

# GRAPHIC COMPOSITION IN PROCESSING\*

Davide Rocchesso

This work is produced by The Connexions Project and licensed under the  
Creative Commons Attribution License †

## Abstract

Elements of 2D and 3D graphic composition (including curves) for the Processing language and environment (in Italian)

## 1 Primitive grafiche

In Processing si possono disporre punti, linee, superfici, e volumi (cioè oggetti geometrici a 0, 1, 2, o 3 dimensioni) in uno spazio tridimensionale ovvero, laddove la cosa abbia un senso, nello spazio bidimensionale della finestra immagine. Il numero di parametri delle primitive corrispondenti determinerà se tali oggetti debbano essere collocati nello spazio X-Y o nello spazio X-Y-Z.

### 1.1 0D

#### Punti

La `point()` è la più semplice delle primitive grafiche. Se invocata con due coordinate, colloca un punto nello spazio X-Y. Se invocata con tre coordinate, colloca il punto nello spazio X-Y-Z. Un punto, in senso geometrico, non ha dimensione, ma ad esso può essere associata una occupazione in pixel ed un colore, mediante le funzioni `strokeWeight()` e `stroke()`, rispettivamente. Ad esempio, il semplicissimo sketch processing

```
stroke(180,0,0);  
strokeWeight(10);  
point(60,60);
```

disegna un quadratino nella finestra immagine.

#### Collezioni di punti

Un insieme di punti può essere raggruppato in un unico oggetto (nuvola di punti) mediante i due delimitatori `beginShape(POINTS)` e `endShape()`, tra i quali ogni punto va specificato con la funzione `vertex()`. Trasformazioni di rotazione, traslazione, o scalamento non si applicano internamente agli oggetti composti descritti da `beginShape()` e `endShape()`, ma possono precedere la definizione di un composito e applicarsi all'insieme.

---

\*Version 1.7: Jun 16, 2005 1:20 pm GMT-5

†<http://creativecommons.org/licenses/by/2.0/>

## 1.2 1D

### Rette

La `line()` traccia un segmento di retta tra due punti del piano o dello spazio, con spessore e colore eventualmente impostati tramite `strokeWeight()` e `stroke()`, rispettivamente.

### Collezioni di Segmenti

Un insieme di segmenti può essere definito, analogamente a quanto visto per i punti, mediante i delimitatori `beginShape()` e `endShape()`, tra i quali si elencano gli estremi dei segmenti mediante la funzione `vertex()`. Usando la invocazione `beginShape(LINES)` i vertici vengono presi a coppie, ciascuna identificante un segmento, mentre con la invocazione `beginShape(LINE_STRIP)` i vertici, presi uno dopo l'altro, definiscono una spezzata. Con l'invocazione `beginShape(LINE_LOOP)` la spezzata viene chiusa su se stessa mediante collegamento del primo e dell'ultimo vertice. Notare che il poligono così ottenuto non può essere manipolato, ad esempio mediante coloratura.

### Curve

La funzione `curve()`, invocata con otto parametri, traccia una curva sul piano immagine, con punto iniziale e finale determinati, rispettivamente, dalle seconda e dalla terza coppia di coordinate passate come argomenti. Le due coppie di coordinate iniziale e finale definiscono due punti di controllo della curva tracciata, la quale è una spline interpolante, e quindi passa per tutti e quattro i punti.

#### Definition 1: Spline

Curva polinomiale a tratti, con polinomi connessi con continuità ai nodi (**knot**)

NOTE: Si veda Introduction to Splines<sup>1</sup> e, per una introduzione al tipo specifico di spline (Catmull-Rom) usato in processing, la voce **spline** in Wikipedia.

Per avere un numero arbitrario di punti di controllo bisogna usare la funzione `curveVertex()` per specificare ciascun punto all'interno di un blocco delimitato da `beginShape(LINE_STRIP)` e `endShape()`.

A differenza della `curve()`, nella funzione `bezier()` i due punti di controllo specificati dai quattro parametri intermedi non sono punti attraversati dalla curva. Essi servono solo a definire la forma della **curva approssimante di Bezier**, la quale ha le seguenti proprietà notevoli:

- è interamente contenuta nell'involuppo convesso definito dai punti estremali e dai punti di controllo;
- trasformazioni di traslazione, rotazione, o scalamento applicate ai punti estremali e di controllo determinano una identica trasformazione della curva.

Come si vede eseguendo il codice

```
stroke(255, 0, 0);
line(93, 40, 10, 10);
line(90, 90, 15, 80);
stroke(0, 0, 0);
bezier(93, 40, 10, 10, 90, 90, 15, 80);
```

i punti di controllo stanno sulla tangente che passa per gli estremi della curva. Per avere un numero arbitrario di punti di controllo bisogna usare la funzione `bezierVertex()` per specificare ciascun punto all'interno di un blocco delimitato da `beginShape(LINE_STRIP)` e `endShape()`. In questo modo si può tracciare una curva arbitrariamente involuta in uno spazio a tre dimensioni. In due dimensioni, la `bezierVertex()` ha sei parametri che corrispondono alle coordinate di due punti di controllo e di un punto di ancoraggio. La prima invocazione di `bezierVertex()` va preceduta da una invocazione di `vertex()` che stabilisce il primo punto di ancoraggio della curva.

Ci sono ulteriori metodi che consentono di leggere le coordinate o la direzione della tangente in un punto qualsiasi lungo la curva di Bezier o spline interpolante, stabilito mediante un parametro  $t$  che può andare

<sup>1</sup>"Introduction to Splines" <<http://cnx.org/content/m11153/latest/>>

da 0 (primo estremo) a 1 (secondo estremo). E' anche possibile impostare la precisione di tracciamento di curve approssimanti e interpolanti in 3D. Per i dettagli si rimanda al manuale di riferimento<sup>2</sup> di Processing.

Lo sketch processing di tabella (Table 1) illustra la differenza tra curva interpolante spline e curva di Bezier

NOTE: Si veda la voce **Bezier curve** in Wikipedia.

.

---

<sup>2</sup>[http://www.processing.org/reference/index\\_ext.html](http://www.processing.org/reference/index_ext.html)

applet che  
confronta  
curva di  
Bezier  
(rossa) e  
spline in-  
terpolante  
(nera) <sup>3</sup>

```
void setup() {
  c1x = 120;
  c1y = 110;
  c2x = 50;
  c2y = 70;
  background(200);
  stroke(0,0,0);
  size(200, 200);
}

int D1, D2;
int X, Y;
int c1x, c1y, c2x, c2y;

void draw() {
  if (mousePressed == true) {
    X = mouseX; Y = mouseY;
    // selezione del punto che si modifica
    D1 = (X - c1x)*(X - c1x) + (Y - c1y)*(Y - c1y);
    D2 = (X - c2x)*(X - c2x) + (Y - c2y)*(Y - c2y);
    if (D1 < D2) {
      c1x = X; c1y = Y;
    }
    else {
      c2x = X; c2y = Y;
    }
  }
  background(200);
  stroke(0,0,0);
  strokeWeight(1);

  beginShape(LINE_STRIP);
  curveVertex(10, 10);
  curveVertex(10, 10);
  curveVertex(c2x, c2y);
  curveVertex(c1x, c1y);
  curveVertex(190, 190);
  curveVertex(190, 190);
  endShape();

  stroke(255,30,0);
  bezier(10,10,c2x,c2y,c1x,c1y,190,190);
  strokeWeight(4);
  point(c1x,c1y);
  point(c2x,c2y);
}
```

Table 1

### 1.3 2D

NOTE: Le figure in due o tre dimensioni assumono un colore che può essere determinato dalla illuminazione, come spiegato nella Section 3 (Illuminazione), oppure stabilito mediante il metodo `fill()`, il quale prevede anche la possibilità di impostare la trasparenza.

#### Triangoli

Il triangolo è l'elemento costruttivo fondamentale per la grafica 3D, in quanto mediante giustapposizione di triangoli si approssimano superfici continue. In Processing, però, i triangoli sono elementi specificati nel 2D mediante la primitiva `triangle()`, la quale ha sei parametri corrispondenti alle coordinate dei tre vertici nella finestra immagine. Pur essendo definito nel 2D, ogni triangolo può essere soggetto a trasformazioni di rotazione e traslazione all'interno dello spazio 3D, come avviene nello sketch Processing

```
void setup(){
  size(200, 200, P3D);
  fill(210, 20, 20);
}

float angle = 0;

void draw(){
  background(200); // clear image
  stroke(0,0,0);
  angle += 0.005;
  rotateX(angle);
  triangle(10,10,30,70,80,80);
}
```

#### Collezioni di Triangoli

Un insieme di triangoli può essere definito, analogamente a quanto visto per i punti e segmenti, mediante i delimitatori `beginShape()` e `endShape()`, tra i quali si elencano i vertici dei triangoli mediante la funzione `vertex()`. Usando la invocazione `beginShape(TRIANGLES)` i vertici vengono presi a terne, ciascuna identificante un triangolo, mentre con la invocazione `beginShape(TRIANGLE_STRIP)` i vertici, presi uno dopo l'altro, definiscono una striscia di superficie a facce triangolari. Se le `vertex()` hanno tre argomenti, i vertici vengono collocati nello spazio 3D e i corrispondenti triangoli individuano superfici planari nello spazio.

#### Quadrilateri

I rettangoli si definiscono, in Processing, mediante la funzione `rect()` di quattro parametri, in cui i primi due specificano, per default, la posizione nel piano 2D dell'angolo in alto a sinistra, e il terzo e quarto specificano, rispettivamente, la larghezza e la altezza. Il significato dei primi due parametri si può cambiare con la funzione `rectMode()`: `rectMode(CORNER)` corrisponde al posizionamento di default; `rectMode(CENTER)` corrisponde al posizionamento del centro del rettangolo nel punto specificato dalle prime due coordinate; `rectMode(CORNERS)` fa in modo che i quattro parametri vengano interpretati come le coordinate del vertice in alto a sinistra e del vertice in basso a destra. Un quadrilatero generico si definisce mediante le coordinate dei suoi quattro vertici, passate come parametri alla funzione `quad()`. E' importante notare che in 3D, mentre un triangolo rimane in ogni caso planare, una quadrupla di punti può dare luogo ad una superficie curva. Viceversa, i quadrilateri definiti mediante roto-traslazioni 3D di quadruple di vertici 2D rimangono

<sup>3</sup>[http://cnx.org/content/m12665/1.7/bezier\\_curve.html](http://cnx.org/content/m12665/1.7/bezier_curve.html)

planari. Processing permette di passare solo otto parametri alla `quad()`, in tal modo forzando la definizione del quadrilatero mediante una quadrupla di vertici 2D.

### Collezioni di Quadrilateri

Un insieme di quadrilateri può essere definito, analogamente a quanto visto per i triangoli, mediante i delimitatori `beginShape()` e `endShape()`, tra i quali si elencano i vertici dei quadrilateri mediante la funzione `vertex()`. Usando la invocazione `beginShape(QUADS)` i vertici vengono presi a quadruple, ciascuna identificante un quadrilatero, mentre con la invocazione `beginShape(QUAD_STRIP)` i vertici, presi uno dopo l'altro, definiscono una striscia di superficie a facce quadrilatere. Se i `vertex()` usati hanno tre parametri, non è garantita la planarità delle facce risultanti, e quindi la resa grafica può essere fuorviante. Ad esempio, eseguendo il codice

```
size(200,200,P3D);
lights();
beginShape(QUADS);
vertex(20,31, 33);
vertex(80, 40, 38);
vertex(75, 88, 50);
vertex(49, 85, 74);
endShape();
```

ci si rende conto che il quadrilatero viene reso come giustapposizione di due triangoli giacenti su piani diversi.

### Poligoni

Un poligono generico è definito da una successione di vertici, ed è un oggetto dotato di una superficie che può essere colorata. In Processing tali vertici si elencano dentro a una coppia `beginShape(POLYGON); - endShape()`; In realtà il poligono è da intendersi in senso generalizzato, essendo possibile usare le `bezierVertex()` e `curveVertex()` per specificare profili curvi. Ad esempio, si provi a disegnare la luna:

```
fill(246, 168, 20);
beginShape(POLYGON);
vertex(30, 20);
bezierVertex(80, 10, 80, 75, 30, 75);
bezierVertex(50, 70, 60, 25, 30, 20);
endShape();
```

### Ellissi

La funzione `ellipse()` traccia una ellisse nel piano 2D. I quattro parametri sono interpretati, come nel caso della `rect()`, come posizione seguita da larghezza e altezza. La posizione può essere regolata mediante la chiamata `ellipseMode()`, il cui parametro può assumere valori `CORNER`, `CORNERS`, `CENTER`, `CENTER_RADIUS`. I primi due di questi quattro possibili valori vanno riferiti al rettangolo circoscritto alla ellisse.

## 1.4 3D

Processing offre un repertorio assai limitato di primitive di oggetti tridimensionali, essenzialmente solo sfere e parallelepipedi.

### Scatole

La funzione `box()` produce un cubo se invocata con un solo parametro (lato), un parallelepipedo se invocata con tre parametri (larghezza, altezza, profondità).

## Palle

La funzione `sphere()` produce, mediante un poliedro approssimante, una sfera il cui raggio è specificato come parametro. La funzione `sphereDetail()` può essere usata per specificare il numero di vertici del poliedro che approssima la sfera ideale.

## 2 Lo stack delle trasformazioni

Una rotazione o una traslazione possono essere pensate come operazioni che ruotano o traslano il sistema di riferimento cartesiano. In altri termini, dopo una `rotate()` o una `translate()` le successive operazioni di posizionamento di oggetti grafici avranno un nuovo sistema di assi coordinati. Quando si posizionano diversi oggetti in vario modo nello spazio, è utile tenere traccia dei diversi sistemi di assi coordinati che via via si vanno ad impostare. La struttura dati atta a contenere tali sistemi è il cosiddetto stack (o pila) delle trasformazioni (**matrix stack**). Con la funzione `pushMatrix()` il sistema di coordinate corrente viene impilato in testa allo stack. Invece, per recuperare il sistema di coordinate precedente alla ultima trasformazione operata, bisogna operare una `popMatrix()`. Di fatto, lo stack contiene le matrici di trasformazione affine, secondo quanto prescritto da **OpenGL** e descritto in Section 5 (Pillole di OpenGL).

### Example 1

In questo esempio vengono collocati due oggetti, uno quadrato planare ed un cubo, nello spazio 3D. Il primo `push()` salva sullo stack il sistema di coordinate, quindi vengono applicate alcune trasformazioni che precedono il disegno del quadrato. Per tornare al sistema di coordinate precedente, e quindi operare nuove trasformazioni per collocare il cubo, si applica una `popMatrix()`. Di fatto, le `pushMatrix()` e `pop()` delimitano lo scope relativo al posizionamento geometrico di un oggetto.

```
float angle;

void setup(){
    size(100, 100, P3D);
    int angle = 0;
}

void draw(){
    background(200);
    angle += 0.003;
    pushMatrix();
    translate(25,50);
    rotateZ(angle);
    rotateY(angle);
    rectMode(CENTER);
    rect(0,0,20,20);
    popMatrix();
    translate(75,50,-25);
    rotateX(angle);
    box(20);
}
```

### 3 Illuminazione

Il modello di illuminazione di Processing ricalca direttamente quello di **OpenGL**, cioè lo **shading di Phong**. Tale modello non è giustificato fisicamente, ma è particolarmente efficiente. OpenGL considera illuminato ogni poligono la cui normale forma un angolo acuto con la direzione di provenienza della luce. Questo a prescindere dalla presenza di oggetti mascheranti, e quindi le ombre non vengono rese. Si dice che OpenGL adotta un modello locale di illuminazione, in quanto le riflessioni multiple tra le superfici e le proiezioni di ombre non vengono automaticamente elaborate.

E' disponibile una sorgente di luce di ambiente, cioè non proveniente da alcuna direzione particolare, il cui colore viene specificato mediante i parametri della chiamata di attivazione `ambientLight()`. Una sorgente di luce direzionale viene impostata con `directionalLight()`, i cui parametri specificano colore e direzione di provenienza. Il metodo `lights()` attiva una combinazione di default di luce ambientale grigia e luce direzionale, anch'essa grigia ma più intensa, proveniente dalla direzione frontale. E' possibile impostare una sorgente luminosa puntiforme in una data posizione dello spazio, mediante la chiamata `pointLight()`. Infine, il metodo `spotLight()` attiva un fascio di luce di cui, oltre a colore, posizione e direzione di irraggiamento, è possibile controllare l'apertura dell'angolo e la concentrazione intorno all'asse, mediante l'esponente  $e$  che regola la rapidità di decadimento all'allontanarsi dall'asse:

$$\cos^e(\phi) \quad (1)$$

Considerata una superficie piana, una luce direzionale che incida su di essa produce luce riflessa secondo varie direzioni, in maniera dipendente dalle proprietà superficiali. Nel caso di riflessione perfettamente diffusiva (o **lambertiana**), l'irraggiamento è prodotto in tutte le direzioni allo stesso modo, con una intensità tanto maggiore quanto più la direzione di incidenza è vicina alla direzione ortogonale (o normale) alla superficie. Viceversa, se la superficie è perfettamente riflettente, la luce viene riflessa solo lungo la direzione simmetrica, rispetto alla normale, alla direzione di incidenza. In OpenGL, per avere la massima flessibilità nella definizione della illuminazione, ogni sorgente racchiude in sé le tre componenti di illuminazione ambientale, lambertiana, e speculare. Queste tre componenti sono definibili separatamente, e interagiscono con le rispettive tre componenti che definiscono le proprietà superficiali degli oggetti. I colori definiti nei parametri dei metodi `directionalLight()`, `pointLight`, e `spotLight()` definiscono la componente lambertiana di illuminazione. La `lightSpecular()` specifica il colore della componente di luce incidente che viene trattata con riflessione speculare.

In Processing è possibile controllare le proprietà delle superfici mediante i metodi `ambient()` (che agisce sulla componente ambientale di luce) e `specular()` (che agisce sulla componente speculare). Quest'ultimo metodo prevede anche la possibilità di specificare, mediante un parametro `alpha`, se la superficie debba essere di tipo traslucido. La `shininess()` controlla la concentrazione del fascio riflesso specularmente, mediante un coefficiente che agisce in modo analogo all'esponente della (1). Gli oggetti rappresentati possono anche essere considerati sorgenti di luce, e quindi si può assegnare loro una luce di emissione, mediante la `emissive()`. Tuttavia, le sorgenti definite in questo modo non producono illuminazione sugli altri oggetti della scena.

In OpenGL, le luci puntiformi, a spot, e ambientali subiscono una attenuazione dipendente dalla distanza, secondo il modello

$$\text{attenuation} = \frac{1}{a + bd + cd^2} \quad (2)$$

Il metodo `lighFalloff()` consente di specificare i parametri  $a$ ,  $b$ , and  $c$

#### Example 2

Qui un cubo e una `QUAD_STRIP` vengono collocati nello spazio ed illuminati da una sorgente rotante. Inoltre, viene impostata una tenue luce fissa. Si noti l'assenza di ombre e la apparente planarità delle superfici della `QUAD_STRIP`.

```
import processing.opengl.*;
```

```
float r;
float lightX, lightY, lightZ;

void setup() {
  size(400, 400, OPENGL);
  r = 0;
  ambient(180, 90, 0);
  specular(0, 0, 240, 0.99);
  lightSpecular(200, 200, 200);
  shininess(5);
}

void draw() {
  lightX = 100*sin(r/3) + width/2;
  lightY = 100*cos(r/3) + height/2;
  lightZ = 100*cos(r);
  background(0,0,0);
  noStroke();
  ambientLight(153, 102, 0);

  lightSpecular(0, 100, 200);
  pointLight(100, 180, 180,
    lightX, lightY, lightZ);
  pushMatrix();
  translate(lightX, lightY, lightZ);
  emissive(100, 180, 180);
  sphere(4); //Put a little sphere where the light is
  emissive(0,0,0);
  popMatrix();
  pushMatrix();
  translate(width/2, height/2, 0);
  rotateX(PI/4);
  rotateY(PI/4);
  box(100);
  popMatrix();
  pushMatrix();
  translate(width/4, height/2, 0);
  beginShape(QUAD_STRIP);
  vertex(10,13,8);
  vertex(13,90,13);
  vertex(65,76,44);
  vertex(95,106,44);
  vertex(97,20,70);
  vertex(109,70,80);
  endShape();
  popMatrix();
  r+=0.05;
}
```

## 4 Proiezioni

### 4.1 Proiezione prospettica

Una proiezione prospettica è caratterizzata da un **centro di proiezione** e da un **piano di proiezione**. I **raggi proiettori** collegano i punti della scena con il centro di proiezione, individuando i corrispondenti punti proiettati sul piano di proiezione. La Figure 1 mostra una vista in sezione in cui il piano di proiezione è individuato da una retta di ascissa  $-d$ , e il centro di proiezione è collocato nell'origine.

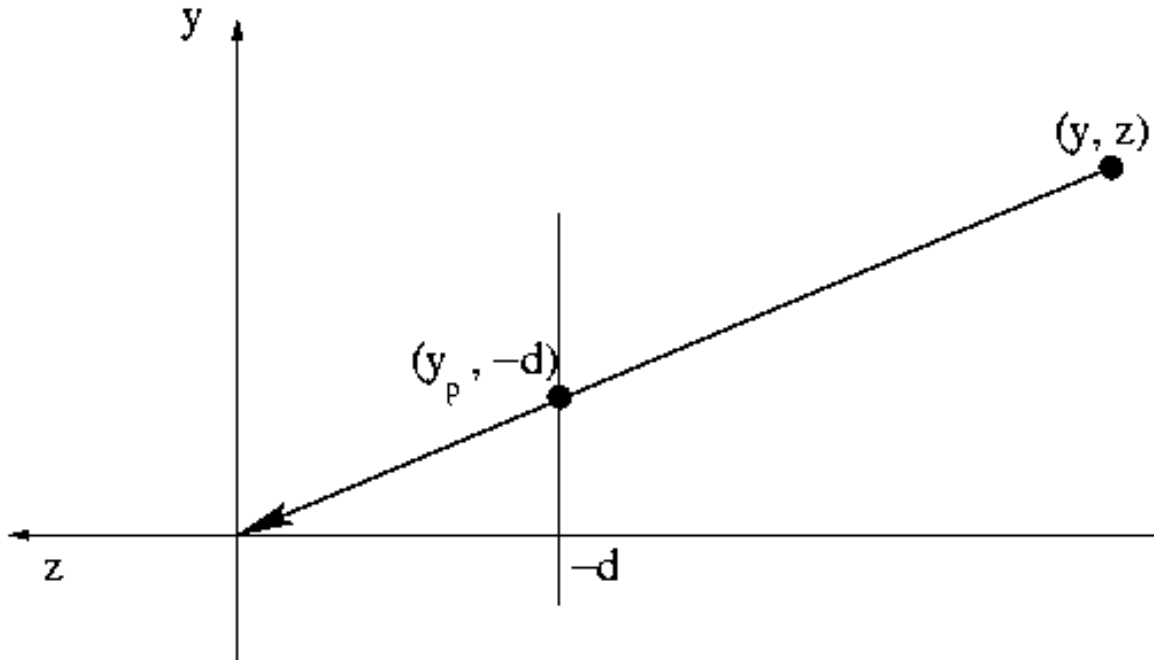


Figure 1

Mediante similitudine dei due triangoli rettangoli, è facile rendersi conto che il punto di ordinata  $y$  viene proiettato sul piano nel punto di ordinata  $y_p = -\left(\frac{yd}{z}\right)$ .

Più in generale, la proiezione di un punto di coordinate omogenee  $\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$  su un piano perpendicolare all'asse  $z$  e secante tale asse in posizione  $-d$  si effettua, in coordinate omogenee, per moltiplicazione con la matrice  $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\left(\frac{1}{d}\right) & 0 \end{pmatrix}$ . Il punto proiettato diventa  $\begin{pmatrix} x \\ y \\ z \\ -\left(\frac{z}{d}\right) \end{pmatrix}$ , il quale può essere normalizzato

moltiplicando tutti i suoi elementi per  $-\left(\frac{d}{z}\right)$ . Si ottiene quindi

$$\begin{pmatrix} -\left(\frac{xd}{z}\right) \\ -\left(\frac{yd}{z}\right) \\ -d \\ 1 \end{pmatrix}$$

## 4.2 Viste parallele

Le viste parallele sono quelle che si ottengono facendo retrocedere a infinito ( $\infty$ ) il centro di proiezione. In questo modo, i raggi proiettori sono tutti paralleli.

### 4.2.1 Proiezione ortografica

La proiezione ortografica produce una classe di viste parallele mediante conduzione di proiettori perpendicolari al piano di proiezione. Se il piano di proiezione viene posto perpendicolare all'asse  $z$  e passante per

l'origine, la matrice di proiezione risulta particolarmente semplice:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Tra le proiezioni or-

tografiche, le **assonometrie** si basano sulla possibilità di descrivere l'oggetto da proiettare secondo tre **assi** ortogonali, e sull'orientazione del piano di proiezione rispetto a questi assi. In particolare la assonometria isometrica<sup>4</sup> è tale che le proiezioni degli assi formano tra loro angoli di  $120^\circ$ . Essa ha la proprietà che lunghezze uguali lungo i tre assi rimangono tra loro uguali lungo le proiezioni degli stessi. Per ottenere una assonometria isometrica di un oggetto i cui assi principali sono inizialmente paralleli agli assi coordinati, si può procedere mediante una prima rotazione di  $45^\circ$  intorno all'asse  $y$ , seguita da una seconda rotazione di  $\arctan\left(\frac{1}{\sqrt{2}}\right) = 35.264^\circ$  intorno all'asse  $x$

### 4.2.2 Proiezione obliqua

Si parla di proiezione obliqua ogniqualvolta il piano di proiezione è obliquo (non perpendicolare) rispetto ai raggi proiettori. Per introdurre una deviazione dei raggi proiettori dalla perpendicolare secondo angoli  $\theta$  e  $\phi$

si deve usare una matrice di proiezione

$$\begin{pmatrix} 1 & 0 & -(\tan(\theta)) & 0 \\ 0 & 1 & -(\tan(\phi)) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

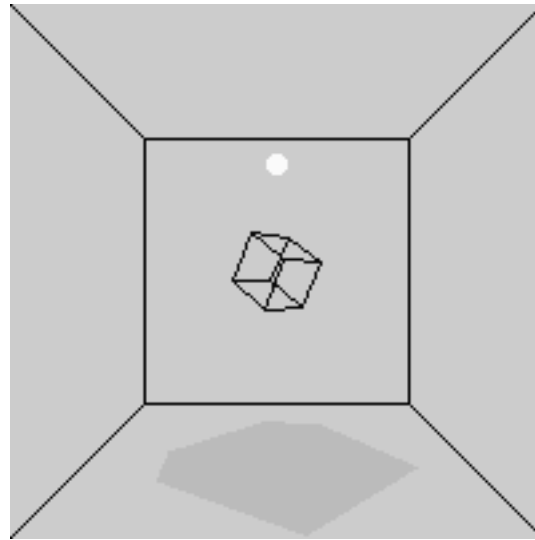
## 4.3 Proiezione di ombre

Come abbiamo visto, Processing offre un modello di illuminazione locale, e quindi non è possibile proiettare ombre (**shadow casting**) per via diretta. Tuttavia, la manipolazione delle matrici di trasformazione affine ci consente di proiettare abbastanza agevolmente ombre su piani. La tecnica che si usa è detta **flashing in the eye**, con questo termine intendendo che si sposta il centro ottico della scena nel punto in cui è collocata la sorgente luminosa, e si procede con una trasformazione prospettica con piano di proiezione coincidente con quello su cui si vuole proiettare l'ombra.

<sup>4</sup>[http://en.wikipedia.org/wiki/Isometric\\_projection](http://en.wikipedia.org/wiki/Isometric_projection)

**Example 3**

Il codice seguente proietta sul pavimento l'ombra generata da una sorgente di luce collocata sull'asse  $y$ . Il risultato è mostrato in Figure 2 (Proiezione di un'ombra)

**Proiezione di un'ombra****Figure 2**

```

size(200, 200, P3D);
float centro = 100;
float yp = 70; //distanza pavimento dal centro
float yl = 40; //altezza luce dal centro
translate(centro, centro, 0); //centra tutto sul cubo
noFill();
box(yp*2); //disegna la stanza
pushMatrix();
  fill(250); noStroke();
  translate(0, -yl, 0); // sposta in alto (rispetto al centro)
                        // la lampadina
  sphere(4); //disegna la lampadina;
  stroke(10);
popMatrix();
pushMatrix(); //disegna il cubo wireframe
  noFill();
  rotateY(PI/4); rotateX(PI/3);
  box(20);
popMatrix();
translate(0, -yl, 0); // riporta sorgente di luce e
                    // pavimento al loro posto
applyMatrix(1, 0, 0, 0,

```

```

    0, 1, 0, 0,
    0, 0, 1, 0,
    0, 1/(yp+yl), 0, 0); // proietta sul pavimento
                          // spostato in basso di yl
translate(0, yl, 0); // porta sorgente di luce al centro
                    // e pavimento in basso di yl
pushMatrix();
  fill(120, 50);
  noStroke();
  rotateY(PI/4); rotateX(PI/3);
  box(20);
popMatrix();

```

## 5 Pillole di OpenGL

**OpenGL** è un insieme di funzioni che consentono al programmatore di accedere al sistema grafico. In gergo tecnico è una **Application Programming Interface (API)**. Principalmente essa si occupa della resa grafica (**rendering**) di una scena popolata di oggetti tridimensionali e luci, da un certo punto di vista. Per quanto riguarda il programmatore, OpenGL consente di descrivere oggetti geometrici ed alcune loro proprietà, nonché di stabilire come tali oggetti debbano essere illuminati e visti. Dal punto di vista realizzativo, OpenGL è basato su una **pipeline** grafica composta di moduli, come riportato in Figure 3 (La pipeline di OpenGL). Un ottimo libro di grafica interattiva in OpenGL è quello di *Angel*[1].

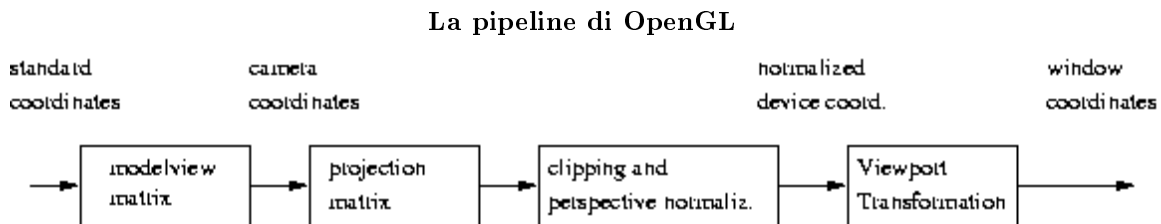


Figure 3

In Processing (ed in OpenGL), l'utente specifica gli oggetti mediante coordinate mondo (**standard coordinates**). La **model-view matrix** è la matrice di trasformazione usata per passare da coordinate mondo a uno spazio associato alla camera da presa. Ciò consente di cambiare dinamicamente il punto di vista e l'orientazione della camera. In OpenGL ciò avviene mediante la funzione `gluLookAt()`, la quale è riprodotta in Processing dalla `camera()`. I primi tre parametri identificano la posizione, in coordinate mondo, del centro ottico della camera (**eye point**). La seconda terna di parametri identifica un punto al quale è rivolta la camera (**center of the scene**). La terza terna di coordinate identifica un vettore atto a specificare la verticale di ripresa. Ad esempio, il codice

```

void setup() {
  size(100, 100, P3D);
  noFill();

```

```

    framerate(20);
}

void draw() {
  background(204);
  camera(70.0, 35.0, 120.0, 50.0, 50.0, 0.0,
    (float)mouseX /width, (float)mouseY /height, 0.0);
  translate(50, 50, 0);
  rotateX(-PI/6);
  rotateY(PI/3);
  box(45);
}

```

traccia il **wireframe** di un cubo e permette la rotazione della camera da presa.

La **projection matrix** si occupa di fare la proiezione sulla finestra di vista, proiezione che può essere parallela (o ortografica) o prospettica. La proiezione ortografica si può attivare con la chiamata `ortho()`. La proiezione prospettica è quella di default, ma si può esplicitamente attivare con `perspective()`. Proiezioni particolari, quali quelle oblique, possono essere ottenute per distorsione degli oggetti mediante applicazione della `applyMatrix()`. C'è inoltre la **texture matrix**, di cui ci si occupa in un altro modulo.

Per ogni tipo di matrice, OpenGL mantiene uno stack di matrici, la matrice corrente essendo quella in cima. Il meccanismo dello stack consente di salvare uno stato (mediante `pushMatrix()`) prima di operare nuove trasformazioni, o di rimuovere lo stato corrente evidenziando uno stato precedente (mediante `popMatrix()`). Ciò si riflette nelle operazioni Processing descritte in Section 2 (Lo stack delle trasformazioni). Le trasformazioni in OpenGL sono applicate secondo lo schema:

1. Push sullo stack;
2. Applica tutte le trasformazioni desiderate mediante moltiplicazione con la matrice in cima allo stack;
3. Disegna l'oggetto (che risulterà affetto dalle trasformazioni);
4. Pop dallo stack.

Un **viewport** è un'area rettangolare della finestra di display. Il passaggio dal piano di proiezione prospettica al viewport avviene in due passi: (i) trasformazione in una finestra 2 x 2 centrata nell'origine ( **normalized device coordinates** ) (ii) mappatura della finestra normalizzata sul viewport. Usando la rappresentazione intermedia a coordinate normalizzate, l'operazione di **clipping**, cioè di eliminazione degli oggetti o loro parti non visibili attraverso la finestra, diventa banale. In Processing, le funzioni `screenX()`, `screenY()`, e `screenZ()` consentono di ottenere le coordinate X-Y prodotte dalla trasformazione di viewport e dagli elementi precedenti della catena di Figure 3 (La pipeline di OpenGL).

Il **frustum** di vista è l'angolo solido attraverso il quale si attua la proiezione prospettica, come indicato in Figure 4 (Il frustum di vista). Vengono visualizzati gli oggetti (o le loro parti) presenti nel volume di vista, le rimanenti parti essendo soggette a clipping. In Processing (e in OpenGL) il frustum si può definire mediante posizionamento dei sei piani che lo individuano (`frustum()`), oppure mediante specificazione di angolo verticale, **aspect ratio**, posizione dei piani anteriore e posteriore (`perspective()`). Ci si può chiedere come avvenga la rimozione delle facce nascoste, cioè di quelle facce che sono mascherate da altre facce presenti nel volume di vista. OpenGL usa l'algoritmo dello **z-buffer**, che è supportato dalle schede grafiche. La scheda tiene un'area di memoria bidimensionale (lo z-buffer) corrispondente ai pixel della finestra di vista e contenente valori di profondità. Prima di proiettare un poligono sulla finestra di vista la scheda va a controllare che i pixel affetti da tale poligono non abbiano già un valore di profondità inferiore rispetto a quello del poligono in oggetto. In questo caso significherebbe che c'è un oggetto grafico che maschera quello che si sta disegnando.

---

### Il frustum di vista

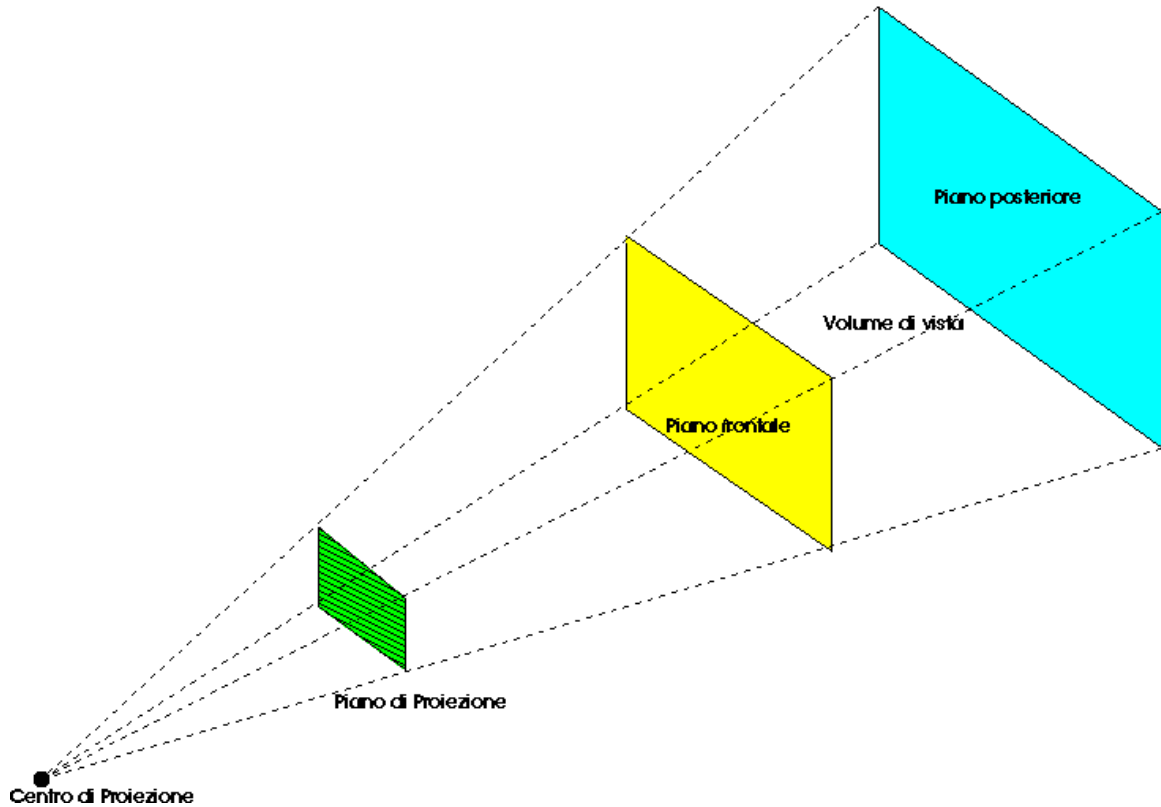


Figure 4

---

Elaborazioni geometriche sofisticate si possono ottenere manipolando direttamente le matrici di proiezione e model-view. Ciò è possibile, in Processing, a partire dalla matrice unitaria, caricata con `resetMatrix()`, mediante moltiplicazioni matriciali eseguite con `applyMatrix()`.

#### Exercise 1

*(Solution on p. 17.)*

Si esegua e si analizzi il codice Processing

```
size(200, 200, P3D);
println("Default matrix:"); printMatrix();
noFill();
ortho(-width/2, width/2, -height/2, height/2, -100, 100);
translate(100, 100, 0);
println("After translation:"); printMatrix();
rotateX(atan(1/sqrt(2)));
println("After about-X rotation:"); printMatrix();
rotateY(PI/4);
println("After about-Y rotation:"); printMatrix();
box(100);
```

Cosa visualizza e con che tipo di proiezione? Come si interpretano le matrici stampate nella console? Posso invertire l'ordine delle rotazioni?

**Exercise 2***(Solution on p. 17.)*

Si scriva un codice Processing che produca la proiezione obliqua di un cubo.

**Exercise 3***(Solution on p. 17.)*

Si visualizzi un cubo che proietti la sua ombra sul suolo, assumendo che la sorgente luminosa sia a distanza infinita (quale si può considerare, in pratica, la distanza del sole).

## Solutions to Exercises in this Module

### Solution to Exercise 1 (p. 15)

Il wireframe di un cubo viene visualizzato in assonometria isometrica. Le ultime tre matrici rappresentano, accumulandole una dopo l'altra, le tre operazioni di traslazione (per allineare il cubo al centro della finestra), rotazione intorno all'asse  $x$ , e rotazione intorno all'asse  $y$ . Una successione di due rotazioni corrisponde al prodotto di due matrici di rotazione, e il risultato non è indipendente dall'ordine delle matrici (il prodotto non è commutativo). Il prodotto di due matrici di rotazione  $R_x R_y$  corrisponde ad eseguire prima la rotazione intorno a  $y$ , e poi la rotazione intorno a  $x$ .

### Solution to Exercise 2 (p. 16)

Ad esempio:

```
size(200, 200, P3D);
float theta = PI/6;
float phi = PI/12;
noFill();
ortho(-width/2, width/2, -height/2, height/2, -100, 100);
translate(100, 100, 0);
applyMatrix(1, 0, -tan(theta), 0,
            0, 1, -tan(phi), 0,
            0, 0, 0, 0,
            0, 0, 0, 1);
box(100);
```

### Solution to Exercise 3 (p. 16)

Si opera in modo simile a quanto visto in Example 3, ma la trasformazione proiettiva è di tipo ortografico:

```
size(200, 200, P3D);
noFill();
translate(100, 100, 0);
pushMatrix();
  rotateY(PI/4); rotateX(PI/3);
  box(30);
popMatrix();
translate(30, 60, 0); //proietta un'ombra da infinito (sole)
applyMatrix(1, 0, 0, 0,
            0, 0, 0, 0,
            0, 0, 1, 0,
            0, 0, 0, 1);

fill(150);
pushMatrix();
noStroke();
rotateY(PI/4); rotateX(PI/3);
box(30);
popMatrix();
```

## References

- [1] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach With OpenGL primer package-2nd Edition*. Prentice-Hall, Inc., 2001.