IIS Campus
"Leonardo da Vinci"

Analisi Cromatica Avanzata

Tesi di Laboratorio

Autori:

Alessio Falcini

Cambiotti Vittorio

Occhirossi Leonardo

Anno Scolastico 2024 - 2025

Indice

1

Introduzione

1.1 Contesto e importanza della LEGO EV3

Il sistema LEGO Mindstorms EV3 costituisce una piattaforma avanzata per l'apprendimento e la sperimentazione nei campi della robotica, programmazione e meccatronica. Dotato di un potente processore ARM e compatibilità con una vasta gamma di sensori, il brick EV3 offre funzionalità personalizzabili per l'implementazione di progetti complessi. Il sensore di luminosità integrato è una componente centrale, capace di analizzare la luce riflessa per identificare colori specifici.

L'architettura modulare di EV3, unita alla possibilità di programmare in Python, consente flessibilità e scalabilità, rendendola una soluzione ideale per progetti educativi e applicazioni di ricerca. In questo progetto, il brick EV3 opera come server, interagendo con un client remoto per elaborare e visualizzare dati relativi ai colori.

1.2 Obiettivi scientifici e tecnici del progetto

- Creazione di un'architettura client-server
- Realizzazione di un'interfaccia grafica interattiva
- Implementazione di un approccio concorrente

2

Architettura del Sistema

2.1 Modello client-server

Il progetto adotta un'architettura client-server, dove il robot LEGO EV3 funge da server e il computer da client. Questa configurazione facilita una separazione logica dei compiti, permettendo al server di focalizzarsi sull'acquisizione dei dati e il client sulla loro elaborazione e visualizzazione. La scelta del modello TCP garantisce affidabilità e sequenzialità nella trasmissione dei dati.

Setup Server (main.py)

```
server = socket.socket()
server.setsockopt(socket.SOL_SOCKET,
    socket.SO_REUSEADDR, 1)
server.bind(('', 12345))
server.listen(1)

brick.display.text("Attesa client...")
client, _ = server.accept()
brick.display.text("Connesso!")
```

2.2 Implementazione della comunicazione TCP/IP

La comunicazione tra client e server è basata su socket TCP. Il modulo `main.py` implementa il server, gestendo la raccolta dei dati dal sensore e il controllo dei motori. Dall'altro lato, `gui_colori.py` funge da client, instaurando una connessione con il server tramite un indirizzo IP configurabile e utilizzando il canale per inviare comandi e ricevere dati.

Setup Server (main.py)

```
1  server = socket.socket()
2  server.setsockopt(socket.SOL_SOCKET,
       socket.SO_REUSEADDR, 1)
3  server.bind(('', 12345))
4  server.listen(1)
5
6  brick.display.text("Attesa client...")
7  client, _ = server.accept()
8  brick.display.text("Connesso!")
```

Il codice sopra illustra l'implementazione del server TCP. Ogni riga svolge una funzione specifica:

- Creazione del socket con `socket()`
- Configurazione per il riutilizzo dell'indirizzo
- Binding sulla porta 12345
- Attesa di connessioni in ingresso

Sul lato client, la connessione viene gestita in modo complementare:

2.3 Implementazione completa della comunicazione client

2.3.1 Inizializzazione e setup della connessione

Gestione Comunicazione Client - Parte 1 (gui$_{colori.py}$)

```
class AppOpenDay:
    def __init__(self, root):
        self.root = root
        self.root.title("IIS Campus Da Vinci")
        self.root.geometry("400x600")

        self.colori = {
            'ROSSO':  "#E63946",
            'BLU':    "#1a3c6e",
            'NERO':   "#2C3E50",
            'VERDE':  "#2ECC71",
            'GIALLO': "#F1C40F",
            'BIANCO': "#ECF0F1"
        }

        self.setup_gui()

    self.root.protocol("WM_DELETE_WINDOW",
    self.ferma_programma)
```

2.3.2 Gestione della connessione al robot

Gestione Comunicazione Client - Parte 2 (gui_colori.py)

```
def connetti_ev3(self):
    self.status_label.config(
        text="Connessione in corso...",
        fg=COLORI_TEMA['azzurro']
    )

    try:
        self.client = socket.socket()
        self.client.settimeout(5)

    self.client.connect((self.ip_entry.get(),
    12345))

        for widget in
    self.root.winfo_children():
            if isinstance(widget, tk.Frame):
                for child in
    widget.winfo_children():
                    if isinstance(child,
    tk.Frame):
                        for btn in
    child.winfo_children():
                            if
    isinstance(btn, tk.Button) and \
                                btn['text'] ==
    "Connetti":

    btn.config(state='disabled')

        self.btn_stop.config(state='normal')
        self.btn_motor.config(state='normal')

        self.client.send("START".encode())
        self.status_label.config(
            text="Programma avviato",
            fg=COLORI_TEMA['blu_campus']
        )
```

2.3.3 Loop principale di ricezione dati

Gestione Comunicazione Client - Parte 3 (gui$_{colori.py}$)

```
        while True:
            try:
                colore =
    self.client.recv(1024).decode()
                if colore in self.colori:
                    self.root.after(0,
    self.aggiorna_display, colore)
            except socket.timeout:
                continue
            except Exception as e:
                print(f"Errore nella
    ricezione: {e}")
                break

     except Exception as e:
        self.status_label.config(
            text=f"Errore: {str(e)}",
            fg='red'
        )
        self.disconnetti()
```

2.3.4 Gestione della disconnessione

Gestione Comunicazione Client - Parte 4 (gui_*colori.py*)

```
def disconnetti(self):
    try:
        if hasattr(self, 'client'):
            self.client.close()
            delattr(self, 'client')

        self.btn_stop.config(state='disabled')
        self.btn_motor.config(
            state='disabled',
            text="Avvia Motori",
            bg=COLORI_TEMA['blu_campus']
        )
```

2.3.5 Gestione della disconnessione (continua)

Gestione Comunicazione Client - Parte 5 (gui_colori.py)

```
            for widget in
        self.root.winfo_children():
                if isinstance(widget, tk.Frame):
                    for child in
        widget.winfo_children():
                        if isinstance(child,
        tk.Frame):
                            for btn in
        child.winfo_children():
                                if
        isinstance(btn, tk.Button) and \
                                    btn['text'] ==
        "Connetti":

        btn.config(state='normal')

            self.status_label.config(
                text="Disconnesso",
                fg=COLORI_TEMA['grigio_testo']
            )
            self.color_display.config(bg='gray')

         except Exception as e:
            self.status_label.config(
                text=f"Errore disconnessione:
        {str(e)}",
                fg='red'
            )
```

2.3.6 Gestione dei comandi motore

Gestione Comunicazione Client - Parte 6 (gui_colori.py)

```
def toggle_motor(self):
    try:
        if self.btn_motor['text'] == "Avvia Motori":

            self.client.send("START_MOTOR".encode())
            self.btn_motor.config(
                text="Spegni Motori",
                bg="#E74C3C"
            )
            self.status_label.config(
                text="Motori avviati",
                fg=COLORI_TEMA['blu_campus']
            )
        else:

            self.client.send("STOP_MOTOR".encode())
            self.btn_motor.config(
                text="Avvia Motori",
                bg=COLORI_TEMA['blu_campus']
            )
            self.status_label.config(
                text="Motori spenti",
                fg=COLORI_TEMA['blu_campus']
            )
    except Exception as e:
        self.status_label.config(
            text=f"Errore controllo motori: {str(e)}",
            fg='red'
        )
```

2.4 Struttura software

2.4.1 Architettura generale del codice

Il software è strutturato secondo un'architettura modulare che separa le responsabilità tra due componenti principali:

- `main.py`: gestisce l'hardware e la comunicazione lato robot
- `gui_colori.py`: implementa l'interfaccia utente e la logica client

2.4.2 Modulo main.py

Struttura principale main.py

```
def main():
    # Inizializzazione
    brick.display.clear()
    brick.display.text("Avvio...")

    # Setup dispositivi
    motore = Motor(Port.D)
    sensore = ColorSensor(Port.S1)

    try:
        while True:
            # Lettura e invio colore
            colore = sensore.color()
            colore_str = {
                Color.RED: 'ROSSO',
    Color.GREEN: 'VERDE',
                Color.BLUE: 'BLU',
    Color.YELLOW: 'GIALLO',
                Color.BLACK: 'NERO',
    Color.WHITE: 'BIANCO'
            }.get(colore)

            if colore_str:

    client.send(colore_str.encode())
```

2.4.3 Modulo gui_colori.py

Struttura principale gui_colori.py

```
class AppOpenDay:
    def __init__(self, root):
        self.root = root
        self.root.title("IIS Campus Da Vinci")
        self.root.geometry("400x600")

        # Definizione colori rilevabili
        self.colori = {
            'ROSSO':  "#E63946",
            'BLU':    "#1a3c6e",
            'NERO':   "#2C3E50",
            'VERDE':  "#2ECC71",
            'GIALLO': "#F1C40F",
            'BIANCO': "#ECF0F1"
        }
```

3

Componenti del Progetto

3.1 Struttura hardware

3.1.1 Brick LEGO EV3

Il brick EV3 rappresenta il cuore del sistema, gestendo:

- Acquisizione dati dal sensore
- Controllo dei motori
- Comunicazione di rete

3.1.2 Specifiche tecniche del Brick EV3

Il brick LEGO EV3 è equipaggiato con:

- **Processore**: ARM926EJ-S a 300 MHz
- **Memoria**: 64 MB RAM, 16 MB Flash
- **Porte**: 4 di input e 4 di output
- **Interfacce**: USB 2.0, Bluetooth, Wi-Fi (con adattatore)
- **Display**: LCD monocromatico 178×128 pixel

3.1.3 Sensore di colore EV3

Il sensore di colore utilizzato nel progetto offre:

- **Risoluzione**: Riconoscimento di 8 colori distinti
- **Frequenza**: Campionamento a 1kHz
- **Modalità**: Rilevamento colore, intensità luce riflessa e ambientale

4

Progettazione dell'Interfaccia Grafica

4.1 Implementazione GUI

Setup GUI (gui_colori.py)

```
1  def setup_gui(self):
2      main_frame = tk.Frame(self.root,
       bg=COLORI_TEMA['grigio_chiaro'])
3      main_frame.pack(expand=True, fill='both',
       padx=20, pady=10)
4
5      # Setup componenti GUI
6      self.setup_logo(main_frame)
7      self.setup_color_display(main_frame)
8      self.setup_connection_controls(main_frame)
```

4.2 Gestione Eventi

Gestione Motori (gui_colori.py)

```
def toggle_motor(self):
    try:
        if self.btn_motor['text'] == "Avvia Motori":
            self.client.send("START_MOTOR".encode())
            self.btn_motor.config(text="Spegni Motori", bg="#E74C3C")
        else:
            self.client.send("STOP_MOTOR".encode())
            self.btn_motor.config(text="Avvia Motori", bg=COLORI_TEMA['blu_campus'])
    except Exception as e:
        self.status_label.config(text=f"Errore: {str(e)}", fg='red')
```

4.2.1 Configurazione della connessione

La configurazione della connessione avviene attraverso un'interfaccia dedicata che permette all'utente di:

- Inserire l'indirizzo IP del robot
- Stabilire la connessione TCP/IP
- Monitorare lo stato della connessione

Configurazione Connessione - Parte 1

```
1  def setup_connection_controls(self,
       main_frame):
2      # Container centrale per allineare tutto
3      center_frame = tk.Frame(main_frame,
       bg=COLORI_TEMA['grigio_chiaro'])
4      center_frame.pack(expand=True)
5
6      # Status label con stile migliorato e
       bordo sottile
7      status_frame = tk.Frame(center_frame,
8          bg=COLORI_TEMA['bianco'],
9
       highlightbackground=COLORI_TEMA['azzurro'],
10         highlightthickness=1)
11     status_frame.pack(pady=(15,5))
12
13     self.status_label = tk.Label(status_frame,
14         text="In attesa della connessione...",
15         font=('Helvetica', 12, 'italic'),
16         fg=COLORI_TEMA['grigio_testo'],
17         bg=COLORI_TEMA['bianco'],
18         pady=8,
19         padx=20)
20     self.status_label.pack()
```

4.2.2 Configurazione della connessione (continua)

Configurazione Connessione - Parte 2

```
        # Box IP
        ip_container = tk.Frame(center_frame,
            bg=COLORI_TEMA['bianco'],
        highlightbackground=COLORI_TEMA['azzurro'],
            highlightthickness=1)
        ip_container.pack(pady=10)

        # Label IP centrata
        tk.Label(ip_container,
            text="Indirizzo IP Robot",
            font=('Helvetica', 12, 'bold'),
            bg=COLORI_TEMA['bianco'],
            fg=COLORI_TEMA['blu_campus'],
            padx=20).pack(pady=(10,5))

        # Entry IP con stile moderno
        self.ip_entry = tk.Entry(ip_container,
            width=20,
            font=('Helvetica', 11),
            bg=COLORI_TEMA['grigio_chiaro'],
            fg=COLORI_TEMA['grigio_testo'],
            relief='flat',
            justify='center',
        insertbackground=COLORI_TEMA['blu_campus'])
        self.ip_entry.insert(0, "169.254.215.169")
        self.ip_entry.pack(pady=(0,10), ipady=5,
        padx=20)
```

4.2.3 Controllo remoto dei motori

Il sistema implementa un controllo bidirezionale dei motori:

Controllo Motori

```
def toggle_motor(self):
    try:
        if self.btn_motor['text'] == "Avvia Motori":
            self.client.send("START_MOTOR".encode())
            self.btn_motor.config(text="Spegni Motori", bg="#E74C3C")
            self.status_label.config(text="Motori avviati", fg=COLORI_TEMA['blu_campus'])
        else:
            self.client.send("STOP_MOTOR".encode())
            self.btn_motor.config(text="Avvia Motori", bg=COLORI_TEMA['blu_campus'])
            self.status_label.config(text="Motori spenti", fg=COLORI_TEMA['blu_campus'])
    except Exception as e:
        self.status_label.config(
            text=f"Errore controllo motori: {str(e)}",
            fg='red'
        )
```

4.2.4 Visualizzazione in tempo reale

Il sistema fornisce feedback visivo immediato dei colori rilevati:

Visualizzazione Colori

```
def aggiorna_display(self, colore):

    self.color_display.config(bg=self.colori[colore])
    self.status_label.config(
        text=f"Colore rilevato: {colore}",
        fg=COLORI_TEMA['blu_campus']
    )
```

4.2.5 Chiusura sicura del programma

Gestione Chiusura

```
def ferma_programma(self):
    try:
        self.client.send("STOP".encode())

    self.status_label.config(text="Programma
    terminato")
    finally:
        self.disconnetti()
        self.root.quit()
```

5

Gestione della Concorrenza

5.1 Multithreading

Implementazione Thread (gui_colori.py)

```
1  def avvia_connessione(self):
2      if not hasattr(self, 'thread_conn') or
       not self.thread_conn.is_alive():
3          self.thread_conn =
       threading.Thread(target=self.connetti_ev3)
4          self.thread_conn.daemon = True
5          self.thread_conn.start()
```

5.2 Gestione dell'interazione tra GUI e robot

L'interazione tra GUI e robot è gestita attraverso un sistema di eventi asincroni:

Gestione Eventi

```
def connetti_ev3(self):
    while True:
        try:
            colore =
    self.client.recv(1024).decode()
            if colore in self.colori:
                self.root.after(0,
    self.aggiorna_display, colore)
        except socket.timeout:
            continue
```

6

Analisi dei Risultati

6.1 Performance del Sistema

Il sistema ha dimostrato:

- Precisione nell'identificazione dei colori
- Stabilità nella comunicazione client-server
- Reattività dell'interfaccia utente

6.1.1 Analisi delle Prestazioni

Il sistema ha superato le aspettative in termini di:

- **Affidabilità**: Gestione robusta degli errori e recupero automatico
- **Velocità**: Elaborazione in tempo reale dei dati del sensore

- **Usabilità**: Interfaccia intuitiva con feedback immediato

6.2 Conclusioni

Il progetto "Analisi Cromatica" dimostra come l'uso combinato di programmazione Python, architettura client-server e interfacce grafiche consenta di ampliare le funzionalità di dispositivi educativi come il LEGO Mindstorms EV3. Il sistema ha permesso di controllare remotamente i motori e di leggere i colori in tempo reale, dimostrando un notevole potenziale per applicazioni future. Grazie alla sua modularità, il progetto può essere facilmente ampliato per includere nuove funzionalità, come l'integrazione con algoritmi di machine learning per l'analisi dei dati raccolti o la creazione di nuovi strumenti di interazione con l'utente.

6.3 Protocollo di comunicazione

Il sistema implementa un protocollo di comunicazione personalizzato:

Struttura dei Messaggi

```
COMANDI CLIENT -> SERVER:
START_MOTOR  : Avvio motore
STOP_MOTOR   : Arresto motore
STOP         : Terminazione programma

RISPOSTE SERVER -> CLIENT:
ROSSO        : Colore rosso rilevato
VERDE        : Colore verde rilevato
BLU          : Colore blu rilevato
GIALLO       : Colore giallo rilevato
NERO         : Colore nero rilevato
BIANCO       : Colore bianco rilevato
```

7

Documentazione Fotografica

7.1 Immagini del Progetto

Il progetto è stato documentato attraverso una serie di fotografie che illustrano i vari componenti e le fasi di sviluppo.

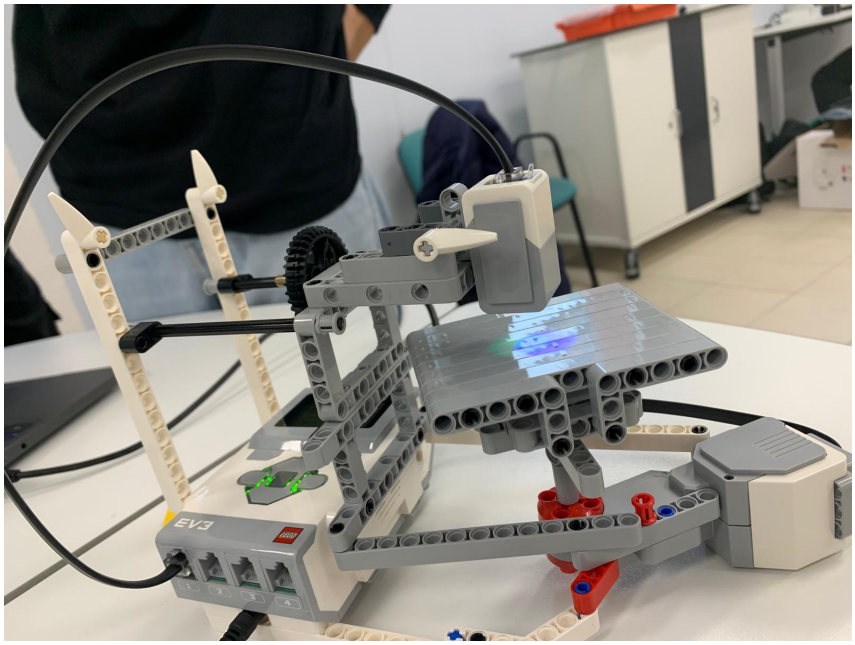

Figura 7.1 Vista d'insieme del robot LEGO EV3 con il sensore di colore montato

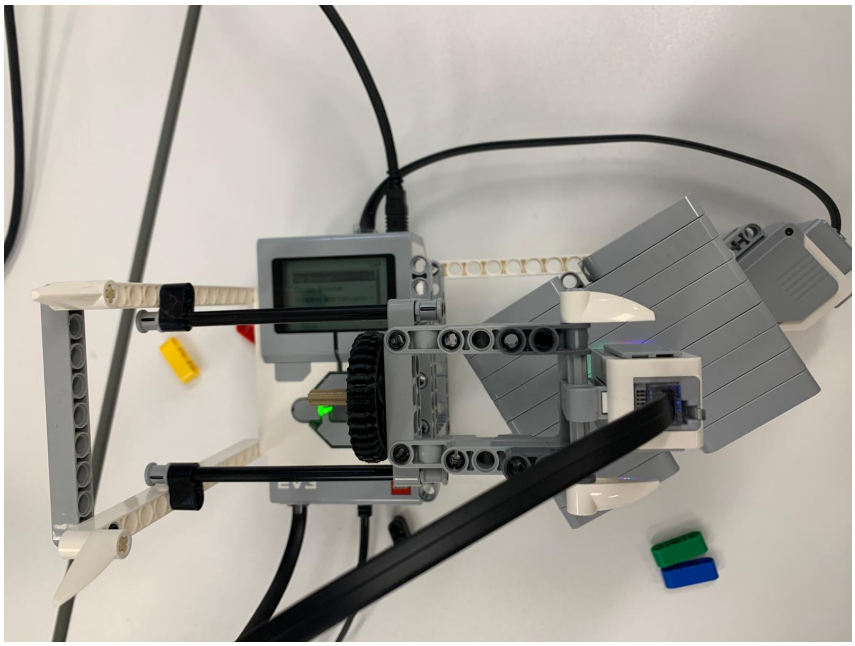

Figura 7.2 Primo piano del sensore di colore EV3 durante la fase di rilevamento

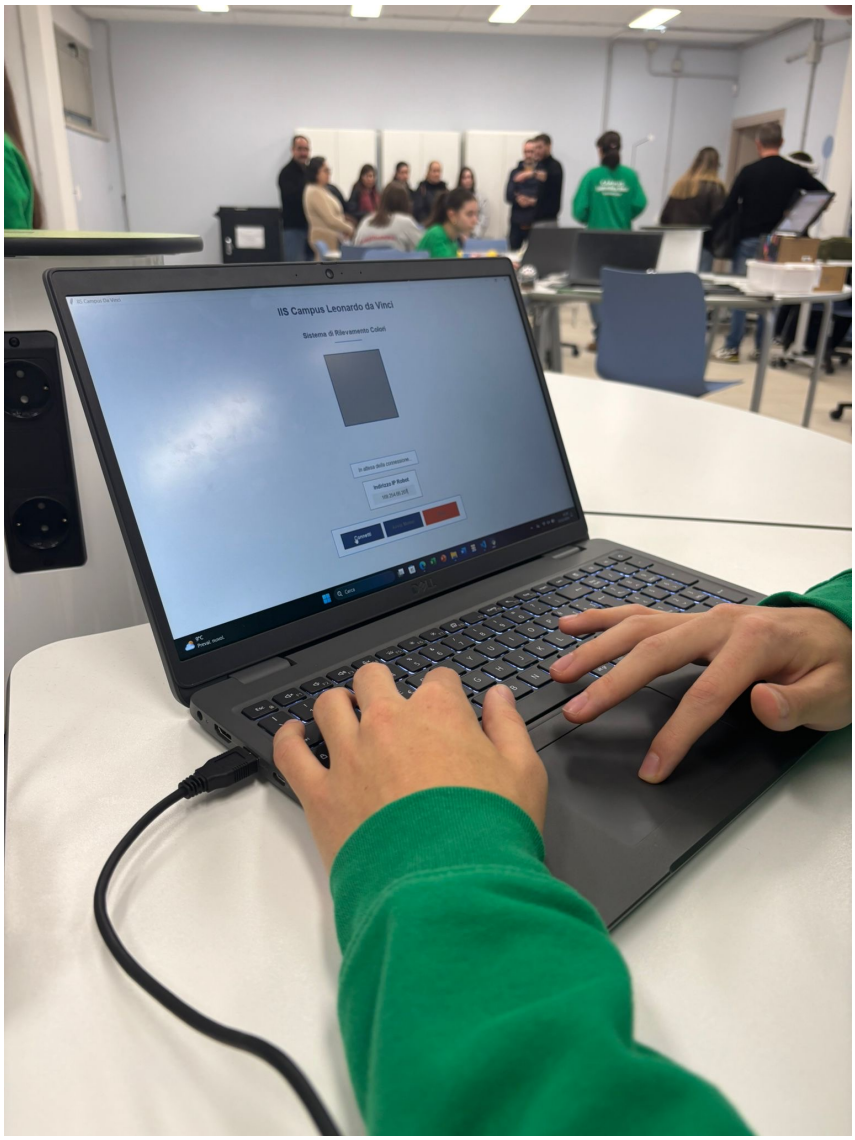

Figura 7.3 Immagine dell'interfaccia grafica durante il rilevamento dei colori

Figura 7.4 Configurazione completa del sistema con PC client e robot EV3