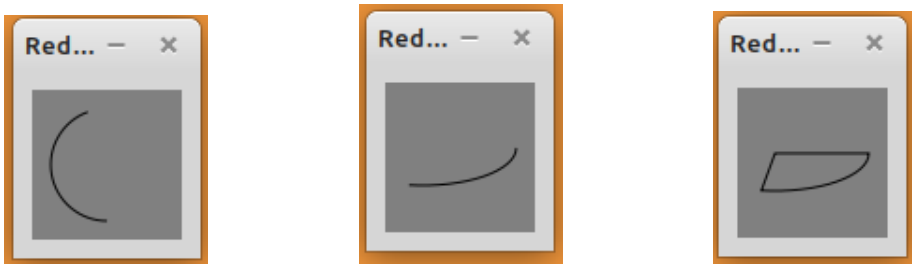
otteniamo la figura di sinistra,

```
view [base draw [arc 20x35 50x20 0 110]]
```

otteniamo la figura di centro,

```
view [base draw [arc 20x35 50x20 0 110 closed]]
```

otteniamo la figura di destra.



3.1.8 curve

Costruisce una curva di Bezier con
`curve` <inizio> <controllo> <fine>
dove
<inizio> è un pair! che indica il punto di inizio della curva,
<controllo> è un pair! che indica il punto di controllo,
<fine> è un pair! che indica il punto dove finisce la curva.

Con un solo punto di controllo abbiamo una curva di Bezier quadratica. Se indichiamo due punti di controllo abbiamo una curva di Bezier cubica.

Con

```
view [base draw [curve 20x60 13x11 58x43]]
```

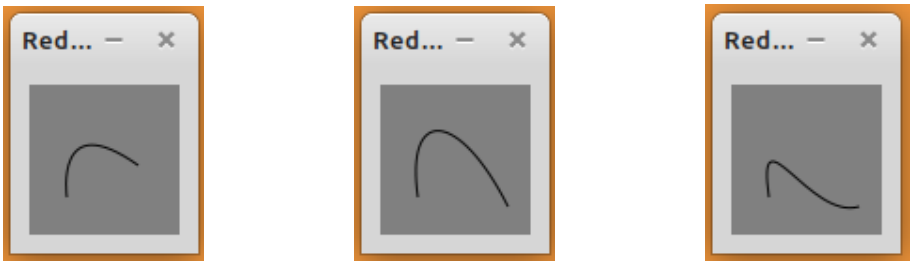
otteniamo la figura di sinistra,

```
view [base draw [curve 20x60 13x11 42x13 68x65]]
```

otteniamo la figura di centro,

```
view [base draw [curve 20x60 13x11 42x73 68x65]]
```

otteniamo la figura di destra.



3.1.9 spline

Traccia una curva che segue una sequenza di punti con
`spline` <punto> <punto> <punto> <punto> . . . .
dove i punti sono indicati con altrettanti pair!.

Aggiungendo la parola `closed` dopo l'elenco dei punti la curva si chiude.

Con

```
view [base draw [spline 20x60 13x11 58x43]]
```

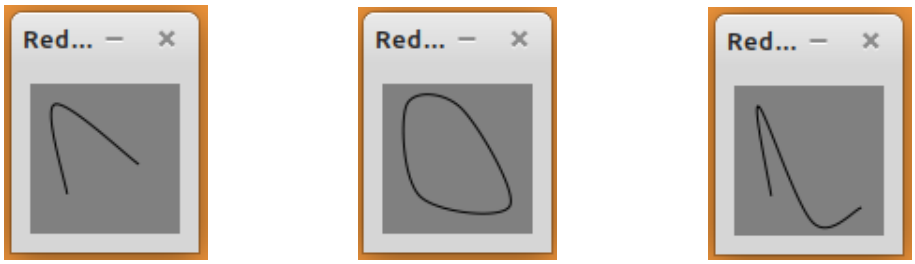
otteniamo la figura di sinistra,

```
view [base draw [spline 20x60 13x11 42x13 68x65 closed]]
```

otteniamo la figura di centro,

```
view [base draw [spline 20x60 13x11 42x73 68x65]]
```

otteniamo la figura di destra.



Ho utilizzato gli stessi punti utilizzati nel precedente paragrafo per le curve di Bezier per un utile raffronto.

A differenza di `curve`, che accetta un massimo di quattro parametri, `spline` ne accetta quanti vogliamo.

3.1.10 text

Con

`text` <punto> <stringa>

inseriamo una scritta, proposta come <stringa> tra doppi apici, il cui angolo superiore sinistro coincide con il <punto>, indicato con un pair!

Con

```
view [base draw [text 20x20 "Ciao"]]
```

otteniamo

* * *

E' possibile inserire più elementi elencandoli uno di seguito all'altro.

Per esempio, con

```
view [base draw [line 10x10 70x70 text 10x10 "Cerchio" circle 40x60 15]]
```

otteniamo

Gli esempi sono stati fatti utilizzando il face `base` di default (80 x 80 pixel e sfondo grigio).

Per modificare questa impostazione di default basta far seguire alla parola base un pair! per modificare la dimensione e/o un nome di colore o una tupla RGB per modificare il colore.

Con

```
view [base 200x100 yellow draw [circle 50x50 30]]
```

avremo il cerchio disegnato su un piano di 200 pixel per 100 con sfondo giallo.

Anche linee e figure sono state disegnate con impostazioni di default (spessore standard minimo, colore nero, senza riempimenti di colore per le figure).

Ora vediamo come governare queste impostazioni per ottenere disegni più avvincenti.

3.2 Proprietà delle linee

Tra quelli previsti dal linguaggio, penso che il più importante strumento di regolazione della proprietà delle linee, oltre a quello di dare loro un colore che vedremo nel prossimo paragrafo, sia quello di dare loro una dimensione.

Lo si può fare facendo precedere all'istruzione per disegnare la linea quest'altra istruzione

`line-width` <numero>

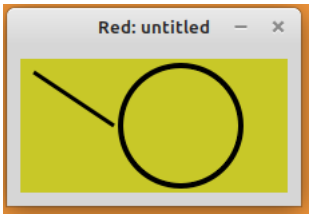
dove <numero> è lo spessore in pixel della linea.

Per default le linee hanno lo spessore di un pixel.

Per esempio, con

```
view [base 200x100 200.200.40 draw [line-width 3 line 10x10 70x50 line-width 4 circle 120x50 45]]
```

otteniamo



3.3 Manipolazione delle figure

Le figure che costruiamo con le istruzioni viste nel paragrafo 3.1 sono tutte orientate in modo parallelo agli assi della lavagna in cui sono disegnate e, nel caso di box, con tutti gli angoli retti. Ciò non si nota per cerchi e archi ma per le figure squadrate e per l'orientamento dell'ellisse si.

Red ci offre due possibilità per modificare questo stato di cose: la prima ci consente di disegnare figure non parallele agli assi ma variamente inclinate rispetto ad essi (`rotate`), la seconda ci consente di disegnare figure con lati inclinati rispetto ad essi (`skew`).

3.3.1 rotate

La sintassi è

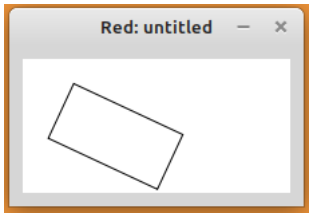
`rotate` <angolo> <centro>

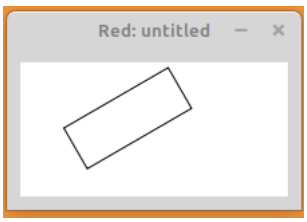
dove <angolo> è l'angolo di inclinazione della figura rispetto all'orientamento orizzontale, girando in senso orario e <centro> è il punto su cui si incentra la rotazione.

Con

```
view [base 200x100 white draw [rotate 25 20x10 box 40x10 130x55]]
view [base 200x100 white draw [rotate 25 20x10 box 40x10 130x55]]
```

otteniamo, rispettivamente, le figure di sinistra e di destra.





3.3.2 skew

E' un comando con il quale possiamo inclinare gli assi in modo che la lavagna su cui disegniamo non sia più ortogonale, ottenendo così che le figure che vi inseriamo abbiano i lati parimenti inclinati.

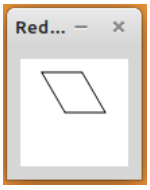
La sintassi del comando è

`skew` <angolo_lungo_asse_x> <angolo_lungo_asse_y>

Con

```
view [base white draw [skew 30 0 box 10x10 40x40]]
```

otteniamo



3.4 Colore

Per disegnare a colori dobbiamo procurarci una penna adatta e Red ce ne fornisce due: una per tracciare linee e una per colorare l'interno di figure.

3.4.1 pen

E' la penna per tracciare linee e si predispone con

`pen` <colore>

dove <colore> è il nome riconosciuto per un colore o una tupla RGB.

La predisposizione deve avvenire prima di tracciare.

Con

```
view [base 200x100 white draw [line-width 3 pen red circle 100x50 50 30]]
```

otteniamo

Per disattivare la penna si scrive `pen off`.

3.4.2 fill-pen

E' la penna per riempire una figura e si predispone con

fill-pen <colore>

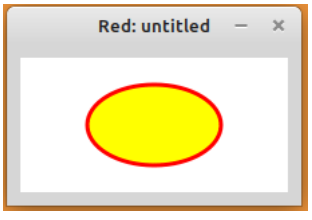
dove, al solito, <colore> è il nome riconosciuto per un colore o una tupla RGB.

La predisposizione deve avvenire prima di tracciare.

Con

```
view [base 200x100 white draw [line-width 3 pen red fill-pen yellow circle 100x50 50 30]]
```

otteniamo



Per disattivare la penna si scrive `fill-pen off`.

3.4.3 Gradiente

Un gradiente di colore è una successione progressiva di tonalità cromatiche.

Red ci dà modo di arricchire le penne per tracciare linee o per riempire figure della capacità di farlo con gradienti di colori scelti da noi.

In più, con la possibilità di comporre il gradiente in forma lineare (da sinistra a destra) o in forma radiale (irradiando da un centro).

La penna viene predisposta aggiungendo alla parola `pen` o `fill-pen` la parola `linear` o la parola `radial` seguite dall'elenco dei colori da coinvolgere (indicati con nome o tupla RGB).

In particolare le sintassi sono le seguenti:

. **linear**

`pen linear` <colore> <colore> . . . <inizio> <fine>

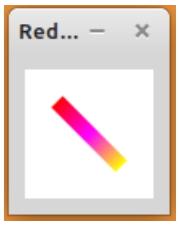
`fill-pen linear` <colore> <colore> . . . <inizio> <fine>

Se <inizio> e <fine> non sono indicati il gradiente viene applicato a tutta la linea o figura in orizzontale da sinistra a destra.

Con

```
view [base white draw [pen linear red magenta yellow line-width 10 line 20x20 60x60]]
```

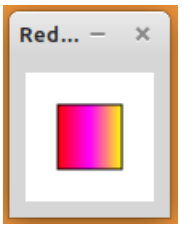
otteniamo

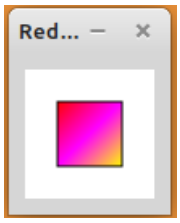


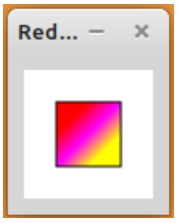
Con

```
view [base white draw [fill-pen linear red magenta yellow box 20x20 60x60]]
view [base white draw [fill-pen linear red magenta yellow 20x20 60x60 box 20x20 60x60]]
view [base white draw [fill-pen linear red magenta yellow 30x30 50x50 box 20x20 60x60]]
```

otteniamo, rispettivamente, le figure di sinistra, centro e destra.







. **radial**

`pen linear` <colore> <colore> . . . <centro> <raggio>

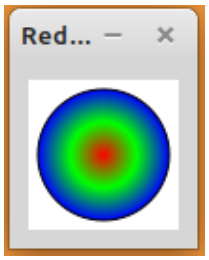
`fill-pen linear` <colore> <colore> . . . <centro> <raggio>

dove <centro> e <raggio>, rispettivamente da indicare con un pair! e con un numero, indicano la zona in cui operare il gradiente.

Con

```
view [base white draw [fill-pen radial red green blue 40x40 35 circle 40x40 35]]
view [base white draw [fill-pen radial red green blue 30x30 40 box 10x10 70x70]]
```

otteniamo, rispettivamente, le figure di sinistra e di destra.



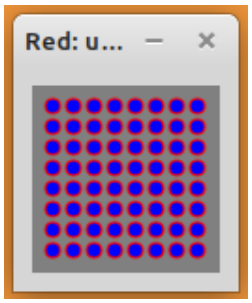
3.5 Disegno programmato e animazione

Ricorrendo al linguaggio Red, particolarmente ai controlli di flusso e al calcolo, possiamo scrivere programmi per produrre disegni particolarmente elaborati o animazioni.

Con questo programma

```
Red [needs:  view]
disegna:  [pen red fill-pen blue]
view/no-wait [foglio:  base 100x100]
repeat x 8 [
    repeat y 8 [
        posizione:(x * 11x0) + (y * 0x11)
        append disegna reduce ['circle posizione 4]
    ]
]
foglio/draw:  disegna
print "Fatto"
print "Scrivi quit per chiudere"
```

otteniamo



Il programma inizia con la predisposizione di un blocco per il disegno, per intanto scegliendo le penne, rossa per le linee, blu per i riempimenti.
Poi abbiamo un blocco `view` all'interno del quale, dopo aver creato un foglio per il disegno con le dimensioni giuste per contenere ciò che vogliamo disegnare, completiamo il blocco per il disegno predisponendo una griglia di posizioni corrispondenti al centro dei cerchi che vogliamo inserire nella griglia e, con la funzione `append`, inseriamo nel blocco per il disegno, valorizzandole con `reduce`, le istruzioni per il disegno dei cerchi: la parola `circle`, dovendo passare tale e quale senza valorizzazione, è preceduta da apice singolo ed è seguita dalla posizione indicante il centro dei cerchi da disegnare e dal numero indicante il raggio dei cerchi stessi.
Nell'impostare il blocco `view` abbiamo aggiunto il metodo `no-wait` per fare in modo che si avvii il ciclo per la predisposizione della griglia: senza questa aggiunta il programma si fermerebbe nel blocco `view`.
Finalmente assegniamo alla funzione `draw` del foglio per il disegno l'esecuzione del blocco per il disegno.
Dal momento che ciò avviene fuori dal blocco `view`, il disegno coinvolge la console e le istruzioni di stampa che seguono servono per tenere aperta la console stessa fino a chiusura volontaria: in mancanza di ciò la console si chiuderebbe subito dopo aver prodotto il disegno e non avremmo nemmeno il tempo per vederlo.

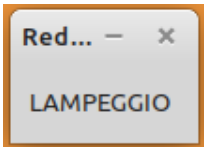
E' noto che le animazioni si ottengono attraverso la successione rapida di figure che si modificano.

Per ottenere la successione di azioni legate al tempo Red prevede un facet, che si chiama `rate`, contenente un timer che pulsa a velocità regolabile e ad ogni tick genera un evento, chiamato `on-time`.
La velocità della pulsazione si ottiene aggiungendo alla parola rate un numero intero che indica il numero di tick al secondo.
Pertanto `rate 2` genera impulsi molto più lenti che non `rate 20`.

Per esempio, per generale una scritta lampeggiante possiamo utilizzare questo programmino

```
Red [needs:  view]
view [ t:  text "" rate 5
on-time [either t/text = "" [t/text:  "LAMPEGGIO"] [t/text:  ""]]
]
```

con cui otteniamo

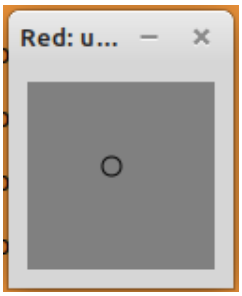


dove la scritta LAMPEGGIO scompare e ricompare 5 volte al secondo.

E' questa la base delle animazioni.

Con il seguente programmino generiamo un piccolo cerchio che attraversa la finestrella dall'angolo in alto a sinistra all'angolo in basso a destra.

```
Red [needs view]
posizione:  0x0
aggiorna-lavagna:  func [] [
    posizione:  posizione + 1x1
    lavagna/draw:  reduce ['circle posizione 5]
]
view [
    lavagna:  base 100x100 rate 25
    on-time [aggiorna-lavagna]
]
```



Cominciamo con inizializzare la posizione, che corrisponde al centro del cerchio che vogliamo far muovere, nell'angolo in alto a sinistra (coordinate rappresentate dal pair! 0x0).
Poi creiamo una funzione che, ad ogni chiamata, disegna il cerchio in una posizione lievemente discosta dalla precedente (vedi posizione che si incrementa del pair! 1x1 ad ogni chiamata).
Infine prevediamo una lavagna, che generalmente, nei programmi di grafica, si chiama canvas (tela), con un rate (nel caso abbastanza veloce) e richiamiamo la funzione per aggiornare la lavagna ad ogni tick con l'evento on-time.