POLITECNICO DI TORINO

Collegio di Collegio di Ingegneria Informatica, del Cinema e Meccatronica

Corso di Laurea Magistrale
in Ingegneria del Cinema e dei Mezzi di Comunicazione

Tesi di Laurea Magistrale

Progetto e implementazione di un sistema di controllo per plotter verticale

Relatore

prof. Giovanni Malnati

Tutor Aziendale

dott. ing. Andrea Bulgarelli

Candidato

Danilo Ronchi

Aprile 2019

Ringraziamenti

Questo lavoro sarebbe stato impossibile da portare a termine senza avere le persone giuste al mio fianco. Innanzitutto, vorrei ringraziare il mio relatore per avermi dato la libertà di scegliere questo progetto, fidandosi di me e delle mie capacità.

È doveroso ringraziare chi con me ha lavorato attivamente alla realizzazione di un oggetto unico, che ci ha dato tante ansie ma tantissime soddisfazioni: tutto il team di Scribit. Ho appreso molto nel corso di questi mesi grazie all'impegno e alla presenza costante dei miei colleghi, anche nei momenti difficili. Un ringraziamento particolare va ad Andrea Bulgarelli, che mi ha seguito sin dall'inizio fornendomi sempre tutto il supporto necessario, costituendo per me un vero esempio di costanza. Ringrazio il maestro Ivan Lunardi, per aver creduto in me facendomi da guida e insegnandomi moltissimo. Ringrazio anche Andrea Baldereschi e Davide Marazita, per le opportunità di crescita e la fiducia che mi hanno dato. Inoltre, vorrei ringraziare i due colleghi con cui ho più lavorato a stretto contatto e con ritmi serratissimi, i mitici Nour Saeed e Piergiuseppe Zolfo: senza il loro contributo e i nostri caffè, molti dei risultati ottenuti sarebbero ancora cosa ignota.

Ringrazio i miei amici storici, senza i quali non sarei neanche riuscito a decidere di partire da Catania per intraprendere questa avventura. Sono loro i fari che non mi hanno mai abbandonato e con cui spero un giorno di recuperare tutto il tempo "perduto".

Ringrazio le fantastiche persone conosciute in questa città: i membri di Studio Stage, amici veri, che mi hanno insegnato quanto sia edificante lavorare insieme ai progetti più disparati, accademici e non, divertendosi e realizzando sempre tutto al meglio delle possibilità e con il massimo dell'impegno; e i miei coinquilini, che mi hanno sempre supportato, ma anche sopportato. Senza di loro, alla mia permanenza qui, e conseguentemente a me, sarebbe mancata più di qualcosa: praticamente tutto.

Ringrazio i miei colleghi e amici dell'Università di Catania, che mi hanno sempre stimolato a dare il meglio e a seguirli qui per creare insieme il nostro futuro.

Infine, non posso far altro che ringraziare la mia famiglia, fonte inesauribile di speranza e motivazione, senza la quale non avrei potuto inseguire i miei sogni.

A chi c'è, a chi non c'è, a chi avrebbe voluto esserci, a chi non c'è più: grazie.

SOMMARIO

1. Introduzione

Scribit (Fig.1) è il primo robot al mondo in grado di scrivere e cancellare su una qualunque parete verticale. Il progetto prende vita al MIT Senseable City Lab, grazie all'intuizione del direttore, ingegnere e architetto torinese Carlo Ratti e dell'arcihitetto Pietro Leoni; il progetto è stato finanziato su KickStarter e IndieGogo con una campagna di crowdfunding incubata da Makr Shakr, azienda creatrice del primo *robotic bar* al mondo, anch'esso idea di Carlo Ratti e CRA (Carlo Ratti Associati), guidata da Andrea Bulgarelli e Andrea Baldereschi, attualmente CTO e CMO della start-up. La campaga ha raccolto più di 2 milioni di dollari, determinando un grande coinvolgimento della comunità di *makers* in giro per il globo. L'interesse è da far risalire ai vari tentativi di creazione di un *vertical plotter* (stampante verticale) in grado di scrivere e cancellare un qualsiasi contenuto che non dipendesse dal formato dei file, né tantomeno dalla superficie: esistono già diversi modelli di plotter e vertical plotter facilmente realizzabili a partire da una scheda programmabile come le serie Arduino o ESP, ma i risultati non sono paragonabili all'efficacia e versatilità di Scribit, rendendolo la tecnologia allo stato dell'arte. La possibilità per un robot di poter disegnare su una parete verticale a prescindere dalle sue dimensioni trova applicazione nell'utilizzo di *immagini vettoriali* (Scalable Vector Graphics Format, che d'ora in avanti indicheremo come "SVG"), adattabili a qualunque risoluzione senza comprometterne la qualità (approfondiremo nel Cap. 2 di questo elaborato).

Figura 1 - Scribit

1.1 Obiettivo

All'interno del contesto di una start-up giovane, ospitata negli uffici di Makr Shakr, l'elaborato si propone di raccontare il nuovo paradigma lavorativo dell'industria creativa, della prototipazione e del design thinking, mostrando il funzionamento del robot con particolare attenzione al software e illustrando quindi gli algoritmi creati e testati a partire da un corposo lavoro di ricerca e sviluppo iniziato già nel 2012 da parte di studiosi e freelancer vicini a CRA, e portato avanti dal candidato a partire da Giugno 2018 insieme al team di Scribit. Verranno quindi illustrate le modalità di gestione delle immagini che, dopo essere state trattate ad hoc per l'applicazione e scalate alla "risoluzione" della parete, andranno tradotti in istruzioni NC (*Numerical Control*, a controllo numerico) da impartire al robot; esso è progettato su base Marlin, firmware open source tra i più utilizzati nel mondo delle stampanti 3D, e richiede appunto un listato di istruzioni NC, in formato GCode, per spostarsi sul piano e disegnare.

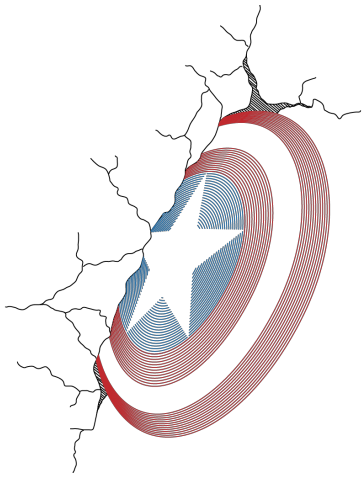

Figura 2a -Grafica in SVG

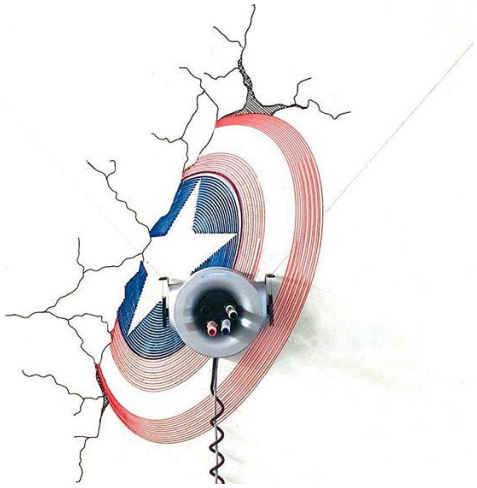

Figura 2b – Realizzazione di 2a con Scribit

1.2 Vertical Plotter

L'idea di *vertical plotter* esiste già da molto tempo: uno dei primi esperimenti in materia risale al 2001 ed è stato realizzato alla Cornell University dal team VP Squared **[1]**. Il progetto prevede l'utilizzo di una scheda programmabile che controlla due motori passo-passo (*stepper motors*) e un tastierino utilizzato per controllare il plotter, ossia un pennarello inserito al centro di una piccola struttura semi-piana di forma quadrata appesa ai motori tramite due cavi (Fig. 2).

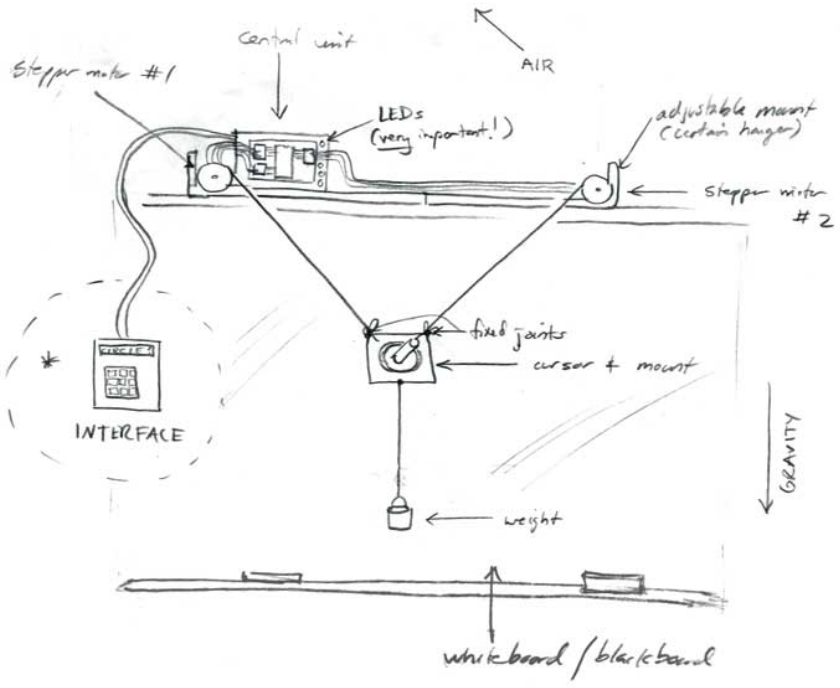

Figura 3 - Sketch del progetto VP Squared (© Cornell University)

L'idea è rivoluzionaria e apre le porte alla creazione di nuovi prototipi, implementazioni e installazioni artistiche, come nel caso del designer svizzero Jürg Lehni, Fig.3 **[2]**).

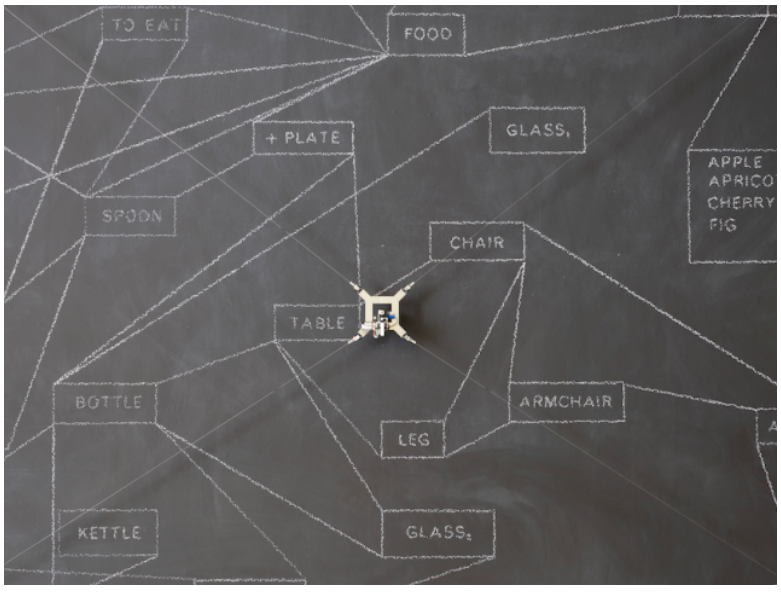

Figura 4 - Otto, il robot che disegna su blackboard. Utilizza 4 motori stepper fissati alle estremità della lavagna e un gessetto al centro del corpo metallico (© Jürg Lehni, Wilm Thoben)

Il web è oggi pieno di guide e *instructables* di ogni genere, che ricalcano l'impostazione generale di VP Squared, sostituendo al tastierino una scheda che possa svolgere calcoli e che

permetta al setup di comunicare con l'esterno e ricevere istruzioni più complesse e sequenziali (es. Raspberry Pi), o del software. Nel 2012 il primo vertical plotter di CRA presenta quest'impostazione, con piccole modifiche di design volute per la presentazione dell'OSArc Manifesto (Open Source Architecture) **[3]**.

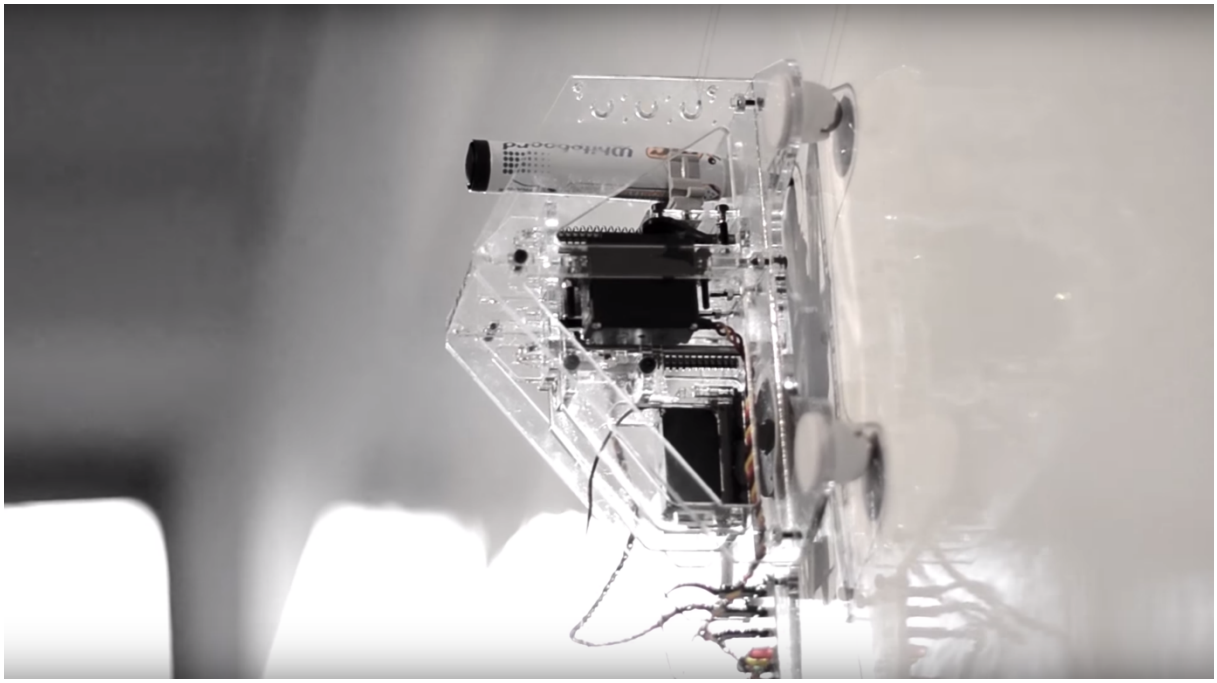

Figura 5 - Prototipo di openWall per la presentazione dell'OSArc Manifesto (© CRA)

Nel 2013 il prototipo si concretizza nel progetto open source "Open Wall", completo di impostazione meccanica, elettronica, software, schematici e istruzioni di montaggio disponibili al pubblico **[4]**; esso si presenta con una configurazione inedita montando i motori (passo-passo) all'interno dello chassis attaccati a due pulegge in cui viene avvolto un cavo, rendendo quindi necessaria la presenza di soli due chiodi agli estremi della parete a cui appendere i due capi. Inoltre, mette a disposizione, oltre al firmware per Arduino, un software scritto in Processing per convertire file vettoriali in istruzioni NC.

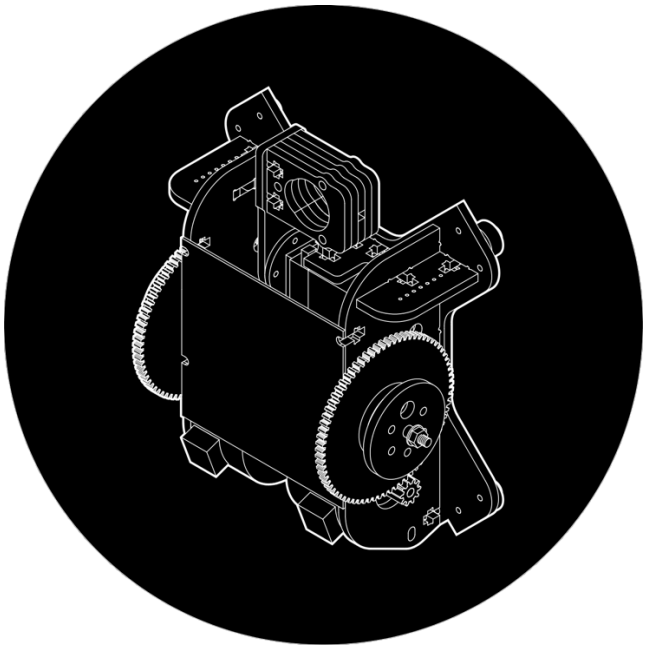

Figura 6 - openWall così come appare sul libretto di istruzioni (© CRA)

Nel corso degli anni Open Wall diventerà struttura portante di Scribit, modificando design, elettronica, meccaniche, e aggiungendo il supporto per più pennarelli, quindi più colori; il software rimane inizialmente identico a quello di Open Wall. L'idea è quella di creare un prodotto completo e user-friendly che permetta a qualsiasi utente di usufruirne con semplicità, senza ricorrere a strumenti di prototipazione per ricostruirlo, aprendo quindi non solo al target dei makers, ma includendo anche consumers standard insieme ai già entusiasti prosumers. Inoltre, grande parte delle installazioni e dimostrazioni di vertical plotters sono state eseguite su lavagne (bianche con scrittura a pennarelli, o nere con scrittura a gesso) o tele espositive: la seconda soluzione prevedeva che la tela, una volta stampata, fosse sempre visibile o sostituita per una nuova stampa; la prima soluzione permetteva invece di cancellare per poter riscrivere.

Scribit implementa per la prima volta un sistema di cancellazione automatizzata grazie alla presenza di un disco in ceramica che, all'aumentare della temperatura, è in grado di far evaporare uno speciale inchiostro a base acqua testato dall'azienda e utilizzato nei pennarelli forniti con il robot, aprendo le porte alla stampa verticale su praticamente qualunque parete stuccata di casa, uffici, negozi, senza rendere irreversibile il processo.

Il meccanismo utilizzato dal tamburo che contiene fino a 4 pennarelli è detto *desmodromico*, poiché basato sul meccanismo di controllo della corsa sia in apertura che in chiusura **[4]**; ovviamente, questo meccanismo non si occupa di gestire la combustione di carburante come

avviene nei motori *desmo*, piuttosto coadiuva contemporaneamente la rotazione del tamburo con l'inserimento e l'ingaggio dei pennarelli.
I meccanismi *desmodronico* e di cancellazione sono stati brevettati dall'azienda.

Figura 7 - Prima apparizione di Scribit durante la Design Week 2018 a Milano. Vengono realizzati disegni a tema floreale sulle vetrate del padiglione sulla Living Nature

1.3 Scribit

Come anticipato, Scribit si discosta parecchio dagli altri vertical plotter che è possibile creare con un po' di tempo, ingegno e abilità. Trattandosi del primo vertical plotter sul mercato, è stato realizzato sin dall'inizio con lo scopo preciso di essere usabile da chiunque e nel modo più semplice possibile, offrendo un'esperienza mai frustrante ma immediata e creativa.

1.3.1 Hardware e meccanica

Scribit è un robot di forma pressoché circolare: è basato su uno chassis in magnesio del diametro di 20cm, forato in basso per permettere al cavo d'alimentazione di uscire e oscillare, al centro per ospitare il tamburo e il suo motore dedicato, in basso nel lato opposto per far affiorare il dischetto di ceramica per la cancellazione, e bombato agli estremi destri e sinistri per far spazio agli altri due motori passo-passo interni (Fig. 8, 10).
Grazie a un foro applicato nei pressi dell'albero dei motori, essi fuoriescono dallo chassis e vengono inseriti nelle due pulegge: in ognuna di esse viene avvolto il cavo che avrà il capo

superiore legato a un chiodo a mo' di cappio, posto ognuna sull'estremo destro e sinistro della parete su cui appendere il plotter. Invece di arrivare direttamente al chiodo, il cavo passa prima da un **tendicavi** che ha la funzione di mantenere sempre retto il cavo quando viene avvolto e svolto per evitare di spezzarlo, ma anche e soprattutto per tenere Scribit quanto più possibile aderente alla parete per evitare che dondoli durante il movimento (Fig. 8).

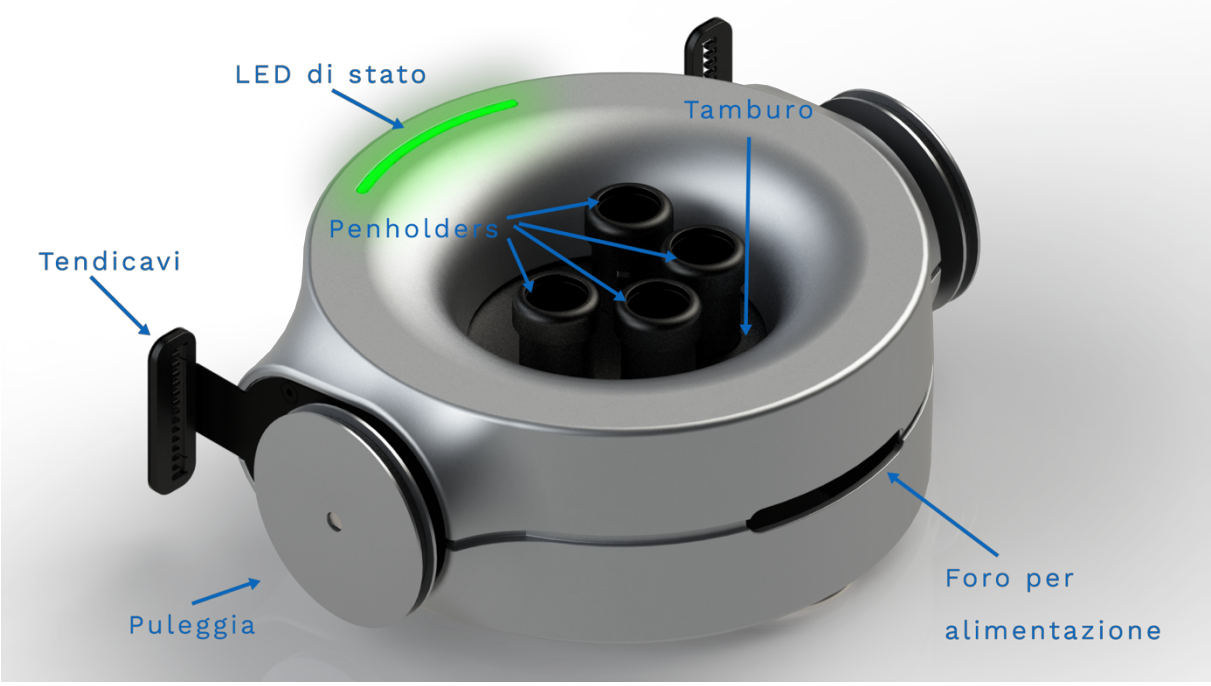

Figura 8 - Posizionamento degli elementi esterni di Scribit

Considerato il modo in cui il robot viene appeso, impostare un sistema di coordinate cartesiane con origine nel punto in alto a sinistra della parete, come avviene nella maggior parte degli standard relativi alle immagini (vengono rappresentate come matrici di pixel), risulta sconveniente per la mole di incognite che avrebbe il sistema di equazioni necessario al calcolo della posizione di esso all'interno della superficie: si è quindi deciso quindi di sfruttare la lunghezza dei fili sinistro e destro per definire la posizione del plotter secondo un sistema di coordinate polari in cui entrambi i chiodi sono **poli** e la lunghezza del filo rappresenta la **coordinata radiale** r di ognuno di essi (Fig. 9). In ogni caso, il chiodo di sinistra rimane l'origine del disegno che verrà stampato sulla parete per similitudine, dato che la grafica dovrà prendere le dimensioni esatte della superficie prima di essere stampata e il canvas, come detto, ha origine nel punto *top-left* di essa.

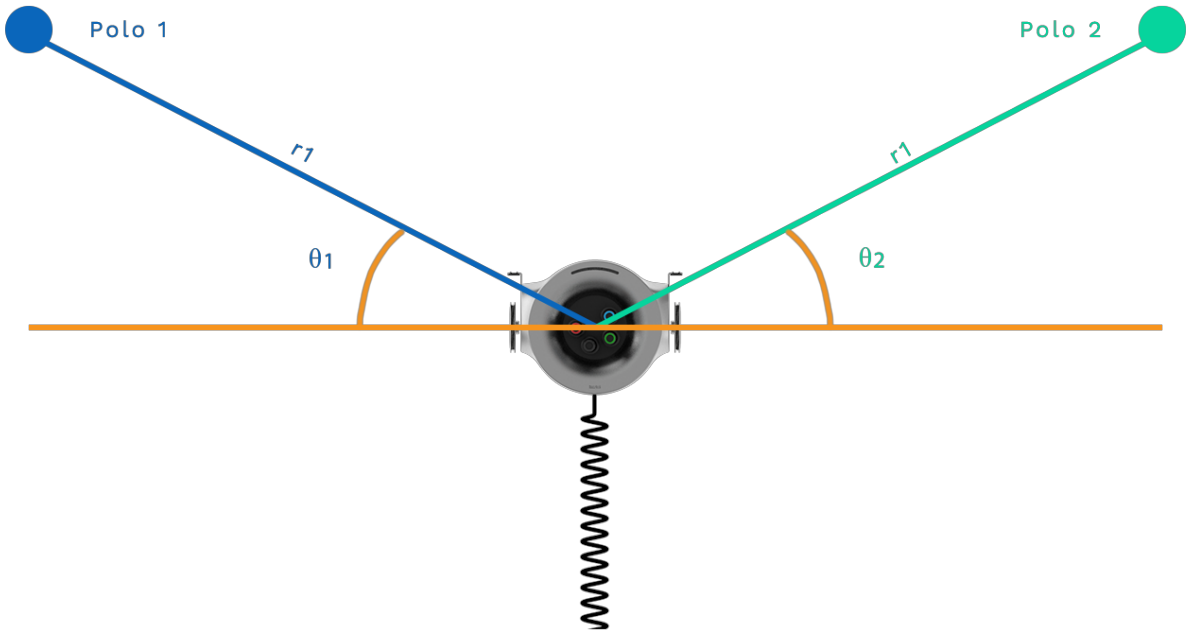

Figura 9 - Identificazione delle coordinate polari di Scribit

Al centro, il plotter è dominato dal tamburo (Fig. 8, 10). Esso espone 4 fori, uno per pennarello: questi possono essere inseriti direttamente se della giusta misura. In alternativa, è possibile svitare il *penholder* e sostituire l'*o-ring* al suo interno per ospitare un qualunque altro pennarello di misura inferiore. Il piano contenente i fori visibili dall'esterno è in realtà una ruota dentata: essa si comporta da **corona** in un sistema corona-pignone, in cui il **pignone** vero e proprio è attaccato all'albero del motore stepper centrale, che permette la rotazione e quindi il funzionamento del sistema *desmodromico.*

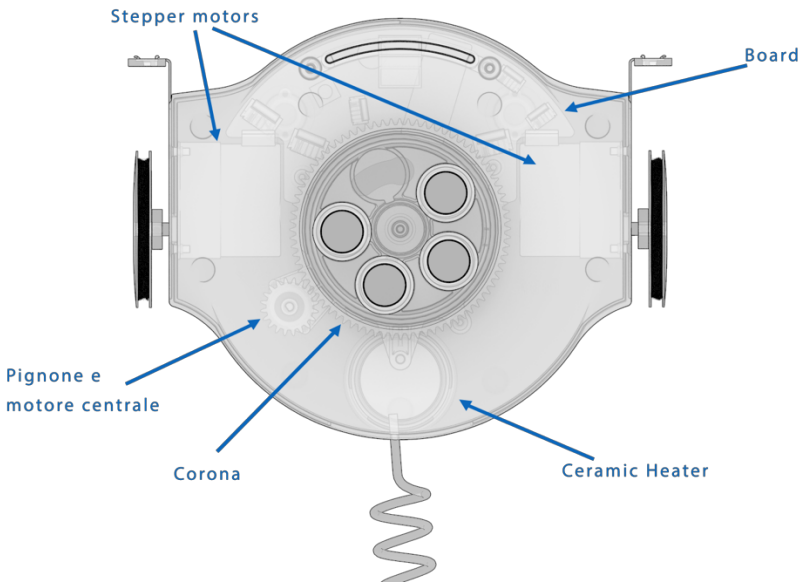

Figura 10 - Componenti di Scribit

Il *ceramic heater* viene esposto in basso al centro nel lato frontale (poggiato al muro, non visibile finché poggiato alla parete, Fig. 10): ha un diametro di 3 cm ed è incastonato in un cilindro composta da materiale isolante che sporge dal plotter di appena 2 mm per evitare che la ceramica tocchi direttamente il muro. Essa viene attivata solo durante la routine di cancellazione secondo regole ben precise. Raggiunge temperature molto elevate (~130 C°) per essere certi che cancelli l'inchiostro a prescindere dalle condizioni atmosferiche (umidità, parete molto fredda…). Al momento, l'unica parete su cui è possibile applicare tale metodo è una qualunque parete stuccata, che sia cartongesso, mattoni, pietra; è sconsigliato l'utilizzo su qualunque altro tipo di superficie. All'interno del cilindro viene inserito dell'isolante in modo da dissipare il colore, che non vada a riscaldare sia il corpo del plotter, che è conduttore, sia come misura di sicurezza per la scheda elettronica.

La scheda, per l'appunto, è installata all'interno del plotter, nella parte superiore di esso, lontana dalla ceramica. È stata realizzata apposta per l'applicazione, sia per forma che per composizione.

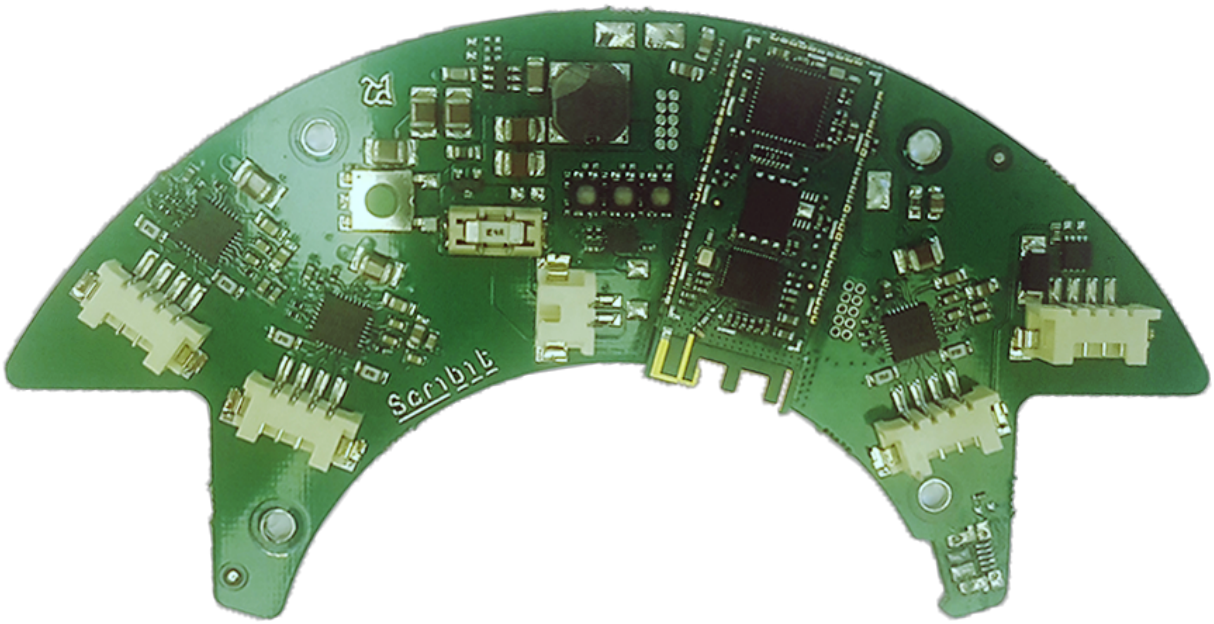

Figura 11 - Board elettronica di Scribit

Il cuore di essa è composto da due processori: un SAMD21, utilizzato esclusivamente per pilotare il plotter e tutti i suoi componenti attraverso firmware Marlin, e una ESP32, utilizzata per tutte le utility di connettività che illustreremo nel paragrafo successivo, dedicato al software. Unico aspetto hardware controllato dall'ESP sono i led di stato, integrati nella

scheda. Essi risultano visibili dall'esterno attraverso un intaglio nella parte anteriore del plotter su cui è stata inserita una guida luce che, se pigiata all'estrema sinistra, funge anche da tasto di reset.

1.3.2 Software

Ciò che Scribit può fare è sostanzialmente disegnare su una superficie verticale. Tutto il processo di stampa, dalla scelta della parete al disegno, dall'installazione del robot al lancio della routine, è gestito tramite applicazione mobile. L'app è stata sviluppata sia per Android che per iOS e attualmente si trova già disponibile nei rispettivi store digitali. È stato deciso che l'interazione tra l'utente e la macchina sarebbe avvenuta tramite cellulare in quanto si tratta del dispositivo elettronico più utilizzato durante la giornata del consumatore medio: attraverso esso, come ben sappiamo, è possibile ricercare contenuti sulla rete, condividerli, ma anche controllare dispositivi *IoT*[1] sia in remoto che in locale. Scribit si inserisce a metà strada tra i due aspetti appena elencati: è un dispositivo IoT, collegato a un server che contiene una serie di disegni e possibilità creative per l'utente che può sia esplorare sia utilizzare.

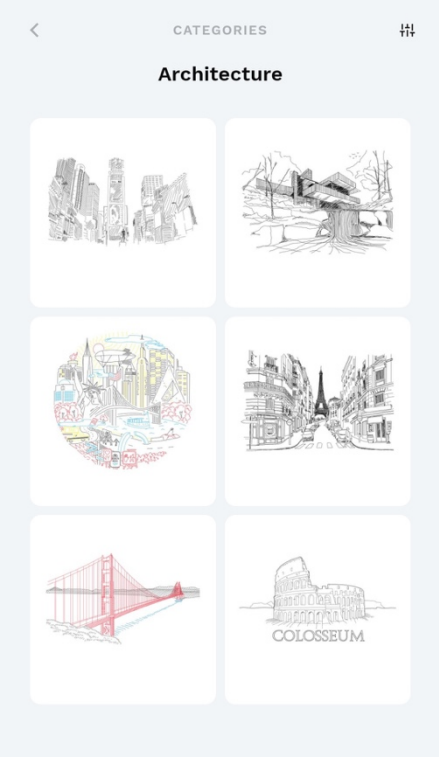

Figura 12 – Immagini appartenenti alla categoria "architettura" sull'app

[1] Con IoT, acronimo di "Internet of Things", si intendono quei dispositivi presenti nella vita di tutti i giorni che, collegati alla rete, acquisiscono "intelligenza" e ruolo attivo (domotica, assistenti vocali…) **[5]**.

Si è deciso di centralizzare contenuti e comunicazioni su server: ogni utente potrà legare al suo account più Scribit indipendenti tra loro e installati potenzialmente ovunque nel mondo, e potrà sia controllarli che scegliere la grafica o il ***widget*** che più preferisce all'interno della stessa piattaforma.

Ancor prima di installare Scribit a parete, dall'app è possibile visualizzare dei tutorial per compiere i gesti e passaggi necessario affinché tutto vada per il meglio e senza intoppi. Dopo l'installazione l'utente, registratosi sull'app, dovrà legare Scribit al proprio account attraverso un processo di pairing, ormai comune nell'universo IoT: il robot genera il proprio *access point*, l'utente comunica le credenziali di accesso alla rete locale connettendosi ad esso e il plotter sarà pronto per l'utilizzo collegandosi alla rete. Prima operazione da compiere una volta effettuato il pairing è la **calibrazione**: l'utente, dopo aver indicato l'ampiezza e la tipologia della parete (stuccata, whiteboard, o vetrina), vedrà Scribit spostarsi dal centro della parete compiendo dei movimenti volti a comprendere, data l'inclinazione del plotter, la propria posizione all'interno della superficie. Alla fine del processo andrà a posizionarsi in basso a sinistra[2].

Essendo adesso Scribit pronto per l'utilizzo, l'utente è libero di dare navigare all'interno dell'applicazione alla ricerca di un disegno o di un *widget* da stampare sul muro: oltre a convertire i disegni proposti dallo staff da vettoriali a GCODE, vengono forniti all'utente dei ***template*** dinamici, che è possibile modificare su richiesta. Sarà infatti possibile stampare a muro le previsioni meteo giornaliere o per una settimana, gli ultimi 3 *tweet* pubblicati da un utente o legati a un *hashtag* specifico, scrivere liberamente del testo da stampare scegliendo il font e convertire un'immagine raster in vettoriale. Tutte queste funzioni sono in lavorazione e saranno disponibili con le future release dell'applicazione.

Le conversioni dai vari formati di input (immagini raster e vettoriali, testo) a GCODE vengono effettuate attraverso l'utilizzo di ***funzioni remote***, chiamate esplicitamente sulla piattaforma Cloud di Google tramite richiesta ("*trigger*") http. L'app comunica inoltre con un'infrastruttura/applicazione NodeJS progettata a *microservizi* che espone delle API **REST** utilizzate sia per recuperare tutte le informazioni relative alle grafiche e alle preferenze degli utenti, sia per la comunicazione con i plotter legati ad ognuno di essi.

[2] La calibrazione automatica è uno degli studi da eseguire sul plotter nel futuro prossimo. Al momento, la calibrazione del robot sarà manuale: esso verrà posizionato nell'utente in uno dei punti fissi consigliati per la misura della parete in uso.

Per illustrare la pipeline di lancio di un disegno, immaginiamo di aver scelto quale sarà la grafica da stampare sulla parete: nel caso di un widget (meteo, twitter, testo, immagine raster) immetteremo le informazioni necessarie alla generazione di esso tramite chiamata alla funzione corrispondente (rispettivamente: città, utente/hashtag, testo, file raster) e riceveremo l'anteprima del disegno sull'app. Nel caso di un disegno già presente nella piattaforma, saremo già in grado di visualizzarne l'anteprima (Fig.13).

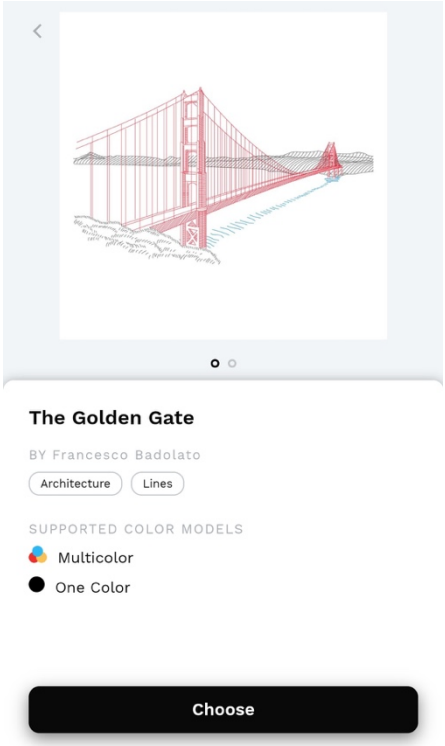

Figura 13 - Anteprima della grafica scelta sull'app

Dopo aver deciso se effettuare la stampa con un solo colore o con più di uno qualora il disegno lo permetta, il fattore di scala del disegno (piccolo, medio o grande rispetto all'area disegnabile) e se specchiare la grafica in caso di parete trasparente, possiamo richiedere l'inizio della stampa dopo aver inserito i pennarelli nel plotter: viene allora inviata una richiesta http alla funzione remota dedicata alla conversione SVG -> GCODE che restituisce il listato NC per la stampa e contestualmente per la cancellazione (qualora il tipo di parete lo consenta), oltre a informazioni utili per l'utente come il tempo stimato e la distanza in metri che ogni pennarello percorrerà. Il listato viene inviato al server che lo inoltra al plotter tramite broker **MQTT** modificandone lo stato, visibile dall'utente dal cambio dell'animazione dei led e dalla presenza dell'interfaccia di stampa sull'applicazione. Da questo momento sarà possibile mettere in pausa o fermare del tutto la stampa (Fig. 14): il primo caso si rende molto utile

nell'eventualità in cui l'utente abbia dimenticato di sostituire pennarelli già usati per lungo tempo, così da poterlo fare senza aver bisogno di rimandare la grafica selezionata in stampa (il plotter comunicherà comunque il suo cambio di stato al server attraverso lo stesso broker). Una volta terminato il processo, Scribit tornerà a posizionarsi nella posizione decisa in fase di calibrazione, e modificherà nuovamente il suo stato.

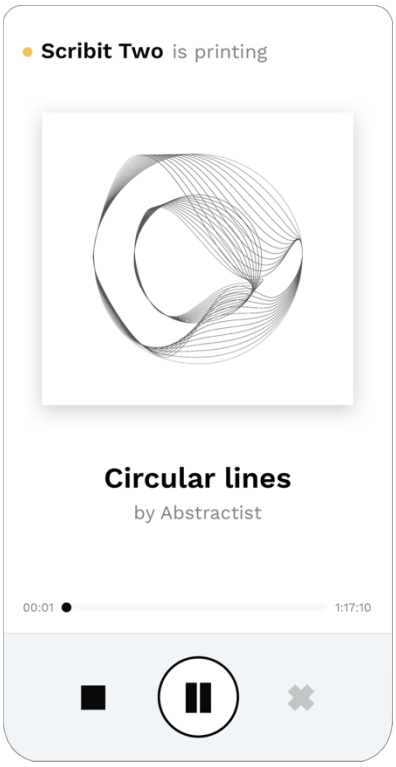

Figura 14 - Scribit Player

Per avviare la routine di cancellazione, basterà rientrare nella schermata relativa ai plotter sincronizzati all'account utente: se l'ultima stampa è stata effettuata su parete stuccata e non è stato inviato alcun altro disegno, il comando di cancellazione sarà disponibile. Qualora l'utente avesse invece deciso di cancellare manualmente il disegno (l'inchiostro per le pareti stuccate è facilmente rimovibile con l'acqua e il calore) e inviarne un altro, il precedente listato sarà cancellato dai sistemi e rimpiazzato con quello attuale.

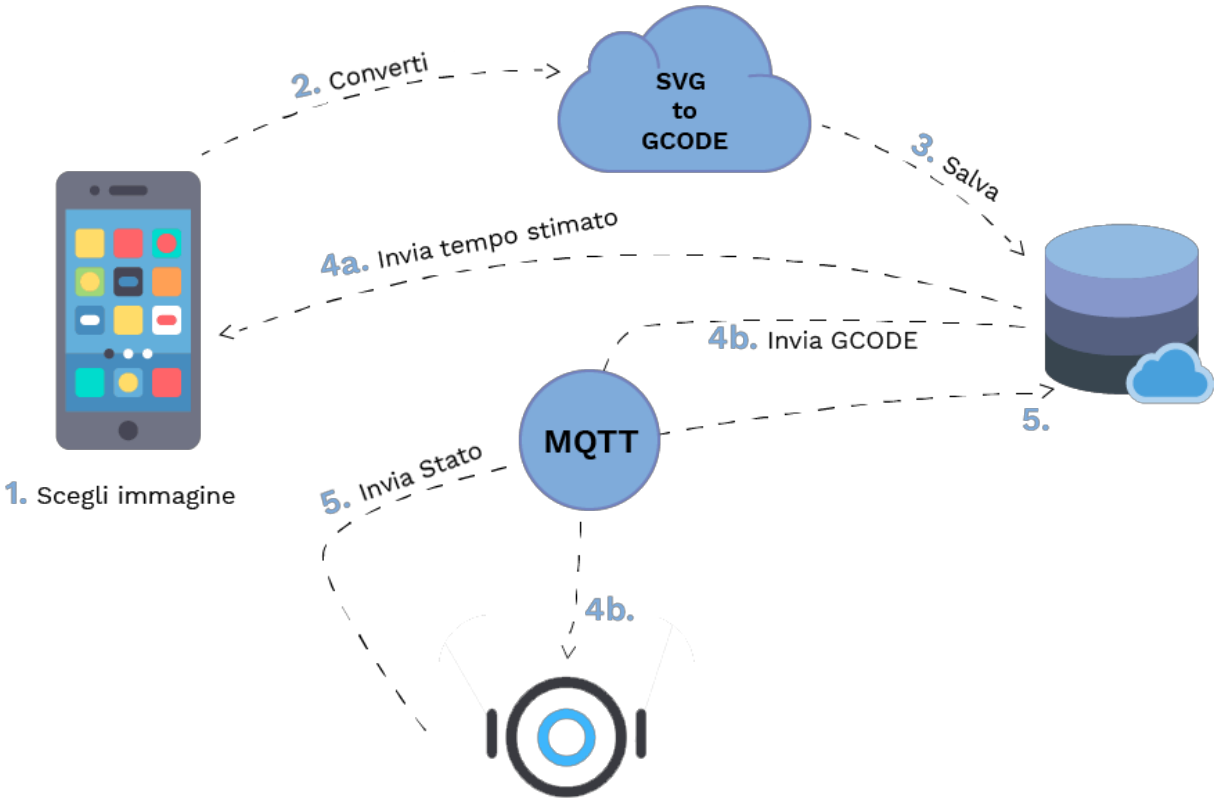

Figura 15 -Processo di stampa tramite app.

Icone di Matteo Palmieri (Scribit), FreePik e Smashicons (flaticon.com)

2. Studio preliminare

In questo capitolo verranno discusse le *conditio sine qua non* su cui si basa il funzionamento software del plotter. In base alla sua struttura, la meccanica, e il firmware su cui è basato l'intero l'oggetto, sono stati necessari studi e approfondimenti su funzionamento, organizzazione, sintassi e utilizzo dei formati SVG e GCODE per permettere una conversione più affidabile e ottimizzata possibile. Verranno quindi introdotti i due formati e illustrate le basi teoriche dietro lo sviluppo del software di conversione, che seguirà nel capitolo successivo. Inoltre, verrà illustrata la struttura del back-end, in cui gli script di conversione saranno utilizzati come *Google Cloud Functions* scritte in Node JS.

2.1 Scalable Vector Graphics

2.1.1 Panoramica

SVG è un linguaggio utilizzato per descrivere grafiche bidimensionali, adoperando sintassi XML. Entra in vigore come standard del World Wide Web Consortium (W3C) a partire dal 2001, frutto del compromesso tra il linguaggio proposto da Microsoft e Macromedia, VML, e PGML, proposto invece da Adobe e Sun Microsystems (poi Oracle) **[6]**.

La peculiarità del formato è quella di descrivere l'immagine in tecnica vettoriale **[7]**, ossia definendo la posizione all'interno dell'area di lavoro di una serie di primitive geometriche (es. segmento, punto, curva di Bézier) che, collegate tra loro, andranno a comporre il risultato finale; il vantaggio della grafica vettoriale risiede nella possibilità di scalare (ingrandire, rimpicciolire) indefinitamente l'immagine così composta secondo necessità, dato che al software di grafica utilizzato basterà ridefinire la posizione degli elementi geometrici in funzione del fattore di scala desiderato.

Il formato SVG permette di inserire tre tipologie di oggetti all'interno dell'area di disegno, quali *paths* o *tracciati* (letteralmente "percorsi"), immagini (*raster*[3]) e testo: va da sé che le immagini raster si sposano poco con gli altri elementi all'interno del contesto vettoriale, indefinitamente scalabile al contrario di un'immagine rasterizzata. Vengono solitamente

[3] La grafica raster o bitmap è contrapposta alla grafica vettoriale. Si intendono così le immagini digitali formate da una griglia ortogonale di righe e colonne i cui elementi costitutivi sono i *pixel*, unità minima per definire la superficie di un'immagine digitale. Dimensione ridotta e densità dell'unità permettono all'immagine di apparire uniforme agli occhi dell'osservatore, come accade con la tecnica del puntinismo. **[7]**

utilizzate come riferimento per lavori di ricalco o di adattamento. Spesso ci si riferisce ad un livello di astrazione più alto dei tracciati, le *shapes* (forme), ossia forme geometriche vere e proprie (rettangolo, linea, circonferenza…) convertite automaticamente in tracciati dal motore di rendering utilizzato dallo standard. È inoltre possibile raggruppare più elementi sotto il tag <g> ("*group*", gruppo) e assegnargli un nome **[8]**.

Il formato è ampiamente utilizzato nei più vari contesti: pagine web, loghi, icone, grafica pubblicitaria, arte: nel caso delle pagine web, ad esempio, viene sfruttata la compatibilità sintattica dello SVG implementandone le grafiche nella pagina e scalandone il contenuto in funzione della sua risoluzione. Inoltre, per descrivere le proprietà di ogni oggetto di cui l'immagine vettoriale è composta (colori di riempimenti, sfumature, spessore e colore del tratto…), si utilizza la sintassi propria del linguaggio CSS.

Nonostante la derivazione comune di HTML e SVG, il formato non è tuttora supportato da alcuni browser che necessitano dei plug-in per visualizzare correttamente le immagini vettoriali.

2.1.2 Tracciati

L'elemento **tracciato** è definito dal tag <path> all'interno del file SVG **[9]**. Come detto pocanzi, il tag può essere descritto da degli attributi, come stroke e fill (rispettivamente, colore del contorno e del riempimento). L'attributo **d** descrive la forma del tracciato, presentandosi come lista di primitive geometriche seguite dalla loro posizione all'interno del canvas. Ogni primitiva è definita da una lettera, maiuscola o minuscola: nel primo caso si tratta di comando *assoluto*, nel secondo di comando *relativo*. La differenza tra i due approcci risiede nella definizione dei punti occupati dalle primitive che, nel primo caso, si riferiscono alla posizione *assoluta* di punto iniziale e finale del tracciato all'interno dell'area, nel secondo invece ci si riferisce alla posizione *relativa* dei punti rispetto al punto finale cui è giunto il tracciato grazie alla primitiva precedente; immaginando il tracciato come disegnato da una penna, le istruzioni comandano la posizione di essa secondo un verso predefinito (che può essere modificato tramite attributo).

Di seguito verranno elencate le primitive geometriche e le istruzioni utilizzate dallo standard SVG e i metodi di conversione da path relativo al corrispondente assoluto (Tab. 1):

Tabella 1 - Comandi dei Path SVG

Comando	Nome	Parametri	Descrizione
M **m**	move To	(x y)	Comando utilizzato per iniziare a disegnare un nuovo tracciato. Muove la penna nel punto (x, y) senza che essa scriva
L **l**	line To	(x y)	Disegna una linea retta fino al punto (x, y)
H **h**	horizontal Line To	(x)	Ereditando il valore di y0 dall'ultima posizione della penna, disegna una linea retta fino al punto (x, y0)
V **v**	vertical Line To	(y)	Ereditando il valore di x0 dall'ultima posizione della penna, disegna una linea retta fino al punto (x0, y)
C **c**	curve To	(x1 y1 x2 y2 x y)	Disegna una curva di Bézier cubica fino al punto (x, y) con (x1, y1) e (x2, y2) punti di controllo
S **s**	shorthand Curve To	(x2 y2 x y)	Disegna una curva di Bézier cubica fino al punto (x, y) con punti di controllo la riflessione del secondo punto di controllo della curva precedente e (x2, y2). Se il comando non è preceduto da nessuna C/c, (x1, y1) si assume uguale al punto corrente (ossia il punto finale della precedente istruzione).
Q **q**	quadratic Bézier Curve To	(x1 y1 x y)	Disegna una curva di Bézier quadratica fino al punto (x, y) con (x1, y1) punto di controllo
T **t**	shorthand quadratic Bézier Curve To	(x y)	Disegna una curva di Bézier quadratica fino al punto (x, y) con punto di controllo pari alla riflessione di (x1, y1) della curva precedente. Se il comando non è preceduto da nessuna Q/q, (x1, y1) si assume uguale al punto corrente (= punto finale della precedente istruzione).
A **a**	elliptical Arc	(rx ry x-axis-rotation large-arc-flag sweep-flag x y)	Disegna un arco ellittico fino al punto (x, y). I parametri rappresentano: raggi (rx, ry), fattore di rotazione in gradi (x-axis rotation) e due flag di ottimizzazione (large-arc-flag, sweep-flag). Il centro dell'arco viene calcolato in funzione degli altri parametri.
Z **z**	close Path	/	Chiude il tracciato portando la penna al primo punto del tracciato o sotto-tracciato definito dall'istruzione M/m.

Quando si tratta di comandi relativi, gli unici parametri che vengono modificati rispetto ai comandi assoluti sono i punti: assumendo il punto corrente pari a *(cpx, cpy)*, ogni path relativo può essere convertito in assoluto sommando le coordinate del punto corrente al corrispettivo relativo: ciò vale anche per i punti di controllo delle curve. A un punto *(x, y)* di un path relativo corrisponderà quindi la posizione *(cpx + x, cpy + y)* (Fig. 16).

```
<svg height="210" width="400">
  <path d="M150 0 l-75 200 l150 0 z"/>
</svg>
```

```
<svg height="210" width="400">
  <path d="M150 0 L75 200 L225 200 Z" />
</svg>
```

Figura 16 - I due tracciati, che utilizzano rispettivamente istruzioni relative e assolute, sono equivalenti.

2.1.3 Testo

Per far sì che un testo sia inserito all'interno del canvas, è possibile indicare, oltre alle informazioni relative al posizionamento, il font e la relativa grandezza in punti tipografici **[10]**. La posizione del testo si riferisce per le x all'inizio di esso a partire da sinistra, per le y invece alla parte inferiore in altezza, ossia alla riga immaginaria su cui si procede a "scrivere".
Quest'approccio fa sì che, una volta generato il file, qualora venisse utilizzato da un client deficiente il font in questione, il testo verrà visualizzato con il font di sistema, la sua spaziatura ed eventualmente le sue grazie, sconvolgendo la resa grafica del file. Per questo motivo, in presenza di testi, è pratica comune convertire questi in tracciati tramite tool appositi messi a disposizione dai più famosi ed utilizzati editor di grafica vettoriali (Adobe Illustrator, Corel Draw, InkScape…). L'operazione di conversione sostanzialmente ricalca i contorni del testo, creando un tracciato di *outline*, appunto "contorno", formato dal solo attributo di stroke senza riempimento (fill) che può comunque essere aggiunto al nuovo tracciato, ottenendo nuovamente il risultato originale con la differenza di non poterlo più modificare nel contenuto ma solo nella forma, come qualunque altro tracciato.

2.1.4 Forme

Una *shape* (forma) è un elemento grafico definito da una combinazione di linee e curve **[11]**. Matematicamente, l'elemento "shape" è equivalente al tracciato che, tramite i comandi prima elencati, costituisce la stessa forma; essi condividono infatti le stesse proprietà (stroke, fill, transform…), vengono solo invocati attraverso due tag XML differenti. L'utilizzo delle shapes è molto utile qualora fosse necessario utilizzare la forma in sé, senza apportare modifiche alla primitiva geometrica di riferimento. Di seguito verranno elencate le forme disponibili nello standard SVG (Tab. 2, Fig. 17); l'unità di misura utilizzata è il pixel.

Tabella 2 - Shapes SVG

Tag	Primitiva	Parametri	Descrizione
rect	Rettangolo	x, y, width, height, rx, ry	(x, y) = posizione (top-left) width = larghezza height = altezza rx, ry = raggio di smussatura dei vertici
circle	Circonferenza	cx, cy, r	(cx, cy) = centro della circonferenza r = raggio
ellipse	Ellisse	cx, cy, rx, ry	(cx, cy) = centro dell'ellisse rx, ry = raggi x, y dell'ellisse
line	Segmento	x1, y1, x2, y2	(x1, y1) = punto iniziale (x2, y2) = punto finale
polyline	Segmenti concatenati	<points>	L'elemento polyline necessita coppie di coordinate (punti), dal primo all'ultimo del segmento complessivo. Nel caso in cui la formattazione dei punti sia errata, verrà rappresentato il segmento fino al punto errato
polygon	Poligono	<points>	Molto simile a *polyline*, richiede un set di punti

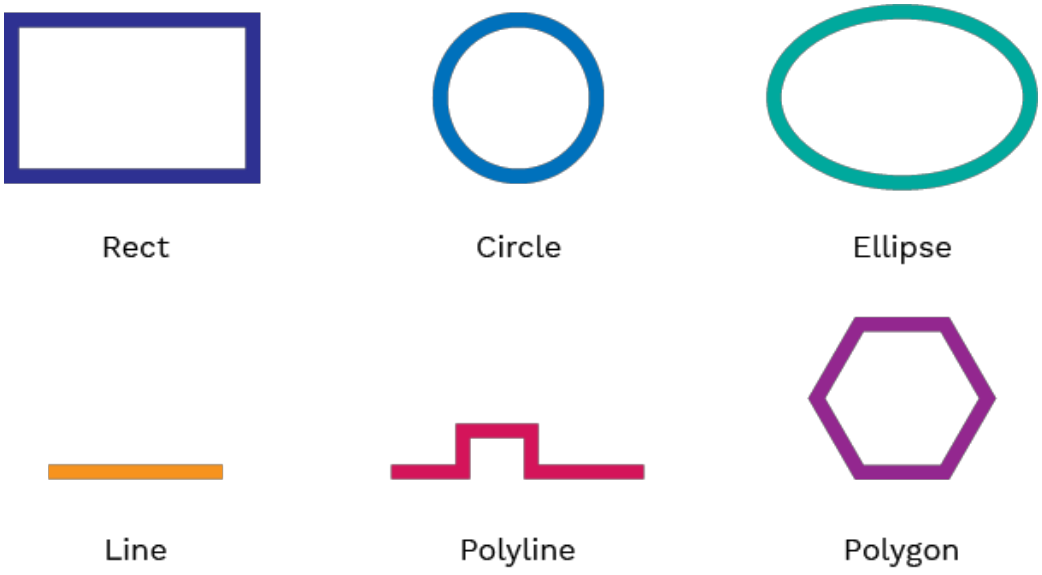

Figura 17 - Shapes SVG

2.1.5 Trasformazioni geometriche

L'attributo `transform` esplicita la presenza di trasformazioni geometriche della forma e/o del tracciato nel piano o area di lavoro (*canvas*). Si dice **trasformazione geometrica** una corrispondenza biunivoca del piano in sé, associando ad un punto del piano uno ed un solo punto del piano stesso **[12]**:

Equazione 1 - Trasformazione geometrica

$$t: P \rightarrow P'$$

Le trasformazioni possono essere isometrie, omotetie, similitudini, affinità.

Si dice **isometria** "una trasformazione geometrica che conserva le distanze" **[12]**. Dati quindi due punti A, B l'isometria fa ad essi corrispondere due punti A', B' tali che $\overline{AB} = \overline{A'B'}$: È il caso della **traslazione** e della **rotazione**.

La prima è definita da equazioni del tipo:

Equazione 2 - Isometria

$$\tau: \begin{cases} x' = x + p \\ y' = y + q \end{cases} \qquad \text{ossia} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} p \\ q \end{pmatrix}$$

con p e q costanti reali, x, y coordinate iniziali e x', y' le coordinate di destinazione.

Per quanto riguarda la rotazione, facendo riferimento al grafico qui presente (Fig. 18), poniamo le seguenti condizioni:

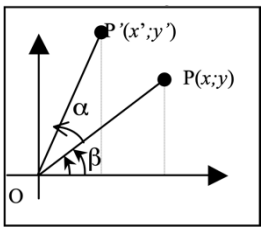

Figura 18 - Rotazione di un punto

$$x = \overline{OP} cos\beta$$
$$y = \overline{OP} sin\beta$$
$$x' = \overline{OP'}\cos(\alpha + \beta) = \overline{OP'}(cos\alpha cos\beta - sin\alpha sin\beta)$$
$$y' = \overline{OP'}\sin(\alpha + \beta) = \overline{OP'}(sin\alpha cos\beta - cos\alpha sin\beta)$$

ma, essendo $\overline{OP} = \overline{OP'}$, otteniamo:

$$x' = xcos\alpha - ysin\alpha \text{ e } y' = xsin\alpha - ycos\alpha$$

Le equazioni della rotazione centrata in O di un punto $P(x, y)$ per un angolo α, in senso antiorario, sono allora:

Equazione 3 - Rotazione di un punto P

$$\rho_{O,\alpha}: \begin{cases} x = x'cos\alpha + y'sin\alpha \\ y = -x'sin\alpha + y'cos\alpha \end{cases} \quad \text{ossia} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos\alpha & -sin\alpha \\ -sin\alpha & cos\alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Le **omotetie** corrispondono alla trasformazione di scalamento: "dato un numero reale non nullo h e un punto P del piano, l'omotetia di rapporto h e centro O è quella trasformazione che associa a P il punto P' tale che $\overline{OP'} = h\overline{OP}$." **[12]** Se $P(x, y)$ segue che $P'(hx, hy)$. P' è detto *omotetico* di P, O è il *centro di omotetia* e h *rapporto di omotetia*.

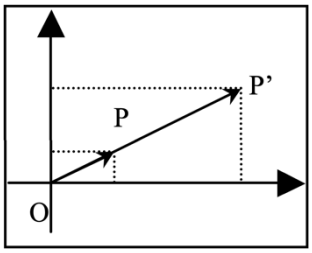

Figura 19 - Omotetia

Si chiama **affinità** (o *trasformazione affine*) una corrispondenza biunivoca T che fa corrispondere al punto $P\ (x, y)$ il punto $P'(x', y')$ attraverso le equazioni:

Equazione 4 - Affinità

$$\omega_{O,h}: \begin{cases} x' = ax + by + c \\ y' = a'x + b'y + c' \end{cases} \quad \text{ossia} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ a' & b' \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} c \\ c' \end{pmatrix}$$

dove a, b, c, a', b', c' sono numeri **reali [12]**. L'applicazione è biiettiva se e solo se:

$$\det(A) = \begin{vmatrix} a & b \\ a' & b' \end{vmatrix} \neq 0$$

La matrice A è detta *matrice delle affinità*.

Nel nostro caso, l'unica trasformazione affine che andremo a descrivere è la **dilatazione**, unica affinità applicabile tramite l'attributo `transform` **[13]**.

La dilatazione è una particolare affinità ottenuta dalle equazioni riportate immediatamente sopra ponendo $b = 0 \ \wedge a' = 0$:

Equazione 5 - Dilatazione

$$\begin{cases} x = hx + p \\ y = h'y + q\ ' \end{cases}$$

con $h, h' \neq 0$ detti *rapporti* **[12]**.

L'attributo `transform` funge da contenitore al cui interno vengono specificate le funzioni di trasformazione sopra descritte secondo la sintassi:

```
<path transform="f(a, b) g(c, d) o(e, f)">
```

Le funzioni applicabili con `transform` sono le seguenti:

- **`translate`**`(p, q)`: se `q` non è specificato, `x` e `y` vengono traslati entrambi di `p`
- **`rotate`**`(alpha, x, y)`: `(x,y)` sono le coordinate dell'origine della rotazione
- **`scale`**`(h, h')`: se `h'` non è specificato, `x` e `y` vengono scalati entrambi di `h`
- **`skewX`**`(h)`: distorsione dell'asse X con fattore `h`
- **`skewY`**`(h')`: distorsione dell'asse Y con fattore `h'`
- **`matrix`**`(a, b, c, d, e, f)`: come abbiamo visto in precedenza, praticamente tutte le trasformazioni geometriche possono essere applicate come matrici. La funzione `matrix` permette di creare la propria matrice di trasformazione organizzando i parametri in questo modo:

$$\begin{matrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{matrix}$$

Essa mappa le coordinate sul piano come:

$$\begin{cases} x' = ax + cy + e \\ y' = bx + dy + f \end{cases}$$

2.2 GCODE

G-Code è il nome del formato assegnato al linguaggio di programmazione NC (composto da istruzioni a controllo numerico) più ampiamente utilizzato, specialmente in ambito CAM (Computer Aided Manifacturing, aggiungi nota) **[14]**. La funzione del G-Code è quella di comandare i movimenti e le azioni di macchine utensili a controllo numerico (MCN) come fresatrici, presse, torni, macchine da taglio laser et altera.

Una tecnologia nata nei tardi anni '50 al MIT, limitata a lavorazioni di alta precisione fino agli anni '80 e quindi tendenzialmente inaccessibile ai più, è oggi tra le più utilizzate nel mondo

del making dopo la scadenza, nel 2009, del brevetto Chuck Hull 1986 riguardante la **stereolitografia**, tecnica attraverso cui è possibile realizzare oggetti tridimensionali a partire da file creati su software CAD (Computer Aided Drafting, disegno tecnico assistito dall'elaboratore), che ha quindi permesso a centinaia di aziende di rendere disponibili sul mercato consumer, *prosumer* e delle PMI le prime stampanti 3D a prezzi accessibili. Lo strumento ha assunto importanza preponderante in quei lavori e progetti che prevedono la creazione di prototipi a vari livelli di fedeltà durante ideazione e sviluppo di un prodotto fisico: basta infatti creare un modello 3D e lasciar realizzare l'oggetto alla stampante che, attraverso il proprio software di riferimento, organizzerà la stampa in istruzioni NC secondo le preferenze dell'utente. Le istruzioni gestiscono ogni parte della macchina e della stampa: movimento dei motori per lo spostamento dell'ugello (o estrusore) nello spazio, sua temperatura, velocità dello spostamento, quantità di filamento da fondere, profondità dell'estrusione, eccetera. Il set di istruzioni NC relativo a una stampante 3D è incredibilmente vasto, ed esistono diversi firmware in grado di comprendere e trattare le istruzioni differentemente, in base alla destinazione e al segmento d'utilizzo della macchina; attualmente, il più utilizzato e apprezzato nel campo è **Marlin**, firmware open source dal 12 Agosto 2011 adoperato sia da appassionati che da importanti marchi come Ultimaker e Prusa per la sua semplicità, stabilità e versatilità (la piattaforma hardware di riferimento del progetto Marlin è un Arduino Mega2560).

Essendo Scribit basato su firmware Marlin, le istruzioni che gli permettono di muoversi nello spazio, selezionare i pennarelli e cancellare l'inchiostro dalle pareti vengono fornite in G-Code dopo la trasformazione da SVG, come vedremo nel capitolo successivo.

Le istruzioni utilizzate dal robot sono le seguenti:

- **G1** [E<pos>] [F<rate>] [X<pos>] [Y<pos>] [Z<pos>]

  L'istruzione di movimento lineare *G1* ha come effetto quello di muovere la macchina secondo una traiettoria dritta da un punto all'altro nello spazio, assicurandosi che tutte le coordinate arrivino alla posizione finale nello stesso momento (tramite interpolazione lineare) **[15]**. I parametri del comando includono le tre coordinate spaziali *xyz*, che vengono utilizzate da Scribit rispettivamente per muoversi sul piano-parete attraverso le due pulegge, e per ruotare il tamburo con i 4 slot per i pennarelli.

Tabella 3 - Parametri dell'istruzione G1

Parametro	Descrizione
[E<pos>]	Lunghezza del filamento da inserire nell'estrusore tra l'inizio e la fine dello spostamento
[F<rate>]	Feedrate, approssimativamente la velocità di avanzamento
[X<pos>]	Coordinata dell'asse X (in mm)
[Y<pos>]	Coordinata dell'asse Y (in mm)
[Z<pos>]	Coordinata dell'asse Z (in mm)

- **G4** [P<time in ms>] [S<time in sec>]

  G4 è una semplice istruzione di pausa **[15]**; di solito utilizzata per far raffreddare il filamento durante la stampa 3D, in Scribit si utilizza principalmente per mantenere la posizione durante il processo di cancellazione, assicurandosi di cancellare l'inchiostro. I parametri P e S indicano l'attesa, rispettivamente in millisecondi e in secondi.

- **G92** [E<pos>] [X<pos>] [Y<pos>] [Z<pos>]

  L'istruzione di impostazione della posizione serve a far comprendere al plotter quale sia la propria posizione di partenza **[15]**; senza questa informazione, le coordinate delle successive istruzioni di spostamento verrebbero interpretate in modo scorretto, in quanto dipendenti dal punto di riferimento iniziale. I parametri sono banalmente le coordinate della nuova posizione.

- **M17** – **M18**

  Istruzioni relative ai motori stepper (passo-passo); M17 li abilita, M18 li disabilita **[15]**.

- **M92** [E<steps>] [T<index>] [X<steps>] [Y<steps>] [Z<steps>]

  M92 è un comando fondamentale, che imposta il numero di passi necessari a un motore per ogni unità di movimento **[15]**. Questa preferenza dipende dalle proprietà meccaniche dei motori e dai movimenti che dovranno compiere: Scribit utilizza due motori passo-passo uguali per le due pulegge, la cui unità di riferimento sono i millimetri, e un altro più piccolo e di minore coppia per la rotazione del tamburo, la cui unità sono i gradi. Anche qui, i valori da dare ai parametri sono di immediata comprensione, eccezion fatta per T, il cui campo "*index*" (indice) rappresenta l'estrusore di riferimento a cui impostare gli *step-per-unit*.

- **M104** – **M109** [B<temp>] [F<flag>] [S<temp>] [T<index>]

I comandi M104 e M109 impostano la temperatura dell'ugello o dell'estrusore della stampante, da cui dipende la fusione del filamento con cui verrà realizzata la stampa **[15]**. Le istruzioni differiscono nell'approccio: la prima imposta la temperatura che in background il firmware si occuperà di far salire o scendere come deciso, proseguendo con le istruzioni; la seconda, attende l'arrivo a temperatura prima di procedere. Scribit utilizza i comandi durante la routine di cancellazione, attendendo l'arrivo alla temperatura necessaria prima di iniziare e regolandola durante.

Tabella 4 - Parametri delle istruzioni M104 e M109

Parametro	Descrizione
[B<temp>]	Se viene utilizzato il flag di AUTOTEMP (temperatura automatica), rappresenta la massima temperatura raggiungibile
[F<flag>]	Flag di AUTOTEMP, se non presente disabilita la temperatura automatica
[S<temp>]	Temperatura desiderata. In caso di AUTOTEMP abilitato, rappresenta la minima temperatura raggiungibile.
[T<index>]	Indice dell'oggetto che dovrà arrivare a temperatura. Se omesso, verrà selezionato automaticamente l'oggetto attivo.

- **M201** [E<accel>] [T<index>] [X<accel>] [Y<accel>] [Z<accel>]
- **M203** [E<units/s] [T<index>] [X<units/s>] [Y<units/s>] [Z<units/s>]

  Con questi comandi è possibile impostare rispettivamente accelerazione e feed-rate massimi della stampa su ogni asse e dell'estrusore **[15]**. Come visto in altri casi, il parametro T richiede l'indice dell'estrusore da impostare, che nel caso di Scribit, non è presente.

- **M204** [P<accel>] [R<accel>] [T<accel>]

  L'istruzione M204 permette di impostare l'accelerazione iniziale/preferita di tre diversi movimenti **[15]**:

Tabella 5 - Parametri dell'istruzione M204

Parametro	Descrizione
[P<accel>]	P -> "*print*", accelerazione di stampa
[R<accel>]	R -> "*retract*", accelerazione dell'estrusore durante il ritiro del filamento
[T<accel>]	T -> "*travel*", accelerazione degli spostamenti senza estrusione

- **M205** [B<µs>] [E<jerk>] [J<deviation>] [S<units/s>] [T<units/s>] [X<jerk>] [Y<jerk>] [Z<jerk>]

  M205 è un comando che permette di decidere una serie di impostazioni molto diverse tra di loro che modificano il comportamento della stampante **[15]**. In Scribit viene

utilizzato solamente per gestire il "*jerk*" negli assi X e Y, ossia il tremolio cui il robot è soggetto al modificarsi brusco della direzione di stampa.

- **`M400`**
  Comando che attende l'esaurirsi dei movimenti precedenti prima di procedere con l'esecuzione delle istruzioni successive **[15]**.

2.3 Backend

Come brevemente anticipato nel capitolo introduttivo, è stato deciso che Scribit possedesse alle sue spalle un'architettura server, basata su *MongoDB* (database non relazionale[4]), che permettesse di controllarlo e di immagazzinare informazioni provenienti dalla società e dagli utenti stessi: grafiche editoriali, personalizzate dagli utenti, preferenze relative alle superfici, statistiche di utilizzo. L'interezza del *backend* non è però costituita dal solo server inteso come "magazzino" di dati: sul server è presente un client **MQTT** che ha l'onere di comunicare con il dispositivo, instradando i pacchetti d'informazione di ogni utente allo Scribit cui sono indirizzate le istruzioni. Tutte le funzioni più pesanti a livello computazionale, ad esempio la conversione da immagine vettoriale a GCODE o da immagine raster a vettoriale, scritte prevalentemente in *NodeJS*, sono state e verranno invece implementate su *Google Cloud Platform*. L'intera struttura è concepita secondo architettura **REST** (*Representational State Transfer*).

2.3.1 Representational State Transfer e Microservizi

Il Representational State Transfer, noto ai più con l'acronimo REST, è un approccio per architetture di sistema distribuite, utilizzatissimo nei servizi orientati al web e frutto degli studi di Roy Fielding, già esperto e co-autore di *Hypertext Transfer Protocol* (http) **[17]**.
L'approccio si basa sull'idea che il rapporto tra *client* e *server* debba essere ben definito in quanto le mansioni di entrambi sono molto diverse e non vanno ibridate causando problemi di scalabilità e di evoluzione dei singoli componenti. Le richieste provenienti dal client devono essere *esplicite*, utilizzando le tipologie di richieste già messe a disposizione dal protocollo **http**

[4] Un database non relazionale non necessita si definire a priori le tabelle degli attributi di un oggetto, al contrario dei classici database di tipo SQL. Tutti gli attributi di un singolo elemento sono salvati all'interno di un file JSON garantendo maggiore versatilità e scalabilità grazie anche a schemi flessibili **[16]**.

(GET per ottenere una risorsa, POST per modificarla, DELETE per eliminarla…) ed esponendo per intero l'oggetto della richiesta all'interno dell'**URI**. Tipico esempio di una richiesta partita da un servizio *RESTful* potrebbe essere:

```
http://myservice.com/users?id=192
```

in cui oggetto della *query* è l'utente il cui identificativo è 192.

La comunicazione tra client e server si appresta quindi a diventare **stateless** ("senza stato"): essendo tutte le informazioni richieste dal client all'interno della query, al server viene semplicemente inviata la richiesta e non necessiterà di immagazzinare alcun altro file relativo a sessioni attive dell'utente e simili. È lapalissiano quindi immaginare che il server a sua volta dovrà solo inviare la risorsa richiesta al client senza appesantire ulteriormente la risposta. Così facendo si perde totalmente il concetto di "stato" della comunicazione, client e server comunicano esplicitamente: il "cosa" è presente nell'URI, il "come" nella tipologia di richiesta. Essere stateless obbliga il server a trattare ogni richiesta come una novità, data la decisione di eliminare file di stato potenzialmente immagazzinabili da esso: sarà il server stesso che allora potrà decidere, rispondendo al client, che la risorsa richiesta e restituita sia *cachabile* o meno. In caso positivo, il client utilizzerà la propria cache per ottenere risorse corrispondenti a una serie di richieste identiche, alleggerendo il carico computazionale del server e permettendogli maggiore scalabilità liberando velocemente le proprie risorse.

L'opposto di ciò che descriviamo accade e accadeva con approcci di tipo **SOAP** (*Simple Object Access Protocol*) in cui le richieste vengono costruite ad-hoc utilizzando metodi tipici della programmazione a oggetti per modificare le risorse ottenute da server e servizi web **[18]**.

Tipico esempio di servizi RESTful, quindi adatti all'architettura REST, sono i cosiddetti **microservizi**. Come è facile evincere dal nome, si tratta di servizi (nel nostro caso **web**) dalla grandezza limitata: insieme si comportano a tutti gli effetti come sistemi distribuiti che spartiscono il carico computazionale di un task potenzialmente complesso in task minori, approccio in contrapposizione alla classica struttura "monolitica" **[19]**. I vantaggi principali di utilizzare microservizi e *middleware*[5] sono la loro **scalabilità** e **resilienza**: trattandosi di piccoli servizi non occupano grandi risorse lasciando quindi spazio a richieste e dati in ingresso;

[5] Viene definito "middleware" ogni software, solitamente ridotto in termini di funzionalità e consumo di risorse, che si interpone tra due servizi come intermediario.

inoltre, se qualcuno di essi dovesse smettere di funzionare, non è detto che l'intero sistema ne subisca le conseguenze essendo i servizi legati tra loro ma pur sempre entità a sé stanti. Ciò si traduce in semplicità di gestione, correzione degli errori e sviluppo rapido, ma anche in uno sforzo non indifferente nello studio preliminare dell'architettura: in presenza di grandi web services, composti da un numero altrettanto grande di microservizi, è chiaro che garantire la corretta comunicazione tra essi assicurando ad ognuno i dati richiesti e necessari per il funzionamento del servizio in sé non sia cosa banale (ricordiamo che la comunicazione è stateless).

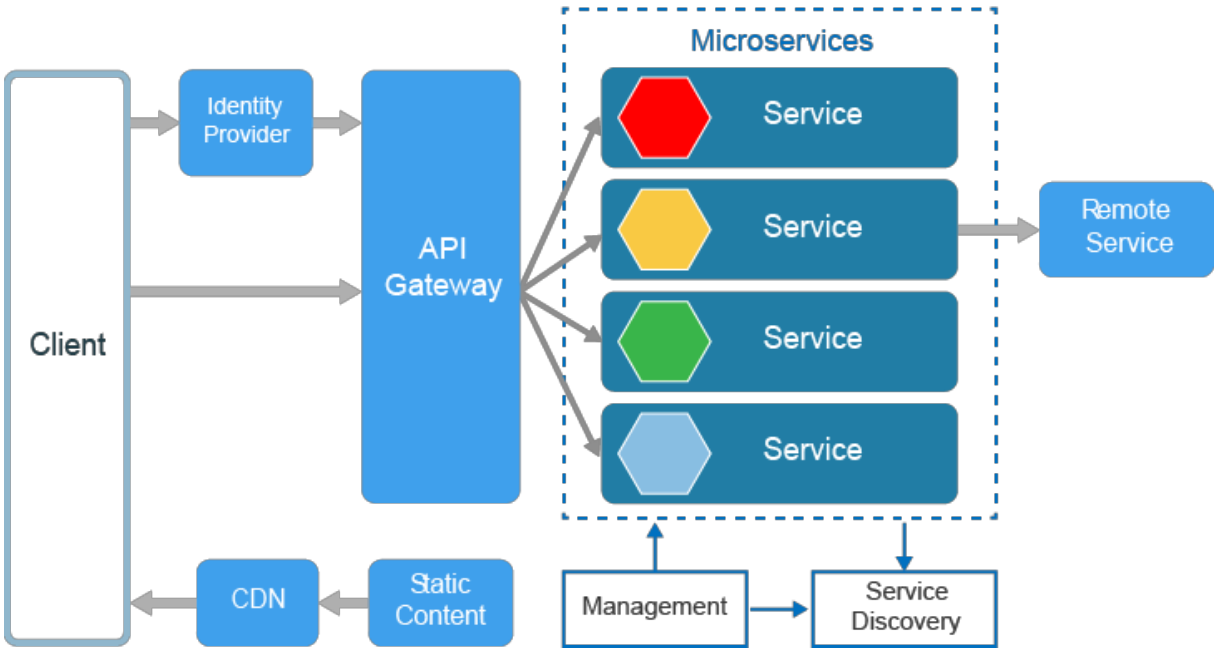

Figura 20 - Architettura a Microservizi (© Microsoft Azure)

2.3.2 MQTT e suo utilizzo

MQTT (*Message Queue Telemetry Transport*) è un protocollo di scambio messaggi basato su TCP/IP inventato all'interno di IBM nel 1999 e reso open source poco dopo **[20]**. È ampiamente utilizzato nel mondo IoT per la sua versatilità, scalabilità ed efficienza anche in presenza di scarsa ampiezza di banda e bassa potenza computazionale. Il suo funzionamento ruota attorno al meccanismo **publish-subscribe**: mittenti e destinatari (rispettivamente *publisher* e *subscriber*) comunicano in modo asincrono attraverso un *broker*, un intermediario, che si occupa di immagazzinare i messaggi *pubblicati* dai mittenti per inoltrarli ai destinatari che si sono *sottoscritti* ad esso **[21]**. Ogni messaggio è identificato da un *topic*, un argomento, oltre a contenere il messaggio vero a proprio e l'identità del mittente: esso comunica il messaggio

relativo a un argomento sul broker, senza rivolgerlo a nessun destinatario in particolare. Ogni subscriber è un potenziale destinatario del messaggio se ha deciso di "abbonarsi" allo stesso topic, o al publisher (Fig. 21).

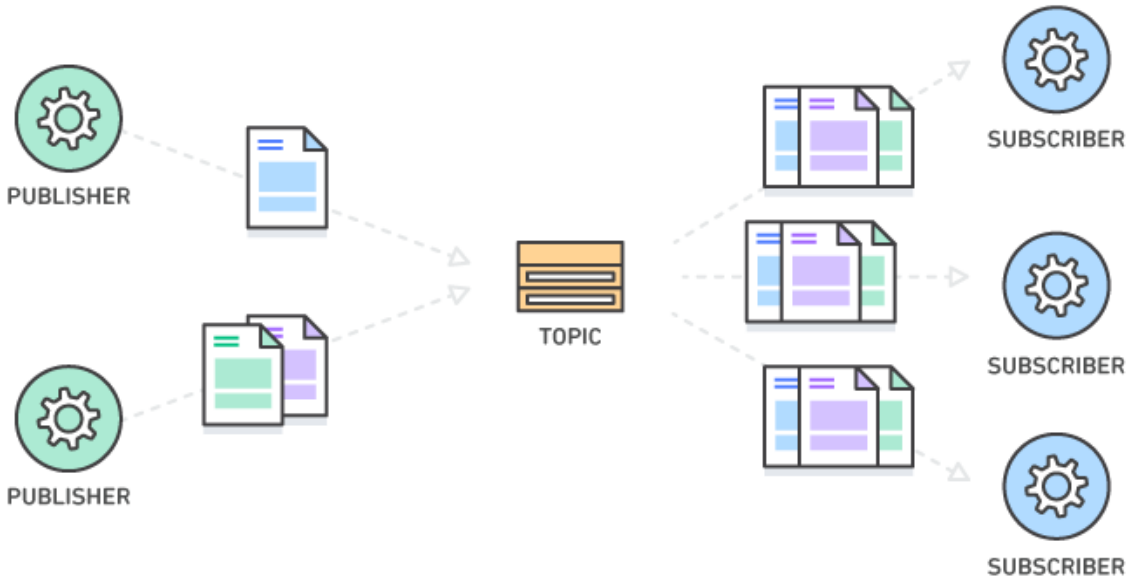

Figura 21 - Messaggistica pub-sub (© Amazon AWS)

Il broker, per la funzione ricoperta, si configura come *middleware* all'interno di una struttura distribuita, essendo un software che funge da intermediario tra due entità: nel nostro caso, Scribit e il server. Il protocollo viene utilizzato per indirizzare le corrette informazioni al robot in base allo stato in cui si trova e per forzare lo stesso in caso di stampa o cancellazione.
Sono cinque i topic inviati da e a Scribit, che si configurano come stati in cui il robot può trovarsi:

- ***idle***: Scribit è fermo in posizione iniziale e in attesa di istruzioni
- ***print***: Scribit è all'interno del processo di stampa
- ***erase***: Scribit è all'interno del processo di cancellazione
- ***reset***: Scribit sta annullando il lavoro attuale e si appresta a tornare in posizione iniziale
- ***pause****:* Scribit si trova all'interno di un lavoro (stampa o cancellazione) ma è in pausa

Il robot invia periodicamente informazioni riguardo il suo stato al broker, così da informare il server riguardo la sua possibilità di accettare lavori o meno. Tutti gli stati, ad eccezione di *idle*, vengono forzati dal server dopo esplicito comando da parte dell'utente. Se all'interno dell'app l'utente deciderà di inviare una stampa o una cancellazione, l'intenzione verrà inoltrata al

server che, una volta generati i rispettivi GCODE (come vedremo nel paragrafo e capitolo successivi) ne includerà il link come *payload*[6] del messaggio MQTT. Esso viene inviato al broker ed avrà come topic una stringa composta dall'**ID** dello Scribit per cui l'utente ha richiesto il lavoro e il nuovo **stato**. Alcuni di essi verranno inoltrati al robot solo sotto determinate condizioni: il server immagazzina lo stato attuale di Scribit sottoscrivendosi all'ID di ognuno sul broker, quindi solo se il robot in questione sta stampando o cancellando sarà possibile metterlo in pausa o annullare il lavoro, come se e solo se il robot è in idle potrà accettare nuovi lavori. Avendo queste informazioni sempre a disposizione sul server, si evita di ingrandire la coda di messaggi sul broker con contenuti inutilizzabili.

2.3.3 Servizi di Cloud Computing

Il **Cloud Computing** è un paradigma secondo il quale è possibile accedere a risorse computazionali, software e hardware, collegate tra loro e condivise con altre entità nella rete **[22]**. Si contrappone all'idea classica di avere uno o più server fisici da mantenere e a cui riferirsi quando si creano servizi web: salvare delle risorse o eseguire del codice sul *cloud* significa infatti allocare i propri dati e servizi su una macchina o una serie di esse collegate tra loro e posizionate fisicamente anche a due poli opposti del pianeta. Utilizzare questo approccio quando si costruiscono servizi web che possono crescere in misura e in necessità di risorse computazionali nel tempo apporta sicuramente una serie di benefici non indifferenti, considerando le principali caratteristiche presenti nel modello:

- **Servizio "misurato" e on-demand**: il cloud alloca solo ed esclusivamente le risorse necessarie allo svolgimento del task richiesto, sia esso salvare dei dati su storage fisico, ospitare un web service, eseguire del codice. Ciò si traduce in massima efficienza e spreco di risorse pressoché nullo. Considerando il tempo e il denaro necessari per organizzare uno o più server fisici, questa soluzione si rivela particolarmente efficace soprattutto in presenza di un utilizzo non massiccio, in quanto i provider che offrono servizi cloud propongono pagamenti pesati sulle risorse effettivamente utilizzate.

[6] Carico utile, parte dei dati trasmessa effettivamente utilizzabile dal destinatario/richiedente. In questo caso, esattamente il messaggio relativo al topic.

- **Resource Pooling**: per *resource pooling* si intende l'atto di raggruppare determinate risorse in presenza di più client intenti a richiederne diverse. Nel cloud computing, le risorse da raggruppare sono perlopiù computazionali, e l'organizzazione di esse dovrà sempre essere quanto più in linea con le richieste dell'utente. Ogni macchina è potenzialmente molto distante dall'altra a livello fisico ma ogni risorsa è utilizzabile da chiunque ne faccia richiesta all'interno del cloud. Per questo motivo, data l'indipendenza spaziale delle risorse, che siano hardware o software, esse vengono assegnate e ri-assegnate continuamente cercando di offrire la maggior efficienza possibile in termini di tempo di risposta, velocità di elaborazione, eccetera. La riassegnazione di una risorsa in base alla vicinanza spaziale tra client e singolo server potrebbe essere una delle discriminanti, ma di norma il client non è in grado di conoscere la localizzazione fisica dei propri dati.

- **Elasticità e Scalabilità**: essendo il cloud composto da innumerevoli macchine collegate tra loro indipendentemente dalla distanza fisica che le separa, modificare quantità e tipologia di risorse da allocare per un utente che dovesse cambiare necessità non è sicuramente un problema. Ogni servizio e database propinato al cloud è potenzialmente scalabile all'infinito, e le risorse di esso vengono continuamente ottimizzate al modificarsi dei bisogni dell'utente e allo scalare dei suoi dati.

Il primo provider di servizi cloud, pioniere del *serverless* computing a livello commerciale è Amazon con i suoi Amazon Web Services (AWS), seguito poi da Google Cloud Platform (GCP) e Microsoft Azure. Nel nostro caso è stato deciso di utilizzare i servizi proposti da GCP per knowhow aziendale e personale del team, e inseriti gli algoritmi di cui tratteremo nel prossimo capitolo come funzioni remote *Google Cloud Functions* (corrispettivo di AWS Lambda). Come accennato in precedenza, Google e altri provider suoi competitor propongono pagamenti basati sull'effettivo tempo di calcolo impiegato dalla/e macchina/e per eseguire il codice contenuto dalle funzioni. Esse possono essere scritte in diversi linguaggi di programmazione (NodeJS, Python, C#, Java GoLang…) e vengono invocate su richiesta al verificarsi di determinati eventi, siano essi scatenati da altre applicazioni "vive" sul cloud o richieste http REST.

3. Sviluppo Software

Dopo aver specificato i formati e gli standard necessari al funzionamento lato software di Scribit, in questo capitolo verranno illustrati gli algoritmi principali sviluppati per essere eseguiti su Google Cloud Platform.

3.1 SVGtoGCODE

La funzione di conversione da SVG a GCODE è stata pensata per essere eseguita su richiesta; si tratta quindi uno script che, dati dei parametri in ingresso, restituisce il file desiderato prima di esaurirsi.

L'idea dietro l'algoritmo di conversione parte da una considerazione relativa ai **path** SVG: come visto nel capitolo precedente, ogni file vettoriale è composto da forme e tracciati e le loro posizioni all'interno dell'area di lavoro sono accessibili direttamente dai *tag* che li costituiscono. La differenza tra i due elementi sta essenzialmente nel modo in cui sono definiti i punti per cui idealmente passa la penna che disegna il tracciato: le **shapes** espongono il solo punto iniziale del tracciato partendo dal più in alto a sinistra nel caso di linee e poligoni, o il centro nel caso di circonferenze ed ellissi, mentre i path espongono punto iniziale e finale di ogni segmento, oltre ai punti di controllo per le curve.

Per restituire in output un file che dovrà contenere comandi discreti punto per punto è opportuno **convertire** per prima cosa le eventuali **forme geometriche** contenute nel file SVG **in tracciati**, e trattare esclusivamente quelli nel momento in cui si procederà alla conversione vera e propria dei punti di ognuno in istruzioni NC.

Ottenuto il file composto da soli tracciati, è possibile accedere a ogni punto iniziale e finale; il primo utilizzo dei tracciati ottenuti è l'operazione di **ridimensionamento** dell'area di lavoro alla dimensione desiderata del disegno sulla parete.

È possibile accedere alle informazioni relative all'ampiezza del *canvas* all'interno del tag `<svg>`, il primo ad essere scritto nel file SVG che annuncia il formato, negli attributi `width` (larghezza), `height` (altezza) e `viewBox`, che contiene ambe le informazioni precedenti secondo il formato:

```
viewBox = "0 0 1024 1024",
```

in cui i primi due attributi rappresentano il punto iniziale del canvas (il punto P(0,0) è posto in alto a sinistra), gli ultimi due il punto finale (posto in basso a destra, le cui coordinate sono identificabili con larghezza e altezza effettiva).
Gli attributi di cui sopra vengono modificati con la dimensione desiderata, e il rapporto tra le due dimensioni, originali e desiderate, utilizzato per scalare ogni tracciato punto per punto.
Dopo lo scalamento è possibile modificare ulteriormente il file solo se espressamente richiesto dall'utente, **riflettendo** la grafica rispetto all'asse Y: questa opzione è stata resa disponibile per gli utenti che intenderanno stampare qualcosa su vetrine e superfici trasparenti in genere: essendo Scribit progettato per funzionare indoor, una stampa che dovrà essere apprezzata e/o compresa dall'esterno (si pensi ad esempio all'annuncio dei saldi nei negozi) andrà specchiata se stampata dal lato interno della parete.

Adesso è possibile procedere con l'**estrazione dei comandi** all'interno dei tracciati, e conseguentemente dei punti da essi esposti: ogni istruzione propria di ogni tracciato va trattata separatamente e tradotta in GCODE in base al suo significato. Sicuramente, tutti i comandi relativi ai punti verranno tradotti in istruzioni "G1", come visto nel capitolo precedente. Piuttosto i comandi dei path configurano diversamente l'utilizzo dei punti che espongono: il primo discriminante è costituito dalla presenza di comandi assoluti o relativi. È necessario poter conoscere in ogni istante il punto corrente cui si trova la penna per sommare alle sue coordinate l'eventuale punto relativo; inoltre, ogni comando tratta diversamente i punti esposti, ad esempio un comando "h" o "H" (*horizontal line to*) espone la sola coordinata dell'asse X, ereditando la coordinata Y dal punto corrente. Un caso a parte è rappresentato dalle curve, che esponendo solo punto finale e punti di controllo necessitano di essere interpolate in più comandi di spostamento lineare: suddividendo la curva in segmenti, la loro lunghezza (e conseguentemente il numero totale di essi) rappresenta il grado di approssimazione in proporzionalità inversa.

Dopo la generazione del file GCODE corrispondente alla grafica richiesta, lo script si occupa della creazione del file relativo alla **cancellazione** della stessa grafica, se richiesto dall'utente. Per cancellare viene creata un'immagine vettoriale composta da soli segmenti orizzontali e verticali che ripercorrono la grafica scalata passando con la ceramica (calda) sull'inchiostro.

A questo punto, creato anche il secondo file, lo script si esaurisce restituendo altri dati utili come la stima del tempo di stampa e la distanza in metri percorsa da ogni pennarello sulla parete per permettere all'utente di valutarne la sostituzione dopo un certo numero di utilizzi.

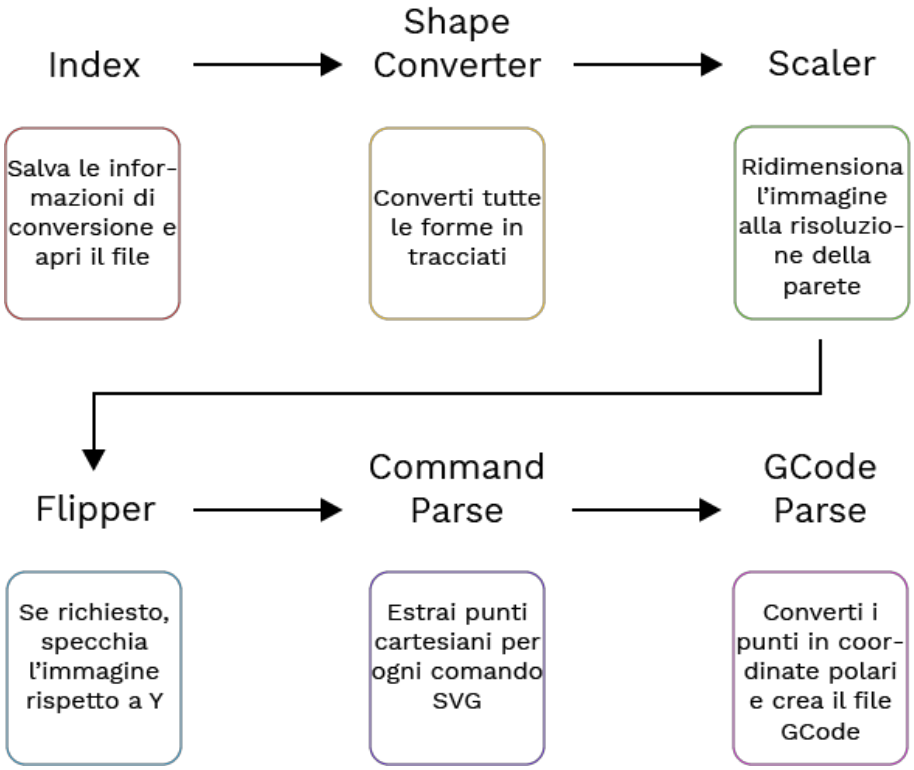

Figura 22 - Diagramma semplificato dell'algoritmo

Vedremo adesso tutti i moduli che compongono lo script in dettaglio.

3.1.1 Indice e richiesta

Il modulo principale della funzione, *index*, contiene in ordine tutti i passi elencati precedentemente. Da qui passa la richiesta http che è così strutturata:

Request:

```
{
    "userId": string,
    "inputFile": string,
    "scaleFactor": float
    "flipflag": boolean,
    "eraseFlag": boolean,
    "multiColor": boolean,
    "width": int,
    "height": int,
    "startX": int,
    "startY": int
```

```
}
```

- **`userId:`** id univoco dell'utilizzatore che ha invocato la funzione. È essenziale per permettere ai GCODE generati di essere caricati nello spazio dedicato all'utente stesso all'interno del bucket
- **`inputFile:`** il file SVG richiesto per la stampa
- **`scaleFactor:`** percentuale dell'area massima disegnabile che si intende mettere a disposizione della stampa. Il valore è compreso tra 0 e 1; tuttavia, nell'applicazione mobile l'utente avrà a disposizione tre soli formati per i disegni, S, M e L, corrispondenti a uno `scaleFactor` rispettivamente pari a 0.5, 0.75, 1.
- **`flipflag:`** se `true`, il disegno verrà specchiato rispetto all'asse Y
- **`eraseFlag:`** se `true`, segnalato verrà generato anche il GCODE di cancellazione. Nel caso di una normale parete, la decisione spetterà all'utente; se invece la stampa avverrà su whiteboard, vetro o qualsiasi altra superficie diversa dal calcestruzzo, e verrà sull'app, il flag sarà automaticamente impostato a `false`
- **`multiColor:`** se `true`, verranno utilizzati tutti i pennarelli necessari per realizzare il disegno selezionato. Altrimenti, la grafica verrà stampata utilizzando solamente il pennarello inserito nel primo slot
- **`width:`** larghezza della superficie
- **`height:`** altezza della superficie
- **`startX:`** coordinata X del punto iniziale di Scribit
- **`startY:`** coordinata Y del punto iniziale di Scribit

Gli ultimi 4 parametri della richiesta vengono definiti in fase di **calibrazione**, procedimento che avviene a cavallo tra l'app e il firmware durante la prima installazione di Scribit su una parete: sarà necessario indicare l'ampiezza della parete per permettere al robot di muoversi al suo interno, cercando il punto migliore in termini di tensione in cui stazionare, partire e ritornare ad ogni stampa. Le sue coordinate verranno inviate al server via MQTT e salvate in un file JSON dal quale verranno prelevate queste informazioni ogni qualvolta una nuova richiesta dovrà essere formulata.

Oltre a richiamare gli altri moduli, l'indice della funzione si occupa di aprire e chiudere lo **stream** di scrittura del file GCODE, e di caricarlo sul *bucket*[7] una volta conclusa la scrittura.

Il metodo di conversione viene invocato per ogni oggetto di tipo **`path`** all'interno del file SVG: per ogni tracciato vengono prima controllati i nodi **genitori**, poi estratti i **comandi**. Come accennato nel capitolo precedente, la divisone in livelli di un'immagine vettoriale corrisponde al raggruppamento degli elementi all'interno sotto il tag **<g>**, "gruppo"; il tag gode

[7] Letteralmente "secchio", termine utilizzato in cloud computing per definire uno spazio sul cloud in cui depositare i file

dell'attributo `id`, ossia il nome assegnato al gruppo. In fase di progettazione, è stato deciso che le grafiche multicolore realizzate per Scribit sarebbero state divise in **livelli**, ognuno con un nome identificativo di un pennarello (m1, m2, m3, m4): all'interno di un livello è possibile dividere ulteriormente i tracciati in sottogruppi, che faranno comunque capo al genitore di tipo "livello", posto gerarchicamente più in alto di default. Inoltre, sempre a livello gerarchico, due tracciati appartenenti a due diversi livelli condivideranno un solo genitore, ossia il tag <svg>, aperto all'inizio di in ogni file vettoriale e chiuso soltanto alla fine. Per questo motivo, per passare l'informazione relativa al colore/pennarello da utilizzare alla routine di conversione istruzioni, è necessario scandagliare **tutti** i nodi genitori di un tracciato e non solamente il padre: nel primo caso avremo la certezza di risalire al livello che contiene l'informazione sul colore, nel secondo correremmo invece il rischio di trovarci all'interno di un sottogruppo che non espone alcuna informazione riguardante il pennarello da usare.
Il ciclo può essere così rappresentato:

```
color = 'm1'
FOR EACH path IN svgFile {
      IF multiColorFlag IS true THEN
            FOR EACH parent IN path {
                  IF parent.name EQUALS TO 'g' AND parent.id IS NOT
                  undefined THEN
                        IF parent.id IS EQUAL TO 'm1' or 'm2' or 'm3'
                  or 'm4' THEN
                              color = parent.id
                        ENDIF
                  ENDIF
            ENDFOR
      ENDIF
      GET commands OF path
      CALL parse OF commands
ENDFOR
```

La funzione di conversione dei comandi (indicata nello specchietto in pseudocodice come "**parse**") ritorna un oggetto che contiene al suo interno il numero di comandi di tipo **M** e **m** trovati nel file che verranno utilizzati dalla stessa routine per il calcolo della stima temporale, e un array contenente le coordinate degli estremi della grafica: queste verranno utilizzate dall'indice come parametro per il lancio della routine di cancellazione, che genererà una

grafica delle stesse dimensioni. L'utilizzo di entrambe le informazioni verrà approfondito nei paragrafi a seguire.

3.1.2 Shape Converter

Il modulo di conversione **da forme a tracciati** è il primo ad essere invocato dall'indice. Il passaggio da questa funzione è obbligato e non dipende dalla volontà dell'utente, in quanto propedeutico al trattamento che l'immagine subirà successivamente. L'apertura del file avviene all'interno della funzione: esso viene trattato come testo **XML** grazie al modulo *npm* `cheerio`, che ricalca in pieno funzionalità e sintassi di *jQuery*[8]. Avviene quindi una **ricerca** dei tag relativi alle forme, che vengono convertite una dopo l'altra in tracciati (prima i `rect`, poi i `circle`, e così via). Come già ampiamente spiegato in precedenza ogni forma può essere convertita in tracciato, combinando i comandi nel giusto ordine (il rendering dei comandi di tracciato avviene in senso orario, rispettarlo significa quindi convertire la forma ordinandoli nello stesso verso). L'algoritmo di conversione si divide in **tre passi**: **recupero delle informazioni** dal tag (posizione, centro, larghezza, altezza, eventuali trasformazioni, eccetera), **creazione del tracciato** corrispondente e **sostituzione della forma**. Alla fine, ogni tracciato che avrà ereditato un attributo di trasformazione (`transform`) viene riscritto applicando ai singoli punti la trasformazione in questione piuttosto che calcolarla in fase di rendering, cosa che Scribit non sarebbe capare di interpretare[9]. Di seguito, vedremo nel dettaglio i singoli algoritmi di creazione del tracciato in funzione di ogni forma, seguendo le indicazioni dello standard SVG 2.

3.1.2.1 Rect

Il trattamento degli attributi dei rettangoli è così strutturato:

- **`(x, y)`**: le coordinate si riferiscono al vertice in alto a sinistra del rettangolo.
- **`width, height`**: larghezza e altezza del rettangolo devono essere interi o floating point maggiori di zero. Valori minori verranno interpretati come errore, valori uguali a zero disabilitano il rendering dell'elemento.

[8] Celeberrima libreria JavaScript utilizzatissima per manipolare file HTML e ottimizzare l'implementazione di web applications lato client.
[9] Trattandosi di una conversione da grafica vettoriale a istruzioni NC, il processo di rendering del file SVG non verrà mai realmente invocato né utilizzato per trattare l'immagine. Ci si rifà ad esso in fase di ricostruzione dei tracciati per non creare artefatti di visualizzazione una volta che le coordinate saranno convertite.

- **rx, ry**: i valori devono combaciare in quanto la smussatura degli angoli dei rettangoli SVG è definita come simmetrica; se non fosse questo l'effetto desiderato, si dovrebbe creare un `path` ad-hoc. È quindi possibile inserire un solo valore di `rx` o di `ry` (l'altro verrà considerato la sua copia), inserirli entrambi ma uguali, o specificare il valore numerico per uno dei due e impostare l'altro ad `auto`. Inoltre, è possibile indicare lo smussamento in percentuale: se descritta da rx, il suo valore in pixel sarà percentuale della larghezza del rettangolo, altrimenti se descritta da y, la medesima considerazione varrà per l'altezza. La presenza dei suddetti attributi non è obbligatoria, la loro assenza o valore pari a zero sottintende la volontà di generare un normale rettangolo; i valori negativi non sono accettati.

Dopo aver verificato le condizioni di cui sopra e salvato i relativi dati, il relativo tracciato viene generato come segue:

```
1. M x+rx y
2. H x+width-rx
3. IF (rx > 0 AND ry > 0) THEN A rx ry 0 0 1 x+width y+ry
4. V y+height-ry
5. IF (rx > 0 AND ry > 0) THEN A rx ry 0 0 1 x+width-rx y+height
6. H x+rx
7. IF (rx > 0 AND ry > 0) THEN A rx ry 0 0 1 x y+height-ry
8. V r+ry
9. IF (rx > 0 AND ry > 0) THEN A rx ry 0 0 1 x+rx y
```

3.1.2.2 Circle

Gli attributi propri di circle sono esclusivamente le coordinate del **punto centrale** `(cx, cy)`, chiaramente non accettabili se negative, e il **raggio** `r`, per cui vale la stessa regola.
La circonferenza viene tradotta in tracciato tramite 4 archi ellittici. Il rendering, come già accennato, avverrebbe in senso orario partendo da ore 3:

```
1. M cx+r cy
2. A r r 0 0 1 cx cy+r
3. A r r 0 0 1 cx-r cy
4. A r r 0 0 1 cx cy-r
5. A r r 0 0 1 cx+r cy
```

3.1.2.3 Ellipse

Come per le circonferenze, `(cx, cy)` rappresenta il **centro** dell'ellisse. I valori dei due raggi **rx** e **ry** invece seguono le regole già viste con i rect, ma con delle differenze: le percentuali andranno intese in funzione di larghezza e altezza dell'intera area di lavoro. Inoltre, se il valore auto verrà assegnato ad uno dei due, inevitabilmente i due raggi coincideranno e la forma sarà pari ad una circonferenza.
Il path corrispondente verrà così creato:

```
1. M cx+rx cy
2. A rx ry 0 0 1 cx cy+ry
3. A rx ry 0 0 1 cx-rx cy
4. A rx ry 0 0 1 cx cy-ry
5. A rx ry 0 0 1 cx+rx cy
```

3.1.2.4 Line, Polyline e Polygon

Banalmente, una linea (segmento) è definita da punto **iniziale** `(x1, y2)` e punto **finale** `(x2, y2)`; nel caso di una polyline si parla di più segmenti **concatenati** tra loro, quindi le coppie di punti oltre l'iniziale possono essere n. L'unico discriminante nel trattamento delle variabili è la presenza di ambe le coordinate all'interno del punto: se così non fosse, il segmento che congiunge l'ultimo punto corretto con quello errato, e tutti i successivi, non verrebbero renderizzati. Quanto detto, vale allo stesso modo per l'elemento `polygon`.
Il path generato consta semplicemente di un absolute moveTo al punto iniziale e una serie di absolute lineTo, uno per ogni punto della polyline:

```
1. M x1 y1
2. L x2 x2
3. L xi yi
4. …
5. L xn yn
```

Nel caso di un `polygon`, andrebbe aggiunto un solo comando alla fine, ossia un *close path command*, **Z**: il comando traccerà un segmento dall'ultimo punto renderizzato al punto iniziale, chiudendo quindi la figura piana e rendendola un poligono a tutti gli effetti. Non ci sono limiti ai punti possibili da inserire in un poligono, ma è chiaro che essi andranno scelti con cognizione di causa per ottenere la forma desiderata.

3.1.2.5 Transform e operazioni finali

Anche se ogni forma è adesso stata convertita in tracciato, tutti gli attributi ad essa legata sono rimasti invariati: `stroke` (colore della traccia), `fill` (riempimento), gruppo di appartenenza (servirà per la scelta del pennarello), eccetera. Tra gli attributi, l'unico che verrà trattato ad-hoc è **`transform`**. Trattandosi di un effetto di rendering esso non verrebbe mai letto e/o interpretato da Scribit[10]; siamo quindi obbligati ad applicare direttamente le trasformazioni ai punti dei tracciati. Come abbiamo già visto nel paragrafo 2.1.5, tutte le trasformazioni geometriche sono definite da equazioni e assimilabili all'applicazione di matrici su punti del piano. L'attributo viene quindi letto per ogni tracciato che lo contiene e ogni funzione espressa al suo interno applicata ai punti nel rispetto delle equazioni. Dopo la trasformazione dei punti, l'attributo viene cancellato dalla stringa.

Come operazione finale, tutti gli archi ellittici (comandi **A** e **a**) vengono convertiti in curve di Bézier [**23**].

3.1.3 Scaler

Adesso che il file presenta solo tracciati al suo interno, possiamo applicare delle trasformazioni per ingrandirlo e adattarlo alla parete su cui l'utente ha deciso di utilizzare Scribit. Le informazioni sulla superficie (`width, height, startX, startY`, come visto nel paragrafo 3.1.1) vengono salvate sul server dopo che l'utente avrà concluso la calibrazione del robot, e utilizzate di default finché l'utente non deciderà di cambiare superficie esplicitandolo tramite l'applicazione.

Ciò che avviene in questa utility, e sostanzialmente l'applicazione di due trasformazioni geometriche, ossia **omotetia** e **traslazione**: come già illustrato, l'omotetia garantisce la relazione $P' = hP$, con $h \in \mathbb{R}$, $h > 0$, e viene qui utilizzata per modificare le dimensioni del disegno. Utilizziamo invece la traslazione per centrare il disegno all'interno del nuovo canvas, in quanto l'omotetia avviene punto-punto: essendo il canvas della maggior parte delle immagini digitali organizzato come una matrice in cui il punto in alto a sinistra rappresenta

[10] Neanche gli attributi *stroke* e *fill* potranno essere apprezzati una volta convertito il file, ma essi non verranno mai toccati dall'originale poiché serviranno per mostrare la preview del disegno con i colori corretti all'utente prima di lanciare la routine di stampa.

l'origine del sistema di riferimento $O(X,Y)$ con coordinate $(0,0)$, l'omotetia restituirà un disegno ingrandito o rimpicciolito, la cui origine sarà però immutata:

Equazione 6 - Omotetia in un'immagine digitale

$$\omega_{O,h}\begin{pmatrix} 0{,}0 & \cdots & x,0 \\ \vdots & \ddots & \vdots \\ 0,y & \cdots & x,y \end{pmatrix} = \begin{pmatrix} 0{,}0 & \cdots & hx,0 \\ \vdots & \ddots & \vdots \\ 0,hy & \cdots & hx,hy \end{pmatrix}$$

L'algoritmo inizia calcolando la porzione massima della parete che può essere utilizzata per il disegno, convertendo la misura in **punti tipografici**: è chiaro che Scribit, essendo appeso al muro tramite due cavi arrotolati attorno alle pulegge, ognuna collegata ad un motore passo-passo, soffra della tensione generata dai due cavi. Quando i motori spostano il robot risalendo il muro, in verso quindi contrario alla forza di gravità, sono costretti ad applicare più coppia a discapito dell'accelerazione. Attraverso i datasheet dei motori e diversi rilievi empirici, è stata individuata la relazione che definisce la massima superficie utilizzabile per un disegno come:

Equazione 7 - Calcolo della massima area disegnabile

$$maxViewBox = \begin{cases} x{:}\,\dfrac{4}{5}wallWidth \\ y{:}\,wallHeight - \dfrac{wallWidth}{2} \end{cases}$$

A questo punto è possibile estrarre le dimensioni del canvas SVG dagli attributi **`width`** e **`height`**, o direttamente da `viewBox`, prima di sovrascriverli con le dimensioni effettive della parete. Il fattore di omotetia h è quindi definito come:

```
h = min[(maxViewBox.x/width), (maxViewBox.y/height)] * scaleFactor
```

in cui, come già illustrato, scaleFactor è un fattore compreso tra 0 e 1, in cui 1 rappresenta la volontà da parte dell'utente di disegnare l'intera porzione parete utilizzabile. Come fattore di omotetia viene scelto un unico valore per mantenere intatto il rapporto originale dell'immagine, si tratta del **minimo** tra i due rapporti, relativo quindi al lato corto dell'immagine, per essere certi che il disegno rientri perfettamente nella porzione minima, solitamente quella verticale considerando le limitazioni fisiche e meccaniche più stringenti per la salita della parete piuttosto che per lo spostamento laterale. Se il canvas ha le stesse dimensioni della parete, l'intera utility viene saltata. Ciò accade in vista degli aggiornamenti futuri della piattaforma, che permetteranno agli utenti, tra le altre cose, di mandare i propri

file vettoriali in stampa con Scribit: se vorranno quindi impostare un disegno decentrato, o porre più elementi in posizioni ben definite della parete, potranno farlo impostando la grandezza del canvas pari a quella della superficie, senza che il disegno subisca alcuna modifica.

Inizia quindi l'estrazione dei tracciati: di ogni tracciato viene estrapolato l'attributo **d**, contenente i comandi che lo definiscono. Tutti i punti andranno moltiplicati per h, ma non tutti andranno centrati tramite traslazione: i punti interni ai comandi relativi sono infatti descritti rispetto alla distanza dal punto precedente (adesso aumentata o diminuita di un fattore h) ma intervenire traslandone la posizione muterebbe irrimediabilmente la grafica. Viene quindi fatta una cernita tra i comandi letti: se il comando è **relativo**, i suoi punti verranno moltiplicati per h; se **assoluto**, entrambe le coordinate dei punti verranno moltiplicate per h e poi centrate.

La trasformazione applicata ai punti assoluti è quindi la seguente:

Equazione 8 - Trasformazione geometrica applicata ai punti assoluti nel modulo "scaler.js"

$$\tau(\omega): \begin{cases} x' = hx + \dfrac{wallWidth}{2} - \dfrac{h}{2} width \\ y' = hy + \dfrac{wallHeight}{2} - \dfrac{h}{2} height \end{cases}$$

L'unico comando che presenta altre informazioni oltre a punti veri è propri è l'arco ellittico (comando A, a), ma non dovremo occuparcene dato che nel passo precedente dello script ogni arco è stato convertito in curva di Bézier.

All'arrivo del comando di chiusura tracciato (Z, z) l'attributo d viene sostituito con una stringa contenente i comandi aggiornati con i nuovi punti, e vengono estratti i comandi del path successivo.

Di seguito, il diagramma di flusso del modulo (Fig. 23):

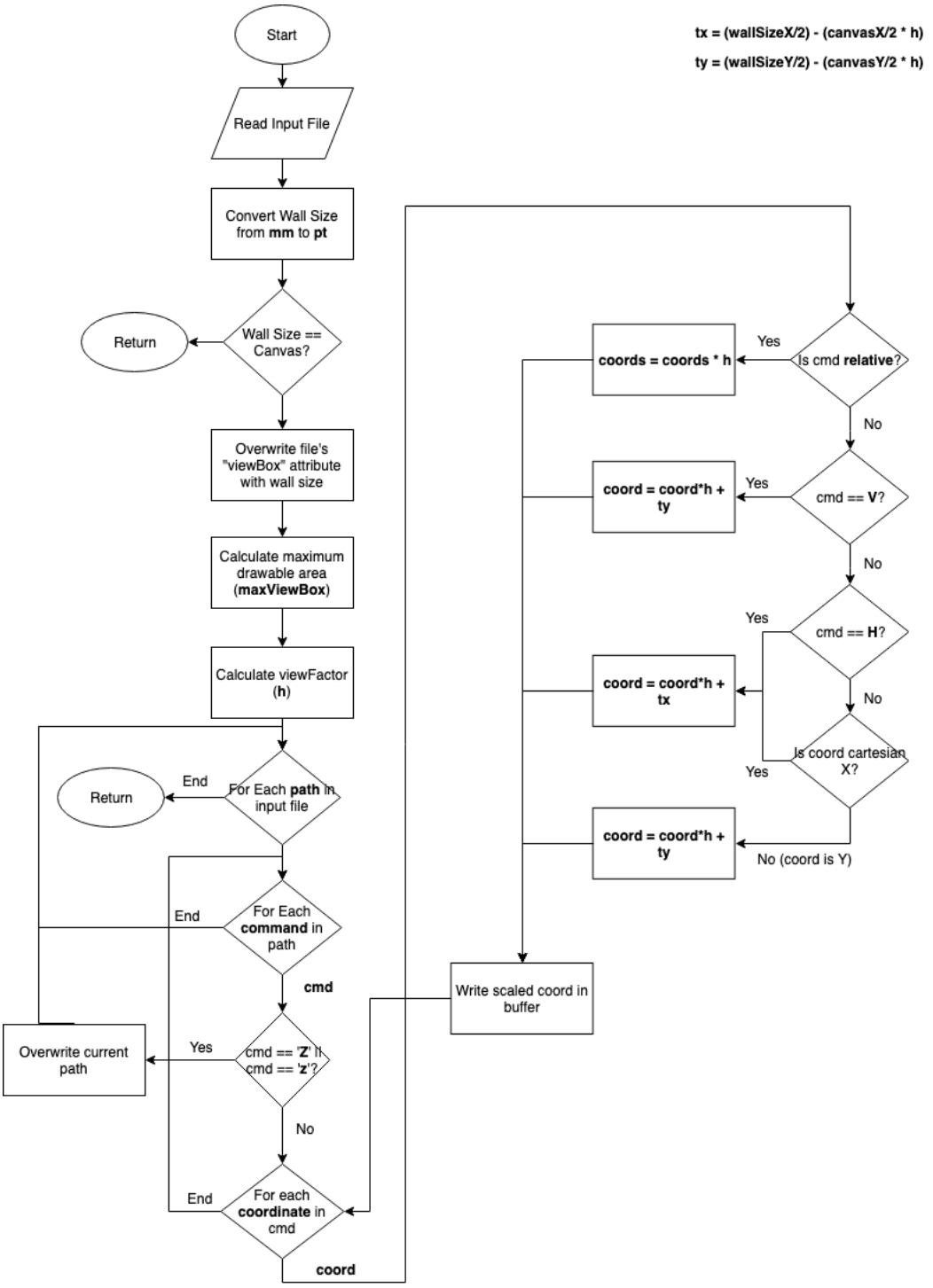

Figura 23 - Diagramma di flusso del modulo "scaler.js"

3.1.4 Flip

La utility di ribaltamento dell'immagine funziona esattamente come quella di ridimensionamento appena descritta: si tratta di una speciale omotetia in cui vengono definiti due fattori, h e h'. In questo specifico caso $h' = 1$ e $\boldsymbol{h} = -1$, non rispettando quindi le condizioni espresse nel paragrafo precedente: in effetti, procedendo con un'omotetia simile, i punti non si troverebbero più nel primo quadrante di un normale sistema di riferimento cartesiano $O(X, Y)$ ma nel quarto, e considerando che la matrice non contiene valori negativi tutti i punti sarebbero non visibili. Siamo costretti quindi ad applicare una trasformazione composta come nel caso dello *scaler*, aggiungendo la larghezza dell'intero canvas ai punti ormai negativi, traslandoli.

La trasformazione dei punti è quindi pari a:

Equazione 9 - Trasformazione geometrica applicata ai punti assoluti nel modulo "flip.js"

$$\tau(\omega): \begin{cases} x' = -x + width \\ y' = y \end{cases}$$

Come per la situazione precedente, la trasformazione completa andrà applicata solamente ai punti espressi da comandi assoluti; i punti all'interno di comandi relativi verranno solo invertiti di segno.

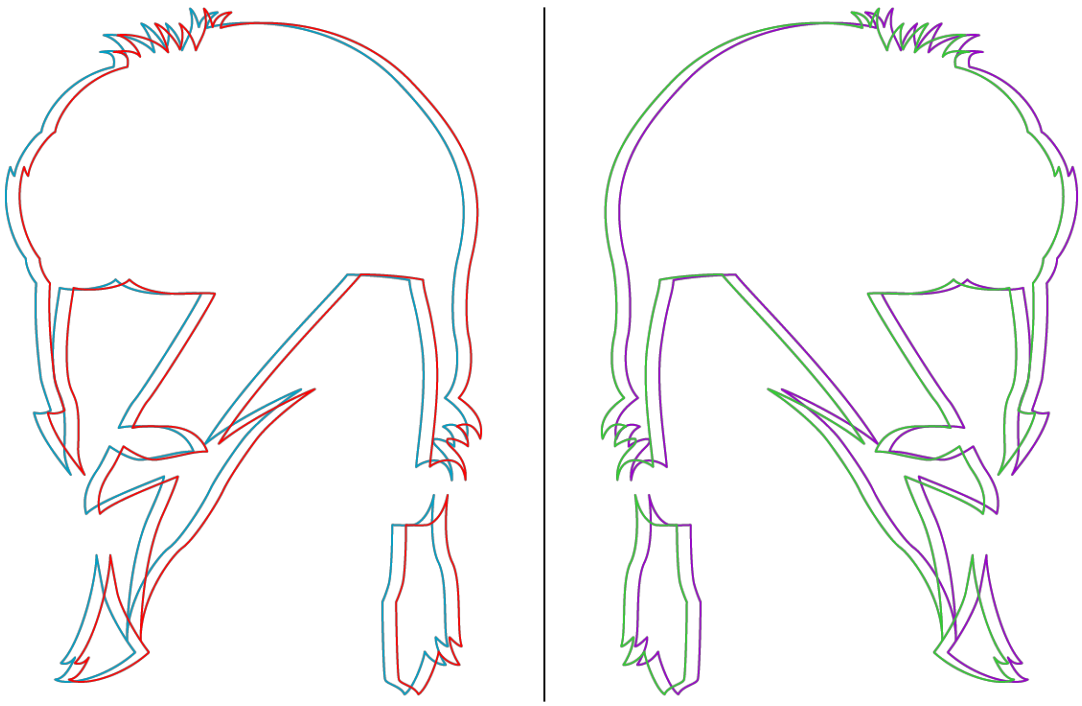

Figura 24 - Risultato della riflessione verticale (dx) della grafica originale (sx)

3.1.5 commandParse

CommandParse è il modulo che si occupa di preparare i punti del file SVG per la conversione a gCode in base ai comandi a cui appartengono. Il modulo prende in ingresso i comandi dell'attributo **d** di un singolo tracciato. Per prima cosa, viene effettuato un controllo sul comando: dalla sua funzione e dal fatto che sia assoluto o relativo, cambia il modo in cui i punti vengono preparati per la conversione.

Non appena lo script viene invocato, il primo punto estratto viene salvato come primo punto del tracciato e **punto corrente**. La prima informazione servirà per chiudere il tracciato in presenza di un comando Z o z, la seconda è invece essenziale in presenza di **comandi relativi**: infatti il punto corrente viene sempre aggiornato dopo ogni trattamento. Per sintassi, qualunque tracciato SVG non può essere definito con un primo comando diverso da M, avendo quindi la certezza che il primo punto considerato sarà sempre assoluto: il comando successivo tratterà quel punto come corrente. Dopo la conversione dei propri punti in gCode, il punto finale dell'istruzione attuale acquisirà lo status di punto corrente, e così via. In presenza di comandi relativi, al punto corrente vengono **sommati** o s**ottratti** i valori esplicitati dal comando per rendere la coordinata da convertire in gCode **assoluta**.

Andando con ordine, dopo le assegnazioni del punto iniziale, la routine cicla tra i comandi finché non si esauriscono, e come detto, ad ognuno è riservato un trattamento specifico dei punti. In presenza di linee rette è necessario esplicitare il solo punto di arrivo del segmento, mentre per le curve subentrano uno o più punti di controllo. Al momento, Scribit non si muove generando autonomamente traiettorie curvilinee, ma tutte le istruzioni dategli in pasto sono costituite da spostamenti lineari: è necessario quindi utilizzare i dati esposti dai comandi in questione per descrivere la curvatura grazie ai punti di controllo e poi suddividerli in linee rette con un passo tale da non rendere l'immagine risultante spigolosa e poco precisa.

Per questa evenienza sono stati adoperati alcuni metodi di *geomerative*, libreria open source scritta in Processing e utilizzata sin dalla prima versione di OpenWall per gestire sostanzialmente tutti gli aspetti legati al trattamento delle immagini vettoriali. Nel nostro caso, il tutto è stato re-ingegnerizzato e riscritto adottando un approccio basato sulle stringhe e non sulle proprietà geometriche ad alto livello dell'immagine, ma i metodi di calcolo delle curve di Bézier, cubiche e quadratiche, sono stati tradotti da Processing in Node.JS e riutilizzati

per praticità. Le funzioni restituiscono l'array dei punti di cui la curva è composta, e sono questi ad essere passati alla utility di conversione.

A seguire, una tabella esplicativa dei singoli trattamenti da eseguire in base all'istruzione di riferimento; la struttura utilizzata per questa specifica applicazione è evidentemente uno ***switch-case*** il cui discriminante è il carattere di comando. Le variabili e funzioni che utilizzeremo in tabella saranno:

- **`currentPoint:`** punto corrente
- **`endPoint:`** punto finale del tracciato
- **`firstPathPoint:`** primo punto del tracciato (del comando `M`)
- **`controlPoints:`** punti di controllo delle curve di Bézier (a volte esplicitati come `controlPoint1` e `controlPoint2`)
- **`command:`** comando attuale
- **`lastCommand:`** comando precedente
- **`points:`** punti generati tramite interpolazione
- **`moveTo`**`():` funzione del modulo di conversione GCode che sposta il plotter all'inizio del tracciato da disegnare
- **`generate`**`():` funzione del modulo di conversione GCode che genera un punto del tracciato da disegnare
- **`interpolateCubicBézier`**`():` funzione di interpolazione per curve di Bézier cubiche
- **`interpolateQuadraticBézier`**`():` funzione di interpolazione per curve di Bézier quadratiche
- **`endsCalc`**`():` funzione di calcolo degli estremi del disegno

Tabella 6 - Estrazione dei punti da path commands SVG

M **m**	• assign currentPoint from command • endPoint = currentPoint • firstPathPoint = currentPoint • moveTo(currentPoint)
L **l**	• assign endPoint from command • generate(endPoint)
H **h**	• assign endPoint.x from command, endPoint.y from currentPoint • generate(endPoint)
V **v**	• assign endPoint.y from command, endPoint.x from currentPoint • generate(endPoint)

C
c
	• assign controlPoints and endPoint from command
	• points = interpolateCubicBézier(currentPoint, controlPoints, endPoint)
	• for each point in points {
	generate(point)
	}

S
s
	• if (lastCommand is C or c or S or s) {
	assign controlPoint1 from lastCommand and currentPoint
	}
	else controlPoint1 = currentPoint
	• assign second controlPoint and endPoint from command
	• points = interpolateCubicBézier(currentPoint, controlPoints, endPoint)
	• for each point in points {
	generate(point)
	• }

Q **q**	• assign controlPoint and endPoint from command • points = interpolateQuadraticBézier(currentPoint, controlPoint, endPoint) • for each point in points { generate(point) • }

T
t
	• if (lastCommand is Q or q or T or t) {
	assign controlPoint from lastCommand and currentPoint
	}
	else controlPoint = currentPoint
	• points = interpolateQuadraticBézier(currentPoint, controlPoints, endPoint)
	• for each point in points {
	generate(point)
	}

Z **z**	• if (currentPoint is equal to firstPathPoint) break else generate(firstPathPoint)
Per ogni comando	• endsCalc (calcolo degli estremi dell'immagine)
Dopo lo switch-case	• currentPoint = endPoint • lastCommand = command

Come accennato precedentemente, all'interno del modulo si provvede anche al calcolo degli estremi del disegno tramite la funzione chiamata endsCalc. Ogniqualvolta un punto viene convertito, viene passato ad una funzione che tratta 4 variabili globali: leftEnd, rightEnd, upEnd e downEnd. Esse si riferiscono rispettivamente ai 4 estremi del disegno (sinistro, destro, superiore, inferiore). Il loro primo valore è pari al primo punto che invoca la funzione; i valori futuri verranno invece sostituiti man mano che i punti passati alla funzione si riveleranno maggiori, o minori, in base all'estremo considerato, dei valori precedenti. Possiamo illustrare il funzionamento di questa semplice funzione in questo modo:

```
endsCalc(pointX, pointY) {
      IF IS THE first calculation THEN
            leftEnd = pointX
            rightEnd = pointX
            upend = pointY
            downEnd = pointY
      ELSE
```

```
            IF (pointX < leftEnd) leftEnd = pointX
            ENDIF
            IF (pointX > rightEnd) rightEnd = pointX
            ENDIF
            IF (pointY < upEnd) upEnd = pointY
            ENDIF
            IF (pointY > downEnd) downEnd = pointY
            ENDIF
      ENDIF
}
```

3.1.6 gCodeParse

In questo paragrafo verranno descritte le funzioni che effettivamente convertono i punti dei tracciati SVG in **istruzioni NC** e che impostano i parametri della stampa per Scribit. All'interno del modulo convergono tutte le informazioni relative a posizione, velocità, distanza che il plotter avrà e compirà durante le routine di stampa e cancellazione, restituendo stime e previsioni riguardo il tempo di stampa o lo stato dell'inchiostro dei pennarelli.

Prima di procedere è opportuno introdurre brevemente le suddette funzioni e le costanti derivate dalle proprietà fisiche di Scribit e dalle varie scelte implementative con cui il file GCODE e le sue istruzioni vengono generati:

Funzioni:

- **`cToPolar()`:** utility di base per la conversione dei punti da coordinate cartesiane a polari
- **`distCalc()`** e **`drawnCalc()`:** calcolo della distanza della stampa
- **`init()`:** inizializzazione del file GCODE
- **`moveTo()`:** generazione spostamento del plotter senza disegnare
- **`eraseTo()`:** generazione spostamento del plotter senza cancellare
- **`generate()`:** generazione spostamento del plotter
- **`estTime()`:** calcolo della durata del disegno
- **`end()`:** chiusura del file GCODE e restituizione informazioni

Costanti:

- **`STEPTOMM`:** numero di step necessari ai due motori laterali, calettati con le pulegge, per avvolgere 1mm di filo (quindi spostare il plotter di un 1mm)

- **ZSTEPS:** numero di step necessari al motore centrale per compiere una rotazione di un 1°
- **STEPTODEG:** fattore di conversione tra step e gradi
- **DEGDIV:** posizione d'ingaggio del primo pennarello all'interno del tamburo
- **DEGDIFF:** posizione di riposo del primo pennarello all'interno del tamburo
- **DEGENGAGE:** posizione di scrittura del primo pennarello all'interno del tamburo
- **PAUSE:** pausa di default per l'istruzione G4 espressa in millisecondi
- **FASTSPEED:** feedrate preferito per la stampa
- **ERASESPEED:** feedrate preferito per la cancellazione
- **ERASETEMP:** temperatura del dischetto di ceramica per la cancellazione
- **PLOTTERW:** distanza tra le due pulegge
- **PLOTTERH:** distanza tra la puleggia e la punta del pennarello in posizione di scrittura
- **MAX_ACC:** accelerazione massima
- **MAX_FEED:** feedrate massimo
- **MAX_START_ACC:** accelerazione iniziale massima
- **JERK:** fattore di correzione per il tremolio

3.1.6.1 cToPolar

La prima funzione da introdurre è **cToPolar**, invocata per ogni punto che dovrà essere convertito in GCODE. La funzione converte il singolo punto da coordinate cartesiane a polari. Come già illustrato nel capitolo introduttivo, i due poli sono identificati nei due chiodi cui i cavi sono attaccati, e ad ogni puleggia è legata la coordinata radiale r pari alla distanza tra il chiodo e la puleggia corrispondente, ossia la lunghezza del cavo.

La funzione prende in ingresso le due coordinate (x, y) da convertire e la larghezza della parete ($CANVASW$): per prima cosa, le coordinate vengono convertite da punti tipografici a millimetri, poi tradotte in coordinate polari e restituite.

La conversione avviene secondo l'espressione (Eq. 10):

Equazione 10 - Calcolo della coordinata polare r

$$\begin{cases} r_X = \sqrt{(x - \frac{PLOTTERW}{2})^2 + y^2} \\ r_y = \sqrt{[(CANVASW - x) - \frac{PLOTTERW}{2}]^2 + y^2} \end{cases}$$

in cui r_X, r_Y sono le lunghezze dei due cavi. Il punto effettivo viene quindi così calcolato:

$$P(\sqrt{{r_X}^2 + PLOTTERH^2}, \sqrt{{r_Y}^2 + PLOTTERH^2})$$

3.1.6.2 Calcolo delle distanze

Le altre due funzioni chiamate all'interno di ogni altra che si occupa di generare istruzioni di spostamento sono deputate al calcolo della distanza compiuta dal plotter. Esse sono state chiamate **distCalc** e **drawnDistCalc** e differiscono dal tipo di distanza calcolata: la prima calcola l'intera distanza compiuta dal plotter durante la futura routine di stampa, la seconda solamente la distanza di scrittura effettiva. Le due informazioni vanno diversificate in quanto serviranno per due utilizzi differenti: la distanza totale servirà poi per calcolare la durata temporale della stampa, la seconda per sapere quanto inchiostro è stato orientativamente consumato da ogni pennarello, così da dare all'utente un'informazione utile alla sostituzione di esso. Il calcolo effettuato è identico, si tratta di una sommatoria delle distanze euclidee tra due punti cartesiani in millimetri, il corrente e il precedente; ciò che cambia sono i punti ad essere trattati in ogni funzione. Vedremo nelle funzioni successive quando queste due utility di calcolo verranno invocate e l'utilizzo che ne faremo. Unica sostanziale differenza sono le variabili in ingresso: la funzione di calcolo di distanza totale richiede il punto corrente e un flag per discriminare la routine da cui è stata invocata. Nel caso della routine di cancellazione, la distanza calcolata sarebbe relativa allo spostamento del plotter lungo il path di cancellazione, in cui non viene utilizzato ovviamente alcun colore. Le distanze e i gli ultimi punti trattati vengono salvati su variabili globali differenti: rispettivamente `distance` e `lastPoint`, `erasedDistance` e `lastErasedPoint`. La funzione di calcolo della distanza "disegnata" non richiede alcun flag, ma necessita dell'informazione relativa al colore corrente: non appena esso cambia, la sommatoria viene salvata come distanza relativa al colore che ha appena

terminato di disegnare, poi azzerata e assegnata al colore successivo. Illustriamo il processo a seguire, introducendo le tre variabili globali che vedremo utilizzate all'interno:

- **actualColor**: colore attuale
- **drawnDistance**: valore della distanza compiuta finora dal pennarello legato al colore attuale
- **drawnDistances**: array di dimensione 4 contenente distanze dei singoli pennarelli
- **lastDrawnPoint**: ultimo punto disegnato
- **newPoint**: punto di cui è stata appena richiesta la conversione

Tutte le variabili sono inizializzate a zero.

```
drawnCalc(pointX, pointY, color) {
      IF actualColor IS EQUAL TO 0 THEN
            actualColor = color
ELSE IS actualColor IS NOT EQUAL TO color THEN
      SWITCH actualColor {
            case 'm1':
                  drawnDistances[0] = drawnDistance
                  BREAK
            case 'm2':
                  drawnDistances[1] = drawnDistance
                  BREAK
            case 'm3':
                  drawnDistances[2] = drawnDistance
                  BREAK
            case 'm4':
                  drawnDistances[3] = drawnDistance
                  BREAK
            default:
                  BREAK
      }
        drawnDistance = 0
        actualColor = color
    ENDIF
IF lastDrawnPoint IS NOT null THEN
      drawnDistance = drawnDistance + distance(newPoint, lastDrawnPoint);
      lastDrawnPoint = newPoint;
```

```
ENDIF
```

3.1.6.3 init

La funzione *init* è la prima ad essere invocata dall'indice non appena il file GCODE e il rispettivo stream su cui poter scrivere sono stati creati. Il suo compito è quello di inizializzare il file con i settaggi che Scribit utilizzerà per la stampa: accelerazione massima, feedrate massimo, eccetera (la lista dei parametri è presente all'inizio del paragrafo riguardante questo modulo). Il valore da assegnare alle istruzioni GCODE relative ai settaggi (le istruzioni '**M**' viste nel paragrafo 2.2) è stato oggetto di cambiamento nel corso dei mesi in cui la configurazione del dispositivo veniva modificata molto spesso, trattandosi di esemplari realizzati in prototipazione rapida e utilizzando componentistica sempre diversa alla ricerca della soluzione più efficiente e stabile. Adesso che Scribit è in produzione, i parametri M non vengono più modificati; l'unico dato sempre diverso all'interno del metodo non dipende da Scribit, ma dalla parete. Fa parte dei settaggi infatti anche il comando **G92** che imposta la posizione iniziale del plotter decidendo che tutte le altre posizioni saranno indipendenti da essa, trattando quindi gli altri punti come assoluti. In alternativa, utilizzando il comando G91 il punto iniziale sarebbe il riferimento di tutti gli altri punti a seguire che verrebbero interpretati come relativi. Infine la posizione iniziale viene impostata come primo punto da cui far partire il calcolo della distanza totale.

3.1.6.4 moveTo

Il nome di questa funzione equivale al comando SVG che la invoca: il comando di moveTo esprime lo spostamento della penna in una nuova posizione senza descrivere una particolare traiettoria in quanto **non sta disegnando**. In questa funzione bisognerà quindi impostare lo spostamento del plotter in posizione e prepararlo all'inizio dell'esecuzione di un nuovo tracciato. Verosimilmente, dopo init, questa sarà la prima funzione generativa ad essere invocata, dato che ogni tracciato inizia sempre un comando di moveTo. La funzione prende in ingresso, tra gli altri parametri, il punto in cui spostare il plotter e il riferimento al colore attivo. Il punto, oltre a dover essere convertito in coordinate polari, serve per il calcolo sia della distanza totale che della distanza "disegnata". Nel secondo caso, però, lo spostamento dall'ultimo punto noto a quello attuale **non deve essere considerato**. Il riferimento all'ultimo punto noto viene quindi settato a *null*: come descritto nel codice al paragrafo 3.1.6.2 la

distanza dei tracciati viene calcolata solo se l'ultimo punto noto esiste: quindi, in questo caso, il punto di `moveTo` verrà solo assegnato come ultimo punto e il calcolo riprenderà nella funzione generativa successiva.

Prima di generare il punto in coordinate polari e quindi raggiungerlo in stampa con il plotter, decidiamo di gestire la posizione del pennarello da usare. In base al colore attivo e alla sua ultima posizione all'interno del tamburo, calcoleremo l'angolo in cui il pennarello dovrà posizionarsi. Ad ogni colore è associato un identificativo numerico compreso tra 1 e 4 inclusi che si riflette su una variabile globale, **`currentTool`**, che ne conserva appunto il riferimento.

Per impartire le corrette istruzioni al robot in merito al pennarello da usare abbiamo chiaramente bisogno sia dell'ultimo colore usato, sia di quello che vorremo usare in questo nuovo tracciato. Il tamburo può sempre girare in senso antiorario, ma deve compiere una rotazione in senso orario per ingaggiare i pennarelli. Gestire male queste misure significherebbe bloccare la rotazione e probabilmente danneggiare il tamburo e il motore che lo governa. Di seguito elenchiamo quindi le tre situazioni tipiche del cambio pennarello:

- Se **`id – currentTool > 0`** ci troviamo nella posizione di dover ingaggiare un nuovo pennarello per la prima volta. Ciò significa che il marker richiesto si trova a una o più posizioni successive rispetto a quella corrente. Bisogna quindi ruotare il tamburo in senso antiorario portandolo alla posizione d'ingaggio del nuovo pennarello, poi farlo ruotare ancora ma in senso orario per portare il pennarello in posizione di riposo, pronto per la scrittura non appena verrà richiesta. Chiamando deg l'angolo di riferimento e pensando al tamburo come una circonferenza goniometrica:

Figura 25 - Tamburo e ordine dei pennarelli

```
IF id – currentTool IS GREATER THAN 0 THEN
```

```
    deg = (id + 1) * DEGDIV; //engage
    CALL writeGCODE('G1 Z[deg]');
    CALL writeGCODE('M400');
    deg = deg – DEGDIFF;    //rest
    CALL writeGCODE('G1 Z[deg]');
    CALL writeGCODE('M400');
ENDIF
```

Se `currentTool` è pari a `0` significa che il tamburo si trova in posizione iniziale, quindi con tutti i pennarelli inattivi. In questo caso ci troveremo sicuramente nella porzione di codice relativa al primo colore, '`m1`' con `id = 1`.

- Se **`id – currentTool == 0`** il colore richiesto è lo stesso utilizzato fino ad ora. L'unica cosa da fare è quindi spostare il pennarello dalla posizione di scrittura a quella di riposo, così da accertarci che Scribit non scriva durante il tragitto verso il punto d'inizio del nuovo tracciato:

```
deg = deg + DEGENGAGE
```

- Infine, se **`id – currentTool < 0`** è stato richiesto un pennarello che si trova in una posizione precedente all'attuale. La posizione dei markers suggerirebbe che la soluzione più veloce sia effettuare una rotazione diretta in senso orario, ma ciò è impossibile poiché, considerando la conformazione del tamburo ed essnedoci un pennarello ingaggiato, il penholder non uscirebbe dalla guida e anzi, forzerebbe il fine corsa potendo anche danneggiare anche l'apparato. Decidiamo quindi di tornare in posizione iniziale:

```
CALL writeGCODE("G1 Z360”);
CALL writeGCODE('M400');
CALL writeGCODE("G92 Z0"); //reset initial position
```

e ripetere i passi del caso **`id – currentTool > 0`**.

Una volta superati i casi appena elencati, viene aggiornato il valore di `currentTool` e posto uguale a `id`.

Viene quindi convertito il punto da cartesiano a polare e scritto sullo stream. Una volta giunto a destinazione, Scribit può finalmente scrivere: se currentTool è diverso da zero siamo certi

che un pennarello si trova in posizione di riposo, lo spostiamo quindi in posizione di scrittura sottraendo a deg, appena calcolato in uno dei 3 casi precedenti, il valore `DEGENGAGE` che ruoterà il marker in senso orario portandolo a fine corsa e facendo uscire la punta di esso dal foro frontale.[11]

3.1.6.5 eraseTo

EraseTo è il corrispettivo di moveTo per la cancellazione. Quando il modulo commandParse viene invocato dalla routine di cancellazione, le istruzioni `M` del pattern di cancellazione invocheranno questa funzione. I settaggi base di Scribit sono identici per entrambe le routine, ma su eraseTo viene impostato l'unico aggiuntivo, la temperatura del disco in ceramica. Quando viene chiamata la funzione eraseTo il punto d'inizio del tracciato di cancellazione viene prima utilizzato per il calcolo della distanza del suddetto tracciato; poi, viene convertito in coordinate polari e scritto sullo stream dedicato. Una volta arrivato al punto in cui iniziare a cancellare, il plotter dovrà fermarsi e attendere il raggiungimento della temperatura di cancellazione prima di proseguire: scriviamo quindi sullo stream una stringa del tipo "**`M109 S[ERASETEMP]`**", cui seguiranno tutti gli altri punti del tracciato generati con il metodo che ci accingiamo ad illustrare nel paragrafo che segue.

3.1.6.6 generate

La funzione generate è deputata alla generazione dei punti dei tracciati che Scribit dovrà disegnare. Prende in ingresso le **coordinate cartesiane** di un solo punto e lo **stream** su cui scrivere insieme a un ***flag*** che rappresenta la routine a cui ci riferiamo: di disegno o di cancellazione.

Nel primo caso, la funzione semplicemente calcolerà sia la distanza di spostamento che di disegno del plotter, e dopo convertirà il punto dato in coordinate polari scrivendolo sullo stream dedicato sotto istruzione di movimento lineare `G1`. Nel secondo caso avremo la necessità di dividere i tracciati in arrivo; questi, come vedremo nel paragrafo 3.1.7 relativo al alla cancellazione, saranno comandi di tipo **H** e **V** (*horizontal e vertical line to*) e ricopriranno il disegno grazie al calcolo degli estremi che abbiamo descritto precedentemente. In fase di

[11] La necessità di scrivere il comando "M400" dopo ogni istruzione relativa al tamburo è quella di attendere che ogni rotazione sia completa prima di avanzare all'istruzione successiva, in quanto anche una piccola incertezza potrebbe non permettere al pennarello di entrare in sede e far funzionare il meccanismo.

testing è stato deciso che il plotter, piuttosto che compiere l'intero path a velocità limitata, dovrà **fermarsi periodicamente** per t secondi lungo il tracciato, per essere certi di aver cancellato l'area sottesa dal dischetto in ceramica. Il tracciato orizzontale viene quindi diviso in segmenti più piccoli, in numero pari alla distanza tra il punto attuale e il precedente (purché sullo stesso segmento orizzontale, quindi a condizione che entrambi condividano la stessa coordinata y) fratto il diametro della ceramica; quindi la coordinata x di ogni punto intermedio avanzerà del rapporto tra la distanza e l'ultimo valore calcolato. Illustriamo l'algoritmo:

```
IF erasing routine THEN
        erasedPoint = lastErasedPoint
        CALL distCalc()
        IF currentPoint_y EQUALS TO erasedPoint_y THEN
            SET dst TO distance between actual point and current point
            SET interpolation TO ceil(dst/ceramic)
            SET diff TO currentPoint_x – erasedPoint_x
            SET factor TO dst/interpolation
            IF currentPoint_x > erasedPoint_x THEN
                  direction = true
            ELSE direction = false
            ENDIF
            currentPoint_x = erasedPoint_x
            FOR (i = 0; i < interpolation; i++) {
                IF direction THEN
                  currentPoint_X = currentPoint_X + factor
                ELSE currentPoint_x = currentPoint_x – factor
                ENDIF
                SET point = CALL cToPolar()
                CALL writeGCODE("M17")
                CALL writeGCODE("G1 X[point_x] Y[point_y]")
                CALL writeGCODE("M18")
                CALL writeGCODE("M109 S[ERASETEMP]")
                CALL writeGCODE ("G4 P[ERASEPAUSE]")
                CALL writeGCODE("M104 S0")
            ENDFOR
ENDIF
```

3.1.6.7 Stima del tempo

La utility di stima del tempo estTime viene utilizzata per restituire all'utente l'informazione relativa alla durata del disegno scelto. Prende in ingresso il solo *flag* relativo alla routine per discriminare la **velocità** del plotter e la **distanza** compiuta. Se chiamato al termine della routine di cancellazione, verrà considerata la distanza compiuta dal plotter del rispettivo pattern e ERASESPEED; alternativamente, nel caso principale, si considererà la distanza totale (spostamenti e scrittura) del disegno e FASTSPEED. Il calcolo deriva dalla basilare espressione fisica $v = {}^{s}/_{t} \rightarrow t = {}^{s}/_{v}$, in cui s è la distanza e v la velocità del plotter. Considerando il numero di istruzioni M o m ricevute, salvate in una variabile globale, viene aggiunto al calcolo il prodotto tra il suddetto conteggio e una costante empirica che rappresenta la media in secondi del tempo necessario a inserire e disinserire il pennarello dalla posizione di scrittura ogni volta che viene richiesto (quindi, ogni volta che viene invocata la funzione moveTo).

3.1.6.8 end

La funzione è l'ultima del modulo ad essere chiamata, dopo di questa il modulo chiamante chiederà la **chiusura dello stream** e lo script si esaurirà. Ad essa si deve la scrittura sullo stream delle istruzioni GCODE che serviranno a Scribit per tornare in posizione iniziale resettando i parametri modificati durante il cammino. Prende in ingresso il *flag* di cancellazione. Se vero, viene scritta l'istruzione di **abbassamento della temperatura**, "`M104 S0`", altrimenti viene riportato il tamburo in **posizione iniziale** tramite "`G1 Z360`". In entrambi i casi, dopo queste istruzioni, il plotter dovrà tornare in posizione iniziale tramite "`G92 X[homeX] Y[homeY]`". Il punto di partenza e di arrivo viene quindi riutilizzato per calcolare nuovamente la distanza per l'ultimo spostamento di Scribit, per l'ultima volta nella routine. A questo punto viene invocata la funzione del calcolo di stima temporale e restituite le informazioni relative al tempo e alle distanze compiute dai pennarelli.

3.1.7 Routine di cancellazione

Sono due i moduli legati alla routine di cancellazione: `eraser` e `batchEraser`. Il primo si comporta da **indice** della routine, il secondo genera il **pattern di cancellazione** vero e proprio. Come abbiamo illustrato a grandi linee nella panoramica dell'algoritmo e nel paragrafo relativo all'indice della funzione, `eraser` viene chiamato dopo la chiusura dello stream del

GCODE di stampa se è stata richiesta anche la cancellazione. La funzione omonima, unica del modulo, prende in ingresso le impostazioni del muro, di Scribit, il nome del file e gli estremi del disegno, precedentemente calcolati dal modulo commandParse.
Il flusso delle operazioni è praticamente identico a index dopo il trattamento preliminare del file SVG: viene creato il file GCODE inizialmente vuoto (con nome "`[filename]_erase`") e il suo *stream*, il file viene inizializzato (`gCodeParse.init`), estratti i comandi dall'SVG, convertiti in GCODE ed infine, salvato il file e chiuso lo stream. Unica differenza è nell'estrazione dei comandi SVG: non c'è nessun file da leggere in questo caso, l'attributo **d** viene generato direttamente da batchEraser. La funzione considera gli estremi del disegno e la dimensione della parete. Il primo comando ad essere scritto è chiaramente un moveTo: decidiamo di iniziare la cancellazione nel **punto più alto a sinistra** (*top-left*) del disegno, quindi "`M [leftEnd],[upEnd]`". L'idea è quella di far creare un percorso formato da linee rette che coprano il disegno per avere la certezza di cancellarlo interamente. Si tratterà quindi di generare **tracciati orizzontali** da un estremo all'altro scostati verticalmente del **raggio** del dischetto in ceramica. Infine, la nuova grafica andrà traslata in alto della distanza tra il pennarello e il centro della ceramica, posto più in basso rispetto al pennarello: siamo costretti a compiere quest'operazione in quanto, come sappiamo, la posizione di Scribit viene sempre calcolata relazionando la lunghezza dei fili delle due pulegge con la posizione del pennarello in scrittura. Detti `markerOffset` la distanza tra il pennarello e il dischetto e `offset` il diametro di quest'ultimo, a seguire illustriamo l'algoritmo utilizzato:

```
SET d TO "M [leftEnd] , [upend]"
SET iterations TO ceil((downEnd – upend) / (offset));
FOR (i = 0;  i < iterations; i++)
  SET d TO d + "H[rightEnd] v[offset] H[leftEnd] v[offset]
ENDFOR
SET d TO d + "H[rightEnd]
CALL translateY(d, –markerOffset)
```

Ciò che ne risulta, è un tracciato che ricopre interamente la grafica selezionata, posto leggermente più in alto del disegno in quanto la "penna" è rappresentata dalla cercamica e non dal pennarello (Fig. 26):

Figura 26 - Grafica con rispettivo pattern di cancellazione sovrapposto (in rosso)

3.2 TextToSVG

L'algoritmo prende in ingresso il testo da convertire e il **font** da utilizzare insieme alle informazioni relative all'utente per salvare l'SVG generato in un'area dedicata. La routine viene richiesta direttamente dall'app e ha luogo in una funzione remota di Google Cloud Platform. La conversione vera e propria avviene tramite utilities provenienti da ***opentype.js***, libreria Javascript open source utilizzata per analizzare e scrivere font OpenType e TrueType su browser o applicazioni Node. Permette di recuperare le informazioni di forma dei singoli caratteri accedendo direttamente ai file .ttf e .otf che definiscono il font, funzione anche della dimensione del font stesso e chiaramente del loro posizionamento nel canvas html o SVG.

L'unica vera eventualità da gestire è mandare il **testo a capo** quando necessario, onde evitare di generare un SVG sviluppato solo in larghezza che, una volta scalato alla risoluzione della parete, ne riempirebbe una sezione molto limitata. Il testo in ingresso viene quindi suddiviso in **parole** e impostato un valore massimo di **caratteri per riga**: non appena si presenterà una parola che porterà il conteggio dei caratteri della riga a superarne il limite consentito la riga verrà allocata, il *counter* resettato e la parola attuale inserita in una nuova riga. Ottenute tutte le righe costituenti il testo, sarà chiesto a OpenType di generare un file SVG convertendo il

testo della prima riga in tracciato; il punto più alto a sinistra del testo partirà dal punto (0,0) del canvas e si svilupperà con un'altezza pari alla **dimensione del font** in punti tipografici (fissata a 64 pt) e in larghezza con la somma delle larghezze delle singole lettere per il numero di esse (ogni lettera di font non monospaziati presenta una larghezza diversa). I valori di x e y (pari all'attuale `width` e `height`, larghezza e altezza del canvas) vengono salvati: la x verrà utilizzata alla fine, la y invece viene incrementata della metà della dimensione del font, così da porre il punto iniziale della riga successiva a (0, y). Le righe successive alla prima vengono generate come semplici tracciati incrementando la y ogni volta. Finite le righe, gli attributi `width` e `height` dell'SVG generato precedentemente vengono modificati rispettivamente con il valore massimo della larghezza di ogni riga, così da inserirle tutte all'interno del canvas, e con la y che è stata incrementata a ogni generazione. Tutti gli altri tracciati vengono appesi sotto il primo con cheerio/jQuery.

Di seguito, l'algoritmo di organizzazione delle righe:

```
ARRAY lines[]
ARRAY x[]
SET y TO 0 //height
SET index TO 0 //line index
SET l TO "" //phrase included in actual line
PUSH l INTO lines //first line is empty when calling the algorithm
FOR EACH word IN text
      IF (lines[index] IS EQUAL TO "" OR text length IS EQUAL TO 1)
            l = word; //if text length is 1 it contains a single word
      ELSE l = lines[index] + " " + word
      ENDIF
      IF (l length IS GREATER THAN maximum_chars)
            PUSH word INTO lines
            index++
      ELSE lines[index] = l
      ENDIF
ENDFOR
FOREACH line IN lines
      IF (index IS EQUAL TO 0)
            CALL createSVG(line, 0, 0)
            GET width AND height FROM line
            PUSH width INTO x
      ELSE
```

```
            y = y + height + fontSize/2
            CALL createPath(line, 0, y)
            APPEND path IN svg
            GET width AND height FROM line
            PUSH width INTO x
            IF (index IS EQUAL TO lines length -1)
                  GET maximumX FROM x
                  SET svg width TO maximumX
                  SET y TO y + height
                  SET svg height TO y
            ENDIF
      ENDIF
ENDFOR
```

3.3 Creazione di Template

I template per il **meteo** e **Twitter** sono stati creati dal team di UX e UI di Scribit. Si tratta di file SVG di 1024x1024px che presentano dei tracciati **costanti**, ad esempio le intestazioni e i loghi di Twitter, ed altri **variabili**. Essi andranno generati in base alle preferenze dell'utente: la **città** per cui sono richieste le previsioni metereologiche, un **hashtag** da cui recuperare gli ultimi tre tweet, i tweet stessi.

Figura 27 – Previsioni metereologiche settimanali. Icone, date, città e temperature sono generat dinamicamente.*

L’algoritmo è molto semplice: ottenuta la *query* contenente i dati recuperati dal provider meteo o da Twitter, viene caricato in memoria il **template** SVG corrispondente che contiene i soli tracciati costanti. Per generare i testi variabili è stato creato un file JSON per template contenente dimensione del font preferita di ogni singolo testo e la sua posizione di partenza (top-left) su cui basare la generazione: il file JSON corrispondente viene quindi anch’esso caricato in memoria e interrogato volta per volta per generare ogni testo nella posizione e grandezza corrette sempre utilizzando OpenType.

Il limite di questo approccio è la bassa libertà di manovra in presenza di testi lunghi che potrebbero invadere lo spazio di altri elementi; per ciò, è stato deciso di allineare tutti i contenuti a sinistra mantenendo così l’approccio top-left di generazione dei testi e basando la grandezza del font sul *worst case*. Considerando la lunghezza variabile dei tweet (1 – 280 caratteri) è stato invece deciso di calcolare il migliore approccio in termini di *fontsize* e numero massimo di caratteri da inserire per riga in base all’effettivo numero di caratteri cui il tweet è composto: un testo più breve sarà più grande e sviluppato in meno righe, per un testo lungo varrà invece il contrario (Fig. 27).

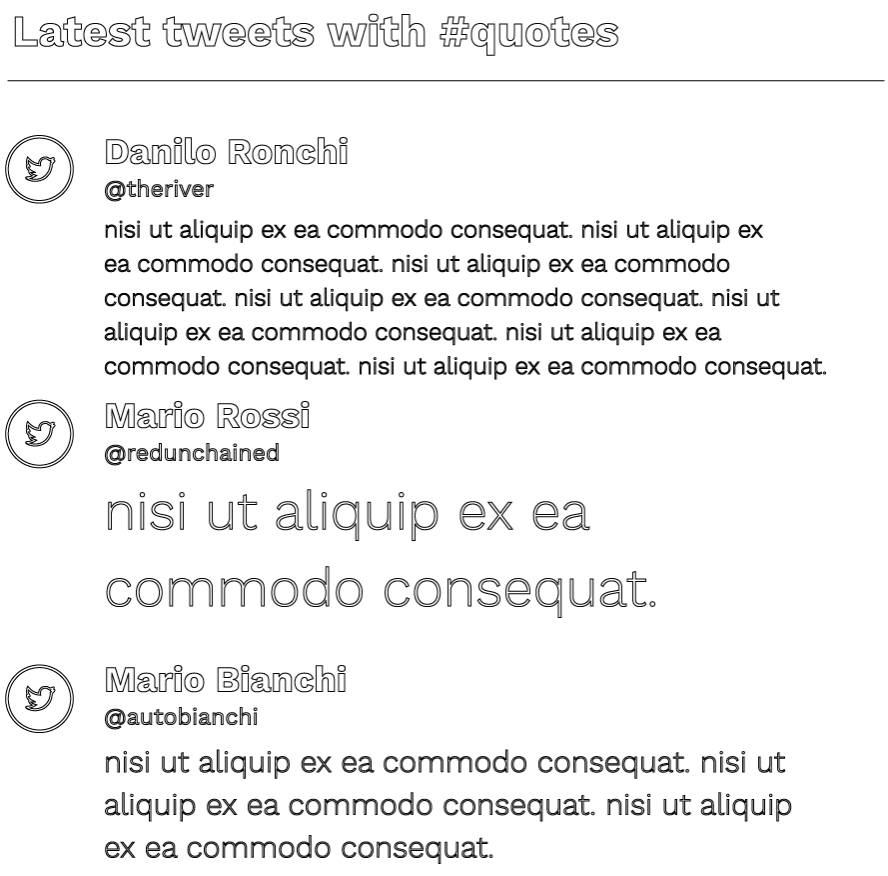

Figura 28 - Ultimi tre tweet sull'hashtag "quotes". Tweet e hashtag sono generati dinamicamente.

4. Conclusioni e sviluppi futuri

A livello puramente algoritmico il trattamento dei file SVG e la conversione in istruzioni NC hanno restituito ottimi risultati: ogni conversione effettuata tramite cloud function impiega da un secondo scarso a circa 3, dipendendo dalla complessità del file richiesto per la conversione (ogni tracciato è una stringa, ogni attributo della stringa diviso in array, tutti i comandi dell'attributo d esportati in altrettanti array, e così via). Il disegno risultante è anche molto accurato e esteticamente piacevole; si riesce ad apprezzare una precisione alla seconda cifra decimale di ogni coordinata.

Figura 29 - Omaggio a Salvador Dalì realizzato da Scribit su una vetrata

Il limite più grande riguarda il calcolo del punto polare corrispondente al cartesiano rappresentato sul file: le equazioni illustrate nel paragrafo relativo alla conversione da coordinate cartesiane a polari (3.1.6.2) convertono il punto rispetto al **baricentro** del plotter, che è posto inclinato e più in basso rispetto al pennarello, vero e proprio responsabile del disegno su cui bisognerebbe centrare la coordinata. Inoltre in presenza di linee rette si può palesemente notare come il pennarello risenta pesantemente dell'inclinazione di Scribit nello spostamento tra gli estremi sinistro e destro della parete: esercitando il filo più corto maggiore tensione il plotter tende ad inclinarsi, modificando conseguentemente la posizione del pennarello che, immaginando uno spostamento da sinistra a destra, inizierà a disegnare più in alto del dovuto, poi andrà ad abbassarsi avvicinandosi al centro (dove la posizione del marker sarebbe corretta) e alzarsi ancora spostandosi sulla destra (*distorsione a cuscino*).

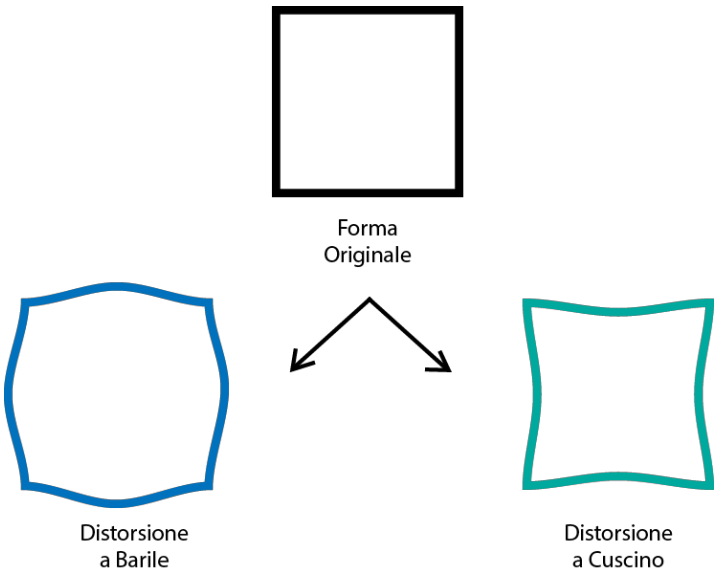

Figura 30 - Distorsione ottica

Questo limite, che prescinde dalla conversione, è stato risolto matematicamente attraverso un sistema di equazioni non lineari che, trattando la coordinata angolare del plotter (ossia l'angolo formatosi tra il cavo, coordinata radiale, e il piano tracciato sul baricentro del plotter) forniscono i fattori di distorsione introdotti dall'inclinazione del robot, così da poterla eliminare nel calcolo della coordinata polare reale del pennarello. Andrà creata una matrice di forze e angoli per ogni ampiezza superficiale consentita dall'applicazione mobile (da 2 a 4 metri di larghezza con passo 0.25 m tra una soluzione all'altra) e utilizzata sotto forma di file JSON dall'algoritmo di conversione. I fattori di annullamento della distorsione contenuti nel

file saranno relativi a posizioni generiche e discrete, ipotizzando ad esempio uno spostamento di 10 mm su ogni singola coordinata ad ogni calcolo; i valori relativi alla posizione reale del plotter andranno quindi interpolati. Il nuovo algoritmo è attualmente in fase di test e sta restituendo degli ottimi risultati (Fig. 30).

Figura 31 - Sopra: linea orizzontale distorta. Sotto: stessa linea con l'algoritmo di compensazione della distorsione

Altro aspetto che verrà sicuramente migliorato riguarda il peso del file GCODE. Come già illustrato le curve di Bèzier vengono interpolate in segmenti retti moltiplicando a dismisura le istruzioni necessarie per descrivere la curva. Inoltre, per gestire la distorsione, le linee rette vengono a loro volta suddivise con passo di 1 cm per calcolare il fattore di correzione durante il tragitto. Verranno effettuati dei test provando a descrivere le curve di Bèzier tramite comando **G5** che, come per le curve SVG di tipo C (spline cubiche di Bèzier), prende in ingresso punto finale e punti di controllo della curva, lasciando l'interpolazione al firmware.

Calcolando il tempo stimato di stampa si evince anche che il tempo che il plotter passa effettivamente a disegnare è solo una porzione del tempo totale: ciò accade in quanto molto spesso i punti d'inizio dei nuovi tracciati (comandi **M**) si trovano molto distanti tra loro, quindi il robot dovrà raggiungerli prima di disegnare realmente, e magari anche per un tempo breve in relazione a quello impiegato per raggiungere il nuovo punto iniziale. Verranno studiati degli algoritmi che permettano quindi una riorganizzazione del file SVG in modo da avvicinare il punto finale di un tracciato al primo punto del successivo, così da minimizzare la distanza e quindi l'attesa tra un tracciato e l'altro.

Per ciò che riguarda gli altri aspetti del prodotto, legati soprattutto all'usabilità da parte dell'utente tramite app, il team è alle battute finali per finalizzare la piattaforma MQTT e il firmware finale, che comunque riceverà periodicamente aggiornamenti OTA. L'app è già presente sugli store, ma allo stato attuale mancano tutti i widget: si progetta di aggiornarla con i riferimenti alle nuove funzioni remote per la generazione testuale, del meteo e dei tweet nella prossima release prevista per la primavera 2019. Nella successiva, si progetta di rendere disponibile anche la conversione da raster a SVG, di cui parte del team IT si sta tutt'ora occupando, e si desidera anche dare agli utenti la possibilità di utilizzare i propri design vettoriali per la stampa. Il lavoro di estrazione dei tracciati dalle forme illustrato nel paragrafo 3.1.2 è stato eseguito pensando alla futura necessità di rendere adatto alla conversione potenzialmente qualunque file SVG ricevuto. Le immagini che Scribit metterà a disposizione degli utenti all'interno della piattaforma sono state realizzate ad-hoc per l'applicazione, presentando solo linee e non figure piene, con più tracciati espansi possibili per velocizzare al massimo il processo di conversione: non è detto che altri utenti vorranno fare lo stesso con i propri design. Verrà comunque creata una guida che permetterà a chiunque vorrà creare un'immagine per il proprio Scribit di farlo nel migliore dei modi evitando di restare delusi del risultato, considerando i limiti strutturali della macchina.

5. Bibliografia

[1] **Aprea V.**, **Grzymkowski P.** (2001), "*VP Squared – Vertical Plotter Solutions*", https://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2001/vp2/

[2] **Lehni J.** (2000 - 2016), "*Works*", http://juerglehni.com/works,
"*Hektor*", http://juerglehni.com/works/hektor,
"*Rita*", http://juerglehni.com/works/rita,
"*Viktor*", http://juerglehni.com/works/viktor,
"*Otto*", http://juerglehni.com/works/otto

[3] **Domus** (2011), "*Open Source Architecture (OSArc)"*, https://www.domusweb.it/en/opinion/2011/06/15/open-source-architecture-osarc-.html
Carlo Ratti Associati (2012), "*OSArc Manifesto*", https://www.youtube.com/watch?v=1po4k1BfXB4

[4] **Wikipedia** (2018), "*Distribuzione Desmodromica*", https://it.wikipedia.org/wiki/Distribuzione_desmodromica

[5] **Wikipedia** (2018), "*Internet delle Cose*", https://it.wikipedia.org/wiki/Internet_delle_cose

[6] **W3C** (2018), "*Scalable Vector Graphics (SVG) 2*", https://www.w3.org/TR/SVG2/intro.html - AboutSVG

[7] **Bottino A.**, "*Immagini Grafiche*", Dispense del corso di "Computer Grafica", Politecnico di Torino, 2017

[8] **W3C** (2018), "*Document Structure*", https://www.w3.org/TR/SVG2/struct.html

[9] **W3C** (2018), "*Paths*", https://www.w3.org/TR/SVG2/paths.html

[10] **W3C** (2018), "*Text*", https://www.w3.org/TR/SVG2/text.html

[11] **W3C** (2018), "*Paths*", https://www.w3.org/TR/SVG2/shapes.html

[12] **Fabbri F.**, "*Trasformazioni geometriche nel piano: dalle isometrie alle affinità*", Istituto d'Istruzione Superiore "Marie Curie", 2014

[13] **W3C** (2018), "*Paths*", https://www.w3.org/TR/SVG2/coords.html

[14] **Wikipedia** (2019), "*G-Code*", https://en.wikipedia.org/wiki/G-code

[15] **Marlin** (2019), "*Documentation – G-code*", http://marlinfw.org/meta/gcode/

[16] **Wikipedia** (2019), "*NoSQL*", https://it.wikipedia.org/wiki/NoSQL

[17] **Fielding R. T.**, "*Architectural Styles and the Design of Network-based Software Architectures*", Doctoral dissertation, University of California, Irvine, 2000

[18] **Wikipedia** (2018), "*SOAP*", https://it.wikipedia.org/wiki/SOAP

[19] **Cordiano S.** (2019), "*Che cosa sono i microservizi?*", https://www.salvatorecordiano.it/che-cosa-sono-i-microservizi/

[20] **Nipper A.**, **Stanford-Clark A.** (1999 - 2014), "*MQTT*", http://mqtt.org/

[21] **Wikipedia** (2019), "*Publish–subscribe pattern*", https://en.wikipedia.org/wiki/Publish–subscribe_pattern

[22] **Grance T.**, **Mell P.**, "*The NIST definition of Cloud Computing*", NIST Special Publication, National Institute of Standards and Technology, Gaithersburg, 2011

[23] **Maisonobe, L.**, "*Drawing an elliptical arc using polylines, quadratic or cubic Bézier curves*", http://spaceroots.org/, 2003