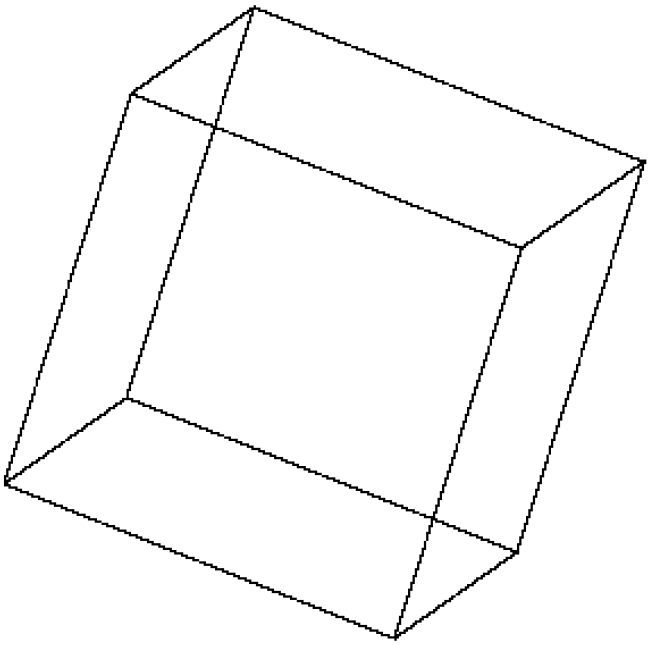

L'Effetto Schermo

Elian Isac

Lavoro di Maturità - Liceo di Locarno
2022

Professori:
Claudia Pedroni
Andrea Rainelli

Indice

Elenco delle figure

Nota: Le figure sono inserite a colori invertiti rispetto all'effettivo funzionamento del codice, per facilità di stampa.

Introduzione

Schermi, processori, Bluetooth, GPS e Wi-Fi: hanno migliorato la vita a moltissime persone e non se ne può quasi più fare a meno. Inoltre, la maggior parte della popolazione le utilizza quotidianamente, senza necessariamente avere il bisogno o il desiderio di comprendere il loro funzionamento.

Con Generazione Z [1] si intendono le persone nate tra il 1997 e il 2012: si tratta della prima generazione ad essersi sviluppata potendo godere dell'accesso ad Internet sin dall'infanzia, perciò viene considerata una generazione di persone avvezze all'uso di tutte le nuove tecnologie. Un'inevitabile conseguenza è che con il continuo sviluppo si dimenticano le tecnologie precedenti che sono state basilari, ad esempio il floppy disk e gli schermi del televisore con tubo catodico. Si arriva persino a saper utilizzare bene soltanto lo smartphone touchscreen, e non sapere come mandare un'e-mail da un PC.

Un'importante tecnologia che tutti utilizzano quasi ogni giorno è lo schermo digitale, che si basa sull'effetto di movimento creato dalla rapida successione di immagini. In questo lavoro è stata studiata questa tecnologia scrivendo un programma in linguaggio Python che consente di inserire in una grafica, come fosse uno "schermo", la colorazione di singoli pixel attraverso coordinate passate anche da altre librerie per ottenere l'effetto di forme geometriche (rettangoli, cerchi...). In una fase successiva, facendo variare le coordinate nel tempo, si possono creare diversi fotogrammi che, riprodotti uno dopo l'altro, producono l'effetto di movimento.

Capitolo 1

Sviluppo del codice - Descrizione

1.1 La Libreria

Nel linguaggio di programmazione Python, più istruzioni progettate per eseguire un'attività specifica possono essere raggruppate come funzione: assegnando a questo blocco di istruzioni un nome, diventa possibile riutilizzarle secondo necessità senza doverle riscrivere ogni volta. A una funzione possono essere passati valori come parametri e la funzione può ritornare un valore o nessun valore. Una funzione può ad esempio restituire la lunghezza di un'ipotenusa di un triangolo rettangolo se le si passano come parametri la lunghezza dei due cateti. Le librerie di Python mettono a disposizione un insieme di funzioni utili per sviluppare codice e ce ne sono molte: The Python Package Index [2] ne cita 408498 attive.

1.2 NumPy, Pygame e Pytamaro

Tra tutte le librerie esistenti sono necessarie per questo lavoro di maturità principalmente tre librerie: NumPy, Pygame e Pytamaro.

NumPy è utilizzato per il calcolo scientifico in Python: permette di creare ed effettuare calcoli con array multidimensionali. Nel codice vengono creati degli array bidimensionali, chiamati anche matrici. Questi sono un insieme di righe e colonne che creano una tabella. Vengono poi effettuati dei calcoli tra matrici e/o vettori usando, per esempio, la funzione per il calcolo del prodotto scalare.

Pygame è principalmente progettata per la programmazione di videogiochi e include moduli per la grafica computerizzata e audio. Nel codice di questo lavoro di maturità viene utilizzata per facilmente e rapidamente visualizzare delle coordinate in una schermata grafica, inserendo dei piccoli cerchi. La questione principale di questo lavoro di maturità sarà riuscire a replicare questa visualizzazione utilizzando una terza libreria.

Questa si chiama Pytamaro, ed è una libreria per la creazione di grafiche primitive, cioè forme semplici che possono essere colorate e visualizzate in una schermata. Le funzioni disponibili per manipolare queste forme sono molto

basilari e non si basano sull'utilizzo di coordinate cartesiane, invece è possibile soltanto definire la posizione di una forma rispetto all'altra: sopra, sotto, a destra, a sinistra, sovrapporre.

Questa limitazione obbliga a sviluppare un codice che semplifichi al massimo il problema con cui ci si confronta, quindi bisogna quindi capire a fondo il meccanismo logico e/o matematico.

1.3 La Proiezione Ortogonale

La proiezione ortogonale [5] è un metodo per la rappresentazione di oggetti tridimensionali in uno spazio bidimensionale, ed è una forma di proiezione parallela in cui tutte le linee di proiezione sono ortogonali al piano. In questo lavoro di maturità è utilizzato questo metodo. Il calcolo è semplificato da una matrice di proiezione che viene moltiplicata alle coordinate tridimensionali, per ottenere le coordinate che sono successivamente fornite al codice per la raffigurazione in una schermata bidimensionale.

La matrice di proiezione: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

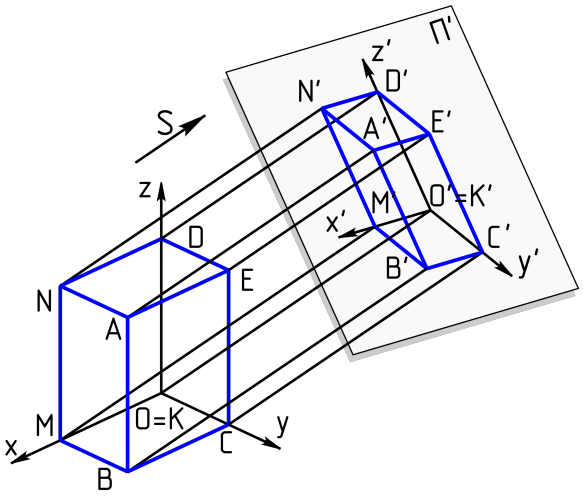

Figura 1.1: La Proiezione Ortogonale

1.4 Obiettivo Finale

L'obiettivo finale di questo Lavoro di Maturità è quello di sviluppare in linguaggio Python, utilizzando la libreria Pytamaro, un programma che accetti delle coordinate come input e le rappresenti in una schermata. Deve quindi offrire una capacità di visualizzazione simile alla funzione per l'inserimento di un punto colorato con Pygame. Inoltre deve essere possibile la creazione di animazioni in formato `.gif`, fornendo una lista di coordinate per ogni fotogramma desiderato.

Il codice sarà scritto in una struttura formale utilizzando le Docstring, la dichiarazione dei tipi e separando il codice in funzioni.

1.5 Primo Approccio

La libreria Pygame permette di inserire diverse forme geometriche in base a parametri di altezza, larghezza, raggio, colore, riempimento, coordinate x e y. Inoltre permette anche di aggiornare lo schermo in modo costante in base a un valore "frequenza dei fotogrammi" per creare l'effetto di movimento in modo molto facile.

Nella parte iniziale di ogni codice scritto in Python bisogna dichiarare le funzioni delle librerie usate nel codice e assegnare un valore alle costanti utilizzate nel programma:

```
1  import pygame
2  from math import (sin, cos)
3  import numpy as np
4
5  BIANCO = (255,255,255)
6  NERO = (0,0,0)
7  VIOLA = (200, 0, 255)
8
9  LARGHEZZA, ALTEZZA = 800, 600
10 SCALA = 100
11 offset =[LARGHEZZA/2, ALTEZZA/2]
```

Alla riga uno, viene importata l'intera libreria Pygame. Ciò significa che ogni volta che si vuole richiamare una funzione contenuta in essa bisogna scriverla nella forma: Pygame.`funzione_desiderata` . Questo procedimento non è efficiente perché l'intera libreria deve essere importata, ciò causa uno sforzo inutile per il dispositivo e crea il rischio di conflitti nel caso in cui esistono due funzioni separate con lo stesso nome.

La funzione `pygame.display.set_mode()` crea una finestra di Pygame di una certa larghezza e altezza; in questo caso vengono forniti come parametri le due costanti di larghezza e altezza. Invece la funzione `pygame.display.set_caption` permette di inserire una stringa che sarà il titolo della schermata. In questo caso viene scelto "3D Projection!"

```
13 schermo = pygame.display.set_mode((LARGHEZZA, ALTEZZA))
14 pygame.display.set_caption("3D Projection!")
```

Una funzione di ciclo, che sia `for` o `while`, sono dichiarazioni del flusso di controllo per specificare l'iterazione di una certa sezione di codice. Il ciclo `for` viene utilizzato per ripeterla un numero noto di volte, mentre quando il numero di ripetizioni necessarie non è noto a priori si utilizza un ciclo `"while"`, che ripete la sezione di codice finché una certa condizione rimane valida. Le tuple e le liste sono simili e permettono di ordinare e raggruppare diversi dati, che vengono assegnati ad una variabile; sarà quindi possibile accedere ai dati accedendo ad un certo indice della variabile. La differenza fra loro è che la tupla non può essere cambiata dopo essere stata creata in un certo ordine, mentre la lista può essere azzerata, modificata, riordinata. Per le tuple si utilizzano le parentesi tonde mentre per la lista le parentesi quadre.

Ci sono tre cicli `for` che vengono eseguiti sulla tupla `(-1, 1)`. Ognuno dei tre cicli rappresenta una delle tre dimensioni, e "sfoglia" quindi gli elementi di

questa tupla. Per esempio osservando il ciclo `for x in (-1 , 1)`, si nota che alla variabile `x` sarà temporaneamente assegnato il valore della tupla a cui il ciclo `for` è giunto, poi passa al prossimo. Inoltre la funzione `.append` permette di aggiungere a una lista già esistente una certo dato.

In questo caso, aggiungendo alla lista `punti` i valori `x, y, z` vengono create delle coordinate tridimensionali che, vista la tupla che viene utilizzata dai cicli, rappresentano le posizioni di tutti i vertici di un cubo.

```
punti =[]
for x in (-1, 1):
    for y in (-1, 1):
        for z in (-1, 1):
            punti.append(np.matrix([x, y, z]))
```

Le otto coordinate create da questi cicli `for`:

```
    [[-1 -1 -1]]
    [[-1 -1  1]]
    [[-1  1 -1]]
    [[-1  1  1]]
    [[ 1 -1 -1]]
    [[ 1 -1  1]]
    [[ 1  1 -1]]
    [[1 1 1]]
```

Poi c'è la sezione di codice che viene ripetuta con una certa frequenza: permette alla finestra di Pygame di essere aggiornata molte volte ogni secondo, creando così un movimento dei punti o oggetti, sullo schermo.

La funzione `pygame.time.Clock()` e il ciclo `for event in pygame.event.get()` sono necessarie per il funzionamento corretto di una finestra Pygame: non fanno parte del lavoro di sviluppo del codice, sono invece una sezione di codice standard. Per questo motivo non verranno descritte.

```
angolo = 0

clock = pygame.time.Clock ()
while True:


    clock.tick (60)
    for event in pygame. event.get() :
        if event.type == pygame.QUIT:
            pygame.quit()
            exit()
        if event.type == pygame.KEYDOWN:
            if event.key == pygame .K_ESCAPE:
                pygame.quit()
                exit ()

    punti_proiezione = []

    matrice_proiezione = np.matrix([
        [1,0,0],
        [0,1,0]
    ])

    rotazione_z = np.matrix([
        [cos(angolo), -sin(angolo), 0],
        [sin(angolo), cos(angolo), 0],
        [0, 0, 1],
    ])
```

```
51
52      rotazione_y = np.matrix([
53          [cos(angolo), 0, sin(angolo)],
54          [0, 1, 0],
55          [-sin(angolo), 0, cos(angolo)],
56      ])
57
58      rotazione_x = np.matrix([
59          [1, 0, 0],
60          [0, cos(angolo), -sin(angolo)],
61          [0, sin(angolo), cos(angolo)],
62      ])
63
64      schermo.fill(NERO)
65
66      i = 0
67      for point in punti:
68          ruotato2d = np.dot(rotazione_x, point.reshape((3, 1)))
```

In questo caso, dopo ogni schermata (frame) viene aumentato il valore della variabile `angolo`: questo è necessario per il calcolo della rotazione dei vertici del cubo, grazie alle matrici di rotazione [3]. Questa è un tipo di matrice di trasformazione e il suo scopo è quello di calcolare la rotazione dei una coordinata o vettore. La geometria ci fornisce quattro tipi di trasformazioni: rotazione, riflessione, traslazione e ridimensionamento.

Clock = pygame.time.Clock(), permette poi di scrivere Clock.tick(framerate): l'intero codice che si trova nel loop verrà ripetuto tante volte ogni secondo come definito dalla costante "framerate".

Il ciclo `for event in pygame.event.get` è il codice che, quando viene registrata la pressione del tasto ESCAPE, chiude la finestra di pygame.

La funzione np.matrix() permette di creare una matrice usando NumPy. Vengono create le matrici di rotazione che serviranno per calcolare la rotazione dei vertici del cubo in tre dimensioni. Per eseguire ciò verrà fatto il calcolo del prodotto scalare tra loro e la coordinata in sè.

Il ciclo for `for point in punti` a riga 37 calcola per ogni punto della lista di coordinate `punti` la rotazione su ogni asse e poi esegue il calcolo della proiezione ortogonale usando la matrice di proiezione. Il risultato è una coordinata bidimensionale che poi viene moltiplicata ad una certa costante SCALA per ingrandire la grafica finale.

Il centro del cubo tridimensionale si trova alla coordinata `(0, 0, 0)` nello spazio tridimensionale, `0, 0` sullo schermo bidimensionale. Per ottenere una grafica finale in cui il cubo perfettamente centrato bisogna addizionare un certo valore di sfalsamento: alla coordinata x bisogna aggiungere il valore LARGHEZZA / 2 e alla coordinata y ALTEZZA / 2.

Questa coordinata viene direttamente visualizzata sullo schermo attraverso la funzione `pygame.draw.circle()`, che permette di disegnare un cerchio; il raggio selezionato è talmente piccolo che nel fotogramma il cerchio sembrerà un piccolo punto. Poi inizia da capo il processo: si ottiene un effetto di rotazione della grafica dei vertici del cubo.

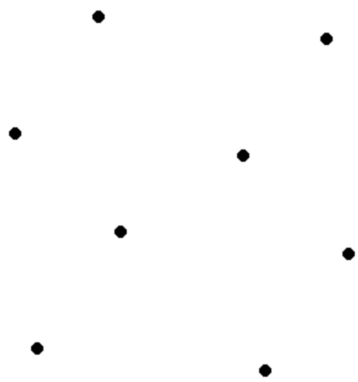

Figura 1.2: La grafica visualizzata nella finestra di Pygame

1.6 Primo Approccio a Pytamaro

Questo codice consente di ottenere una grafica con Pytamaro: una riga formata da quadratini. Uno o più di loro possono essere selezionati e colorati di bianco, mentre gli altri sono neri quindi rappresentano il vuoto dello schermo. Questo "meccanismo" viene poi utilizzato nella versione finale ma a un livello di astrazione superiore, cioè la possibilità di ottenere una schermata rettangolare formata da molte singole righe, in base alle costanti `LUNGHEZZA_RIGA` e `NUMERO_RIGHE`.

Nella parte iniziale del codice vengono dichiarate le funzioni delle librerie usate e assegnato un valore alle costanti: LUNGHEZZA_RIGA, colori, coordinata x, e viene creata una grafica vuota "fila". La funzione `accanto()` permette di affiancare due grafiche create con Pytamaro; in questo caso si tratta di un quadratino che deve essere aggiunto agli altri già affiancati che iniziano a formare la riga, ma cosa succede al primo quadratino che viene affiancato al nulla? La funzione `accanto()` richiede di fornire due argomenti che siano grafiche Pytamaro, e non esiste altro modo per lasciare completamente vuota una variabile: assegnare zero, o una lista vuota, renderebbe la variabile non valida per l'utilizzo nella funzione `accanto()`. Bisogna quindi definire `fila` una `grafica_vuota()`.

```
1 from pytamaro.it import rettangolo, accanto, colore_rgb,
      visualizza_grafica, grafica_vuota
2
3 BIANCO = colore_rgb(255, 255, 255)
4 NERO = colore_rgb(0, 0, 0)
5 LATO = 100
6 LUNGHEZZA_RIGA = 10
7 X = 2
8
9 fila = grafica_vuota()
```

Per ogni indice nell'intervallo "LUNGHEZZA_RIGA" viene determinato il colore del quadratino in base alla coordinata x assegnata precedentemente: se la coordinata x assegnata e l'indice corrente coincidono, il colore sarà bianco, se

invece non coincidono, sarà nero. Viene poi creato il quadratino "rettangolo1" in base alla costante LATO e al colore; questo viene poi affiancato alla "fila" con la funzione `accanto()`.

Quindi il codice funziona soltanto per creare una riga: manca la possibilità di produrne diverse e posizionarle una sotto l'altra creando una schermata rettangolare.

```
10 for i in range(LUNGHEZZA_RIGA):
11     print(i)
12
13     if i != X:
14         colore = NERO
15     else:
16         colore = BIANCO
17
18     rettangolo1 = rettangolo(LATO, LATO, colore)
19
20     fila = accanto(fila, rettangolo1)
21
22 visualizza_grafica(fila)
```

Figura 1.3: La riga ottenuta con Pytamaro

Il prossimo passo è risolvere l'aspetto verticale, cioè avere un numero di righe determinato da una costante selezionata all'inizio e permettere di inserire e visualizzare coordinate bidimensionali: il seguente codice affronta questo problema.

Nella parte iniziale vengono dichiarate le funzioni usate e assegnato un valore alle costanti. Inoltre vengono create due liste `x, y` che contengono i valori di due coordinate selezionate casualmente per scopo dimostrativo:

```
1  from pytamaro.it import accanto, colore_rgba, grafica_vuota,
       visualizza_grafica, rettangolo, sopra
2
3  BIANCO = colore_rgba(255, 255, 255, 255)
4  NERO = colore_rgba(0, 0, 0, 255)
5  numero = 25
6  x = [1, 14]
7  y = [1, 2]
8  coordinate = []
```

Poi viene aggiunta una coordinata [0, 0] alla lista "coordinate" e, nel ciclo `for`, viene controllato se l'indice di quella lista coincide con una delle coordinate in x inserite all'inizio. Se ciò avviene allora il valore all'indice rispettivo viene modificato da 0 a 1. Questo metodo non considera le coordinate in y, e non è nemmeno efficiente per le coordinate in x perché viene creata una sola lista che contiene tutte le coordinate, invece di avere una lista per ogni fila. Questo crea anche un problema, nel caso in cui si tenta di inserire due o più coordinate che hanno lo stesso valore di x: le coordinate verranno inserite tutte allo stesso indice nella lista, quindi si sovrascrivono e rimarrà salvata soltanto l'ultima che è stata inserita.

```
10 for i in range(numero):
11     coordinate.append([0, 0])
12     for j in x:
13         if i == j:
14             coordinate[i][0] = j
15             coordinate[i][1] = 1
```

Questo ciclo `for` crea tre liste di quadratini in base alle coordinate nella lista `coordinate`, scegliendo i colori in base al numero; quindi se in x si ha zero è nero invece se è diverso da zero, cioè uno, sarà bianco. Prendiamo in considerazione le prime quattro righe: i[1] è il valore al secondo indice di i, in questo caso il valore in Y che definisce la riga. Se i[1] è uguale a 0 possiamo essere sicuri che questa è la coordinata di un quadratino che appartiene alla prima riga, se è diverso da 0 appartiene alla seconda o alla terza. Il risultato viene poi aggiunto alla lista pixel della riga rispettiva.

```
23 for i in coordinate:
24     if i[0] != 0 and i[1] == 0:
25         pixel1.append(rettangolo(100, 100, BIANCO))
26     else:
27         pixel1.append(rettangolo(100, 100, NERO))
28
29     if i[0] != 0 and i[1] == 1:
30         pixel2.append(rettangolo(100, 100, BIANCO))
31     else:
32         pixel2.append(rettangolo(100, 100, NERO))
33
34     if i[0] != 0 and i[1] == 2:
35         pixel3.append(rettangolo(100, 100, BIANCO))
```

```
36      else:
37          pixel3.append(rettangolo(100, 100, NERO))
```

Poi tre cicli `for` affiancano ogni quadratino contenuto nelle liste `pixel 1/2/3`, quindi creano delle grafiche `riga 1/2/3`, che vengono poi a loro volta aggiunte alla lista `righe`.

```
40 screen = grafica_vuota()
41 riga = grafica_vuota()
42 riga2 = grafica_vuota()
43 riga3 = grafica_vuota()
44
45 for i in pixel1:
46     riga = accanto(riga, i)
47
48 for i in pixel2:
49     riga2 = accanto(riga2, i)
50
51 for i in pixel3:
52     riga3 = accanto(riga3, i)
53
54 righe.append(riga)
55 righe.append(riga2)
56 righe.append(riga3)
```

Vengono poi posizionati uno sotto l'altro gli elementi della lista `righe` per formare un grande rettangolo, la grafica `schermo`, che viene visualizzata grazie alla funzione `visualizza_grafica()`.

```
54 for i in righe:
55     screen = sopra(screen, i)
56
57 visualizza_grafica(screen)
```

Capitolo 2

La Versione Finale del Codice

2.1 Schermo Pytamaro

2.1.1 Descrizione dell'Algoritmo

Come descritto nel capitolo 1.4, il primo passo è fornire al programma una lista di coordinate che si vogliono visualizzare. Queste possono appartenere a una figura geometrica, oppure possono essere scelte casualmente dall'utente.

Nel caso in cui l'utente seleziona manualmente i punti che desidera colorare, è necessario conoscere il numero di coordinate che desidera inserire, per determinare quante volte bisognerà chiedere di inserire i valori x e y di una coordinata. Quando, invece, si vuole ottenere un disegno più complesso si può utilizzare un secondo programma che restituisce le coordinate, calcolate seguendo una certa logica, della forma desiderata. Questa lista deve poi essere fornita all'algoritmo per la creazione della grafica finale.

La lista è stata creata e bisogna utilizzarla per ottenere la grafica desiderata. La grafica finale è composta da quadratini di diverso colore, bianchi o neri. Se la coordinata di un quadratino appartiene alla lista di coordinate creata precedentemente, il suo colore dovrà essere bianco, mentre se non appartiene il colore dovrà essere nero. Bisogna rappresentare in una griglia o tabella ogni singolo quadratino che andrà poi a fare parte della grafica finale. Quindi si crea una tabella bidimensionale, cioè una lista di liste, in cui ogni sotto-lista è l'equivalente di una riga della tabella. Per rappresentare i quadratini, nella sotto-lista saranno inseriti diversi numeri 1 e 0 che, rispettivamente, indicano un quadratino bianco o uno nero. Questa rappresentazione si chiama lista bidimensionale perché permette di determinare una posizione in x e in y, per ogni quadratino che andrà poi a comporre la grafica finale.

Bisogna quindi generare una lista bidimensionale, il cui numero di sotto-liste sia determinato dall'altezza desiderata per la grafica finale, mentre il numero di elementi in ogni sotto-lista sia determinato dalla larghezza desiderata. La lista bidimensionale deve inizialmente contenere soltanto cifre 0, cioè quadratini neri, che rappresentano il vuoto.

Il passo successivo sarà trasformare lo 0 in 1, dove indicato dalle coordinate che si desidera visualizzare. Per modificare la lista bidimensionale in questo modo, è necessario come prima cosa selezionare una coordinata dalla lista, poi accedere alla lista bidimensionale laddove indicato dall valore su x e su y. In

quella posizione lo 0 sarà trasformato in 1, che rappresenta un quadratino bianco. Si passa poi alla prossima coordinata nella lista di coordinate e si esegue la stessa logica.

Esiste ora quindi una lista bidimensionale che non è più composta solamente da zeri: è stata modificata in base alla lista di coordinate. Questa deve essere utilizzata per definire il colore di ogni quadratino che va a formare le righe, rispettivamente lo schermo.

Per eseguire questa trasformazione bisogna iterare attraverso ogni elemento di ogni sotto-lista, e valutare il colore del quadratino da inserire: vengono creati due contatori che aumentano progressivamente, uno per le colonne e uno per le righe. Quello per le colonne va da zero fino al valore di `LUNGHEZZA_RIGA`, aumentando di uno ogni ciclo, mentre quello per le righe va da zero fino al valore di `NUMERO_RIGHE`, aumentando di uno ogni volta che il primo ha raggiunto il valore massimo dell'intervallo, cioè ha iterato attraverso l'intera sotto-lista, che è la singola riga. Il modo in cui viene letta la lista bidimensionale è comparabile al modo in cui leggiamo: parola dopo parola, riga dopo riga; quando sono state lette tutte le parole di una riga si passa alla prossima. I contatori vengono utilizzati per accedere ad un certo valore contenuto nella lista bidimensionale, 0 o 1, il quale determina il colore del rispettivo cubo.

Osservazione: Per definire un intervallo, in Python viene utilizzata la funzione `range()`. Quando si inserisce in questa funzione, per esempio, un numero intero, l'output sarà la sequenza di tutti i numeri da zero fino a $numeroinserito - 1$, allo stesso modo pure gli indici delle liste partono dalla cifra zero.

Dopo aver determinato il colore necessario, il quadratino viene assegnato ad una variabile e poi affiancato alla grafica della riga corrente; questo funzionamento si ripete per ogni elemento della sotto-lista nella lista bidimensionale, cioè ogni quadratino creato verrà aggiunto alla grafica della riga. In seguito alla creazione dell'intera riga, questa viene inserita nella grafica schermo, che dovrà contenere tutte le righe una sotto l'altra. Questi cicli, ripetendosi per ogni valore della lista bidimensionale, creano un'unica grafica finale che viene poi visualizzata.

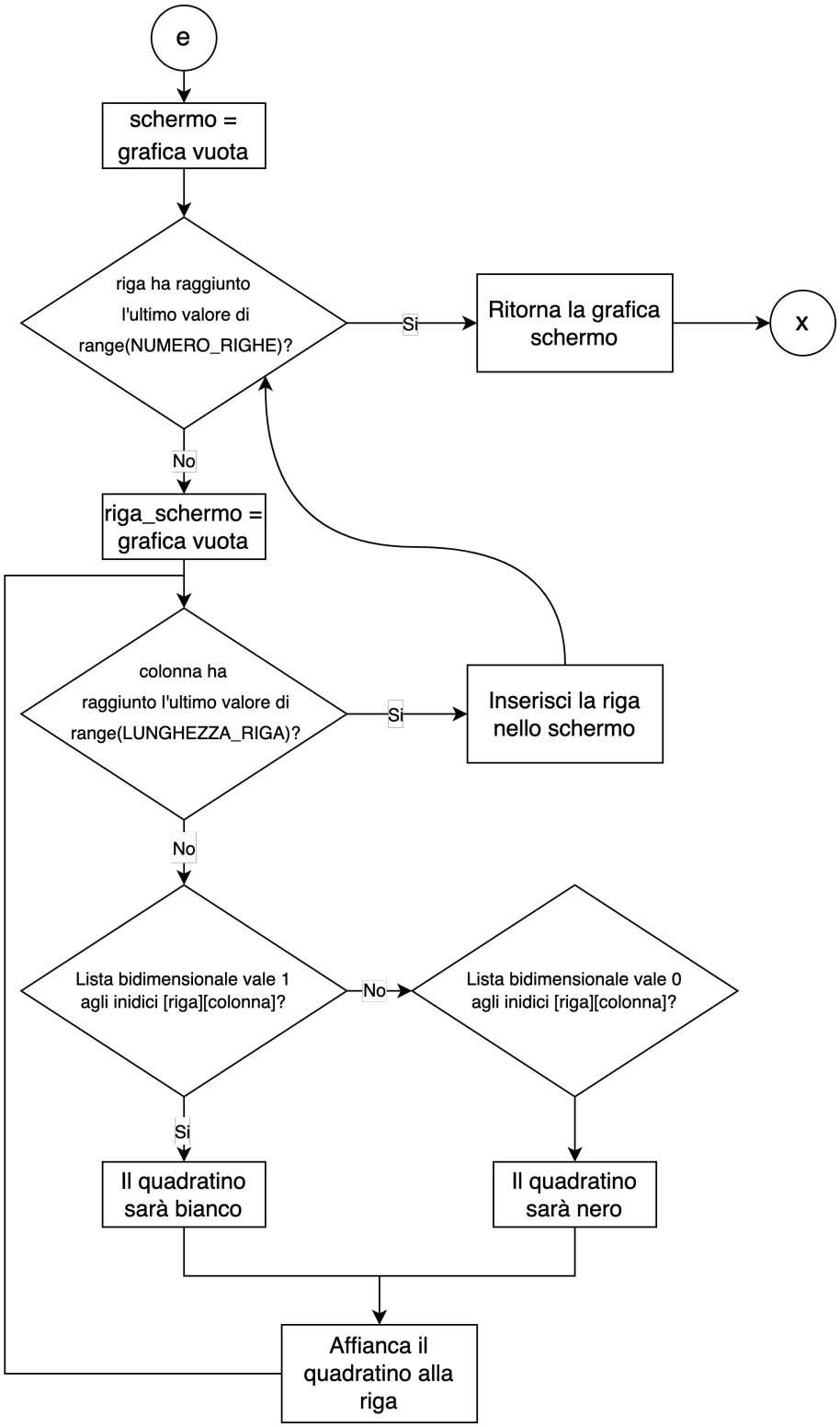

Figura 2.1: CFG della funzione `crea_grafica_schermo()`.

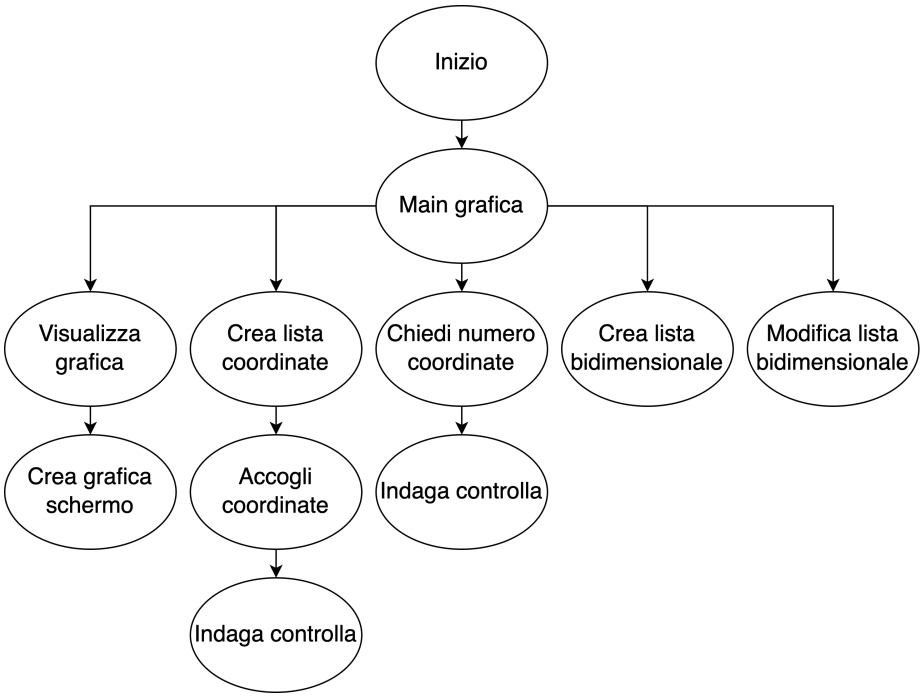

Figura 2.2: Grafo di chiamata delle funzioni per la funzione `main_grafica()`.

2.1.2 Descrizione delle Funzioni

Il programma "Schermo Pytamaro" permette di visualizzare su uno "schermo" delle coordinate scelte dall'utente oppure fornite da un altro programma, attraverso la creazione di molti quadratini di diversi colori che poi vengono affiancati grazie a una funzione della libreria `Pytamaro`. Il codice inizia importando le funzioni utilizzate da `pytamaro.it` e definendo le costanti necessarie, per esempio il numero di righe o la lunghezza di ogni riga.

```
6  from pytamaro.it import (accanto, Colore, Grafica,
7  grafica_vuota, rettangolo, salva_gif, sopra, visualizza_grafica,
       salva_grafica, sovrapponi)
8
9  BIANCO = Colore(255, 255, 255, 255)
10 NERO = Colore(0, 0, 0, 255)
11 SFONDO_SCHERMO = NERO
12 COLORE_SU_SCHERMO = BIANCO
13
14 LUNGHEZZA_RIGA = 300
15 NUMERO_RIGHE = 300
16
17 NUMERO_MASSIMO_COORDINATE = 8
18 LATO_QUADRATINO = 5
```

La funzione `crea_lista_bidimensionale` crea una lista di liste che contengono la cifra 0, la quale rappresenta lo "schermo" ancora vuoto. La lunghezza della lista che contiene gli zeri (`riga_zeri`) rappresenta la larghezza della grafica finale che è uguale alla larghezza della singola riga; mentre il numero di volte

che questa viene inserita nella lista "principale" (`matrice`) rappresenta l'altezza della grafica finale.

Per aggiungere più volte la "sub-lista", questa viene moltiplicata alla costante `LUNGHEZZA_RIGA` e il risultato viene assegnato alla variabile `riga_zeri`. Questa variabile viene inserita nella lista `matrice` e poi si ripete il processo grazie a un ciclo `for` che considera il `range`, cioè l'intervallo, da zero a `NUMERO_RIGHE`.

```
20 def crea_lista_bidimensionale() -> list[list[int]]:
21     """
22     Crea una lista di liste di zeri che rappresenta lo "schermo"
       vuoto.
23     :returns: lista bidimensionale che rappresenta lo schermo,
       ancora "vuoto"
24     """
25     matrice = []
26     for riga in range(NUMERO_RIGHE):
27         riga_zeri = [0] * LUNGHEZZA_RIGA
28         matrice.append(riga_zeri)
29     return matrice
```

L'output di questa funzione con,
`NUMERO_RIGHE` = 3; `LUNGHEZZA_RIGA` = 5
sarà:

```
1     [0, 0, 0, 0, 0]
2     [0, 0, 0, 0, 0]
3     [0, 0, 0, 0, 0]
```

La funzione `indaga_controlla` verifica che il valore inserito dall'utente sia valido e appartenga all'intervallo fornito come argomento: nel caso in cui questa condizione non si verifica viene proposto di inserire un altro valore. In qualsiasi caso in cui è richiesto un'input da parte dell'utilizzatore può essere utilizzata questa funzione per assicurare che il valore inserito permetta un corretto funzionamento.

Inizialmente, la variabile `condizione` viene definita `False`, questa rappresenta lo stato della validità dell'input; viene inizialmente considerata `False` perché non è ancora verficato se valido o no. Poi c'è un ciclo `while` che viene ripetuto fino a quando a `condizione` viene assegnato `True`.

La funzione `try` permette di controllare se ci sono errori in un segmento di codice: nel caso in cui `valore_inserito` non appartiene all'intervallo di valori possibili viene utilizzata la funzione `raise` per segnalare un errore e ulteriormente la funzione `except` per definire come questo deve essere gestito. `Else` invece è la parte di codice che viene eseguita quando il numero è valido, cioè l'utente riceve notifica del riuscito inserimento.

```
31 def indaga_controlla(valore_minimo: int, valore_massimo: int) ->
       int:
32     """
33     Chiede un input e controlla che il valore inserito sia un
       valore intero
34     entro un intervallo definito.
35     Se le condizioni non sono rispettate, viene richiesto un valore
       ; questo fino a quando
36     le condizioni sono rispettate, poi viene restituito il valore.
37
38     :param valore_minimo: valore minimo che deve avere il numero
       intero fornito come input
```

```
39      :param valore_massimo: valore massimo che deve avere il numero
        intero fornito come input
40      :returns: primo valore lecito dell'imain_esempio_video()nput
        fornito dall'utente
41      """
42      condizione = False
43      while condizione == False:
44          try:
45              valore_inserito = int(input("Inserire un numero intero
        nell'intervallo corretto. "))
46              if valore_inserito < valore_minimo or valore_inserito >
         valore_massimo:
47                  raise ValueError
48          except ValueError:
49              print("Input non adeguato. ")
50          else:
51              print("Inserimento riuscito. ")
52              condizione = True
53      return valore_inserito
```

La funzione `accogli_coordinate` chiede all'utente di inserire una coordinata, quindi un certo valore x e un valore y. Viene utilizzata in questo caso la funzione precedente per controllare che la coordinata si trova all'interno dello "schermo" costituito dai quadratini, in base alle costanti `LUNGHEZZA_RIGA` e `NUMERO_RIGHE`. Questa coordinata viene poi restituita come lista.

```
55  def accogli_coordinate() -> list[int, int]:
56      """
57      Chiede all'utente di inserire le coordinate, valore x e valore
        y.
58      :returns: coordinata_x e coordinata_y
59      """
60      print("Inserisci coordinata x")
61      coordinata_x = indaga_controlla(0, LUNGHEZZA_RIGA - 1)
62      print("inserisci coordinata y")
63      coordinata_y = indaga_controlla(0, NUMERO_RIGHE - 1)
64      return [coordinata_x, coordinata_y]
```

La funzione `chiedi_numero_coordinate` chiede all'utente di definire il numero di coordinate che vuole inserire. Viene utilizzata di nuovo la funzione `indaga_controlla` per verificare che il numero inserito non supera il valore della costante `NUMERO_MASSIMO_COORDINATE`.

```
66  def chiedi_numero_coordinate() -> int:
67      """
68      Permette all'utente di definire il numero di coordinate che
        vuole inserire.
69      :returns: il valore int inserito dall'utente
70      """
71      print("Quante coordinate vanno inserite?")
72      numero_coordinate_desiderato = indaga_controlla(0,
        NUMERO_MASSIMO_COORDINATE)
73      return numero_coordinate_desiderato
```

La funzione `crea_lista_coordinate` crea una lista di liste che contengono le coordinate inserite dall'utente. Il funzionamento si basa su un ciclo `for` che si ripete in base ad un numero fornito alla funzione, il quale determina il numero totale di coordinate che verranno inserite. Ogni volta viene chiamata la funzione `accogli_coordinate`, e poi inserita in `lista_coordinate` la coordinata che è stata fornita dall'utente.

```
75 def crea_lista_coordinate(numero: int) -> list[list[int, int]]:
76     """
77     Crea una lista di liste che contengono le coordinate scelte
       dall'utente.
78     :param numero: numero di volte che deve richiamare
       accogli_coordinate
79     :returns: la lista di coordinate in forma di liste di liste
80     """
81     lista_coordinate = []
82     for indice in range(int(numero)):
83         coordinate = accogli_coordinate()
84         lista_coordinate.append(coordinate)
85     return lista_coordinate
```

La funzione `modifica_lista_bidimensionale` modifica la lista bidimensionale che viene fornita in modo che la dove indicato dalle coordinate in `lista_coordinate`, lo 0 che rappresenta un vuoto sullo schermo diventa 1, cioè il quadratino verrà colorato. Viene inserita la cifra "1" in `lista_bidimensionale`, nella "sub-lista" e all'indice definito dalle coordinate in `lista_coordinate`: viene utilizzato un ciclo `for` che richiama ogni coordinata della lista, quindi i[1] sarà la y (quale riga/lista della lista bidimensionale) mentre i[0] sarà la x (quale indice della singola lista/riga).

```
87 def modifica_lista_bidimensionale(lista_coordinate: list[list[int,
       int]], lista_bidimensionale: list[list[int]]) -> list[list[int
       ]]:
88     """
89     Modifica la lista bidimensionale in modo che dove indicato con
       le coordinate dell'utente,
90     lo 0 viene rimpiazzato da 1
91     :param lista_coordinate: lista di coordinate inserite dall'
       utente in forma tuple
92     :returns:lista bidimensionale modificata in base alle
       coordinate inserite dall'utente
93     """
94     for i in lista_coordinate:
95         lista_bidimensionale[int(i[1])][int(i[0])] = 1
96     return lista_bidimensionale
```

La funzione `crea_grafica_schermo` trasforma la lista bidimensionale in modo che gli 0 e 1 diventano quadrati di colore nero (cioè vuoto) oppure bianco. Il funzionamento si basa su due cicli `for`: per ogni riga e per ogni colonna, se il valore in una certa posizione è 0 allora la variabile `quadratino` sarà un quadratino nero, se invece il valore è 1 a quell'indice sarà assegnato un quadratino bianco.

Per ogni ciclo di `for colonna in range(LUNGHEZZA_RIGA)` viene affiancata la variabile `quadratino` alla variabile `riga_schermo`, quindi si forma una riga intera di quadratini. Poi finito questo ciclo, per ogni ciclo di `for riga in range(NUMERO_RIGHE)`, la si affianca alla variabile `schermo`.

Sia `riga_schermo` sia `schermo` sono grafiche vuote all'inizio, quindi quando qualcosa viene affiancato a loro per la prima volta, sarà semplicemente inserito nella grafica vuota il quadratino o la lista, invece quando una grafica è già presente il quadratino o la riga saranno affiancati a quella.

```
99 def crea_grafica_schermo(lista_bidimensionale: list[list[int]]) ->
       Grafica:
100     """
101     Mappa la lista bidimensionale in modo che gli 0 e gli 1
       diventano quadrati
```

```
    di colore dello sfondo rispettivamente di un altro colore
    :param lista_bidimensionale: lista bidimensionale che contiene
    valori 0 e 1
    :returns: lista bidimensionale che contiene oggetti rettangolo
    """
    schermo = grafica_vuota()
    for riga in range(NUMERO_RIGHE):
        riga_schermo = grafica_vuota()
        for colonna in range(LUNGHEZZA_RIGA):
            if lista_bidimensionale[riga][colonna] == 1:
                quadratino = rettangolo(LATO_QUADRATINO,
    LATO_QUADRATINO, COLORE_SU_SCHERMO)
            elif lista_bidimensionale[riga][colonna] == 0:
                quadratino = rettangolo(LATO_QUADRATINO,
    LATO_QUADRATINO, SFONDO_SCHERMO)
            riga_schermo = accanto(riga_schermo, quadratino)
        schermo = sopra(schermo, riga_schermo)
    return schermo
```

La funzione `main_grafica` chiama le precedenti funzioni nell'ordine necessario per ottenere la grafica finale, uno "schermo".

```
def main_grafica():
    numero = chiedi_numero_coordinate()
    lista_coordinate = crea_lista_coordinate(numero)
    lista_bidimensionale = crea_lista_bidimensionale()
    lista_bidimensionale = modifica_lista_bidimensionale(
    lista_coordinate, lista_bidimensionale)
    visualizza_grafica(crea_grafica_schermo(lista_bidimensionale))
```

La funzione `passa_coordinate` può essere chiamata da un altro programma in linguaggio Python, importando il file "SchermoPytamaro.py". Questa permette di realizzare la visualizzazione con Pytamaro utilizzando una lista di coordinate creata da un secondo programma e avere delle grafiche più complesse e interessanti, non soltanto punti selezionati da un utente. Semplicemente richiama le funzioni del programma nell'ordine necessario, ma poi utilizza una lista esterna.

```
def passa_coordinate(lista_coordinate: list[list[int, int]]):
    """
    visualizza una grafica, usando una lista di coordinate fornita
    da un modulo esterno
    """
    lista_bidimensionale = crea_lista_bidimensionale()
    lista_bidimensionale = modifica_lista_bidimensionale(
    lista_coordinate, lista_bidimensionale)
    visualizza_grafica(crea_grafica_schermo(lista_bidimensionale))
```

La funzione `sovrapponi` che appartiene alla libreria Pytamaro permette di sovrapporre due grafiche, una sarà in primo piano e l'altra in secondo piano. Si incontra però un problema nel caso in cui vengono create due schermate separate, una che raffigura i vertici del cubo e l'altra le linee che li collegano: ogni grafica è composta da quadratini bianchi che sono le coordinate dove deve apparire il disegno voluto, e quadratini neri che rappresentano il vuoto; quando queste due grafiche vengono sovrapposte, la grafica in secondo piano sarà completamente coperta dall'altra e non sarà prodotto l'effetto finale desiderato.

Per rimediare a questo problema è necessario sovrapporre le coordinate dei punti che si desidera "colorare" prima che queste diventino vere e proprie grafiche. Utilizzando la funzione `modifica_lista_bidimensionale()` per

ogni lista di ogni grafica che si vuole ottenere sullo schermo viene modificata `lista_bidimensionale` in maniera che nessuna grafica venga coperta da un altra.

```
146 def grafica_due_liste(lista1c, lista2c):
147     lista = crea_lista_bidimensionale()
148     for i in lista1c:
149         lista = modifica_lista_bidimensionale(i, lista)
150     for i in lista2c:
151         lista = modifica_lista_bidimensionale(i, lista)
152     visualizza_grafica(crea_grafica_schermo(lista))
```

2.2 Non Solo Coordinate Inserite dall'Utente

Nel programma `SchermoPytamaro`, la funzione `main_grafica` permette all'utente di inserire manualmente le coordinate che desidera ottenere sullo "schermo". Per ottenere immagini più complesse, invece, bisogna utilizzare un secondo programma che, utilizzando funzioni matematiche, ritorna una lista di coordinate di diverse forme geometriche, per esempio i punti sulla circonferenza di un cerchio o la proiezione ortogonale dei vertici di un cubo.

Questi due esempi vengono messi in pratica nelle sezioni seguenti: Cerchio e Cubo.

2.2.1 Cerchio

Descrizione dell'Algoritmo

L'algoritmo deve fornire come output una lista di coordinate, che siano punti sulla circonferenza di un cerchio con un certo raggio. Bisogna quindi inizialmente richiedere all'utente di inserire un valore per il raggio desiderato.

Non è possibile calcolare ogni singolo punto sulla circonferenza di un cerchio, perché ce ne sono infiniti, quindi è necessario definire un certo "angolo di campionamento". Questo è l'angolo che ci deve essere tra ogni coordinata calcolata, e viene inserito dall'utente. Bisogna poi, in base ai valori di raggio e angolo di campionamento, calcolare i punti utilizzando le funzioni trigonometriche `sin` e `cos`: $P = (x, y); x = r * cos(angolo); y = r * sin(angolo)$

Dopo aver calcolato un punto, alla variabile `angolo` viene aggiunto l'angolo di campionamento, per calcolare la coordinata che si trova più avanti, e quella appena calcolata viene salvata in una lista. Questo ciclo continua fino a quando `angolo` diventa maggiore o uguale a 2π, che è l'angolo 360° in radianti, cioè quando è stato completato il giro sulla circonferenza al cerchio.

L'output finale dell'algoritmo, la lista di coordinate, può poi essere utilizzata dal programma SchermoPytamaro per visualizzare il cerchio.

Descrizione delle Funzioni

Le coordinate di tutti i punti che si trovano sulla circonferenza di un cerchio possono essere calcolati utilizzando funzioni trigonometriche. La parte iniziale:

```
5 import sys
6 sys.setrecursionlimit(1500)
7 from math import sin, cos, pi
```

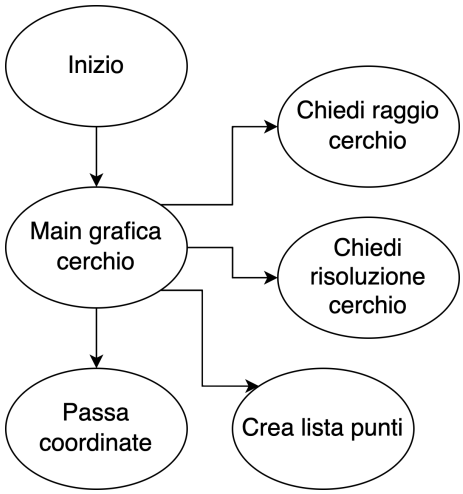

Figura 2.3: Grafo di chiamata delle funzioni - Cerchio.py

```
8 from SchermoPytamaro import passa_coordinate, LUNGHEZZA_RIGA,
      NUMERO_RIGHE, indaga_controlla
```

La funzione `chiedi_raggio_cerchio` permette all'utente di inserire il raggio desiderato per il cerchio. Il valore inserito viene assegnato alla variabile `r`:

```
10 def chiedi_raggio_cerchio() -> int:
11     """
12     Chiede all'utente qual'   il raggio desiderato per il cerchio
13     :returns: Raggio inserito
14     """
15     r = input("Inserire scelta raggio del cerchio: ")
16     return r
```

La funzione `chiedi_risoluzione_cerchio` permette all'utente di inserire la risoluzione desiderato per il cerchio. Il valore inserito viene assegnato alla variabile `risoluzione`:

```
18 def chiedi_risoluzione_cerchio() -> float:
19     """
20     Chiede all'utente qual'   la risoluzione desiderata per il
       cerchio
21     :returns: Risoluzione inserita
22     """
23     risoluzione = float(input("Inserire scelta risoluzione del
       cerchio (Step Size): "))
24     return risoluzione
```

La funzione `crea_lista_punti` crea una lista che contiene le coordinate di tutti i punti sulla circonferenza di un cerchio utilizzando le funzioni trigonometriche seno e coseno. La variabile `offset` rappresenta la coordinata del punto centrale dello "schermo" Pytamaro e viene calcolata dividendo le costanti LUNGHEZZA_RIGA e NUMERO_RIGHE per 2. A `coordinate` viene assegnata una lista per il momento ancora vuota, verrà poi riempita con le coordinate dei punti calcolati. Viene utilizzato un ciclo `while` che permette di ripetere una sezione di codice fino a quando una certa condizione non è più valida. In questo caso viene definita una variabile `t` che è il "contatore" del ciclo. Questo

contatore ha lo stato iniziale 0, poi con ogni ciclo viene aggiunto il valore della variabile `risoluzione`. La coordinata di ogni punto sulla circonferenza di un cerchio può essere definita attraverso:
$x = r * cos(angolo)$; $y = r * sin(angolo)$
Al valore di x e al valore di y viene aggiunto il valore di offset all'indice 0 rispettivamente offset all'indice 1. In questo modo la figura del cerchio avrà come punto centrale il centro della grafica invece del quadratino con coordinata $(0; 0)$

Il programma "SchermoPytamaro.py" piazza un quadratino per ogni numero intero nell'intervallo da 0 al valore di LUNGHEZZA_RIGA o NUMERO_RIGHE scelto dall'utente, quindi non è possibile fornire coordinate che contengono numeri decimali, devono avere soltanto numeri interi. Per risolvere questo problema viene utilizzata la funzione `round` che approssima il numero al numero intero più vicino.

```
26 def crea_lista_punti(risoluzione: float, r: int) -> list[list[int,
       int]]:
27     """
28     Crea la lista di punti del cerchio in base a un valore di
       risoluzione e raggio
29     :param risoluzione: Risoluzione voluta
30     :param r: Raggio voluto
31     """
32     offset =[LUNGHEZZA_RIGA/2, NUMERO_RIGHE/2]
33     coordinate = []
34     angolo = 0
35     while angolo <= 2 * pi:
36         coordinate.append([round((int(r) * cos(angolo)) + offset
       [0]), round((int(r) * sin(angolo)) + offset[1])])
37         angolo += risoluzione
38     return coordinate
```

La funzione `main_grafica_cerchio` crea la grafica del cerchio chiamando `chiedi_raggio_cerchio` e `chiedi_risoluzione_cerchio` per poi fornirli come parametro a `crea_lista_punti`. La lista ritornata da quest'ultima viene poi inserita in `passa_coordinate`. Vedi figura 2.4.

```
40 def main_grafica_cerchio():
41     raggio = chiedi_raggio_cerchio()
42     risoluzione = chiedi_risoluzione_cerchio()
43     punti = crea_lista_punti(risoluzione, raggio)
44     passa_coordinate(punti)
45
46 main_grafica_cerchio()
```

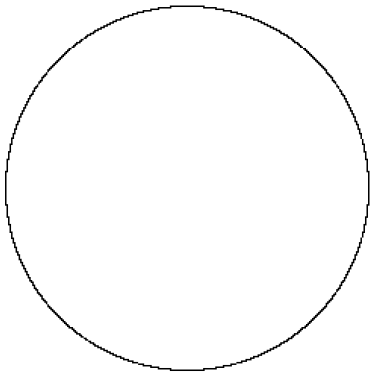

Figura 2.4: La grafica del cerchio che è stata generata dal programma.

2.2.2 Cubo

Il codice nel capitolo 1.5 "Primo approccio" viene riutilizzato per ottenere la lista di coordinate dei vertici del cubo, che sarà poi inserita come parametro nella funzione `passa_coordinate`, importata da `SchermoPytamaro`.

Viene però aggiunta al codice una funzione che permette di calcolare i punti sulla retta che collega due vertici del cubo fra loro. Così facendo si possono poi utilizzare entrambe le liste di coordinate per ottenere la grafica finale. Si ottiene l'immagine di un cubo, non soltanto i suoi vertici. Come mostrato dalla figura 3.3, si ottiene l'immagine di un vero cubo, non soltanto i suoi vertici 2.6.

Descrizione dell'Algoritmo

L'algoritmo può essere separato in due parti: la prima è il calcolo delle coordinate dei vertici del cubo, mentre la seconda è il calcolo dei punti sulla retta che collega due vertici del cubo fra loro. Queste due liste verranno poi utilizzate per ottenere la grafica finale.

Inizialmente vengono create le coordinate dei vertici di un cubo centrato in (0, 0, 0) e con lato 2. Il passo seguente è calcolare la rotazione di ogni vertice sui tre assi x, y, z, applicando il prodotto scalare tra la coordinata e, una dopo l'altra, le tre matrici di rotazione. Le matrici di rotazione contengono diverse funzioni trigonometriche, le quali richiedono un angolo. Questo è definito da una costante, quindi permette di non dovere manualmente riscrivere i diversi valori, nel caso in cui si vuole modificare l'angolo.

Per trasformare le coordinate da tridimensionali a bidimensionali, in modo che possano essere visualizzate in SchermoPytamaro, si può applicare il calcolo della proiezione ortogonale: viene calcolato il prodotto scalare tra queste e la matrice di proiezione. Il risultato del calcolo viene salvato in una lista, poi si ripete il processo per ogni vertice.

Per calcolare i punti sulla retta che collega due vertici del cubo fra loro, bisogna inizialmente definire delle variabili x1, y2, x2, y2; queste sono le due coordinate dei due vertici da collegare con una riga. Vengono utilizzate le coor-

dinate dei vertici dopo l'esecuzione del calcolo della rotazione e della proiezione ortogonale, perché in caso contrario i vertici e le righe non sarebbero allineati correttamente.

Successivamente bisogna ottenere la funzione della retta che collega i due vertici: il primo passo è calcolare la pendenza, dividendo la differenza tra i valori y per la differenza tra i valori y. La pendenza può poi essere utilizzata per calcolare l'ordinata all'origine, cioè il punto dove la retta incontra l'asse O_y. Grazie a questi due valori si riesce a definire la funzione lineare che collega i due vertici, e permette di calcolare le coordinate di diversi punti su di essa, nell'intervallo $[x1, x2]$.

Per "attraversare" questo intervallo è necessario utilizzare un ciclo `while` che si ripete fino a quando il valore della x attuale è minore o uguale a x2: questo aumenta di un piccolo valore dopo ogni ciclo. Alla variabile x attuale viene inizialmente assegnato il valore di x1, mentre alla fine del ciclo dovrà aver raggiunto x2. Si incontra però un problema nel caso in cui x1 è maggiore a x2: siccome la condizione richiesta dal ciclo è falsa già dall'inizio, il codice presente al suo interno non sarà mai eseguito. Per risolvere questo problema è necessario creare un ciclo diverso per il caso in cui x1 è maggiore a x2: la variabile x attuale deve diminuire ogni ciclo, invece di aumentare.

Vengono poi manualmente selezionate le coppie di vertici che devono essere collegate da una linea: non devono esserci linee tra vertici su lati opposti, devono vedersi soltanto gli spigoli esterni.

L'output finale dell'algoritmo sono le due liste: quella degli otto vertici dopo che questi sono stati ruotati e calcolata la proiezione ortogonale, e quella delle linee che rappresentano gli spigoli esterni del cubo. Queste due vengono successivamente fornite alla funzione `grafica_due_liste`, importata da SchermoPytamaro, per ottenere la grafica finale.

Descrizione delle Funzioni

Nella parte iniziale vengono importate le funzioni utilizzate da `math, numpy, SchermoPytamaro, sys,` e vengono definite le costanti necessarie.

```
6  from math import sin, cos
7  import numpy as np
8  from SchermoPytamaro import LUNGHEZZA_RIGA, NUMERO_RIGHE,
       grafica_due_liste
9  import sys
```

Inoltre vengono definite le matrici che sono necessarie per calcolare la proiezione ortogonale dei vertici del cubo e la loro rotazione. Sono costanti: nelle matrici di rotazioni esistono funzioni trigonometriche, le quali vengono calcolate usando un angolo definito in una costante.

```
16 matrice_proiezione = np.matrix([
17     [1,0,0],
18     [0,1,0]
19 ])
20
21 ruota_y = np.matrix([
22           [cos(ANGOLO), 0, sin(ANGOLO)],
23           [0, 1, 0],
24           [-sin(ANGOLO), 0, cos(ANGOLO)],
25     ])
26
```

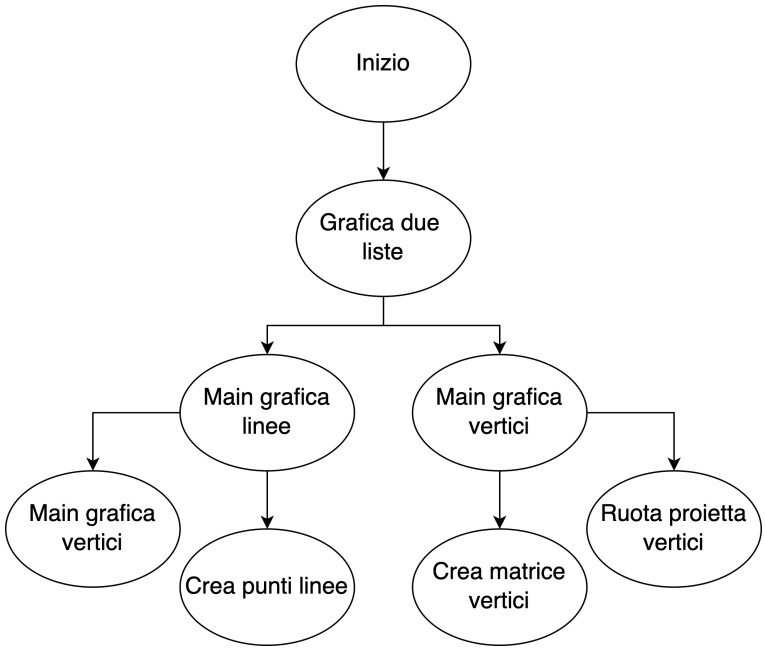

Figura 2.5: Grafo di chiamata delle funzioni - Cubo.py

```
27 ruota_z = np.matrix([
28         [cos(ANGOLO), -sin(ANGOLO), 0],
29         [sin(ANGOLO), cos(ANGOLO), 0],
30         [0, 0, 1],
31     ])
32
33 ruota_x = np.matrix([
34         [1, 0, 0],
35         [0, cos(ANGOLO), -sin(ANGOLO)],
36         [0, sin(ANGOLO), cos(ANGOLO)],
37     ])
```

La funzione `ruota_proietta_vertici()` permette di calcolare la rotazione dei vertici del cubo, e successivamente la proiezione ortogonale. A `coordinate_vertici` viene assegnata una lista vuota. Poi il ciclo `for vertice in vertici`, calcola per ogni vertice il prodotto scalare con le matrici di rotazione per gli assi x, y, z; la coordinata viene assegnata sempre alla variabile `ruota3d`.

Successivamente viene calcolato il prodotto scalare tra la matrice di proiezione e la variabile `ruota3d` per ottenere la coordinata bidimensionale di ogni vertice; questa viene assegnata alla variabile `punti_proiezione`. La coordinata viene successivamente aggiunta alla lista `coordinate_vertici`, che viene ritornata dalla funzione a termine del ciclo.

```
51 def ruota_proietta_vertici(vertici: np.matrix) -> np.matrix:
52     '''
53     Crea una lista che contiene le coordinate dei vertici del cubo,
        dopo che    stata calcolata la loro rotazione per un certo
       angolo, e la proiezione ortogonale.
54     :param vertici: I vertici del cubo in coordinate
       tridimensionali.
```

```
55      :returns: Lista di coordinate in forma bidimensionale dei
        vertici ruotati e proiettati.
56      '''
57      coordinate_vertici = []
58      for vertice in vertici:
59          ruota3d = np.dot(ruota_y, vertice.reshape((3, 1)))
60          ruota3d = np.dot(ruota_z, ruota3d)
61          ruota3d = np.dot(ruota_x, ruota3d)
62          punti_proiezione = np.dot(matrice_proiezione, ruota3d)
63          x = int(punti_proiezione[0][0] * SCALA) + offset[0]
64          y = int(punti_proiezione[1][0] * SCALA) + offset[1]
65          coordinate_vertici.append([x, y])
66      return coordinate_vertici
```

La funzione `crea_punti_linee()` ritorna una lista di coordinate che appartengono alla retta che attraversa due punti bidimensionali. Viene utilizzata per collegare i vertici del cubo con linee per ottenere una grafica realistica. Alla funzione vengono fornite le variabili `vertice1`, `vertice2`, `vertici`: le prime due sono la coppia di vertici da collegare, mentre l'ultima è la lista di vertici ottenuta dalla funzione `ruota_proietta_vertici`.

Le variabili x1, y1, x2, y2, vengono definite accedendo alle coordinate nella lista `vertici` all'indice definito da `vertice1`, oppure per la seconda coordinata `vertice2`. Si accede a delle coordinate, quindi per ottenere il valore di x si aggiunge il sotto-indice [0], mentre per le y il sotto-indice [1].

Per qualsiasi funzione retta è necessario conoscere due valori: la pendenza e l'ordinata all'origine. In questo caso la pendenza, cioè delta, viene calcolata dividendo la differenza su y per la differenza in x delle due coordinate dei vertici. Successivamente è possibile calcolare l'ordinata all'origine inserendo uno dei due punti e la pendenza nell'equazione $f(x) = mx + b$.

Una volta ottenuta la funzione, bisogna calcolare i punti che appartengono a questa nell'intervallo $[x1; x2]$. In base alla situazione in cui ci troviamo, cioè se x1 è minore o maggiore di x2, verrà utilizzato uno di due cicli `while`. Nel caso in cui `x1` è minore di `x2`, `x_attuale` che inizialmente è uguale a `x1` deve aumentare ogni ciclo. Se invece è la situazione contraria, `x_attuale` deve diminuire ogni ciclo. Ognuno dei due cicli esegue lo stesso calcolo: ottiene il valore y inserendo `x_attuale` nella funzione, poi aggiunge la coordinata alla lista `punti_linee`. Infine la funzione ritorna questa lista.

```
68  def crea_punti_linee(vertice1, vertice2, vertici):
69      '''
70      Crea una lista di punti che appartengono alla funzione retta
        che attraversa due vertici del cubo.
71      :param vertice1: Uno dei due vertici che devono essere
        attraversati dalla retta.
72      :param vertice2: Uno dei due vertici che devono essere
        attraversati dalla retta.
73      :param vertici: Lista di coordinate dei vertici del cubo.
74      :returns: Lista di punti che appartengono alla funzione retta
        che attraversa due vertici del cubo.
75      '''
76      punti_linee = []
77      x1 = int(vertici[vertice1][0])
78      y1 = int(vertici[vertice1][1])
79      x2 = int(vertici[vertice2][0])
80      y2 = int(vertici[vertice2][1])
81      delta = (y2 - y1) / (x2 - x1)
82      b = (delta * x1 - y1) * -1
```

```
83
84      x_attuale = x1
85      if x_attuale <= x2:
86          while x_attuale <= x2:
87              y = delta * x_attuale + b
88              punti_linee.append([int(x_attuale), int(y)])
89              x_attuale += 0.01
90
91      if x_attuale >= x2:
92          while x_attuale >= x2:
93              y = delta * x_attuale + b
94              punti_linee.append([int(x_attuale), int(y)])
95              x_attuale += -0.01
```

La funzione `main_grafica_vertici()` chiama le funzioni necessarie per ritornare la lista dei vertici del cubo, ruotati e proiettati.

```
99  def main_grafica_vertici():
100     grafica_vertici = crea_matrice_vertici()
101     grafica_vertici = ruota_proietta_vertici(grafica_vertici)
102     return grafica_vertici
```

La funzione `main_grafica_linee()` chiama la funzione `crea_punti_linee()` per ogni linea tra vertici che deve creare, fornendo come argomento la lista di vertici ritornata da `main_grafica_vertici()`.

```
104 def main_grafica_linee():
105     vertici = main_grafica_vertici()
106     linee = []
107     linee.append(crea_punti_linee(0, 1, vertici))
108     linee.append(crea_punti_linee(2, 0, vertici))
109     linee.append(crea_punti_linee(2, 3, vertici))
110     linee.append(crea_punti_linee(3, 1, vertici))
111
112     linee.append(crea_punti_linee(4, 5, vertici))
113     linee.append(crea_punti_linee(6, 4, vertici))
114     linee.append(crea_punti_linee(6, 7, vertici))
115     linee.append(crea_punti_linee(7, 5, vertici))
116
117     linee.append(crea_punti_linee(0, 4, vertici))
118     linee.append(crea_punti_linee(1, 5, vertici))
119     linee.append(crea_punti_linee(2, 6, vertici))
120     linee.append(crea_punti_linee(3, 7, vertici))
121     return linee
```

Le liste ritornate da `main_grafica_vertici()` e `main_grafica_linee()` vengono fornite come argomento alla funzione `grafica_due_liste()` importata da SchermoPytamaro.

```
123 grafica_due_liste(main_grafica_vertici(), main_grafica_linee())
```

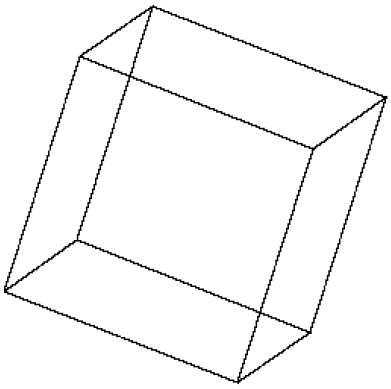

Figura 2.6: La grafica del cubo generata dal programma.

Capitolo 3

Conclusione

Riassumendo, il programma SchermoPytamaro permette di scegliere qualsiasi coordinata bidimensionale, nei limiti della grandezza della grafica finale, e visualizzarla. Ulteriormente, grazie ai programmi Cerchio e Cubo, è possibile visualizzare delle grafiche di forme geometriche più complesse.

Un ulteriore sviluppo di questi tre è il loro raggruppamento in una singola libreria, disponibile anche per altri utenti. Inoltre sarebbe possibile emulare il funzionamento della funzione `pygame.draw`, utilizzata nel capitolo 1.5, grazie a una funzione che crei la grafica di una forma geometrica desiderata, centrata in una coordinata che viene fornita come parametro. Grazie a ciò sarebbe possibile raggiungere in modo ancora più completo l'obiettivo evidenziato nel capitolo 1.4, cioè la stessa capacità di visualizzazione della funzione `pygame.draw`.

La grafica prodotta da SchermoPytamaro utilizza soltanto due colori: bianco e nero. Un ulteriore miglioramento sarebbe l'aggiunta di un algoritmo che applichi l'effetto "anti-aliasing"[4], utilizzando sfumature di grigi per ridurre l'effetto di scalettatura ai bordi delle figure raffigurate, causato dalla bassa risoluzione della grafica finale.

Capitolo 4

Codici in Forma Intera

4.1 Primo Approccio - Pygame

```
import pygame
from math import (sin, cos)
import numpy as np

BIANCO = (255,255,255)
NERO = (0,0,0)
VIOLA = (200, 0, 255)

LARGHEZZA, ALTEZZA = 800, 600
SCALA = 100
offset =[LARGHEZZA/2, ALTEZZA/2]

schermo = pygame.display.set_mode((LARGHEZZA, ALTEZZA))
pygame.display.set_caption("3D Projection!")

punti =[]
for x in (-1, 1):
    for y in (-1, 1):
        for z in (-1, 1):
            punti.append(np.matrix([x, y, z]))


angolo = 0

clock = pygame.time.Clock ()
while True:


    clock.tick (60)
    for event in pygame. event.get() :
        if event.type == pygame.QUIT:
            pygame.quit()
            exit()
        if event.type == pygame.KEYDOWN:
            if event.key == pygame .K_ESCAPE:
                pygame.quit()
                exit ()

    punti_proiezione = []

    matrice_proiezione = np.matrix([
```

```
        [1,0,0],
        [0,1,0]
    ])

    rotazione_z = np.matrix([
        [cos(angolo), -sin(angolo), 0],
        [sin(angolo), cos(angolo), 0],
        [0, 0, 1],
    ])

    rotazione_y = np.matrix([
        [cos(angolo), 0, sin(angolo)],
        [0, 1, 0],
        [-sin(angolo), 0, cos(angolo)],
    ])

    rotazione_x = np.matrix([
        [1, 0, 0],
        [0, cos(angolo), -sin(angolo)],
        [0, sin(angolo), cos(angolo)],
    ])

    schermo.fill(NERO)

    i = 0
    for point in punti:
        ruotato2d = np.dot(rotazione_x, point.reshape((3, 1)))
        ruotato2d = np.dot(rotazione_y, ruotato2d)
        ruotato2d = np.dot(rotazione_z, ruotato2d)

        proiezione2d = np.dot(matrice_proiezione, ruotato2d)
        x = int(proiezione2d[0][0] * SCALA) + offset[0]
        y = int(proiezione2d[1][0] * SCALA) + offset[1]
        pygame.draw.circle(schermo, VIOLA, (x,y), 5)

        punti_proiezione.append([x, y])
        i += 1

    angolo += 0.01
    pygame.display.update()
```

4.2 Primo Approccio a Pytamaro

Primo Codice:

```
from pytamaro.it import rettangolo, accanto, colore_rgb,
    visualizza_grafica, grafica_vuota

BIANCO = colore_rgb(255, 255, 255)
NERO = colore_rgb(0, 0, 0)
LATO = 100
LUNGHEZZA_RIGA = 10
X = 2

fila = grafica_vuota()
for i in range(LUNGHEZZA_RIGA):
    print(i)

    if i != X:
        colore = NERO
    else:
        colore = BIANCO
```

```
17
18      rettangolo1 = rettangolo(LATO, LATO, colore)
19
20      fila = accanto(fila, rettangolo1)
21
22  visualizza_grafica(fila)
```

Secondo Codice:

```
1  from pytamaro.it import accanto, colore_rgba, grafica_vuota,
       visualizza_grafica, rettangolo, sopra
2
3  BIANCO = colore_rgba(255, 255, 255, 255)
4  NERO = colore_rgba(0, 0, 0, 255)
5  numero = 25
6  x = [1, 14]
7  y = [1, 2]
8  coordinate = []
9
10 for i in range(numero):
11     coordinate.append([0, 0])
12     for j in x:
13         if i == j:
14             coordinate[i][0] = j
15             coordinate[i][1] = 1
16
17 pixel1 = []
18 pixel2 = []
19 pixel3 = []
20 righe = []
21
22
23 for i in coordinate:
24     if i[0] != 0 and i[1] == 0:
25         pixel1.append(rettangolo(100, 100, BIANCO))
26     else:
27         pixel1.append(rettangolo(100, 100, NERO))
28
29     if i[0] != 0 and i[1] == 1:
30         pixel2.append(rettangolo(100, 100, BIANCO))
31     else:
32         pixel2.append(rettangolo(100, 100, NERO))
33
34     if i[0] != 0 and i[1] == 2:
35         pixel3.append(rettangolo(100, 100, BIANCO))
36     else:
37         pixel3.append(rettangolo(100, 100, NERO))
38
39
40 screen = grafica_vuota()
41 riga = grafica_vuota()
42 riga2 = grafica_vuota()
43 riga3 = grafica_vuota()
44
45 for i in pixel1:
46     riga = accanto(riga, i)
47
48 for i in pixel2:
49     riga2 = accanto(riga2, i)
50
51 for i in pixel3:
52     riga3 = accanto(riga3, i)
53
```

```
54 righe.append(riga)
55 righe.append(riga2)
56 righe.append(riga3)
57
58 for i in righe:
59     screen = sopra(screen, i)
60
61 visualizza_grafica(screen)
```

4.3 Versione Finale del Codice

4.3.1 Schermo Pytamaro

```
1  """
2  Il programma crea una superficie rettangolare costituita da
       quadratini che assumono
3  il colore dello sfondo o un altro colore in funzione delle
       coordinate inserite dall'utente
4  """
5
6  from pytamaro.it import (accanto, Colore, Grafica,
7  grafica_vuota, rettangolo, salva_gif, sopra, visualizza_grafica,
       salva_grafica, sovrapponi)
8
9  BIANCO = Colore(255, 255, 255, 255)
10 NERO = Colore(0, 0, 0, 255)
11 SFONDO_SCHERMO = NERO
12 COLORE_SU_SCHERMO = BIANCO
13
14 LUNGHEZZA_RIGA = 300
15 NUMERO_RIGHE = 300
16
17 NUMERO_MASSIMO_COORDINATE = 8
18 LATO_QUADRATINO = 5
19
20 def crea_lista_bidimensionale() -> list[list[int]]:
21     """
22     Crea una lista di liste di zeri che rappresenta lo "schermo"
       vuoto.
23     :returns: lista bidimensionale che rappresenta lo schermo,
       ancora "vuoto"
24     """
25     matrice = []
26     for riga in range(NUMERO_RIGHE):
27         riga_zeri = [0] * LUNGHEZZA_RIGA
28         matrice.append(riga_zeri)
29     return matrice
30
31 def indaga_controlla(valore_minimo: int, valore_massimo: int) ->
       int:
32     """
33     Chiede un input e controlla che il valore inserito sia un
       valore intero
34     entro un intervallo definito.
35     Se le condizioni non sono rispettate, viene richiesto un valore
       ; questo fino a quando
36     le condizioni sono rispettate, poi viene restituito il valore.
37
38     :param valore_minimo: valore minimo che deve avere il numero
       intero fornito come input
```

```
39      :param valore_massimo: valore massimo che deve avere il numero
        intero fornito come input
40      :returns: primo valore lecito dell'imain_esempio_video()nput
        fornito dall'utente
41      """
42      condizione = False
43      while condizione == False:
44          try:
45              valore_inserito = int(input("Inserire un numero intero
        nell'intervallo corretto. "))
46              if valore_inserito < valore_minimo or valore_inserito >
         valore_massimo:
47                  raise ValueError
48          except ValueError:
49              print("Input non adeguato. ")
50          else:
51              print("Inserimento riuscito. ")
52              condizione = True
53      return valore_inserito
54
55  def accogli_coordinate() -> list[int, int]:
56      """
57      Chiede all'utente di inserire le coordinate, valore x e valore
        y.
58      :returns: coordinata_x e coordinata_y
59      """
60      print("Inserisci coordinata x")
61      coordinata_x = indaga_controlla(0, LUNGHEZZA_RIGA - 1)
62      print("inserisci coordinata y")
63      coordinata_y = indaga_controlla(0, NUMERO_RIGHE - 1)
64      return [coordinata_x, coordinata_y]
65
66  def chiedi_numero_coordinate() -> int:
67      """
68      Permette all'utente di definire il numero di coordinate che
        vuole inserire.
69      :returns: il valore int inserito dall'utente
70      """
71      print("Quante coordinate vanno inserite?")
72      numero_coordinate_desiderato = indaga_controlla(0,
        NUMERO_MASSIMO_COORDINATE)
73      return numero_coordinate_desiderato
74
75  def crea_lista_coordinate(numero: int) -> list[list[int, int]]:
76      """
77      Crea una lista di liste che contengono le coordinate scelte
        dall'utente.
78      :param numero: numero di volte che deve richiamare
        accogli_coordinate
79      :returns: la lista di coordinate in forma di liste di liste
80      """
81      lista_coordinate = []
82      for indice in range(int(numero)):
83          coordinate = accogli_coordinate()
84          lista_coordinate.append(coordinate)
85      return lista_coordinate
86
87  def modifica_lista_bidimensionale(lista_coordinate: list[list[int,
        int]], lista_bidimensionale: list[list[int]]) -> list[list[int
        ]]:
88      """
89      Modifica la lista bidimensionale in modo che dove indicato con
```

```
    le coordinate dell'utente,
    lo 0 viene rimpiazzato da 1
    :param lista_coordinate: lista di coordinate inserite dall'utente in forma tuple
    :returns:lista bidimensionale modificata in base alle coordinate inserite dall'utente
    """
    for i in lista_coordinate:
        lista_bidimensionale[int(i[1])][int(i[0])] = 1
    return lista_bidimensionale


def crea_grafica_schermo(lista_bidimensionale: list[list[int]]) -> Grafica:
    """
    Mappa la lista bidimensionale in modo che gli 0 e gli 1 diventano quadrati
    di colore dello sfondo rispettivamente di un altro colore
    :param lista_bidimensionale: lista bidimensionale che contiene valori 0 e 1
    :returns: lista bidimensionale che contiene oggetti rettangolo
    """
    schermo = grafica_vuota()
    for riga in range(NUMERO_RIGHE):
        riga_schermo = grafica_vuota()
        for colonna in range(LUNGHEZZA_RIGA):
            if lista_bidimensionale[riga][colonna] == 1:
                quadratino = rettangolo(LATO_QUADRATINO, LATO_QUADRATINO, COLORE_SU_SCHERMO)
            elif lista_bidimensionale[riga][colonna] == 0:
                quadratino = rettangolo(LATO_QUADRATINO, LATO_QUADRATINO, SFONDO_SCHERMO)
            riga_schermo = accanto(riga_schermo, quadratino)
        schermo = sopra(schermo, riga_schermo)
    return schermo

def main_grafica():
    numero = chiedi_numero_coordinate()
    lista_coordinate = crea_lista_coordinate(numero)
    lista_bidimensionale = crea_lista_bidimensionale()
    lista_bidimensionale = modifica_lista_bidimensionale(lista_coordinate, lista_bidimensionale)
    visualizza_grafica(crea_grafica_schermo(lista_bidimensionale))

def animazione_salva_gif(coordinate: list[list[int, int]]):
    progress = 1
    listagif = []
    for i in coordinate:
        lista_bidimensionale = crea_lista_bidimensionale()
        print("Starting Build " + str(progress))
        modifica_lista_bidimensionale(i, lista_bidimensionale)
        grafica_schermo = crea_grafica_schermo(lista_bidimensionale)
        listagif.append(grafica_schermo)
        progress += 1
    salva_gif("gif" + str(progress), listagif, 100)


def passa_coordinate(lista_coordinate: list[list[int, int]]):
    """
    visualizza una grafica, usando una lista di coordinate fornita da un modulo esterno
```

```
141     """
142     lista_bidimensionale = crea_lista_bidimensionale()
143     lista_bidimensionale = modifica_lista_bidimensionale(
        lista_coordinate, lista_bidimensionale)
144     visualizza_grafica(crea_grafica_schermo(lista_bidimensionale))
145 
146 def grafica_due_liste(lista1c, lista2c):
147     lista = crea_lista_bidimensionale()
148     for i in lista1c:
149         lista = modifica_lista_bidimensionale(i, lista)
150     for i in lista2c:
151         lista = modifica_lista_bidimensionale(i, lista)
152     visualizza_grafica(crea_grafica_schermo(lista))
153 
154 def due_liste_gif(lista1c, lista2c):
155     lista = crea_lista_bidimensionale()
156     for i in lista1c:
157         lista = modifica_lista_bidimensionale(i, lista)
158     for i in lista2c:
159         lista = modifica_lista_bidimensionale(i, lista)
160     visualizza_grafica(crea_grafica_schermo(lista))
```

4.3.2 Cerchio

```
1 
2 """
3 Questo modulo permette di creare una lista di coordinate del
      perimetro di un cerchio
4 """
5 import sys
6 sys.setrecursionlimit(1500)
7 from math import sin, cos, pi
8 from SchermoPytamaro import passa_coordinate, LUNGHEZZA_RIGA,
      NUMERO_RIGHE, indaga_controlla
9 
10 def chiedi_raggio_cerchio() -> int:
11     """
12     Chiede all'utente qual'  il raggio desiderato per il cerchio
13     :returns: Raggio inserito
14     """
15     r = input("Inserire scelta raggio del cerchio: ")
16     return r
17 
18 def chiedi_risoluzione_cerchio() -> float:
19     """
20     Chiede all'utente qual'  la risoluzione desiderata per il
       cerchio
21     :returns: Risoluzione inserita
22     """
23     risoluzione = float(input("Inserire scelta risoluzione del
       cerchio (Step Size): "))
24     return risoluzione
25 
26 def crea_lista_punti(risoluzione: float, r: int) -> list[list[int,
       int]]:
27     """
28     Crea la lista di punti del cerchio in base a un valore di
       risoluzione e raggio
29     :param risoluzione: Risoluzione voluta
30     :param r: Raggio voluto
31     """
32     offset =[LUNGHEZZA_RIGA/2, NUMERO_RIGHE/2]
```

```
33      coordinate = []
34      angolo = 0
35      while angolo <= 2 * pi:
36          coordinate.append([round((int(r) * cos(angolo)) + offset
        [0]), round((int(r) * sin(angolo)) + offset[1])])
37          angolo += risoluzione
38      return coordinate
39
40  def main_grafica_cerchio():
41      raggio = chiedi_raggio_cerchio()
42      risoluzione = chiedi_risoluzione_cerchio()
43      punti = crea_lista_punti(risoluzione, raggio)
44      passa_coordinate(punti)
45
46  main_grafica_cerchio()
```

4.3.3 Cubo

```
1
2   """
3   Crea coordinate cubo, ruotate
4   """
5
6   from math import sin, cos
7   import numpy as np
8   from SchermoPytamaro import LUNGHEZZA_RIGA, NUMERO_RIGHE,
        grafica_due_liste
9   import sys
10  sys.setrecursionlimit(1500)
11
12  SCALA = 70
13  ANGOLO = 0.3
14  offset = [LUNGHEZZA_RIGA/2, NUMERO_RIGHE/2]
15
16  matrice_proiezione = np.matrix([
17      [1,0,0],
18      [0,1,0]
19  ])
20
21  ruota_y = np.matrix([
22          [cos(ANGOLO), 0, sin(ANGOLO)],
23          [0, 1, 0],
24          [-sin(ANGOLO), 0, cos(ANGOLO)],
25      ])
26
27  ruota_z = np.matrix([
28          [cos(ANGOLO), -sin(ANGOLO), 0],
29          [sin(ANGOLO), cos(ANGOLO), 0],
30          [0, 0, 1],
31      ])
32
33  ruota_x = np.matrix([
34          [1, 0, 0],
35          [0, cos(ANGOLO), -sin(ANGOLO)],
36          [0, sin(ANGOLO), cos(ANGOLO)],
37      ])
38
39  def crea_matrice_vertici() -> np.matrix:
40      """
41      Crea una matrice che contiene le coordinate di tutti i vertici
        per un cubo di lato 2
42      :returns: Matrice spigoli
```

```
43      """
44      vertici =[]
45      for x in (-1, 1):
46          for y in (-1, 1):
47              for z in (-1, 1):
48                  vertici.append(np.matrix([x, y, z]))
49      return vertici
50
51  def ruota_proietta_vertici(vertici: np.matrix) -> np.matrix:
52      '''
53      Crea una lista che contiene le coordinate dei vertici del cubo,
         dopo che     stata calcolata la loro rotazione per un certo
        angolo, e la proiezione ortogonale.
54      :param vertici: I vertici del cubo in coordinate
        tridimensionali.
55      :returns: Lista di coordinate in forma bidimensionale dei
        vertici ruotati e proiettati.
56      '''
57      coordinate_vertici = []
58      for vertice in vertici:
59          ruota3d = np.dot(ruota_y, vertice.reshape((3, 1)))
60          ruota3d = np.dot(ruota_z, ruota3d)
61          ruota3d = np.dot(ruota_x, ruota3d)
62          punti_proiezione = np.dot(matrice_proiezione, ruota3d)
63          x = int(punti_proiezione[0][0] * SCALA) + offset[0]
64          y = int(punti_proiezione[1][0] * SCALA) + offset[1]
65          coordinate_vertici.append([x, y])
66      return coordinate_vertici
67
68  def crea_punti_linee(vertice1, vertice2, vertici):
69      '''
70      Crea una lista di punti che appartengono alla funzione retta
        che attraversa due vertici del cubo.
71      :param vertice1: Uno dei due vertici che devono essere
        attraversati dalla retta.
72      :param vertice2: Uno dei due vertici che devono essere
        attraversati dalla retta.
73      :param vertici: Lista di coordinate dei vertici del cubo.
74      :returns: Lista di punti che appartengono alla funzione retta
        che attraversa due vertici del cubo.
75      '''
76      punti_linee = []
77      x1 = int(vertici[vertice1][0])
78      y1 = int(vertici[vertice1][1])
79      x2 = int(vertici[vertice2][0])
80      y2 = int(vertici[vertice2][1])
81      delta = (y2 - y1) / (x2 - x1)
82      b = (delta * x1 - y1) * -1
83
84      x_attuale = x1
85      if x_attuale <= x2:
86          while x_attuale <= x2:
87              y = delta * x_attuale + b
88              punti_linee.append([int(x_attuale), int(y)])
89              x_attuale += 0.01
90
91      if x_attuale >= x2:
92          while x_attuale >= x2:
93              y = delta * x_attuale + b
94              punti_linee.append([int(x_attuale), int(y)])
95              x_attuale += -0.01
96
```

```
97      return punti_linee
98
99  def main_grafica_vertici():
100     grafica_vertici = crea_matrice_vertici()
101     grafica_vertici = ruota_proietta_vertici(grafica_vertici)
102     return grafica_vertici
103
104 def main_grafica_linee():
105     vertici = main_grafica_vertici()
106     linee = []
107     linee.append(crea_punti_linee(0, 1, vertici))
108     linee.append(crea_punti_linee(2, 0, vertici))
109     linee.append(crea_punti_linee(2, 3, vertici))
110     linee.append(crea_punti_linee(3, 1, vertici))
111
112     linee.append(crea_punti_linee(4, 5, vertici))
113     linee.append(crea_punti_linee(6, 4, vertici))
114     linee.append(crea_punti_linee(6, 7, vertici))
115     linee.append(crea_punti_linee(7, 5, vertici))
116
117     linee.append(crea_punti_linee(0, 4, vertici))
118     linee.append(crea_punti_linee(1, 5, vertici))
119     linee.append(crea_punti_linee(2, 6, vertici))
120     linee.append(crea_punti_linee(3, 7, vertici))
121     return linee
122
123 grafica_due_liste(main_grafica_vertici(), main_grafica_linee())
```

Sitografia

[1] Generazione Z. "https://www.treccani.it/vocabolario/generazione-z_%28Neologismi%29/".

[2] The Python Package Index. "https://pypi.org".

[3] Tutorial proiezione ortogonale. "https://github.com/Magoninho/3D-projection-tutorial", 2020.

[4] Wikipedia. Anti-Aliasing. "https://en.wikipedia.org/wiki/Spatial_anti-aliasing".

[5] Wikipedia. Proiezione Ortogonale. "https://it.wikipedia.org/wiki/Proiezione_(geometria)".

Dati Tecnici

IDE: Microsoft Visual Studio Code
Figure CFG e grafi della chiamata delle funzioni: Diagrams.net
Versione Python: 3.10.4