PhotoFish
DIGITAL FRAME WIFI

Versione wireless delle cornici digitali per riprodurre immagini ricevute da WiFi grazie a Fishino e allo shield TFT. Dopo l'hardware e le librerie, facciamo un po' di didattica sulla gestione delle immagini. Seconda e ultima puntata.

di MASSIMO DEL FEDELE

Nella prima puntata abbiamo avuto modo di introdurvi il progetto, che utilizzando una board Fishino (tipicamente, non necessariamente, una 32 bit) riceve tramite link wireless file d'immagine che gli inviamo tramite il servizio di instant messaging Telegram e li visualizza su un display LCD TFT montato sull'apposito shield TFT pubblicato nel fascicolo di giugno scorso.
Vi abbiamo descritto l'hardware occorrente e il modo in cui è stato programmato per svolgere il compito affidatogli, dettagliando le librerie create allo scopo e facendo cenno alla decodifica JPEG, implementata nello

sketch e fondamentale per la visualizzazione delle immagini. Abbiamo anche spiegato che, per ragioni di occupazione di memoria, le immagini ricevute vengono decodificate al volo, spezzettate e inviate al display grafico una porzione alla volta, così da avere un buffer di dimensioni ridotte.
Spiegata la decodifica JPEG e come avviene nel sistema, approfondiamo la materia con un'utile disamina sui formati d'immagine e sulla compressione, in particolare quella JPEG.

UN PO' DI TEORIA

Per capire come mai il formato JPEG sia così efficiente ma anche piuttosto pesante da decodificare, vediamo di spiegarne il funzionamento con qualche schema a blocchi; iniziamo dalla conversione dello spazio colore e dalla segmentazione (**Fig. 1**).
L'immagine viene per prima cosa convertita dallo spazio colore RGB ad uno spazio colore differente, composto da luminanza (Y) e da due valori di crominanza (Cr e Cb), che corrispondono più o meno a tinta e saturazione colore.
Questo viene fatto perché l'occhio umano percepisce la luminosità ed i colori attraverso due recettori differenti (bastoncelli e coni); i bastoncelli risultano molto più sensibili, ma non sono in grado di differenziare i colori dell'immagine, ma solo i valori di luminosità, mentre i coni sono meno sensibili ma riescono a distinguere i colori. Ne consegue che l'occhio risulta molto più sensibile alle variazioni ed agli errori sulla luminosità di un pixel piuttosto che alle sue variazioni cromatiche.
Passando allo spazio di colore YCrCb separiamo, appunto, i valori di luminosità dalle componenti cromatiche, che possono seguire vie diverse nel percorso di compressione. Come vedremo, potremo infatti comprimere maggiormente o addirittura scartare alcuni valori di crominanza senza nessun effetto visibile o quasi.
L'immagine viene quindi spezzettata in blocchi da 8x8 pixel, sui quali verranno concentrate le successive operazioni di compressione (**Fig. 1**). Fin qui, infatti, l'immagine è inalterata; la conversione dello spazio colore è un procedimento reversibile senza perdita di qualità (salvo eventuali errori numerici).
Questi blocchi sono detti **MCU, Minimum Coded** Unit, ovvero le più piccole unità codificate.
Ogni blocco viene quindi processato per proprio conto, indipendentemente dagli altri.
Successivamente le componenti di luminanza e crominanza di un singolo blocco vengono separate ed iniziano a seguire vie diverse (**Fig. 2**).

TRASFORMATA DISCRETA DEL COSENO

Come abbiamo detto sopra, finora i pixel sono rimasti invariati: ogni tripletta di byte nelle nostre MCU rappresentano un pixel sullo schermo, poco importa che usiamo unità di misura differenti (RGB o YCrCb). Abbiamo sempre tre valori numerici per ogni pixel.
Se guardiamo un'immagine reale notiamo subito alcune caratteristiche, la prima delle quali è che normalmente l'immagine varia poco tra un pixel e quelli immediatamente adiacenti. È raro trovare grossi sbalzi di luminosità, e ancor meno di colore, tra due pixel "vicini".
Ma come possiamo sfruttare questa cosa? Ebbene, immaginiamo di convertire tutto il blocco di 8x8 pixel, la nostra MCU, facendo una media tra tutti i valori di luminanza e crominanza dei pixel. In pratica, facciamo un grosso "pixelone" mediando tutti i colori dei 64 pixel. Ovviamente otterremo un'immagine molto degradata ma quasi sempre riconoscibile. Se invece di creare un unico "pixelone" ne facciamo 4, ognuno ottenuto facendo la media di 4x4 = 16 pixel, otterremo già un risultato migliore.

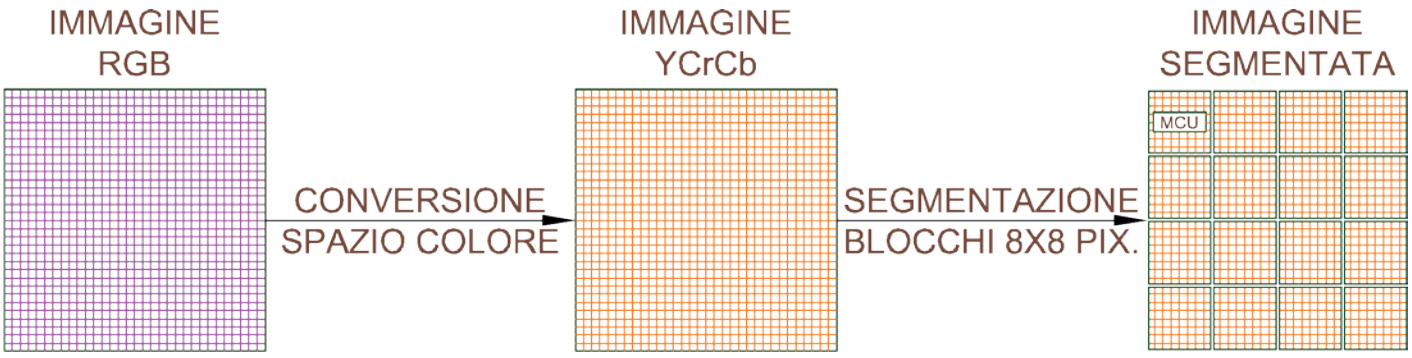


Fig. 1 *- Spazio colore e segmentazione.*

Così via, fino a scendere a 2x2 pixel mediati fino a 1 singolo pixel, che poi è l'immagine originale con la qualità inalterata. Ora, proviamo a procedere in questo modo:

- facciamo un immagine con il "pixelone" iniziale;
- ovvero la media di tutti i pixel;
- ne sovrapponiamo un'altra con 4 blocchi contenenti ciascuno le differenze tra la media dei 4x4=16 pixel di prima ed il "pixelone", e via di seguito.

In pratica, ricostruiamo l'immagine iniziale sovrapponendo varie gradazioni di "qualità". È intuitivo che, facendo bene le cose, si ottiene ancora l'immagine originale. Abbiamo quindi separato l'immagine in una sovrapposizione di immagini parziali, dalla più grossolana fino ai dettagli più fini.
In questa separazione, la prima immagine rappresenta il valore medio su tutti i 64 pixel, ovvero, in termini "elettronici", la componente "DC", ovvero la "corrente continua" del blocco.
Il secondo livello rappresenta le variazioni dei punti più distanti (stiamo lavorando su 4x4 pixel medi, ricordiamo) dalla media, quindi possiamo considerarla come una componente in "bassa frequenza" dal punto di vista elettronico.
Restringendo man mano, le varie sotto-immagini rappresentano dettagli sempre più fini nel blocco, quindi componenti a frequenza maggiore (le variazioni di intensità tra pixel sempre più vicini).
Abbiamo trasformato quindi l'immagine in una sovrapposizione di "onde" a frequenze diverse, ognuna delle quali rappresenta dettagli sempre più precisi dell'immagine del blocco.
Questo lavoro viene fatto tramite una trasformazione matematica dal dominio spaziale a quello delle frequenze. Questa trasformazione si chiama DCT (Trasformata Discreta del Coseno) ed è simile alla più nota trasformata di Fourier (FT, o FFT), e lavora su numeri reali e non su numeri complessi, come invece la FFT.
Sorvoliamo sulla trattazione completa, che è piuttosto complicata; basti sapere che i 64 valori del nostro blocco di pixel, dopo la trasformazione non rappresentano più i singoli punti, ma dei valori di "frequenza" via-via sempre più alta; il primo valore ("in alto a sinistra", nel blocco) per intenderci, rappresenta il valore DC, ovvero la media di tutti i 64 pixel; i successivi rappresentano delle variazioni sempre più dettagliate. La **Fig. 3** può dare l'idea di come funziona la cosa.
Come si nota, il punto in alto a sinistra rappresenta la parte più grossolana della MCU, mentre spostandosi man-mano verso destra e verso il basso si scende sempre più in dettaglio. Sovrapponendo tutte le immagini otterremo ancora l'immagine originale.
Qualcuno si starà chiedendo: a che pro? Abbiamo sempre 64 valori, dai 64 di partenza, quindi dove sta la compressione? Semplice: ancora da nessuna parte! Invertendo la trasformazione otteniamo ancora l'immagine di partenza, senza alcuna perdita di qualità.
Ma, con una rappresentazione di questo tipo, le cose iniziano a farsi interessanti; per esempio, cosa succede se scartiamo qualche valore vicino all'angolo in basso a destra, e poi facciamo la trasformazione inversa? Semplicemente, otterremo un'immagine leggermente degradata. Quanto viene degradata dipende da quanti e quali valori scartiamo. Quindi se, per esempio, dei 64 valori ne scartiamo bru-

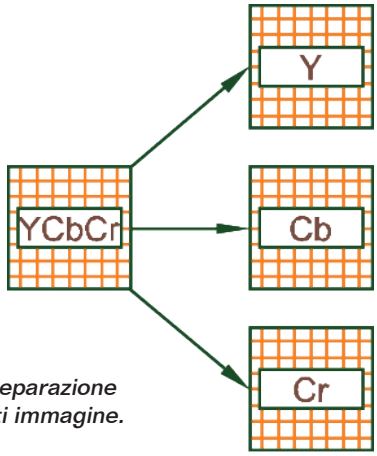


***Fig. 2** - Separazione componenti immagine.*

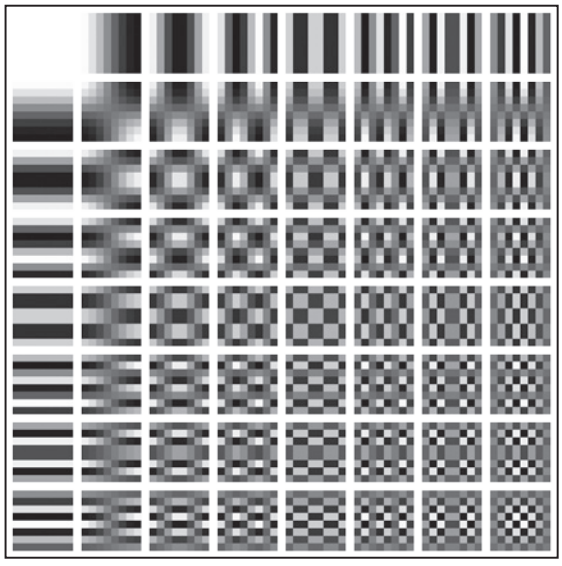


***Fig. 3** - Variazioni di grigio.*

talmente 32, otterremo un'immagine meno nitida (ricordatevi che i valori più vicini all'angolo in basso a destra rappresentano i dettagli più fini!), ma sempre riconoscibile.
Il "trucco" sta proprio qui: facendo una DCT, scartando alcuni valori e poi invertendo la trasformazione, otterremo un'immagine somigliante a quella di partenza, solo meno nitida; la perdita di nitidezza dipende da come scartiamo i valori.
In realtà i valori non vengono proprio scartati, ma quantizzati; per esempio, potremmo decidere che per i valori vicini all'angolo in basso a destra (i dettagli fini) invece di usare 1 byte possiamo rappresentarli con 1-2-3-4 bit, risparmiando memoria, utilizzando delle tabelle di conversione dette "tabelle di quantizzazione" (quantization tables).
Sono queste tabelle che influenzano in gran parte sia la perdita di qualità nella compressione sia la riduzione di dimensione del file.
Ok, abbiamo trasformato il nostro blocchetto di pixel (MCU) con una DCT, abbiamo ottenuto 64 valori numerici, li abbiamo "semplificati", ne abbiamo scartato qualcuno. Che ci rimane da fare, per comprimere ulteriormente il file? Ebbene, per prima cosa ricordiamo che le immagini reali variano lentamente, senza bruschi salti.
Quindi, ogni blocco avrà, per esempio, una luminosità poco diversa dal precedente. Invece di considerare la sua luminosità assoluta, quindi, possiamo salvare solo la differenza di luminosità rispetto al blocco precedente, che sarà presumibilmente un valore "piccolo", quindi codificabile con un numero minore di bit, come vedremo in seguito. Quindi, il valore del primo coefficiente, la componente DC, verrà trasformato nella differenza con il blocco precedente.
Questo procedimento si chiama DPCM (Differential Pulse Code Modulation) e viene utilizzato anche nella compressione di flussi audio. I restanti 63 valori saranno stati quantizzati, quindi ridotti in numero di bit. Man mano che ci avviciniamo all'angolo in basso a destra, questi valori conterranno un numero sempre maggiore di zeri ed il numero di zeri aumenterà con l'aumentare della grossolanità della quantizzazione.
Invece di memorizzare tutti gli zeri e gli uni come d'abitudine, converrà quindi contare gli zeri e memorizzarne il numero prima dei valori non nulli. Per esempio, se abbiamo i seguenti valori:

0 0 0 235 0 0 0 0 0 0 0 0 128 0 0 0 0 0 0 0 0 0 12 (totale 23 byte)
possiamo memorizzarli in questo modo:

3 235 8 128 9 12 (totale 6 byte)

che significa 3 zeri, 235, 8 zeri, 128, 9 zeri, 12. Come vedete, abbiamo risparmiato un bel po' di byte, grazie al fatto che molti numeri sono a zero!
Questo metodo, applicato ai soli 63 coefficienti AC, viene chiamato RLE (Run Length Encoding) e viene utilizzato anche nella compressione di file generici.

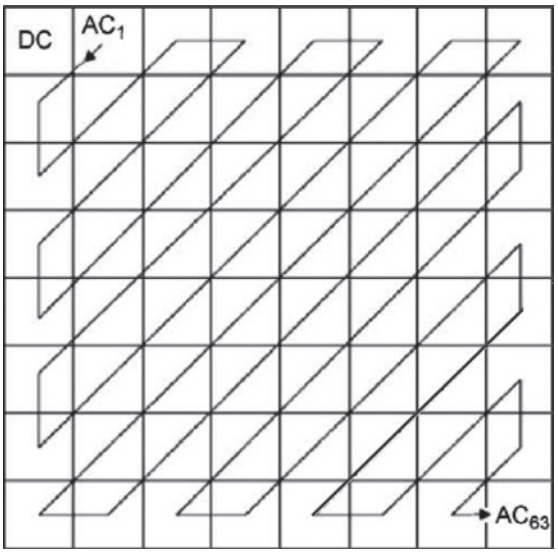


***Fig. 4** - Scansione a zig-zag.*

ZIGZAG SCAN

Abbiamo detto che nei 63 valori il "numero di zeri" aumenta avvicinandoci all'angolo in basso a destra; conviene quindi, prima di comprimerli con l'algoritmo RLE, ordinarli in modo da avere gli zeri il più possibile raggruppati, e precisamente in coda al blocco.
Per questo si utilizza il procedimento di "zigzag scan", visibile in **Fig. 4**.

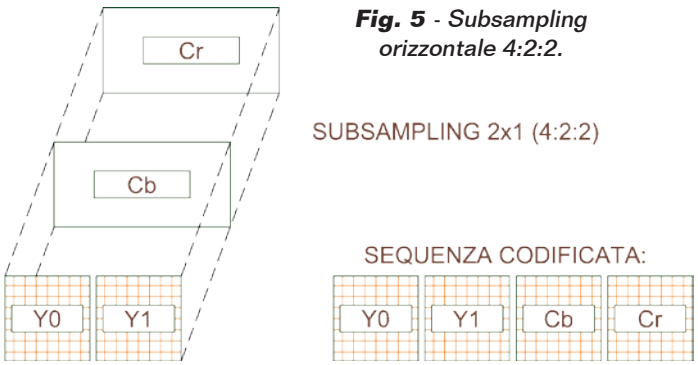


***Fig. 5** - Subsampling orizzontale 4:2:2.*

ALTRE POSSIBILITÀ DI COMPRESSIONE

Finora abbiamo sfruttato i 3 componenti colore allo stesso modo, anche se, probabilmente, con tabelle di quantizzazione diverse per luminosità e componenti cromatiche. Abbiamo inoltre "compresso" le componenti DC dei vari blocchi prendendone la differenza con i blocchi precedenti, e codificato le componenti AC dei blocchi in modo da sfruttare l'abbondanza di zeri presenti.
Finito? Ancora no! Possiamo fare altre due cose, per ridurre ulteriormente la dimensione del file; una con ulteriore piccola perdita di qualità, l'altra senza.

SUBSAMPLING

Abbiamo detto all'inizio che l'occhio umano è molto meno sensibile ai colori, ed alle variazioni dei medesimi, rispetto a quanto non sia sensibile alle variazioni di intensità luminosa.
Possiamo sfruttare ulteriormente questo fatto (anche se forse non era chiarissimo, tramite le tabelle di quantizzazione differenziate l'abbiamo già parzialmente fatto prima) raggruppando le componenti cromatiche di MCU adiacenti; per esempio, possiamo prendere 2 blocchi di luminosità ed un singolo blocco di crominanza (2 in realtà, visto che le componenti sono 2!) facendo la media delle crominanze di due MCU adiacenti. Questo può essere fatto in orizzontale o in verticale. Oppure possiamo addirittura mediare la crominanza su quattro blocchi (2x2) e quindi prendere quattro blocchi di luminanza per uno di crominanza.
Nella **Fig. 5** vedete la rappresentazione del subsampling 4:2:2 orizzontale.
Come si nota, invece di 6 blocchi di valori ne abbiamo presi solo 4, con un risparmio del 30%.
Se volessimo fare un subsampling più spinto, un 4:2:0 (2x2), avremmo la situazione in **Fig. 7**.
In questo caso, al posto di 12 blocchi di valori ne abbiamo presi 6, con un risparmio del 50%.
Il subsampling permette un notevole risparmio di dimensioni dell'immagine con perdite di qualità quasi sempre trascurabili.
La nostra libreria supporta quattro modalità di subsampling: 4:4:4 (no subsampling), 4:2:2 orizzontale, 4:2:2 verticale (mancava nella libreria originale, e si ha spesso in immagini ruotate), e 4:2:0.

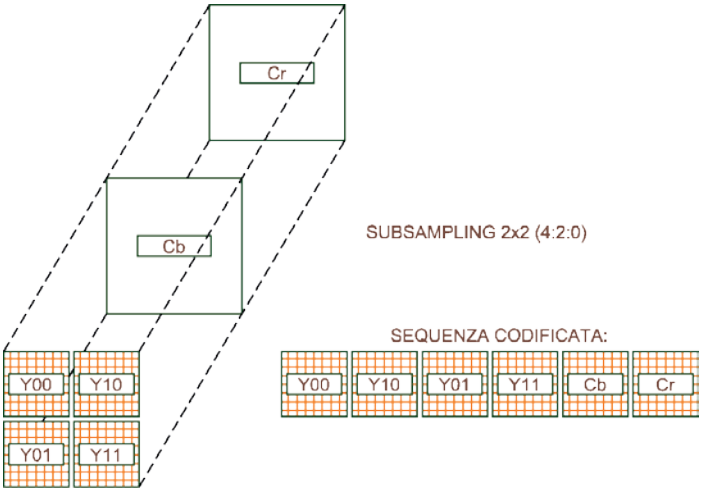


***Fig. 6** - Subsampling orizzontale 4:2:0.*

CODIFICA HUFFMAN

Per chi aveva sperato che fossimo giunti alla fine... brutte notizie! Resta l'ultimo livello di compressione presente nei file JPEG, che è una compressione cosiddetta lossless, ovvero senza perdita di informazioni: la codifica Huffman.
La codifica Huffman si basa sul fatto che i numeri che descrivono il mondo reale non sono proprio casuali, ma tendono ad avere una certa regolarità.
La codifica entropica (Entropy coding, in inglese) sfrutta questo fatto, utilizzando non più un codice a lunghezza fissa di byte, ma un codice a lunghezza variabile, assegnando ai numeri più frequenti dei codici più brevi.
Come funziona? Supponiamo di dover codificare un testo contenente le seguenti lettere:

G, O, P, H, E, R, S e <spazio>

Si tratta di 8 caratteri, quindi con una codifica normale (lunghezza costante di bit) servono 3 bit (23 = 8!!); possiamo fare la **Tabella 1.**
Se vogliamo codificare la frase "GO GO GOPHERS", scriviamo:

000 001 111 000 001 111 000 001 010 011 100 101 110

Utilizzando quindi 39 bit in totale. Se guardiamo bene la frase però vediamo che alcune lettere appa-

G	000
O	001
P	010
H	001
E	100
R	101
S	110
<SPAZIO>	111

Tabella 1

iono più frequentemente di altre; la G e la O appaiono 3 volte ciascuna, lo spazio 2 volte, mentre le altre lettere solo una volta. Come possiamo sfruttare questa regolarità? Scegliamo un codice differente, questa volta a lunghezza variabile, utilizzando un minor numero di bit per i simboli più frequenti (**Tabella 2**).
Con questa codifica la nostra frase viene rappresentata da:

10 11 001 10 11 001 10 11 0100 0101 0110 0111 000

Ovvero 37 bit in totale, con un risparmio di 2/39, ovvero di circa un ventesimo. Ovviamente l'esempio fatto è piccolo, quindi lo è anche il risparmio. In casi reali il risparmio è notevole.
Ma come si fa a codificare e decodificare una serie di dati in questo modo?
La cosa non è semplicissima! Per prima cosa occorre, ovviamente, stabilire una "scaletta" dei numeri che appaiono più di frequente, in modo da poter assegnare loro codici più brevi. In alcuni casi la frequenza è già nota, come per esempio per dei testi in una specifica lingua, dove esistono statistiche ben precise; in altri casi occorre analizzare i dati, o per lo meno una quantità sufficiente di dati, per creare la tabella.
Questo è uno degli svantaggi della codifica entropica: serve un'analisi preventiva dei dati, e questa può essere lenta. Successivamente serve un algoritmo che permetta di trovare le sequenze ottimali di bit per i miei dati, ed è questo l'algoritmo di Huffman, che è piuttosto complesso e su cui sorvoliamo.
Basti sapere che si basa su alberi binari, ed è in grado di trovare una soluzione ottimale al problema, tenendo ovviamente conto che i dati vanno poi anche decodificati.
Per esempio, nel nostro caso abbiamo assegnato alla G il valore 10; ovviamente NESSUN altro carattere, anche se con sequenza più lunga, potrà iniziare con 10, altrimenti non sapremmo come decodificarlo! Lo stesso vale per la O (11). Come potete vedere dalla tabella, infatti, nessun altro carattere inizia né con 10 né con 11; la S e lo <spazio> iniziano, per esempio, con 00 e finiscono rispettivamente con 0 e con 1, ottenendo i 2 codici 000 e 001. Anche in questo caso, NESSUN altro carattere dovrà iniziare con 000 o con 001, ed infatti i rimanenti iniziano tutti con 01 (prefisso libero!) e sono obbligati ad utilizzare 4 bit ciascuno.
Quando andremo a rileggere i bit per ricostruire le nostre lettere potremo quindi farlo univocamente: se leggiamo 10 sappiamo con certezza che si tratta di una G; se leggiamo 00 sappiamo che ci serve un altro bit per ricavare il carattere (S o spazio), mentre se leggiamo 01 sappiamo che ci servono altri 2 bit per capire di cosa si tratta.
L'altro svantaggio della codifica Huffman (ed in genere di tutte le codifiche a lunghezza variabile) è che occorre lavorare a livello di Bit, estraendo dal flusso bit per bit ed analizzandolo, cosa in cui gli elaboratori non sono particolarmente efficienti. Inoltre, se ci servisse, per esempio, solo il terzo carattere del testo, saremo comunque obbligati a leggere anche i 2 precedenti, visto che è l'unico modo che abbiamo per sapere dove inizia quello che ci interessa.
È quest'ultimo il più grosso svantaggio della codifica JPEG che ci impedisce, per esempio, di saltare parti di immagine che non ci interessano. Se di un'immagine da 2000x2000 pixel vogliamo estrarre un quadratino di 100x100 pixel saremo comunque obbligati a leggere almeno TUTTI i pixel che precedono quelli che ci interessano, e la cosa la potete vedere utilizzando la nostra applicazione su immagini molto grandi.

G	10
O	11
P	0100
H	0101
E	0110
R	0111
S	000
<SPAZIO>	001

Tabella 2

I JPEG PROGRESSIVI

Per "facilitare" la visualizzazione sui browser di immagini molto grandi trasferite via internet, soprattutto con connessioni lente com'erano le prime disponibili, qualcuno ha pensato bene di estendere il formato iniziale con una modalità progressiva.
Di cosa si tratta? Semplicemente, un formato che "descrive" l'immagine con livelli di qualità bassi ad inizio file, migliorandoli successivamente man mano che si avanza nella lettura del medesimo.
Il vantaggio, innegabile, è che il browser è in grado di visualizzare un'immagine approssimativa anche solo leggendo una piccola parte dei dati disponibili, cosa che può far piacere a chi sta visualizzando un sito particolarmente lento.
È quasi sempre preferibile vedere delle immagini sfuocate che migliorano pian piano piuttosto che

attendere secondi/decine di secondi senza vedere nulla.
Il grosso svantaggio è che quel tipo di file per poter essere decodificato a velocità accettabili richiede la memorizzazione dell'intera immagine durante la lettura. Quindi, se sto scaricando un'immagine da 1000x1000 pixel, mi ritrovo a dover memorizzare 3 megabyte di dati durante la ricezione. Per un Personal Computer moderno si tratta di un valore tranquillamente accettabile, ma non per un microcontroller, ovviamente.
L'alternativa possibile è quella di rileggere continuamente il file più volte per ogni blocco (MCU) da decodificare. La cosa è possibile, anche se molto lenta, se il file è per esempio su una scheda SD locale; diventa improponibile se il file è remoto, visto che sarebbe necessario richiederlo decine/centinaia/migliaia di volte al server durante la decodifica.
Un'altra alternativa, praticabile ma estremamente macchinosa, è leggere una prima volta il file, creare una mappa dei dati su una SD (ovvero, l'inizio fisico di ogni MCU nelle sue versioni a definizioni migliorate), e poi rileggerlo in modalità random (saltellando qua e la) in modo da poter decodificare completamente ogni MCU per poterla visualizzare. Quest'alternativa è certamente praticabile in caso di file locali, un po' meno per file remoti, per poter leggere i quali in modalità random servono apposite estensioni HTTP non sempre disponibili.
L'ultima alternativa possibile su un microcontroller è utilizzare la scheda SD come un'estensione della RAM dove parcheggiare i dati parziali dell'immagine decodificata. Sembrerebbe una buona soluzione, se non fosse che le SD sono molto più lente della RAM e, soprattutto, hanno un limite di scritture dopo il quale si guastano.
Viste le problematiche e la scarsa diffusione dei JPEG in formato progressivo abbiamo deciso di non supportarli nella nostra libreria, come peraltro nella maggior parte di librerie disponibili sui microcontroller.

JPEG E JFIF

JPEG è l'acronimo per la tipologia di compressione dei dati-immagine; lo standard non descrive il formato di file che contiene le immagini JPEG ma solo il modo di comprimerle, che abbiamo visto in dettaglio nei paragrafi precedenti. Del formato dei file si occupa lo standard JFIF (JPEG File Interchange Format), che specifica come dev'essere composto un file .jpg.
Il file è composto da vari record che contengono le dimensioni dell'immagine, la risoluzione, una serie di byte necessari per riconoscere il file, alcuni dati opzionali ed, infine, l'immagine vera e propria. Fornire ulteriori dettagli risulterebbe lungo e noioso, tenendo conto che questi possono essere trovati facilmente in rete. La nostra libreria è comunque in grado di interpretarli correttamente ed estrarne le informazioni necessarie.

LA DECODIFICA DEI FILE JPEG

Ed eccoci arrivati (quasi) alla fine!
Per decodificare un file JPEG occorre procedere a ritroso rispetto a quanto elencato in precedenza, ovvero:

- Leggere le caratteristiche dell'immagine dagli appositi campi nel file; in particolare servono il numero di pixel, il tipo di subsampling e la profondità di colore (i JPEG supportano anche una modalità monocromatica, da noi non gestita, con 256 livelli di grigio, un byte per pixel).
- Controllare che il file sia di tipo supportato; come detto, non gestiamo la modalità monocromatica né i JPEG progressivi
- Leggere le tabelle usate nella codifica Huffman ed utilizzarle per ripristinare i dati compressi, MCU per MCU
- Invertire il processo DPCM sui valori DC delle MCU (tradotto : il primo valore di ogni MCU, la componente media di luminanza e crominanza, è memorizzato come differenza con la MCU precedente, e la cosa va invertita!)
- De-comprimere i restanti 63 valori delle MCU, che erano stati compressi con l'algoritmo RLE
- Eseguire la trasformata dei coseni inversa (IDCT) per ripristinare i valori YCbCr delle MCU, ovvero i valori di luminanza e crominanza dei singoli pixel
- Eventualmente integrare i valori mancanti di crominanza, se si è scelta una compressione con subsampling

***Fig. 7** - Rotazione d'immagine.*

- Riconvertire il tutto dal formato YCbCr nel formato RGB
- Convertirlo nel formato in uscita, se diverso da RGB (ad esempio nel formato a 16 bit 565)
- Inviare la MCU al display.

ROTAZIONE E SCALING

Solitamente, in un PC, rotazione e scaling sono indipendenti dalla decompressione JPEG; prima l'immagine viene decompressa in memoria e successivamente viene scalata/ruotata per adattarla al display. Come visto in precedenza questo non è possibile farlo su un microcontroller, che non permette la memorizzazione dell'intera immagine decompressa.
Come fare, quindi? Un'immagine fuori scala serve a poco: se abbiamo un'immagine da 1000x1000 pixel e la visualizziamo su un display come il nostro da 240x320 pixel ne vedremo solo una piccola, insignificante parte.
La stessa cosa per la rotazione: se l'immagine è più larga che alta, e la visualizziamo sul nostro display messo "in piedi" lo sfrutteremo male.
La soluzione anche qui è operare in modalità streaming, quindi eseguire scala e rotazione "al volo" man mano che arrivano i dati.

ROTAZIONE

Iniziamo dalla rotazione che, se fatta per multipli di 90 gradi, è concettualmente molto semplice (**Fig. 8**). Qui è rappresentata una MCU di 4 pixel, per semplicità, posizionata in alto a sinistra sull'immagine. Come si nota, occorre ruotare le MCU attorno a se stesse e spostarle sul display. In questo caso la rotazione avviene cambiando la sequenza dei pixel da 1-2-3-4 a 3-1-4-2, mentre l'origine della MCU sull'immagine va traslata per portarla alla posizione desiderata.
Si tratta, in sintesi, di modificare l'ordine dei byte nel buffer contenente la MCU e nel ridefinirne la sua posizione sul display, in base all'angolo di rotazione. La modifica dell'ordine dei byte nella MCU dipende solo dal tipo di rotazione, mentre lo spostamento sull'immagine dipende sia dal tipo di rotazione che dalla posizione originale della MCU.

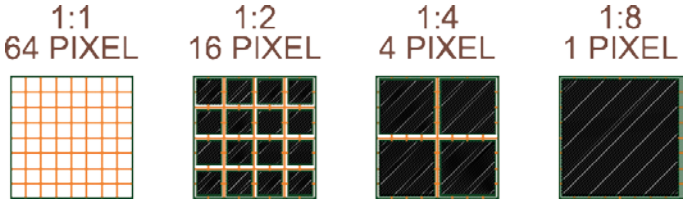


***Fig. 8** - Scaling d'immagine.*

SCALING

Lo scaling, o ridimensionamento dell'immagine risulta decisamente più complesso. Innanzitutto, visto che le immagini solitamente disponibili hanno dimensioni più grandi del nostro display, evitiamo l'ingrandimento, che comporta particolari problematiche che vedremo più avanti.
Ci limiteremo quindi solo alla riduzione dell'immagine. Vista la particolarità della compressione JPEG, ovvero il fatto che è realizzata per unità minime di immagine (MCU) composte da 8x8 pixel, una prima idea è quella di scalarle per multipli di 2, con un massimo di 1/8. Questo può avvenire semplicemente raggruppando i pixel nella MCU a 2 a 2, 4 a 4 oppure tutti e 8, facendone la media di intensità e colori (**Fig. 8**).
Questo tipo di ridimensionamento risulta semplificato, appunto, dal fatto che la MCU ha come dimensioni una potenza di 2; il vantaggio ulteriore scalando sui pixel della MCU è che si fanno le medie dei valori senza "buttar via" nulla, e l'immagine risultante è senza difetti palesi.
Chiaramente, a meno di non avere una fortuna incredibile, è difficilissimo che l'immagine originale, scalata di una potenza di 2, si adatti perfettamente al nostro display.
Se scalando in questo modo ottenessimo anche solo una dimensione molto vicina a quella del display, potremmo fermarci ed utilizzarla, al prezzo di un minor sfruttamento dello schermo oppure al leggero ritaglio delle parti perimetrali dell'immagine, ma anche qui si tratterebbe di casi piuttosto rari.
Poiché vogliamo ottenere la perfezione (!) procediamo quindi nel modo seguente :

- utilizziamo per prima cosa la scala grossolana per multipli di 2, avvicinandoci il più possibile IN ECCESSO alla dimensione voluta. Ovvero, dividiamo x 2 i lati dell'immagine il più possibile fino ad ottenere un'immagine vicina a quella desiderata, ma più grande, NON più piccola. Ovviamente il massimo del fattore di scala è 8, lavorando sulle MCU da 8x8 pixel
- Ottenuta la scalatura grezza, utilizziamo un algoritmo per eliminare righe e colonne "ogni tanto", in modo da ottenere la dimensione esatta con il minor degrado di qualità possibile.

Ma come facciamo a decidere quali righe/colonne eliminare ?
Se l'immagine da ottenere fosse un sottomultiplo "semplice" di quella originale (esempio, 2/3),

ovvero con un **minimo comune multiplo** piccolo rispetto alla dimensione dell'immagine, basterebbe saltare una riga/colonna ogni numero intero di righe colonne; nel caso dell'esempio, 1 ogni 3.
Il sistema funziona se la frazione ha nominatore e denominatore piccoli, come in questo caso. Se avessimo un rapporto del tipo 1245/1346 le cose si complicherebbero.
Anche qui, purtroppo, è meglio non contare sulla fortuna! Solitamente dimensione di partenza e finale sono tutt'altro che multipli semplici, quindi il sistema non va bene!

L'ALGORITMO DI BRESENHAM

Chi mastica un po' di grafica applicata agli elaboratori conoscerà questo nome, piuttosto famoso, e probabilmente si starà chiedendo "Ma Bresenham non serve per disegnare linee???"
Si, si utilizza proprio per quello, e non solo per disegnare linee ma anche altre curve geometriche.
Che ce ne facciamo quindi nel ridimensionamento delle immagini? Per capirlo, guardiamo la **Fig. 10**, che rappresenta un segmento di retta, disegnato con l'accortezza che l'angolo tra questo e l'asse X sia minore di 45 gradi.
Se consideriamo l'asse X come il lato dell'immagine originale, e l'asse Y come il lato dell'immagine scalata, vediamo che il segmento in rosso rappresenta la corrispondenza tra i pixel della prima con quelli della seconda; partendo dall'origine (corrispondente all'estremo sinistro di entrambe le immagini), spostandomi verso destra, ovvero lungo i pixel dell'immagine originale, in verticale otteniamo i pixel dell'immagine scalata. Alla fine, portandoci sulla larghezza totale dell'immagine originale, in verticale otteniamo la dimensione corrispondente scalata.

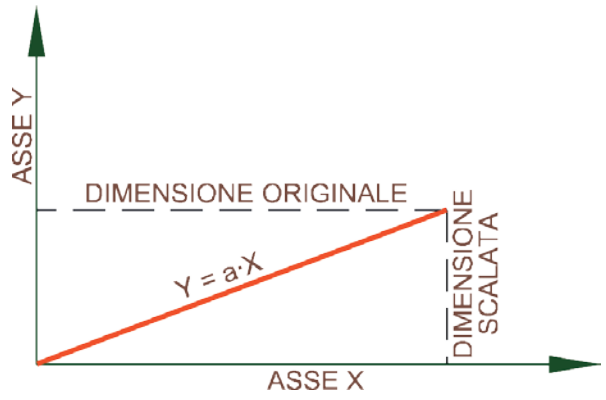


***Fig. 10** - Applicazione dell'algoritmo di Bresenham.*

Il problema è quindi risolvibile più o meno allo stesso modo di quello consistente nel disegnare un segmento di retta!
Facciamo un esempio pratico: diciamo che l'immagine di partenza è larga 1234 pixel, mentre quella di arrivo deve stare in 240 pixel (il nostro display); l'equazione della retta diventa:

$$Y = \frac{240}{1234} \cdot X$$

Quindi, per esempio, il pixel dell'immagine originale 790 corrisponde, nell'immagine scalata a:

$$Y = \frac{240}{1234} \cdot 790 = 153.65$$

Come vedete otteniamo, ovviamente, un numero non intero. 153.65 corrisponde al pixel numero 153 o 154? Intuitivamente arrotondiamo all'intero più vicino, quindi 154.
Il problema è che siamo obbligati a lavorare con numeri decimali, cosa molto inefficiente, tenendo conto che dobbiamo eseguire queste operazione per un numero di pixel spesso molto grande.
Lavorando solo con numeri interi otteniamo sempre il numero in difetto, 153 in questo caso, cosa che da alla linea un'apparenza "segmentata", e lo stesso vale se utilizziamo il sistema per scalare la nostra immagine. Risultano inoltre necessarie una moltiplicazione ed una divisione su numeri interi oppure una moltiplicazione per un numero in floating point

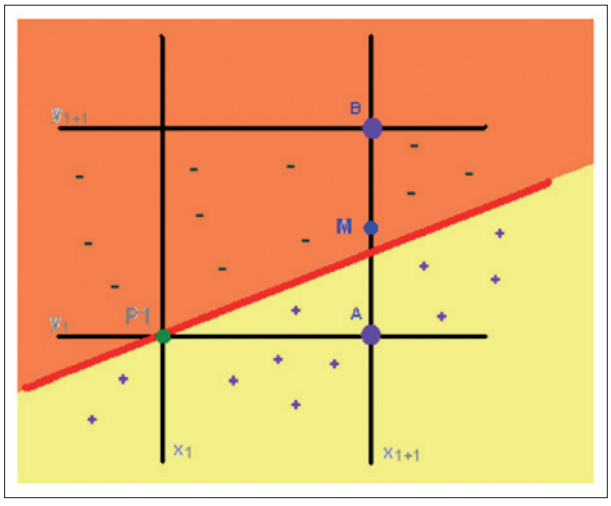


***Fig. 10** - Rappresentazione dell'equazione.*

e la riconversione in numero intero, operazioni che sono piuttosto lente. L'algoritmo di Bresenham risolve brillantemente il problema, riducendo inoltre il numero di calcoli da fare. Per capire come funziona, riscriviamo l'equazione della retta precedente:

$$Y = \frac{D}{O} \cdot X$$

Dove 'D' è la dimensione dell'immagine di destinazione e 'O' è la dimensione dell'immagine originale. La possiamo riscrivere in questo modo:

$$O \cdot Y - D \cdot X = 0$$

Quest'equazione rappresenta una retta su un piano; tutti i punti della retta la soddisfano (danno risultato 0!), mentre i punti fuori dalla retta non la soddisfano, ma danno un risultato minore o maggiore di zero a seconda che stiano SOTTO o SOPRA alla retta (**Fig. 10**).
Immaginiamo di partire dall'origine (0, 0), che soddisfa l'equazione. Ora avanziamo lungo l'asse X (ovvero prendiamo il pixel successivo sull'immagine originale), dobbiamo decidere se avanzare al pixel successivo anche sull'immagine scalata oppure no.
Per questo, guardiamo ancora la figura sopra, dove vediamo il punto A che sta sotto alla retta il punto B che ci sta sopra, ed il punto M che sta a metà tra i due. Chiaramente, se la retta passa sotto al punto M significa che il punto Y è più vicino a A, quindi NON occorre passare al successivo (B); se invece la retta passasse sopra al punto M vorrebbe dire che è più vicina al punto B, quindi occorrerebbe scegliere quello. Per decidere quale prendere inseriamo la posizione del punto M nella nostra equazione; in questo caso il punto M è a metà tra A e B, ovvero tra 0 e 1:

$$delta = O \cdot \frac{1}{2} - D \cdot 1$$

se delta è minore di zero, il punto medio sta SOTTO alla retta, quindi questa è più vicina a B; altrimenti sta SOPRA alla retta che risulta quindi più vicina ad A. Il valore 'delta' ci permette quindi di decidere, in base al suo segno, se passare o meno al prossimo pixel nell'immagine di destinazione.
Passando ai punti successivi, più in generale, possiamo scrivere la nostra equazione come:

$$delta = O \cdot \left(\frac{1}{2} + Y_i\right) - D \cdot X_i$$

Moltiplicando il tutto per 2 (in modo da evitare divisioni e numeri in floating point):

$$2 \cdot delta_i = O \cdot (1 + 2 \cdot Y_i) - D \cdot 2 \cdot X_i$$
$$2 \cdot delta_i = O + 2 \cdot O \cdot Y_i - 2 \cdot D \cdot X_i$$

Ora, immaginiamo di essere arrivati al punto $P_i(X_i,Y_i)$, del quale conosciamo entrambe le coordinate; vogliamo trovare il punto successivo $P_{i+1}(X_{i+1}, Y_{i+1})$, del quale conosciamo SOLO la coordinata X, visto che avanziamo pixel per pixel sull'immagine originale: $X_{i+1} = X_i + 1$; scriviamo quindi la nostra delta per il punto P_{i+1}:

$$2 \cdot delta_{i+1} = O + 2 \cdot O \cdot Y_{i+1} - 2 \cdot D \cdot X_{i+1}$$

e, poiché è $X_{i+1} = Xi + 1$

$$2 \cdot delta_{i+1} = O + 2 \cdot O \cdot Y_{i+1} - 2 \cdot D \cdot X_i - 2 \cdot D$$

Ora, Y_{i+1} l'abbiamo ricavato in base al valore precedente di delta ($delta_i$), e può assumere solo due valori:

$Y_{i+1} = Y_i$	SE $delta_i$ era >= 0
$Y_{i+1} = Y_i + 1$	SE $delta_i$ era < 0

Quindi anche il prossimo valore di delta ($delta_{i+1}$) lo possiamo ricavare dal precedente, secondo questa relazione:

$$2 \cdot delta_{i+1} - 2 \cdot delta_i = 2 \cdot O \cdot (Y_{i+1} - Y_i) - 2 \cdot D$$

Ottenuta sottraendo le due equazioni precedenti, quindi:

$$2 \cdot delta_{i+1} = 2 \cdot delta_i + 2 \cdot O \cdot (Y_{i+1} - Y_i) - 2 \cdot D$$

Tabella 3

Posizione originale	Delta corrente	Copia pixel	Posizione ridotta corrente	Delta successivo	Posizione ridotta successiva
0	0	SI	0	3	1
1	3	NO	1	-4	1
2	-4	SI	1	-1	2
3	-1	SI	2	2	3
4	2	NO	3	-5	3
5	-5	SI	3	-2	4
6	-2	SI	4	1	5
7	1	NO	5	-6	5
8	-6	SI	5	-3	6
9	-3	SI	6	0	7

Visto che $Y_{i+1} - Y_i$ può assumere solo 2 valori (1 o zero) a seconda che deltai sia positivo o negativo, abbiamo:

$$2 \cdot delta_{i+1} = 2 \cdot delta_i - 2 \cdot D \qquad \text{se delta}_i >= 0$$

$$2 \cdot delta_{i+1} = 2 \cdot delta_i + 2 \cdot O - 2 \cdot D \qquad \text{se delta}_i < 0$$

(ci siamo portati dietro il coefficiente 2 fino alla fine, anche se in questo caso potrebbe essere semplificato visto che la nostra retta passa per l'origine (0,0), cosa non sempre vera nell'uso dell'algoritmo, nel qual caso appare anche un valore costante nell'espressione). Abbiamo quindi trovato un modo per:

- scegliere se aumentare o meno il valore di Y in base al valore di delta CORRENTE
- trovare il valore di delta SUCCESSIVO che ci permette di proseguire

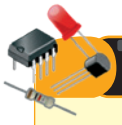
Possiamo quindi scrivere il nostro codice in questo modo, per "stringere" una riga di pixel:

```
int delta = 0;
int posizioneOriginale = 0;
int posizioneRidotta = 0;
while(posizioneOriginale < larghezzaOriginale)
{
  if(delta <= 0)
  {
        CopiaPixel(posizioneOriginale, posizioneRidotta);
        posizioneRidotta++;
        delta += larghezzaOriginale - larghezzaRidotta;
  }
  else
  {
        delta -= larghezzaRidotta:
  }
  posizioneOriginale++;
}
```

Per mostrarvi come si "muovono" le varie variabili, possiamo pensare ad una larghezza originale di 10 pixel ed una ridotta di 7 pixel, e vedere i vari passi dalla **Tabella 3**.
Si nota che vengono saltati 3 pixel su 10, per la precisione i numeri 1, 4 e 7, mentre ne vengono copiati esattamente 7, come quelli disponibili nella larghezza dell'immagine ridotta. I valori nelle colonne "delta" rappresentano una misura dell'errore tra la posizione precisa dei pixel nell'immagine di destinazione e quella ottenuta; più piccoli sono e più ci si avvicina al teorico.
Nel nostro caso reale dovremo, ovviamente, ripetere il procedimento anche in un ciclo esterno, visto che dobbiamo ridimensionare sia in orizzontale che in verticale, ma il principio non cambia minimamente. L'algoritmo è in grado di saltare in modo ottimale le righe e le colonne dell'immagine originale per farla stare nello spazio di destinazione.
Ma questo è un algoritmo perfetto? No, perché gli algoritmi professionali non si limitano a togliere dei pixel, ma "aggiustano" anche quelli a fianco di quelli tolti, in modo da eliminare eventuali artefatti nell'immagine; questo implicherebbe la memorizzazione di linee, cosa che non possiamo fare, e pesanti calcoli per determinare i valori dei pixel. Notare la differenza, in caso di riduzione dell'immagine, è comunque molto difficile: noi nelle nostre prove non ci siamo riusciti!
Ben diverso sarebbe il caso di ingrandimento dell'immagine, cosa che abbiamo appositamente evitato; anche in quel caso si potrebbe usare l'algoritmo di Bresenham per inserire, invece che togliere, righe e colonne di pixel. Il problema è che inserendole dobbiamo "inventarci" dei valori per i pixel aggiunti; se li prendiamo semplicemente dal pixel precedente vengono fuori degli artefatti nell'immagine, che, in questo caso, si notano molto facilmente. Servirebbe quindi un'interpolazione tra pixel precedente e successivo, in modo da avere una transazione di colore più progressiva, tuttavia, come abbiamo accennato prima, la cosa è complicata e richiede la memorizzazione di parti consistenti dell'immagine. E non possiamo permettercelo.

CONCLUSIONI

Bene, si conslude qui la nostra esposizione dell'applicazione PhotoFish, sperando che la pur pesante trattazione sulla gestione delle immagini e sui formati di compressione vi sia stata utile e che chi vorrà potrà applicare poficuamente le nozioni.



per il MATERIALE

Tutti i componenti utilizzati in questo progetto sono di facile reperibilità. Il master del circuito stampato può es… della rivista così come il f… program-mare il microcon… ensore ad infrarossi IR38D… entre l'integrato Microchip MCP3905A è disponibile 4,20 Euro.

Sostituire testo

Il materiale va richiesto a:
Futura Elettronica, Via Adige 11, 21013 Gallarate (VA)
Tel: 0331-799775 · http://www.futurashop.it