Table of Contents

Fondamenti di Programmazione

Andrea Sterbini

lezione 15 - 24 novembre 2022

In [1]:
```
1 %load_ext nb_mypy
```

```
Version 1.0.4
```

RECAP: CLASSI e oggetti

- classi: attributi (di classe) e metodi
- istanze: attributi di istanza e metodi
- information hiding e "responsabilità" delle operazioni
- ereditarietà come meccanismo per estendere un tipo di oggetti

Colori

- operazioni matematiche sui colori (somma, prodotto, ...)
- costruttore
- rappresentazione (**`__repr__`** e **`__str__`**)
- conversione in tripla RGB

In [4]:

```python
import images
from random import randint

class Colore:
    white : 'Colore'
    black : 'Colore'
    red   : 'Colore'
    green : 'Colore'
    blue  : 'Colore'
    cyan  : 'Colore'
    purple: 'Colore'
    yellow: 'Colore'
    grey  : 'Colore'

    def __init__(self, R : float, G : float, B : float):↔

    # può far comodo avere un secondo costruttore che genera colori casuali
    @classmethod
    def random(cls, m : int =0, M : int = 255) -> 'Colore' :   # -> Colore casuale
        "torna un colore casuale con i valori delle luminosità in [m .. M]"
        return cls(randint(m,M), randint(m,M), randint(m,M))

    def luminosità(self) -> float:   # luminosità del colore
        "calcolo la luminosità media di un pixel (senza badare se è un valore intero)
        return (self._R + self._G + self._B)/3

    def __add__(self, other : 'Colore') -> 'Colore':
        "somma tra due colori"
        if not isinstance(other, Colore):
            raise ValueError("Il secondo addendo non è un Colore")
        return Colore(self._R + other._R, self._G + other._G, self._B + other._B)

    def __sub__(self, other : 'Colore') -> 'Colore':
        "somma tra due colori"
        if not isinstance(other, Colore):
            raise ValueError("Il secondo addendo non è un Colore")
        return Colore(self._R - other._R, self._G - other._G, self._B - other._B)

    def __mul__(self, k : float) -> 'Colore':
        "moltiplicazione di un colore per una costante"
        # FIXME: controllare che k sia un numero
        return Colore(self._R*k, self._G*k, self._B*k)

    def __truediv__(self, k : float) -> 'Colore':
        "divisione di un colore per una costante"
        # FIXME: controllare che k sia un numero != 0
```

```
        return Colore(self._R/k, self._G/k, self._B/k)

    def _asTriple(self) -> tuple[int, int, int]:
        "creo la tripla di interi tra 0 e 255 che serve per le immagini PNG"
        def bound(X):↔
        return bound(self._R), bound(self._G), bound(self._B)

    def __repr__(self) -> str :
        "stringa che deve essere visualizzata per stampare il colore"
        # uso una f-stringa equivalente a
        # return "Color(" + str(self._R) + ", " + str(self._G) + ", " + str(self._B)
        return f"Colore({self._R}, {self._G}, {self._B})"
```

In [6]:

```
# solo dopo aver definito Colore posso aggiungere attributi che contengono un Colore
Colore.white = Colore(255, 255, 255)
Colore.black = Colore(  0,   0,   0)
Colore.red   = Colore(255,   0,   0)
Colore.green = Colore(  0, 255,   0)
Colore.blue  = Colore(  0,   0, 255)
Colore.cyan  = Colore(  0, 255, 255)
Colore.purple= Colore(255,   0, 255)
Colore.yellow= Colore(255, 255,   0)
Colore.grey  = Colore.white/2

# Esempi
p1 = Colore(255,  0, 0)
p2 = Colore(  0,255, 0)
p3 = p2 + p1    # uso l'operatore somma tra due colori che ho definito sopra
p4 = p3 * 0.5   # uso l'operatore prodotto per una costante che ho definito sopra

p5 = Colore(120, 34, 200)
p4
```

Out[6]:

```
Colore(127.5, 127.5, 0.0)
```

Semplifichiamo le immagini

Per spostare le operazioni di disegno in classi separate ci servono solo le primitive di disegno minime:

- **set_pixel** e **get_pixel**
- **is_inside**
- **`__init__`** , **`__repr__`** , **save** e **visualizza**

In [7]:

```python
import images
import math


class Immagine:

    def __init__(self, larghezza=None, altezza=None, sfondo=None, filename=None):
        """posso creare una immagine in due modi:
            - leggendola da un file PNG se passo il parametro filename
            - fornendo dimensioni e colore di sfondo se non lo passo
        """
        if filename:
            img = images.load(filename)
            # letta la immagine la converto in una matrice di Colore
            self._img = [ [ Colore(r,g,b) for r,g,b in riga  ] for riga in img  ]
            self._W = len(img[0])
            self._H = len(img)
        else:
            # altrimenti creo una immagine monocolore
            # FIXME: dovrei controllare che ci siano tutti e 3 gli altri argomenti
            if sfondo is None:
                sfondo = Colore.black
            self._W = larghezza
            self._H = altezza
            self._img  = [ [sfondo for _ in range(larghezza) ] for _ in range(altezza

    def __repr__(self) -> str:
        "per stampare l'immagine ne mostro le dimensioni"
        return f"Immagine(larghezza={self._W},altezza={self._H}, ... )"

    def save(self, filename : str) -> None:
        "si salva l'immagine dopo averla convertita in matrice di triple"
        images.save(self._asTriples(), filename)

    def _asTriples(self) -> list[list[tuple[int,int,int]]] :
        "conversione della immagine da matrice di Color a matrice di triple"
        return [ [ pixel._asTriple() for pixel in riga ] for riga in self._img  ]

    def is_inside(self, x : float, y : float) -> bool:
        "verifico se le coordinate x,y sono dentro l'immagine"
        return 0 <= x < self._W and 0 <= y < self._H

    def set_pixel(self, x : float, y : float, color) -> None:
        "cambio un pixel se è dentro l'immagine"
        x = round(x)
        y = round(y)
```

```
47          if 0 <= x < self._W and 0 <= y < self._H:
48              self._img[y][x] = color
49
50      def get_pixel(self, x : float, y : float) -> Colore:
51          "leggo un pixel se è dentro l'immagine oppure torno None"
52          x = max(0, min(round(x), self._W-1))
53          y = max(0, min(round(y), self._H-1))
54          return self._img[y][x]
55
56      def visualizza(self):
57          "visualizzo l'immagine in Spyder"
58          return images.visd(self._asTriples())
59
60      # fa comodo poter trovare i colori intorno ad un punto, fino a distanza k
61      def vicini(self, x : int, y : int, k : int) -> list[Colore]:
62          "torno i colori nei pixel 2k x 2k intorno al punto x,y"
63          return [ self.get_pixel(X,Y)
64                       for X in range(x-k,x+k+1)
65                       for Y in range(y-k,y+k+1)
66                       if self.is_inside(X,Y)]
67
68  # copia?
```

Filtri come oggetti

Notate come la vita di un oggetto (istanza) sia fatta di DUE fasi

- **creazione** con tutti i parametri e le info sue personali
- **uso** con i suoi metodi

(in altri linguaggi dobbiamo anche "distruggere/disallocare" l'oggetto ma Python lo fa automaticamante)

Nell'uso dei filtri abbiamo dovuto usare trucchi come una lambda per passare parametri

Se li trasformiamo in oggetti i parametri li possiamo passare nella creazione e l'applicazione diventa più semplice

Definiamo il filtro generico e poi lo specializziamo aggiungendo funzionalità

- nell' **`__init__`** riceve tutti i parametri che gli serviranno
- ha un metodo **nuovo_pixel(self, pixel)** per i filtri che non dipendono dalla posizione
- ha un metodo **nuovo_pixel(self, immagine, x, y)** per i filtri che dipendono dalla posizione

- ha un metodo **reset(self)** per azzerarlo ed usarlo su altre immagini
- ha il metodo **applica(self, immagine)**

In [10]:
```
class GenericFilter:  # tutti i filtri specializzano questa classe
    def nuovo_pixel(self, pixel : Colore) -> Colore :
        raise NotImplementedError()
    def nuovo_pixel_XY(self, immagine : Immagine, x : int, y : int) -> Colore :
        raise NotImplementedError()
    def reset(self) -> None :
        raise NotImplementedError()
    def applica(self, immagine : Immagine) -> Immagine:
        raise NotImplementedError()
```

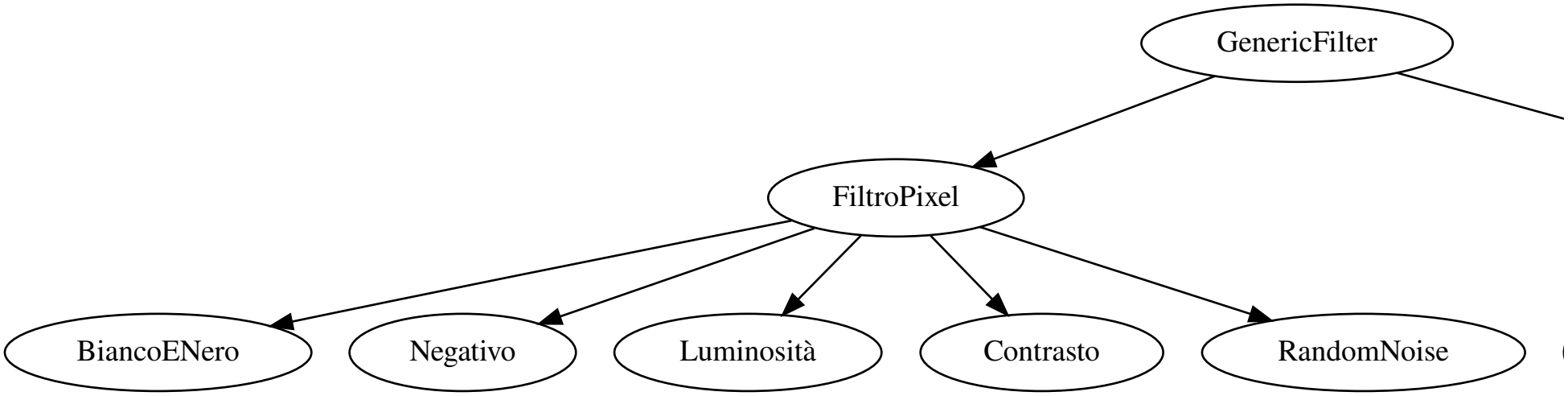
In [11]:
```
class FiltroPixel (GenericFilter):  # tutti i filtri indipendenti dalla posizione
    "un filtro che NON dipende dalla posizione del pixel"
    # applicazione di un filtro per ottenere una nuova immagine
    def applica(self, immagine : Immagine) -> Immagine:
        "costruisco una nuova immagine con ciascun pixel trasformato tramite il filtr
        assert isinstance(immagine, Immagine), f"l'oggetto {immagine} non è una Immag
        nuova_immagine = Immagine(larghezza=immagine._W,
                                  altezza=immagine._H)
        for y,riga in enumerate(immagine._img):
            for x,pixel in enumerate(riga):
                nuova_immagine._img[y][x] = self.nuovo_pixel(pixel)
        return nuova_immagine
    def nuovo_pixel(self, pixel : Colore ) -> Colore :
        "trasformazione nulla di un pixel"
        return pixel                # per default ritorno lo stesso pixel
```

In [12]:
```python
class FiltroXY (GenericFilter):    # tutti i filtri dipendenti dalla posizione
    "un filtro che DIPENDE dalla posizione del pixel"
    def applica(self, immagine : Immagine) -> Immagine:
        "applicazione di un filtro che conosce la posizione del pixel"
        assert isinstance(immagine, Immagine), f"{immagine} non è una Immagine"
        nuova_immagine = Immagine(larghezza=immagine._W, altezza=immagine._H)
        self.reset()   # se il filtro contiene info da usi precedenti le azzero
        for y in range(immagine._H):
            for x in range(immagine._W):
                nuova_immagine._img[y][x] = self.nuovo_pixel_XY(
                                                    immagine,
                                                    x,y)
        return nuova_immagine
    def nuovo_pixel_XY(self, immagine : Immagine, x : int, y : int) -> Colore :
        "trasformazione nulla di un pixel a coordinate x, y"
        return immagine.get_pixel(x,y)     # per default ritorno lo stesso pixel
    def reset(self) -> None:  # per default non faccio nulla
        pass
```

In [2]:
```python
import graphviz
figura = graphviz.Digraph()
figura.body.append('''↔
```

In [12]:
```python
figura
```

Out[12]:



BiancoENero (FiltroPixel)

- genera pixel grigi della stessa luminosità dell'originale

In [13]:
```
class BiancoENero(FiltroPixel):
    def nuovo_pixel(self, pixel : Colore ) -> Colore :
        L = pixel.luminosità()
        return Colore(L,L,L)   # grigio di luminosità L

trecime = Immagine(filename='3cime.png')
BiancoENero().applica(trecime).visualizza()
```

Negativo (FiltroPixel)

- inverte la scala di luminosità

In [14]:
```
class Negativo(FiltroPixel):
    def nuovo_pixel(self, pixel : Colore ) -> Colore :
        return Colore.white - pixel

Negativo().applica(trecime).visualizza()
```

Cambio Luminosità (FiltroPixel)

- moltiplica il colore per un fattore k

In [15]:
```
class CambioLuminosità (FiltroPixel):
    def __init__(self, k : float ) -> None:
        self._k = k
    def nuovo_pixel(self, pixel : Colore ) -> Colore :
        return pixel * self._k

CambioLuminosità(0.5).applica(trecime).visualizza()
CambioLuminosità(1.5).applica(trecime).visualizza()
```

Contrasto (FiltroPixel)

- avvicina/allontana i colori scuri e chiari al/dal grigio

In [16]:

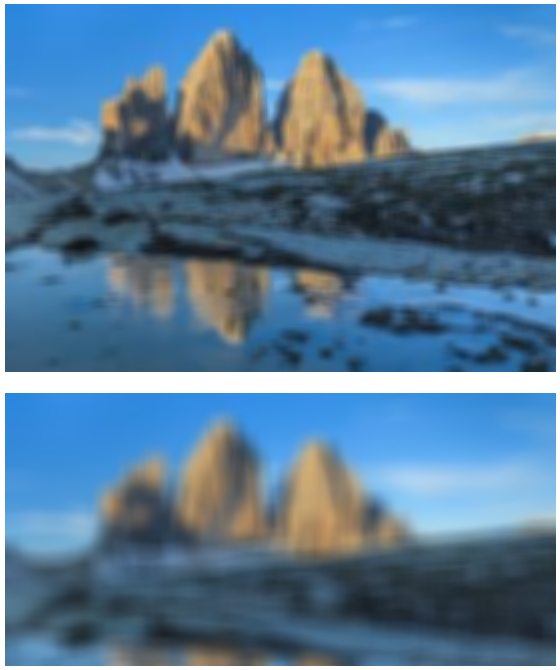
```python
class Contrasto(FiltroPixel):
    def __init__(self, k : float) -> None:
        "Contrasto(k)"
        self._k = k
    def nuovo_pixel(self, pixel : Colore )  -> Colore :
        return Colore.grey + (pixel-Colore.grey)*self._k

Contrasto(0.8).applica(trecime).visualizza()
Contrasto(1.2).applica(trecime).visualizza()
```

Sfocatura/Blur (FiltroXY)

- dà al pixel il colore medio dei vicini fino a distanza k

In [19]:

```python
class Blur(FiltroXY):
    def __init__(self, k : int):
        "inizializzo il filtro col parametro k"
        self._k = k
    def nuovo_pixel_XY(self, immagine : Immagine, x : int, y : int) -> Colore :
        "ciascun pixel è la media del gruppo grande 2k*2k che lo circonda"
        vicini = immagine.vicini(x, y, self._k )
        '''
        somma = Colore.black
        for v in vicini:
            somma += v
        return somma / len(vicini)
        '''
        return sum(vicini, Colore.black)/len(vicini)

Blur(2).applica(trecime).visualizza()
Blur(5).applica(trecime).visualizza()
```



Effetto pixellato (FiltroXY)

- per ogni pixel trova il quadretto che lo contiene
- calcola la media dei pxel nel quadretto
- si ricorda questo valore (per non doverlo ricalcolare tante volte)

- da a tutti i pixel del quadretto lo stesso valore medio

In [20]:

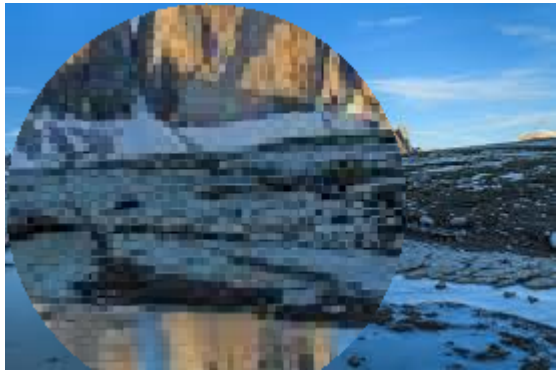
```python
class Pixellato(FiltroXY):
    """filtro che pixella l'immagine con la MEDIA dei pixel del quadretto
    """
    def __init__(self, size : int):
        "inizializzo il filtro con la dimensione del quadretto"
        self._size = size
        self._valori : dict[tuple[int,int], Colore ] = {}      # per ricordare i quad
    def reset(self) -> None :
        "per riusare lo stesso filtro su una diversa immagine lo devo resettare"
        self._valori = {}
    def nuovo_pixel_XY(self, immagine : Immagine, x : int, y : int) -> Colore:
        "ciascun pixel è la media del gruppo grande 2k*2k che lo circonda"
        X = x - x % self._size + self._size//2    # centro del quadrato
        Y = y - y % self._size + self._size//2
        if not (X,Y) in self._valori:             # ottimizzazione
            vicini = immagine.vicini(X, Y, self._size//2 )
            self._valori[X,Y] = sum(vicini, Colore.black)/len(vicini)
        return self._valori[X,Y]

Pixellato(5).applica(trecime).visualizza()
Pixellato(10).applica(trecime).visualizza()
Pixellato(15).applica(trecime).visualizza()
```

Effetto Lente (FiltroXY)

- all'esterno del cerchio della lente lascia l'immagine uguale
- all'interno legge i pixel che stanno ad una distanza dal centro della lente maggiorata/diminuita di un fattore k

In [22]:

```
class Lente(FiltroXY):
    def __init__(self, x : int, y : int,
                 raggio : int, ingrandimento : float) -> None:
        "inizializzo il filtro con posizione e raggio della lente e fattore di ingran
        self._x = x
        self._y = y
        self._raggio2 = raggio*raggio
        self._ingrandimento = ingrandimento

    def nuovo_pixel_XY(self, immagine : Immagine , x : int, y : int):
        """ciascun pixel che sta dentro la lente
            è preso da quello che sta sulla retta dal centro della lente
            ad una distanza aumentata di ingrandimento
        """
        dx = x - self._x
        dy = y - self._y
        if dx*dx + dy*dy <= self._raggio2:
            # cerca il pixel giusto
            X = self._x + dx * self._ingrandimento
            Y = self._y + dy * self._ingrandimento
            return immagine.get_pixel(X,Y)
        else:
            # altrimenti lascio il pixel così com'è
            return immagine.get_pixel(x,y)

Lente(100,100,100,0.5).applica(trecime).visualizza()
Lente(100,100,100,1.5).applica(trecime).visualizza()
```



- RandomNoise: aggiunge al pixel una variazione di **colore** casuale da -k a +k per ogni canale
- RandomLight: aggiunge al pixel una variazione di **grigio** casuale da -k a +k uguale per tutti i canali

In [25]:

```
class RandomNoise(FiltroPixel):
    def __init__(self, k : int) -> None :
        self._k = k

    def nuovo_pixel(self, colore : Colore) -> Colore :
        return colore + Colore.random(-self._k, +self._k)

class RandomLight(RandomNoise):
    def nuovo_pixel(self, colore : Colore) -> Colore :
        L = randint(-self._k, +self._k)
        return colore + Colore(L,L,L)

RandomNoise(50).applica(trecime).visualizza()
RandomLight(50).applica(trecime).visualizza()
```

Rumore sulla posizione del pixel (FiltroXY)

- sceglie un pixel vicino entro distanza k

In [26]:
```
from random import randint

class RandomNoiseXY(FiltroXY):
    def __init__(self, k : int) -> None :
        self._k = k

    def nuovo_pixel_XY(self, immagine : Immagine, x : int, y : int) -> Colore:
        X = x + randint(-self._k, self._k)
        Y = y + randint(-self._k, self._k)
        return immagine.get_pixel(X,Y)

RandomNoiseXY(5).applica(trecime).visualizza()
```

CONCLUSIONI: grazie all' OOP

- **INTERFACCIA UGUALE**: anche se i due tipi di filtro sono diversi si applicano allo stesso modo!!!
  - nomi uguali dei metodi -> oggetti intercambiabili (ricordate il "Duck Typing"?)
  - l'applicazione che li **usa** non deve preoccuparsi di come sono fatti
    (tranne in questo caso per la loro creazione)
- **RIUSO DEL CODICE**: ciascun filtro eredita le funzionalità comuni dalla superclasse e la estende
  - una unica realizzazione delle funzionalità comuni -> meno errori di copy/paste e di aggiornamento

Altro esempio: Disegno di figure sulle immagini

- una Figura ha:
  - un colore
  - eventuali altri parametri
- e può:
  - essere disegnata su una immagine in una certa posizione

In [3]:
```
figura = graphviz.Digraph()
figura.body.append('''↔
```

In [41]:
```
figura
```

Out[41]:

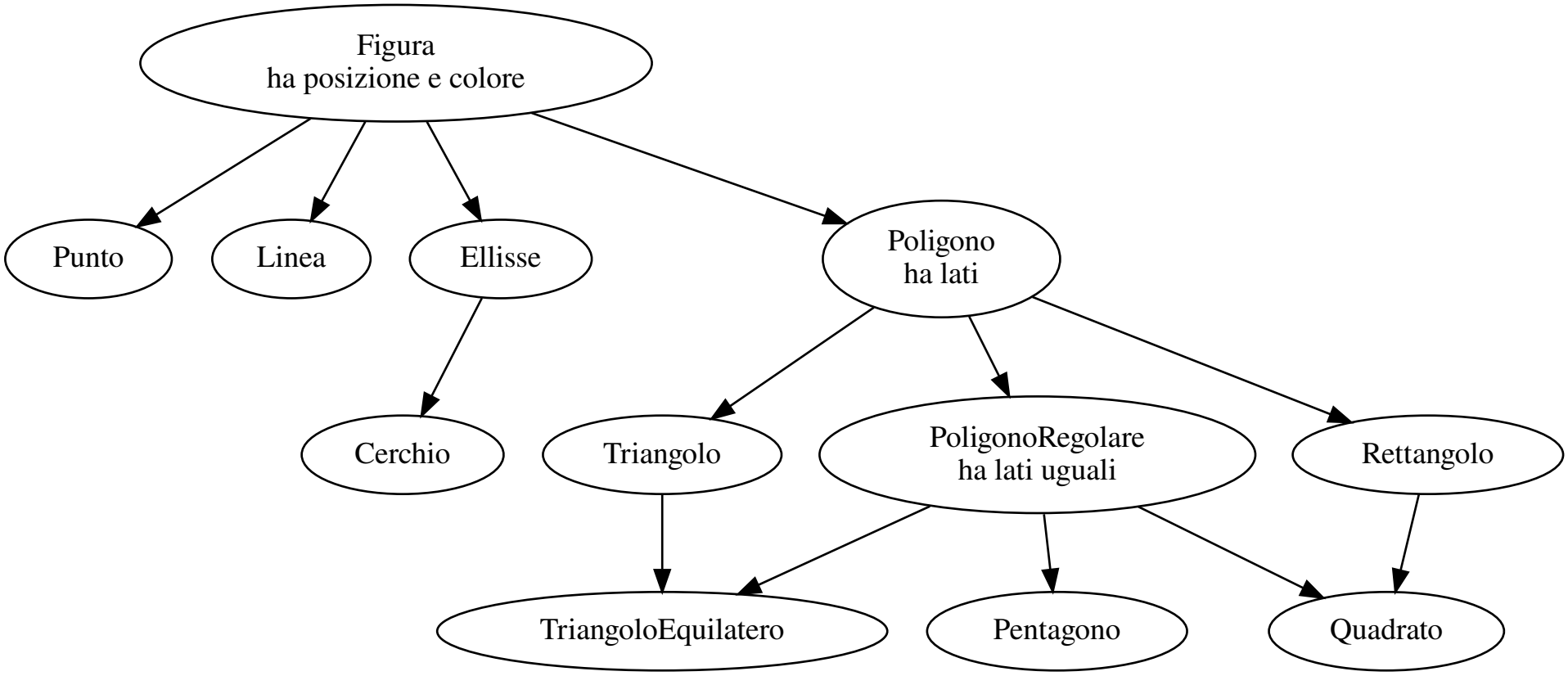


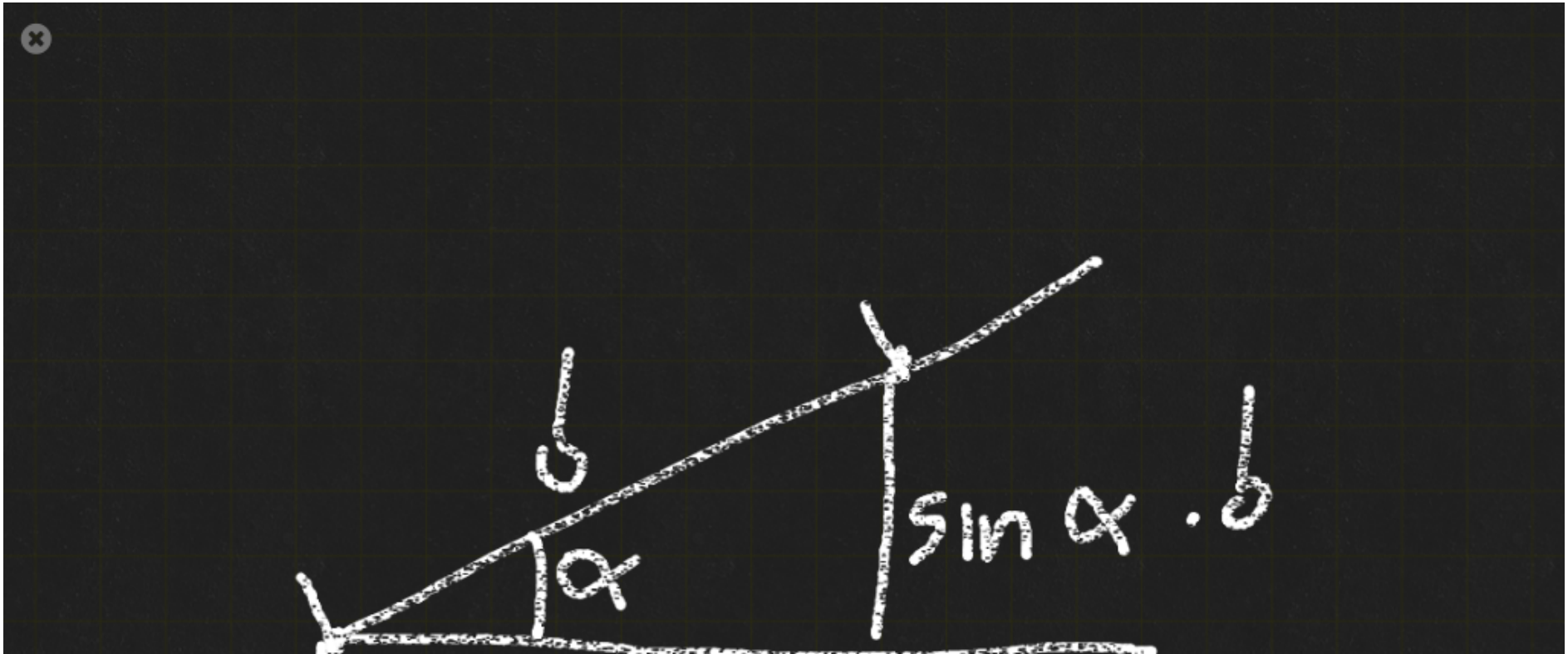
Figura e Punto

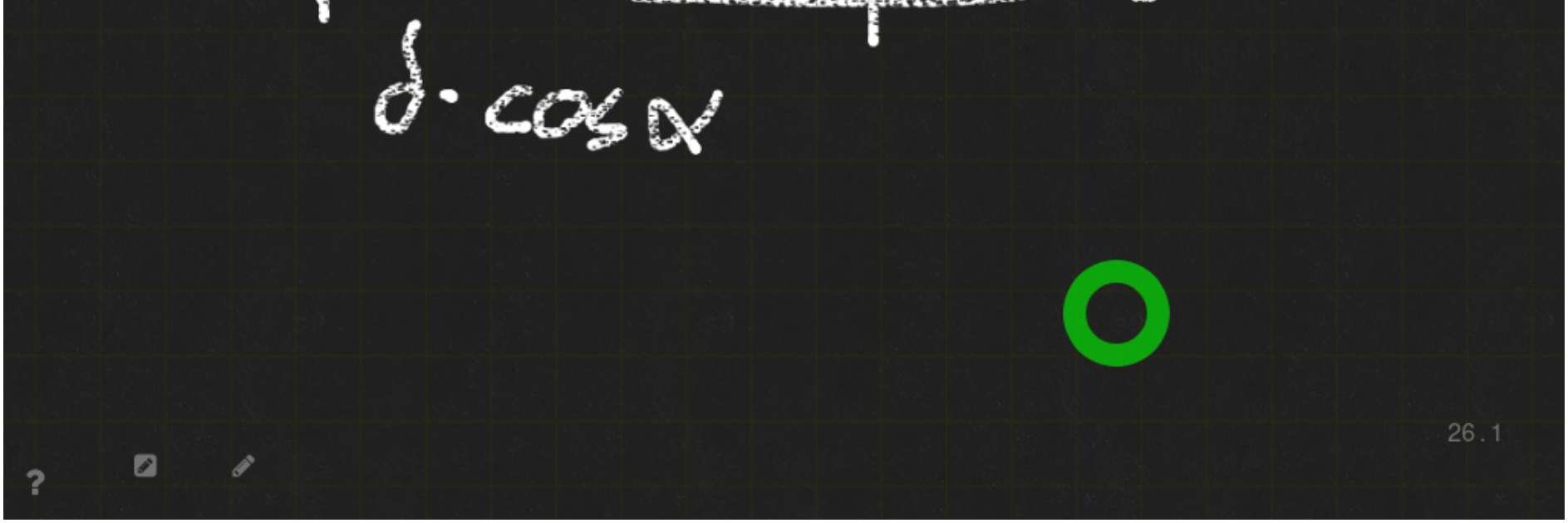
- tutte le Figure hanno un colore ed una posizione
- il punto disegna un pixel di quel colore nella posizione

In [29]:
```python
class Figura:
    def __init__(self, colore : Colore, x: float, y: float):
        self._colore = colore   # ricordo il colore
        self._x = x             # e la
        self._y = y             # posizione

    def disegna(self, immagine : Immagine) -> None:
        pass    # per default non faccio nulla

class Punto (Figura):
    def disegna(self, immagine : Immagine) -> None:
        immagine.set_pixel(self._x, self._y, self._colore)
```

Linea (segmento)

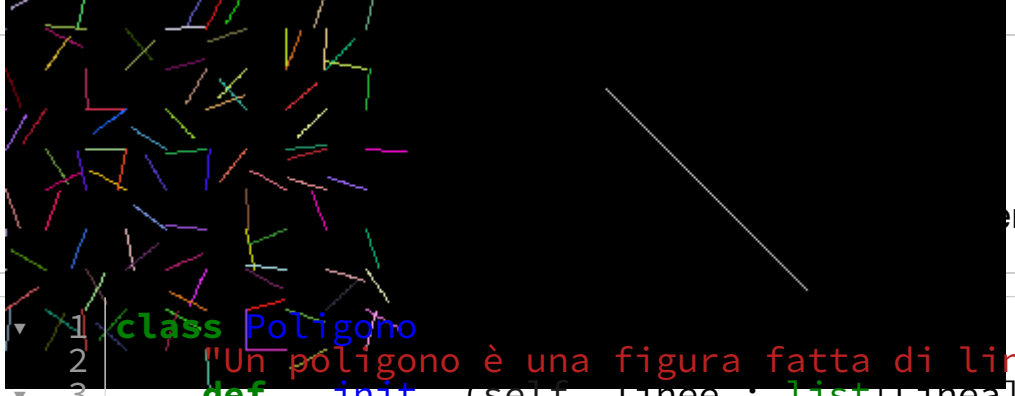
- rappresentata da un punto iniziale (x,y) e da una lunghezza ed angolo
- mi memorizzo il seno e coseno dell'angolo (in radianti) per riusarli comodamente




d · cos α
26 . 1

In [33]:

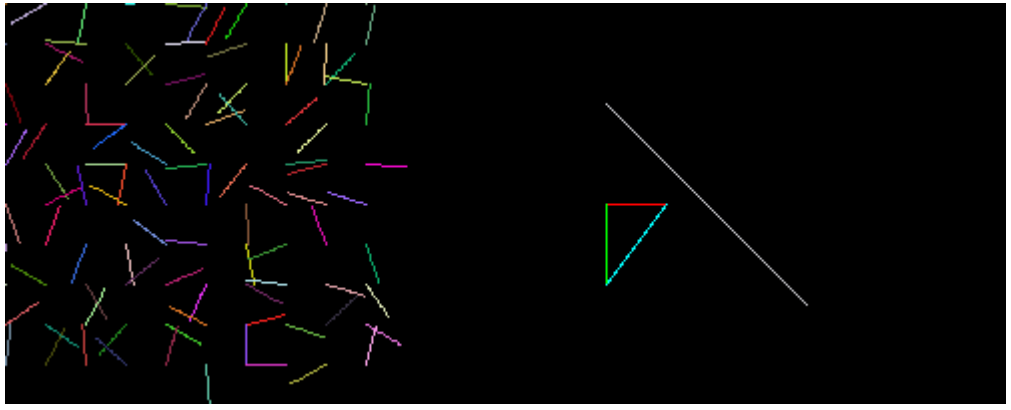
```python
import math, random
class Linea (Figura):
    def __init__(self, colore : Colore,
                 x0 : float, y0 : float,
                 lunghezza : float, angolo : float):
        "una linea che parte da un punto, e ha lunghezza e direzione"
        super().__init__(colore, x0, y0)
        self._lunghezza = lunghezza
        self._angolo    = angolo
        self._cos = math.cos(math.radians(angolo))
        self._sin = math.sin(math.radians(angolo))

    # creo un secondo costruttore (classmethod) che
    # dalle coordinate degli estremi calcola distanza ed angolo
    @classmethod
    def lato(cls, colore, x,y, x1, y1) -> 'Linea' :
        "costruttore che parte dagli estremi della linea"
        dx = x1 - x
        dy = y1 - y
        lunghezza = math.dist((x,y),(x1,y1))
        angolo = math.degrees(math.atan2(dy, dx))
        return cls(colore, x, y, lunghezza, angolo)

    # viceversa, data una linea fa comodo trovare l'altro estremo
    def estremo(self):
        "torna l'altro estremo"
        return (self._x + self._cos*self._lunghezza,
                self._y + self._sin*self._lunghezza)

    def disegna(self, immagine : Immagine) -> None:
        "disegna la linea usando trigonometria"
        # scandisco la lunghezza della linea per disegnarne i punti
        for i in range(round(self._lunghezza)+1):
            X = self._x + i*self._cos
            Y = self._y + i*self._sin
            immagine.set_pixel(X,Y,self._colore)

canvas = Immagine(500, 200)
for x in range(0,200,20):
    for y in range(0,200,20):
        Linea(Colore.random(), x, y, 20, random.randint(1,360)).disegna(canvas)

Linea.lato(Colore.random(200,250), 300, 50, 400, 150).disegna(canvas)

canvas.visualizza()
```



erficie chiusa)

In [34]:

```
class Poligono
    "Un poligono è una figura fatta di linee"
    def __init__(self, linee : list[Linea]):
        self._linee  = linee
    def disegna(self, immagine: Immagine):
        for l in self._linee:
            l.disegna(immagine)

Poligono([
    Linea.lato(Colore.red,   300, 100, 330, 100),
    Linea.lato(Colore.green, 300, 100, 300, 140),
    Linea.lato(Colore.cyan,  330, 100, 300, 140),
]).disegna(canvas)
canvas.visualizza()
```

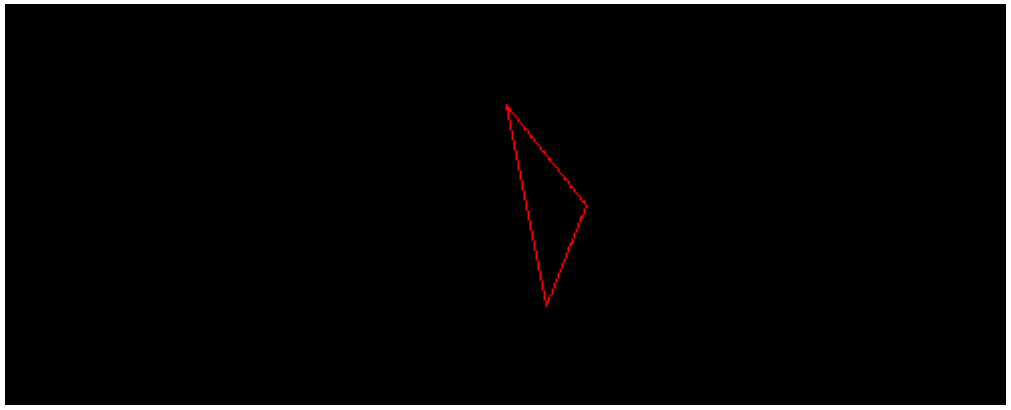


Triangolo

- dati 3 punti, ne costruisce le tre linee

In [35]:

```
class Triangolo(Poligono):
    "Un Triangolo è un poligono con 3 lati"
    def __init__(self, colore : Colore,
                       x  : float, y  : float,
                       x1 : float, y1 : float,
                       x2 : float, y2 : float):
        lato1 = Linea.lato(colore, x, y, x1,y1)
        lato2 = Linea.lato(colore, x1,y1,x2,y2)
        lato3 = Linea.lato(colore, x2,y2,x, y)
        super().__init__([lato1, lato2, lato3])

canvas = Immagine(500, 200)
Triangolo(Colore.red, 250,50, 290, 100, 270, 150).disegna(canvas)
canvas.visualizza()
```

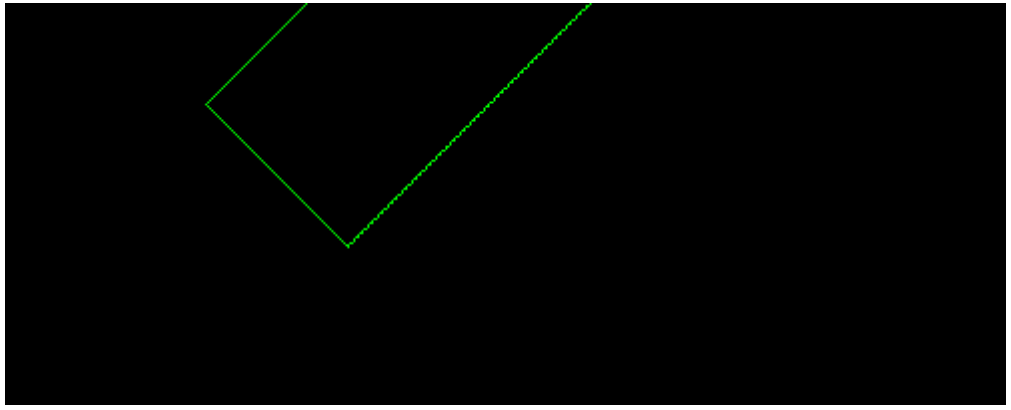


Rettangolo

- dato l'angolo sopra a sinistra, larghezza, altezza e inclinazione
- costruisce le 4 linee

In [37]:

```
class Rettangolo(Poligono):
    "Un Rettangolo è un poligono con 4 lati perpendicolari"
    def __init__(self, colore : Colore, x : float, y : float,
                 larghezza : float, altezza : float, angolo : float):
        self._larghezza = larghezza
        self._altezza   = altezza
        self._angolo    = angolo
        sopra           = Linea(colore, x, y, larghezza, angolo)
        sinistra        = Linea(colore, x, y, altezza,   angolo+90)
        x1,y1           = sopra.estremo()       # vertice in alto a destra
        destra          = Linea(colore, x1, y1, altezza, angolo+90)
        x2,y2           = sinistra.estremo()    # vertice in basso a sinistra
        sotto           = Linea(colore, x2, y2, larghezza, angolo)
        super().__init__([sopra, sotto, sinistra, destra])


canvas = Immagine(500, 200)
Rettangolo(Colore.green, 100, 50, 300, 100, -45).disegna(canvas)
canvas.visualizza()
```



un Quadrato è un rettangolo con lati uguali

In [59]:

```
class Quadrato(Rettangolo):
    def __init__(self, colore : Colore, x: float, y: float,
                 lato : int, direzione : float):
        super().__init__(colore, x, y, lato, lato, direzione)

Quadrato(Colore.red, 200, 100, 100, 30).disegna(canvas)
canvas.visualizza()
```