Cicli, test e calcoli angolari per disegni non banali con METAPOST

Gianluca Pignalberi

Sommario

In rete è presente una buona quantità di documenti introduttivi a METAPOST; anche con le distribuzioni TEX viene fornita un'adeguata selezione di documenti. Purtroppo, non sempre gli autori riescono a esaurire tutti gli argomenti trattati: alcuni dettagli si confondono o si perdono, o sono volutamente lasciati ad altri documenti simili. In questo articolo vedremo come sono stati realizzati alcuni disegni non banali per una tesina su Galileo Galilei, approfondendo alcune delle spiegazioni dei manuali.

Abstract

A fair quantity of introductory guides to METAPOST is available online; a good selection of them comes along with the TEX distributions. Unfortunately, sometimes the authors don't succeed in treating the topics fully: some details gets hidden, lost or left to other similar documents. In this paper we'll see how some non-trivial drawings for a short thesis on Galileo Galilei were done, having the chance to study in detail some of the manuals' explanations.

1 Introduzione

METAPOST è un prodotto (compilatore più linguaggio di programmazione) orientato alla realizzazione di disegni geometrici e derivato da METAFONT. La differenza sostanziale (Hobby, 2009) sta nel fatto che, mentre METAFONT produce file in grafica *bitmap*, METAPOST produce file PostScript (un suo sottoinsieme), quindi vettoriali e scalabili a piacere senza perdita di qualità. In rete e con le distribuzioni TEX si trovano buone guide introduttive (Heck, 2005; Hobby, 1997; Hurlin, 2007) che a volte sono, a mio giudizio, troppo ermetiche o comunque non esaustive. Non si può prescindere dalla lettura di Goossens *et al.* (2007).

Il funzionamento di METAFONT, e quindi anche di METAPOST, è lo stesso di TEX: in un file sorgente si descrive l'aspetto del documento finale tramite un linguaggio di programmazione, quindi si compila il sorgente per ottenere il documento finale. Sebbene il tempo di stesura della prima versione del disegno sia molto maggiore rispetto alla stessa operazione fatta con un editor WYSIWYG, le modifiche successive sono in genere molto più veloci e accurate.

Recentemente ho dovuto realizzare alcuni disegni per una tesina su Galileo Galilei. La tesina, incentrata sulle osservazioni della luce, aveva bisogno di disegni inerenti la riflessione. Per quanto semplici, questi disegni dovevano essere precisi; era dunque necessaria una conoscenza non banale di METAPOST. Non sono stato in grado di desumere tutte le informazioni a me necessarie da uno solo dei manuali citati. La mia esperienza d'uso di METAPOST precedente a questo progetto era molto limitata e, soprattutto, breve. Klaus Höppner (Höppner, 2008) ha pubblicato un articolo introduttivo che mi è servito per iniziare, ma questa volta avevo bisogno di più.

Non ho dubbi che i codici che sono riuscito a scrivere siano ampiamente migliorabili; sono comunque serviti allo scopo, cioè aiutarmi a capire alcuni argomenti di METAPOST che mi erano rimasti poco chiari. Sono altresí consapevole che essi mi hanno fornito un aiuto immenso con la programmazione parametrica: avere fatto dei disegni in funzione di un parametro dimensionale mi ha permesso di ricompilarli al volo variandone le dimensioni ma lasciando inalterato il font in uso.

2 I tipi di METAPOST

METAPOST è insieme linguaggio di programmazione e compilatore. Il linguaggio è tipizzato, e i suoi tipi sono necessariamente correlati al compito da svolgere. Abbiamo dunque il tipo `numeric`, adatto a rappresentare i numeri interi e decimali (ma dotati di unità di misura); il tipo `pair`, che serve a indicare delle coordinate, cioè una coppia di valori di tipo `numeric`; il tipo `path`, necessario a memorizzare entità geometriche più complesse, che vedremo più avanti.

Ogni operazione viene espressa per mezzo di equazioni o di comandi (definiti *espressioni*). Naturalmente, a causa della tipizzazione, bisogna controllare che le equazioni scritte rispondano a criteri di correttezza (non solo sintattica come per i comandi), pena la non compilazione e la contestuale emissione di errori.

Ad esempio, se `z0` è una variabile di tipo `pair`, `z0 = (1,2);` è un'equazione lecita mentre `z0 = 1;` non lo è. Nella sezione 3 parleremo di equazioni. Non dimentichiamo che, ogni volta che ci riferiamo a una variabile o a un valore di tipo `numeric`, lo intendiamo dotato di unità di misura: se questa non è espressa sarà implicitamente

pt, altrimenti sarà esplicitamente una di quelle in genere riconosciute da TEX.

METAPOST permette di dichiarare variabili singole e vettori:

```
pair var; %variabile singola
pair vvar[]; %vettore
```

Per riferirsi a una specifica cella del vettore basta posporre il numero di posizione (a partire da 0) al nome della variabile, *senza* parentesi quadre: `vvar0` si riferisce al primo elemento del vettore `vvar`. L'indice espresso tramite variabile va invece posto tra parentesi quadre, come vedremo negli esempi riportati nella sezione 5.1.

METAPOST fornisce "gratis" i vettori `x[]` e `y[]` di tipo `numeric`, e `z[]` di tipo `pair`, il cui contenuto è correlato a `x` e `y`: `z0=(x0,y0);`. Tutte le altre variabili vanno dichiarate per poter essere usate. In realtà potremmo anche non dichiarare niente, ma ogni variabile istanziata senza l'opportuna dichiarazione sarà considerata `numeric`, dando luogo a eventuali errori.

Supponiamo di avere la seguente dichiarazione:

```
pair moon[];
```

è semplice desumere le singole coordinate da assegnare a variabili `numeric`:

```
numeric xm, ym; % si può omettere
xm = xpart(moon0);
ym = ypart(moon0);
```

Un percorso (`path`) contiene un oggetto ancora più complesso, ossia la rappresentazione di una *forma* (aperta o chiusa), dal segmento in poi. Sono diverse le forme che un percorso può memorizzare, in base agli operatori che useremo. Le prossime sezioni dell'articolo mostreranno diversi esempi di percorsi, con o senza direzioni obbligate.

Nel prosieguo dell'articolo vedremo esempi d'uso (implicito) di `boolean`, `color` e `pen`. Per gli altri tipi (`transform`, `(rgb)color`, `cmykcolor`, `string`) rimando alle opere citate in bibliografia.

3 Le equazioni

Un'espressione della forma

$$\langle \text{membro sinistro} \rangle = \langle \text{membro destro} \rangle$$

è un'equazione lineare, del tutto identica a quelle imparate a scuola: il ⟨membro sinistro⟩ deve essere uguale al ⟨membro destro⟩. Un'assegnazione è invece caratterizzata dall'operatore `:=`; un'espressione del tipo

$$\langle \text{lvalue} \rangle := \langle \text{rvalue} \rangle$$

indica che a ⟨lvalue⟩, una variabile riferita per indirizzo, viene assegnato il valore espresso da ⟨rvalue⟩.

METAPOST risolve da solo i sistemi di equazioni lineari, a patto che questi siano coerenti. Se all'interno dello stesso codice, non importa se nella stessa figura o in due figure diverse, provassimo a scrivere un codice del genere:

```
z0 = (1,1);
z0 = (2,2);
```

otterremmo un messaggio di "Inconsistent equation (off by 1)" e l'arresto della compilazione: è chiaro che un sistema di equazioni del genere non ha senso. Potremmo pensare di sostituire l'uguaglianza con l'assegnazione, ma non si può fare un'assegnazione a variabili di tipo `pair`. Non abbiamo invece problemi a fare assegnazioni a variabili di tipo `numeric` e `path`.

L'unica occasione in cui le equazioni precedenti sono lecite è quando si riferiscono a diverse istanze di variabili locali. Una variabile è locale solo se è stata esplicitamente dichiarata all'interno di un blocco `beginfig(x);...endfig;`. Questo è l'unico modo per non causare conflitti nei sistemi di equazioni.

4 Le procedure

Può succedere che diversi disegni abbiano bisogno di elementi comuni. Se METAPOST fornisce delle forme base (circonferenza, quadrato), per altre dovremo fare da noi. Per la tesina suddetta ho avuto bisogno di un omino stilizzato (avrei preferito un Dalek (Griffiths, 2008), ma forse non sarebbe stato il caso; ho propeso per "l'impiccato", il cui uso è legittimato dalla convenzione dei diagrammi UML (Vetti Tagliati, 2003) e dal fatto, altamente culturale, che potrebbe essere una rappresentazione rupestre dell'Uomo Vitruviano). Siccome l'omino (un punto di osservazione dei raggi luminosi riflessi) si trovava in più disegni (in due di essi in più esemplari), sarebbe stato lungo, noioso e a rischio di errori disegnarlo ex novo ogni volta. Ci viene in soccorso il meccanismo delle *procedure* (non uso il termine *funzioni* dato che non ci sono valori da restituire), o macro(istruzioni).

Una procedura in METAPOST è caratterizzata dalle parole chiave `def` (iniziale) ed `enddef` (finale). A `def` si fa seguire il nome della macro, e poi il suo corpo. Eventualmente, tra nome e corpo possiamo specificare alcuni parametri da passare alla macro. Analizziamo la macro per disegnare l'omino (Wire Frame Man):

```
1 def wfm (expr bx, by, h) =
2   body := h/3.3;
3   legs := h/2;
4   face := h/7.1;
5   neck := h/17;
6   draw fullcircle scaled face
      shifted (bx,by);
7   draw (bx,by-face/2)--(bx,by-
      face/2)-(neck+body)*dir(90)
      ;
```

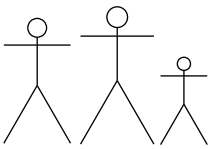

FIGURA 1: Omini stilizzati resi da una macro METAPOST

```
8     draw (bx,by-face/2-neck)--(bx,
          by-face/2-neck)-.5legs*dir
          (0);
9     draw (bx,by-face/2-neck)--(bx,
          by-face/2-neck)-.5legs*dir
          (180);
10    draw (bx,by-face/2-neck-body)
          --(bx,by-face/2-neck-body)-
          legs*dir(60);
11    draw (bx,by-face/2-neck-body)
          --(bx,by-face/2-neck-body)-
          legs*dir(120);
12  enddef;
```

I parametri passati sono preceduti dal termine `expr`, non propriamente un tipo. Nel codice dovremo noi fare in modo che non ci siano conflitti dei tipi.

Nell'esempio in esame, la procedura `wfm` riceve tre parametri: posizione del centro della testa dell'omino (`bx` e `by`) e la sua altezza (`h`). Questi parametri saranno trattati nella macro come dati `numeric`. Nelle righe 2–5 calcoliamo con proporzione anatomicamente verosimili l'altezza del tronco e la lunghezza di gambe, braccia e collo. Si passa infine a disegnare la testa (riga 6), il collo e il tronco (riga 7), le gambe (righe 8 e 9) e le braccia (righe 10 e 11).

La funzione suesposta, definita in un file METAPOST e richiamata con la seguente invocazione:

```
wfm(0,0,50);
wfm(30,4,55);
wfm(55,-13.5,35);
```

produce il risultato mostrato nella figura 1.

5 Attenti a quei due

Un linguaggio di programmazione senza *test* e *cicli* non sarebbe molto espressivo; in virtù del suo essere un linguaggio di programmazione, METAPOST dispone dei costrutti citati. Nei disegni realizzati per la tesina su Galilei sono stati grandissimi alleati, senza i quali la realizzazione dei disegni sarebbe stata più lunga e imprecisa.

5.1 Cicli

Per realizzare i cicli METAPOST usa l'istruzione `for`. La sua sintassi è la seguente:

```
for <parametro> = <espressione>
  upto <espressione>:
  <corpo del ciclo> endfor;
```

Un ciclo inverso usa `downto`. Sia `downto` che `upto` sono macro. La macro `upto` è identicamente uguale a `step 1 until`, mentre `step -1 until` è la definizione naturale di `downto`. Dunque il ciclo in METAPOST avrebbe la forma

```
for <parametro> = <espressione>
  step 1 until <espressione>:
  <corpo del ciclo> endfor;
```

Possiamo variare il passo (l'incremento o il decremento) del ciclo ridefinendo le macro, per esempio:

```
def upto = step 2 until enddef;
```

L'argomento della tesina riguardava gli studi di Galileo sulla luce, in particolare quella diretta dal sole agli oggetti (orientata). Per tutti gli oggetti illuminati serviva avere una serie di raggi incidenti, tutti paralleli. La realizzazione di tali raggi è stata semplicissima:

```
for i = 0 upto 10:
  ir[i] = (-5u,0)+(0,i*u);
  fr[i] = (5u,0)+(0,i*u);
  ray[i] = ir[i]--fr[i];
  draw ray[i]
endfor;
```

dati `ir[]` e `fr[]` i vettori dei punti iniziali e finali dei raggi e `ray[]` il percorso dei singoli raggi, si ottiene una serie di segmenti paralleli disegnando ogni segmento all'interno del ciclo. Nel caso in esame sono raggi orizzontali, visto che l'unica coordinata che varia col ciclo è quella relativa all'asse y. Alcuni numeri sono moltiplicati per u; u è una costante arbitraria; nei casi in esame è posta a .5 cm.

Siccome è più elegante avere delle frecce indicanti la direzione dei raggi, la costruzione sarà di poco più lunga:

```
1  for i = 0 upto 10:
2    ir[i] = (-5u,0)+(0,i*u);
3    fr[i] = (5u,0)+(0,i*u);
4    drawarrow ir[i]--(ir[i]+(2u,0)
         );
5    draw (ir[i]+(2u,0))--fr[i];
6  endfor;
```

Nelle righe 2 e 3 calcoliamo i punti iniziale e finale del raggio i-esimo; nella riga 4 disegniamo un segmento orientato di lunghezza $2u$; infine, grazie alla riga 5, disegniamo il tratto restante del raggio. Vediamo il risultato del precedente codice nella figura 2.

Allo stesso modo dei raggi solari, usare un ciclo mi ha aiutato a disegnare una sfera stilizzata:

```
1  for i = 120 downto 1:
2    fill fullcircle scaled (i*u
         /10) shifted ((-6.4+i/20)*u
         ,0) withcolor 70/i*white;
3  endfor;
```

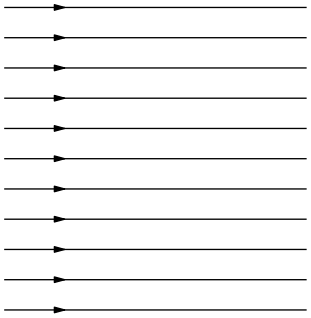

FIGURA 2: Raggi diretti

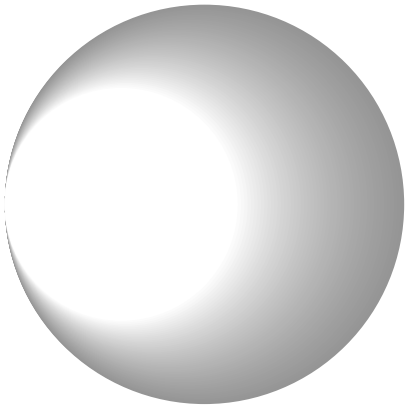

FIGURA 3: Sfera stilizzata

il cui risultato è riportato nella figura 3.

Un ultimo esempio di uso dei cicli è quello usato per disegnare il sole nello schema descrivente il fenomeno della *luce cinerea* (la luce riflessa della Terra che illumina la parte di Luna non direttamente illuminata dal Sole). Evitare al sole la forma "da bambino" (con i raggi che potrebbero sembrare dei vettori uscenti) ha comportato l'uso di triangoli. La loro posizione sul bordo si trova facilmente usando un ciclo e poche semplici formule; il contatore del ciclo viene usato come valore angolare.

La figura 4 mostra lo schema della luce cinerea, del cui sorgente è qui riportata solo la parte relativa alla macro dei raggi e al loro posizionamento intorno al sole.

```
def tri (expr b, h, ang, tx, ty) =
  pair v[];
  v0 = (-.5b, 0);
  v1 = (.5b, 0);
  v2 = (0, h);
  fill v0--v1--v2--cycle
    rotated ang shifted (tx,ty)
    withrgbcolor (255,255,0);
enddef;
⋮
for i = 0 step 30 until 330:
  tri(3,6,i,-3.1u*sind(i),3.1u*cosd(i));
endfor;
```

Il valore del contatore `i` (che non dimentichiamo essere di tipo `numeric`) viene passato sia alle

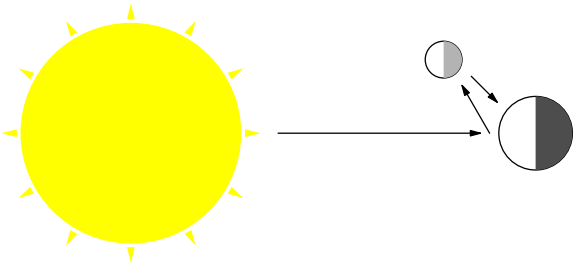

FIGURA 4: Luce cinerea

funzioni `sind` e `cosd` (per il calcolo del posizionamento del raggio) che alla macro `tri` (per la rotazione del raggio stesso).

5.2 Test

Nei linguaggi di programmazione siamo in grado di differenziare le esecuzioni di uno stesso codice grazie alla possibilità di eseguire dei test, e quindi intraprendere l'azione adeguata. Come tutti i linguaggi di programmazione, anche METAPOST mette a disposizione un costrutto per i test. La sintassi del comando `if` è la seguente:

```
if <espressione1>:
  <corpo dell'if>
else:
  <corpo dell'else>
fi
```

Se `<espressione1>` è vera (cioè la sua valutazione restituisce un valore *true*) viene eseguito il `<corpo del if>`, altrimenti viene eseguito il `<corpo dell'else>`. Il ramo "else" non è obbligatorio, dunque può essere omesso. Il `<corpo dell'else>` può iniziare con un altro test; METAPOST fornisce in proposito l'istruzione più compatta `elseif`, usabile nell'ambito del test. Nell'ambito della tesina il test mi è servito per determinare l'incrocio di due percorsi (un raggio luminoso e un oggetto) e determinare quindi la lunghezza effettiva del raggio. Stabilire quale espressione usare nel test ha richiesto più di qualche minuto di lettura dei manuali e di prove.

Da HÖPPNER (2008) sappiamo che possiamo usare `whatever` per determinare un punto sconosciuto su una variabile di tipo `path`. Ad esempio, il codice seguente:

```
z0 = origin;
z1 = (12,7.458);
p0 = z0--z1;
z2 = (3,20);
x3 = 6;
z3 = whatever[z0,z1];
p1 = z2--z3;
draw p0;
drawarrow p1;
```

produce il disegno riportato nella figura 5. Saremmo tutti soddisfatti perché `z3`, e quindi il segmento

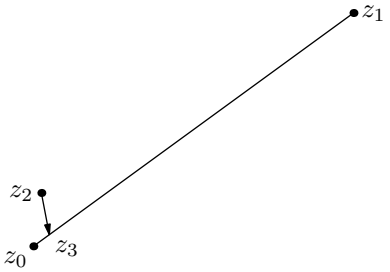

FIGURA 5: Il punto indicato dalla freccia è calcolato da METAPOST

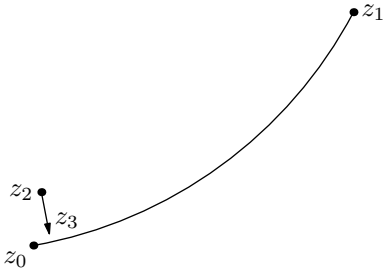

FIGURA 6: Il punto indicato dalla freccia è calcolato da METAPOST, ma non è quello di incontro tra il segmento e la curva

`z2--z3`, viene determinato dal programma in base al percorso di `z0--z1` una volta che abbiamo fissato `x3`: dunque il test lavorerebbe adeguatamente per noi.

Ma supponiamo ora che il percorso sia curvo, come nel listato seguente:

```
z0 = origin;
z1 = (120,87.458);
z2 = (130,115);
p0 = z0{dir 10}..z1;
z2 = (3,20);
x3 = 6;
z3 = whatever[z0,z2];
p1 = z2--z3;
draw p0;
drawarrow p1;
```

continuerà la freccia a indicare il punto di incrocio tra `p0` e `p1`?

La figura 6 mostra che non lo fa, quindi il punto calcolato da METAPOST è, *apparentemente*, errato.

Una più attenta analisi ci rivela che il punto calcolato è quello posizionato sul segmento che unisce i punti `z0` e `z1` della curva, quindi METAPOST non ha sbagliato: siamo piuttosto stati noi a usare scorrettamente i suoi comandi. La figura 7 svela graficamente l'arcano.

Visto che non è stato possibile usare `whatever` per determinare dove un segmento e una curva si intersecano, si è reso opportuno scoprire l'eventuale esistenza di comandi in grado di determinare tale punto. METAPOST mette a disposizione dei programmatori due operatori: `intersectionpoint` e

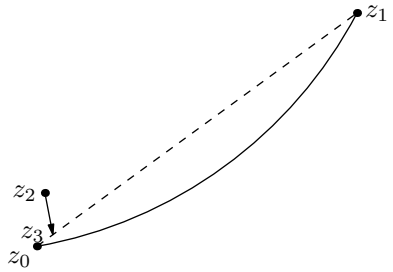

FIGURA 7: Il punto indicato dalla freccia, calcolato da METAPOST, è in realtà quello tra il segmento `z2--z3` e il segmento unente i punti `z0` e `z1` della curva

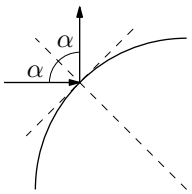

FIGURA 8: Schema di riflessione della luce incidente su uno specchio sferico

`intersectiontimes`, da applicare a due variabili di tipo `path`. Il primo restituisce un valore di tipo `pair` indicante un punto di intersezione, se esiste; il secondo restituisce il tempo di intersezione (intuitivamente la distanza dall'inizio di ogni percorso) per ognuno dei due percorsi, $(-1,-1)$ altrimenti. Data una serie di raggi (segmenti) diretti verso un oggetto, non si può usare l'operatore `intersectionpoint` per stabilire se i raggi toccano l'oggetto, visto che se non c'è contatto la compilazione si interrompe segnalandoci l'inesistenza di un punto di contatto: andrà usato `intersectiontimes`.

Questo test è servito per limitare la lunghezza dei raggi solo se questi incontrano il pianeta, come vedremo tra poco. Andiamo avanti con le esigenze. Un altro grafico che ha sfruttato questo test è stato quello dello specchio sferico. Qui, per ogni raggio incidente, si doveva disegnare il corrispondente raggio riflesso, che ha angolo uguale e contrario a quello compreso tra la direzione di incidenza e la normale alla tangente della circonferenza nel punto di incidenza (figura 8). Il risultato è riportato nel listato seguente, il cui disegno finale è visibile nella figura 9.

```
1  u := .5cm;
2  numeric ang[];
3  pair e[], ipe[], in[], f[], fv
       [], osservatore;
4  path earth[], ray[], vec[];
5  e1 = (-6.4u,0);
6  e2 = (0,6.4u);
7  e3 = (6.4u,0);
8  e4 = (0,-6.4u);
9  earth0 = e1..e2..e3..e4..cycle;
10 earth1 = e4..e1..e2;
```

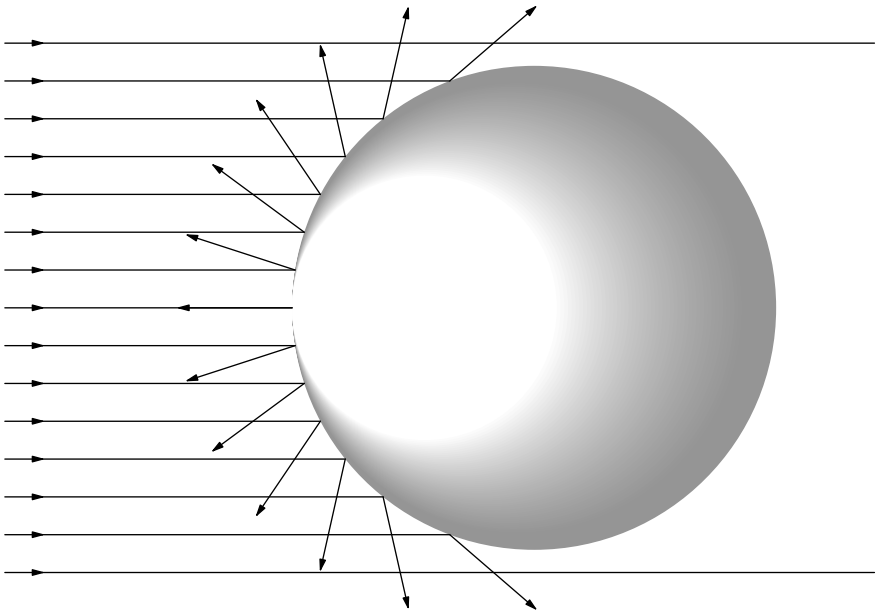

FIGURA 9: La riflessione dei raggi luminosi incidenti su uno specchio sferico così come vista da METAPOST

```
11 fill earth0 withcolor 7/12*white
     ;
12 for i = 120 downto 1:
13   fill fullcircle scaled (i*u
       /10) shifted ((-6.4+i/20)*u
       ,0) withcolor 70/i*white;
14 endfor;
15 in0 = (-14u,-8u);
16 f0 = (9u,-8u);
17 for i=1 upto 15:
18   in[i] = in[i-1] + (0,1u);
19   f[i] = f[i-1] + (0,1u);
20   ray[i] = in[i]--f[i];
21   if ((ray[i] intersectiontimes
       earth1) <> (-1,-1)):
22     ipe[i] = ray[i]
         intersectionpoint earth1;
23     ang[i] = 180+2*(angle(ipe[i
         ])-180);
24     drawarrow in[i]--(in[i]+(1u
         ,0));
25     draw (in[i]+(1u,0))--ipe[i];
26     vec[i] = ipe[i] + 3u*dir(ang
         [i]);
27     drawarrow ipe[i]--vec[i];
28   else:
29     drawarrow in[i]--(in[i]+(1u
         ,0));
30     draw (in[i]+(1u,0))--f[i];
31   fi;
32 endfor;
```

L'intelligenza di questo codice sta nelle righe 21–26. Nella riga 21 si controlla che il raggio intersechi lo specchio sferico (chiamato qui "terra" perché si intendeva mostrare come Galileo confutò la credenza che i pianeti — e la Luna in particolare — riflettessero la luce come uno specchio sferico),

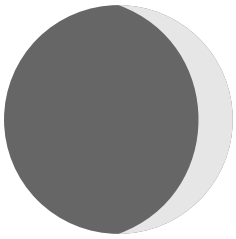

FIGURA 10: Il disegno della Luna crescente ha delle cuspidi: sarebbe difficile da ottenere se non potessimo determinare gli angoli di arrivo e di partenza delle linee da ogni punto

quindi si calcola il punto di intersezione (riga 22). L'equazione della riga 23 calcola l'angolo di riflessione. Dato un punto x, la funzione `angle` calcola l'angolo del segmento (0,0)–x rispetto all'orizzonte (riga orizzontale passante per (0,0) diretta a destra). In questo caso il calcolo è diretto perché il pianeta è centrato in (0,0); in caso non lo fosse stato, avremmo risolto con una traslazione. Il valore restituito (un angolo in gradi sessagesimali) può essere passato direttamente alla funzione `dir` (riga 26). Dopo aver disegnato il raggio incidente (righe 24 e 25) si determina il punto finale del raggio riflesso da disegnare, in base alla sua direzione (riga 26).

6 Direzione

Abbiamo visto esempi di come poter dare una direzione alle linee disegnate nei paragrafi 4 e 5.2. In questi esempi la direzione era sempre posposta al punto da cui partiva una linea. Se questa fosse l'unica sintassi possibile sarebbe ben difficile ottenere dei disegni contenenti linee curve non raccordate; se non diversamente istruito, METAPOST cerca

di calcolare il percorso curvo migliore possibile e, soprattutto, senza cuspidi.

Proprio per permettere un disegno simile a quello mostrato, per ogni punto di un percorso possiamo definire la direzione di arrivo e di ripartenza della curva; la convenzione prevede che 0° è la direzione orizzontale verso destra, e gli angoli crescono procedendo in senso antiorario. Nel caso delle fasi lunari, ottenere il disegno della Luna crescente è semplicissimo:

```
fill fullcircle scaled u
  withcolor .4*white;
fill (0,.5u){dir -20}..
  {dir -160}(0,-.5u){dir 0}..
  (.5u,0)..(0,.5u)..cycle
  withcolor .9*white;
```

il cui risultato è visibile nella figura 10.

Ripetiamo un concetto: quando definiamo un percorso, diamo alla variabile di tipo `path` un elenco di punti per cui la curva (o la spezzata) deve necessariamente passare. Nel caso di una curva, se non diamo vincoli, METAPOST raccorda i punti al meglio, senza variazioni discontinue della derivata prima della curva. Questi vincoli sono proprio gli angoli di arrivo e partenza della curva da un punto; questi ci permettono di forzare il comportamento di METAPOST per piegarlo alle nostre necessità, tra cui ci può essere dover disegnare una curva con cuspidi, cosa che realizziamo dando due angoli opportuni; si veda la figura 11, il cui codice è:

```
a = (-u, u);
b = origin;
c = (-u, -u);
curv0 = a..{dir 270}b{dir 270}..c;
  % curv0 equivale alla curva a..b..c
curv1 = a..{dir 0}b{dir 180}..c;
draw curv0;
draw curv1 dashed evenly;
```

7 Conclusioni

Rispetto a un prodotto visuale, METAPOST obbliga l'utente a studiare e a programmare. Il tempo che sembra perso (mentre un utente "visuale" disegna) viene recuperato prontamente. Poiché è in grado di determinare i risultati dei sistemi di equazioni, METAPOST ci permette di disegnare senza conoscere i valori di molti dei punti in gioco: a noi basterà scrivere le equazioni, essendo sicuri che il risultato sarà ineccepibile, senza la perdita di tempo del disegnatore "visuale". Possiamo poi cambiare le dimensioni del disegno senza dover ridisegnare niente (solo cambiando un parametro dimensionale) e senza che la dimensione del font ne sia affetta, al solo costo di una ricompilazione. In questo modo i disegni saranno sempre ben integrati negli spazi disponibili e la coerenza col testo sarà sempre assicurata.

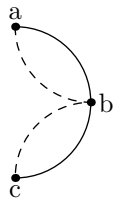

Figura 11: Entrambe le curve uniscono gli stessi punti, ma con percorsi differenti: la curva continua è quella naturale di METAPOST, quella tratteggiata è ottenuta dando angoli arbitrari, e presenta una cuspide

8 Ringraziamenti

Non posso non ringraziare i revisori e i correttori del mio articolo. Solo la loro attenzione e competenza mi hanno permesso di scrivere un articolo migliore, più completo e senz'altro più comprensibile.

Riferimenti bibliografici

Goossens, M., Mittelbach, F., Rahtz, S. *et al.* (2007). *The LaTeX Graphics Companion*. Addison Wesley, 2ª edizione.

Griffiths, N. (2008). *Dalek I Loved You*. Gollancz S.F.

Heck, A. (2005). *Learning METAPOST by Doing*. URL `http://staff.science.uva.nl/~heck/Courses/mptut.pdf`.

Hobby, J. D. (1997). *The MetaPost System*. URL `http://tex.loria.fr/prod-graph/mpintro.pdf`.

— (2009). *METAPOST A USER'S MANUAL*. URL `http://www.tug.org/docs/metapost/mpman.pdf`.

Höppner, K. (2008). «A short introduction to METAPOST». *ArsTEXnica*, (6). URL `http://www.guit.sssup.it/arstexnica/download_ars/arstexnica06.pdf`.

Hurlin, C. (2007). *Practical introduction to METAPOST*. URL `http://www-sop.inria.fr/everest/Clement.Hurlin/misc/Practical-introduction-to-MetaPost.pdf`.

Vetti Tagliati, L. (2003). *UML*. Tecniche Nuove.

▷ Gianluca Pignalberi
Free Software Magazine
`g dot pignalberi at freesoftwaremagazine dot com`