Politecnico di Torino

Politecnico di Torino

Ingegneria Informatica

A.a. 2024/2025

Sessione di laurea Marzo/Aprile 2025

Progettazione e sviluppo di un sistema di Version Control per il Creative Coding

Relatori:

Luigi De Russis

Juan Pablo Sáenz Moreno

Candidato:

Michele Pistan

Ringraziamenti

Ormai in vista del traguardo, della conclusione di un altro capitolo di vita, è il momento di ringraziare tutti coloro che mi hanno spronato fino a questo obiettivo, o mi hanno incoraggiato, dato una mano, consolato, accompagnato per qualche tratto, coloro che mi hanno ostacolato e fatto riflettere sul mio percorso, e coloro che mi hanno regalato un sorriso o, con un semplice gesto, senza accorgersene, mi hanno migliorato una giornata.

Grazie alla mia famiglia. Per tutto. Grazie agli amici che mi sono rimasti accanto, a prescindere dalla distanza fisica. Grazie a Luigi De Russis e Juan Pablo Sáenz Moreno, i relatori di questa tesi, per l'aiuto, i consigli e la pazienza. Grazie a tutti i colleghi con cui ho intrecciato il mio percorso universitario, i compagni di banco e di studio, chi ha collaborato con me nei lavori di gruppo, chi con me ha condiviso i pranzi in mensa o i viaggi in treno, per fare tutti i nomi servirebbe aggiungere qualche pagina. Grazie a tutti gli insegnanti che hanno provato a trasmettermi un po' della loro conoscenza. Grazie a chi ha condiviso con me l'esperienza di volontariato, agli animatori dell'oratorio, a tutta l'AC e in particolare a chi si è avvicendato nell'equipe giovani. Grazie alla musica, al mio pianoforte, ai sax e a chi ha suonato con me in questi anni. Grazie a te che sei arrivato a leggere fino a qui perché, anche se forse io non me ne sono accorto - e capita spesso, credimi -, forse in fondo un po' mi vuoi bene.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Come nasce un'opera d'arte? Come *si fa* l'arte? Cosa rende un manufatto un'*opera d'arte*? A quest'ultima domanda non credo di poter rispondere, non certo in questa sede, almeno. Le prime due mi hanno sempre affascinato.

Se proviamo a pensare a un'opera d'arte ci vengono in mente sicuramente grandi capolavori, perfetti e studiati in tutti i minimi dettagli, che rappresentano un punto d'arrivo e di non ritorno, che non ci permetteremmo mai di pensare di voler modificare. O no?

E allora perché, nonostante le differenze, la *Vergine delle rocce* del Louvre e quella della National Gallery di Londra sono entrambe considerate capolavori della pittura? E invece, cos'ha la *Gioconda* del Louvre in più di quella del Prado di Madrid? (Figura 1.1)

Si comincia a capire che l'arte non sta solo nel risultato finale: se oggi un artista ridipingesse la Gioconda - e non è impossibile, le accademie d'arte sono piene di persone in grado di riprodurla perfettamente - non sarebbe per questo considerato al livello di Leonardo da Vinci.

L'opera d'arte non è altro che il risultato di un processo creativo - neanche per forza il risultato finale, è quello che l'artista sceglie di far vedere al pubblico -, un processo fatto di prove e tentativi, idee avute prima degli altri o sviluppate meglio, ma anche idee approcciate con entusiasmo e poi abbandonate. Per restare con Leonardo, come non citare i suoi bellissimi disegni preparatori, conservati nei suoi famosi codici.

Avvicinandosi nel tempo, l'ultimo dipinto della *Montagne Sainte-Victoire* di Paul Cezanne (Figura 1.2) perderebbe sicuramente gran parte del suo significato se non tenessimo in considerazione le "versioni" precedenti, notando così l'evoluzione del processo di semplificazione e astrazione delle forme che porterà al Cubismo.

Forse questo processo è più comprensibile nell'architettura, dove è più facile immaginare che prima della costruzione di un edificio ci siano state una o più

Figura 1.1: A sinistra: *La Gioconda* conservata nel Museo del Louvre a Parigi; a destra quella conservata nel Museo del Prado, a Madrid, attribuita alla bottega di Leonardo da Vinci

idee, un progetto dettagliato, uno o più modellini e studi preliminari e infine, eventualmente, modifiche in corso d'opera per le più svariate ragioni.

Allontanandoci dalle arti visive, un caso esemplare perché completamente pubblico nei suoi snodi principali è quello dei *Promessi Sposi* di Alessandro Manzoni, pubblicato dapprima come *Fermo e Lucia* e poi rimaneggiato e ripubblicato altre due volte, prima nel 1827 e poi, definitivamente, solo nel 1840.

L'arte dunque è una continua evoluzione, ma è anche strettamente legata allo sviluppo tecnologico dell'epoca in cui nasce. Così come l'introduzione di strumenti come il compasso e la camera oscura ha aiutato artisti come Filippo Brunelleschi e Leon Battista Alberti a sviluppare tecniche di prospettiva lineare, permettendo rappresentazioni tridimensionali più realistiche su superfici bidimensionali, così come l'invenzione dei tubetti di colore - e quasi in contemporanea della fotografia, che ha reso quasi inutile il lavoro dei ritrattisti - ha dato una spinta decisiva alla nascita dell'impressionismo, nell'ultimo mezzo secolo l'arte è sicuramente stata influenzata dalla nascita e dallo sviluppo dell'informatica.

Uno dei modi in cui ciò è più evidente è il *creative coding*: gli artisti prendono gli

Figura 1.2: Alcuni dei dipinti della serie *La Montagne Sainte-Victoire* di Paul Cezanne, che evidenziano la progressiva semplificazione delle forme

strumenti dell'informatica - algoritmi e linguaggi di programmazione, elaborazione tramite calcolatori elettronici, ecc. - e li sostituiscono a tela e pennelli per realizzare le loro opere. Con il tempo questa "corrente artistica" si è evoluta, e sono nati software dedicati appositamente a questa pratica. Recentemente i creative coders sono finiti sotto la lente d'ingrandimento della comunità scientifica, in particolare dei ricercatori in Human-Computer Interaction perché il loro approccio agli strumenti informatici aveva qualcosa di diverso da quello tradizionale.

Come cercherò di spiegare nel corso di questa tesi, si è visto che una delle differenze principali risiede proprio nel processo, dal momento che gli artisti hanno portato nel contesto della programmazione il processo creativo dell'opera d'arte che, con la sua natura esplorativa e iterativa, si contrappone al modello problema-analisi-soluzione tipico del mondo ingegneristico.

Si è visto che il codice poteva funzionare perfettamente per creare arte, ma che alcuni degli strumenti pensati per supportare i programmatori non erano altrettanto

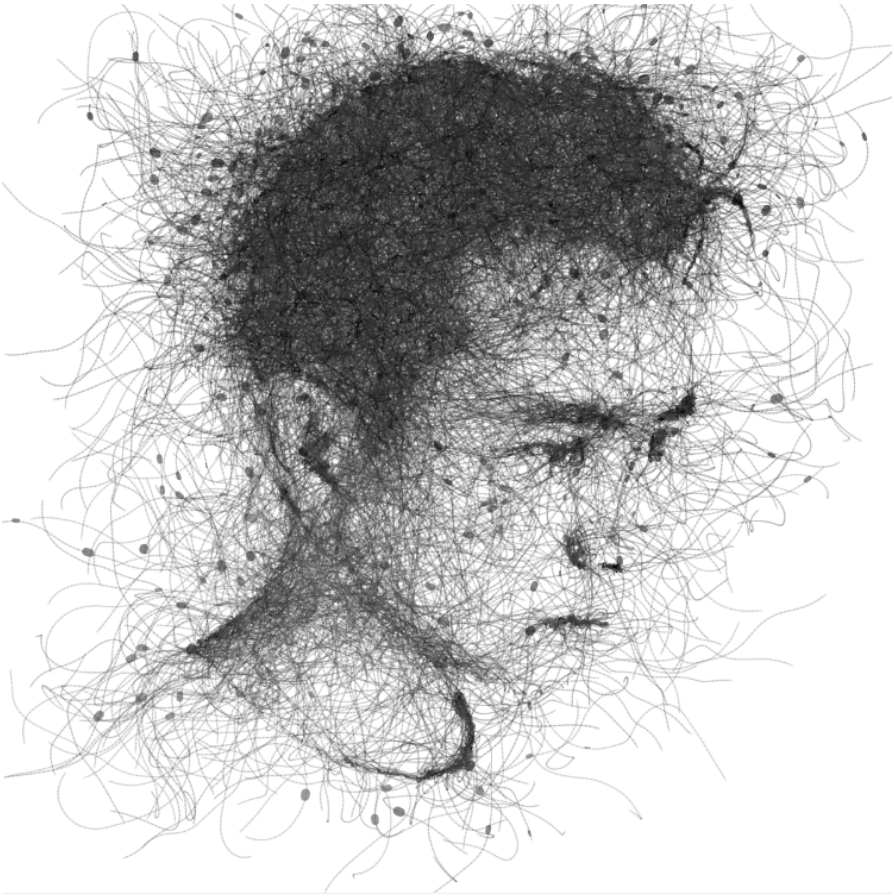

Figura 1.3: *Messy Curve Draw* di Zheng Yuesheng è un esempio di opera realizzata con creative coding. Le linee continuano ad aumentare, aggiungendo man mano dettagli alla figura, e ad ogni click del mouse inizia un nuovo disegno, con un soggetto diverso.

d'aiuto per gli artisti.

Tra questi rientrano gli strumenti di controllo delle versioni, perché rilasciare l'aggiornamento di un'applicazione non è la stessa cosa di dipingere la *Vergine delle rocce* e poi realizzarne un'altra leggermente diversa.

1.1 Obiettivo

Questa tesi si pone in continuità con gli studi di Juan Pablo Saenz [1] sulla comunità dei creative coders e con la tesi di Giacomo Vitali [2], sempre sotto la supervisione di De Russis e Saenz, che proponeva alcuni strumenti pensati appositamente per supportare il lavoro dei creative coders, rendendolo più semplice e intuitivo.

In tutti i lavori precedenti viene evidenziata la mancanza di un sistema di controllo delle versioni realizzato tenendo in considerazione le esigenze dei creative coders; l'obiettivo di questa tesi è dunque quello di ideare, progettare e realizzarne uno.

Seguendo un approccio di human-centered design [3] il punto di partenza è stato l'individuazione dei bisogni degli utenti per cui il prodotto viene pensato, a cui è seguita una prima fase di proposta di soluzioni per i problemi riscontrati. A questa fase preliminare è seguita la realizzazione di un primo prototipo a bassa fedeltà, che presentasse visivamente le soluzioni trovate. Una volta discusso e approvato questo prototipo, si è passati allo scontro con la realtà, a capire come potesse essere effettivamente implementato e quali fossero le tecnologie, i contesti e gli strumenti più adatti per confezionare un prodotto funzionante ed efficace. Infine, una valutazione con gli utenti finali, per capire se le funzionalità implementate sono utili e utilizzabili, e dove è possibile migliorare il prodotto.

1.2 Struttura della tesi

Dopo il Capitolo 1 che introduce in maniera più discorsiva il tema della ricerca, questa tesi contiene altri cinque capitoli.

Il Capitolo 2 *Lavori correlati* inizia spiegando con maggiore dettaglio i concetti di *creative coding* e *version control system*, fondamentali per comprendere il resto della trattazione, per poi passare a una panoramica delle ricerche già pubblicate in merito al creative coding e al suo rapporto con il version control, evidenziando le principali problematiche già note.

Il Capitolo 3 *Progettazione* descrive il processo di ideazione e design della soluzione proposta. Si parte dall'analisi delle interviste ai creative coders, per poi offrire una panoramica delle soluzioni attualmente usate, evidenziandone i problemi e i punti di forza. Si passa quindi alla presentazione dei primi prototipi, con l'approccio visivo e i concetti di *Version* e *Variation*, per concludere con la scelta delle modalità di implementazione.

Il Capitolo 4 *Implementazione* propone un'analisi dettagliata del prodotto realizzato. Si parte dalla presentazione delle tecnologie su cui si basa, come p5.js, Git e Visual Studio Code, si passa quindi alla struttura del codice, discutendo le scelte tecniche, e si conclude con una presentazione delle funzionalità dal punto di vista dell'utente, in cui vengono descritte le operazioni che l'applicazione permette di eseguire e come possono essere raggiunte.

Il Capitolo 5 *Valutazione* presenta le modalità di valutazione dell'usabilità di un prodotto secondo i principi dell'human-centered design. Vengono poi proposte alcune linee guida per la valutazione del progetto implementato in questa tesi.

Il Capitolo 6 *Conclusioni* si propone di tirare le somme, cercando di capire se e in quale misura l'obiettivo proposto sia stato raggiunto, evidenziando gli elementi da migliorare o da inserire in eventuali sviluppi futuri del progetto.

Capitolo 2

Lavori correlati

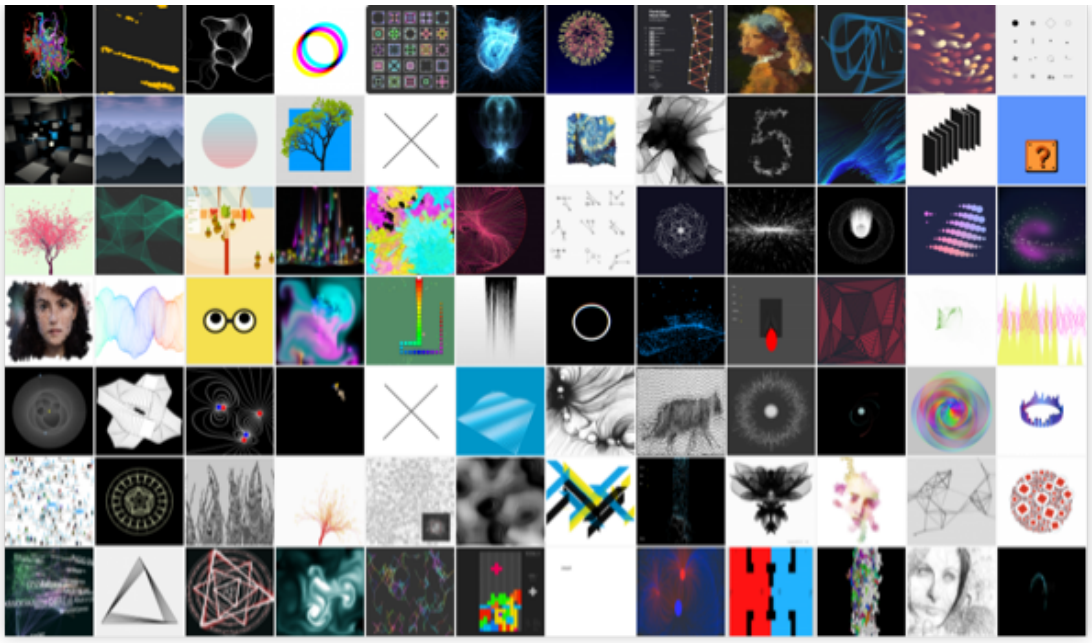

Figura 2.1: Alcune opere realizzate da creative coders caricate sulla piattaforma OpenProcessing

2.1 Creative Coding

Il *creative coding* è la pratica di utilizzare il codice come strumento per creare opere d'arte. Il risultato non è un'opera singola, ma un processo codificato che, ogni volta che viene eseguito da un computer o altro dispositivo, produce un "oggetto"

artistico, potenzialmente sempre diverso se vengono inseriti elementi di casualità o di interazione con il mondo esterno [4].

Un illustre antecedente del creative coding può essere trovato nella scrittura musicale: l'idea dell'artista può essere codificata - in questo caso per mezzo delle note sul pentagramma - per essere poi eseguita in luoghi e momenti diversi, con strumenti diversi. [1]

Tornando all'ambito dell'informatica, i primi esperimenti di creative coding risalgono a non molto tempo dopo la nascita e la diffusione della programmazione, negli anni '60. La pratica ha poi acquisito popolarità a partire dagli anni '80 e '90, fino a diventare ampiamente diffusa negli ultimi due decenni grazie al crescente interesse di artisti, designer e sviluppatori.

In letteratura, il termine creative coding viene spesso associato alla generative art; queste due definizioni, tuttavia, non rappresentano lo stesso concetto. La generative art si concentra su processi autonomi e algoritmi per generare opere: gli algoritmi di generative art riescono a trasformare un input di dati di un certo tipo in risultati artistici di altro tipo, per esempio una descrizione testuale in un'immagine. Dall'altra parte, il creative coding enfatizza il ruolo del programmatore come autore diretto, che sceglie quali elementi creare e combinare per giungere al risultato desiderato.

La pratica del creative coding trova applicazione in diverse forme espressive: immagini, animazioni, scultura - ad esempio in script per la stampa 3D - e altre forme di arti visive, ma anche suoni, videogiochi, prototipi di prodotti e così via. La parte pratica di questa tesi, per gli strumenti utilizzati, si concentra sull'ambito delle arti visive, che può vantare una folta e attiva comunità di artisti. [2]

2.2 Version Control

Si definisce Version Control System (VCS) un sistema che consente di gestire il cambiamento nel tempo di un oggetto in evoluzione. Nel contesto dello sviluppo di software, è un sistema che permette di gestire, tracciare e coordinare ogni modifica ai file realizzata dagli sviluppatori.[5]

Nel processo di sviluppo software, i programmatori apportano modifiche continue a parti di codice e ad altri file. È comprensibile che vengano effettuate diverse revisioni prima di arrivare alla versione finale. Tuttavia, la gestione e l'organizzazione del codice e dei file diventano sempre più complesse man mano che aumentano le revisioni, soprattutto nei sistemi più grandi e articolati. In questo contesto, i sistemi di controllo delle versioni si rivelano fondamentali per accelerare e semplificare il processo di sviluppo.

Senza un VCS, gli sviluppatori tendono a mantenere diverse copie duplicate del codice sui propri computer. Questa pratica è rischiosa, poiché è facile modificare o

eliminare un documento o un file nella versione sbagliata, con il rischio di perdere il lavoro svolto. I sistemi di controllo di versione nascono per risolvere questo problema gestendo tutte le versioni del codice in modo efficace. L'adozione di un VCS è sempre più necessaria per consentire ai programmatori, anche geograficamente distanti, di collaborare in modo efficace.

Nato negli anni '70 con sistemi come SCCS (Source Code Control System), il Version Control si è evoluto per rispondere alle esigenze crescenti di progetti complessi e team distribuiti. Negli anni '80 e '90, strumenti centralizzati come RCS (Revision Control System) e CVS (Concurrent Versions System) hanno introdotto funzionalità più avanzate, mentre Subversion (SVN) ha perfezionato questo approccio. Con l'avvento di Git nel 2005, il panorama dei VCS ha subito un cambiamento radicale: Git, sviluppato da Linus Torvalds, ha reso popolare un approccio distribuito, che permette a ciascun collaboratore di avere una copia completa del repository, migliorando flessibilità, velocità e resilienza. [6] I due principali approcci al Version Control, centralizzato e distribuito, differiscono per il numero e la posizione dei *repository*. [7] [8] [9] Nella sottosezione 4.1.2 viene approfondito l'approccio distribuito e, in particolare, Git.

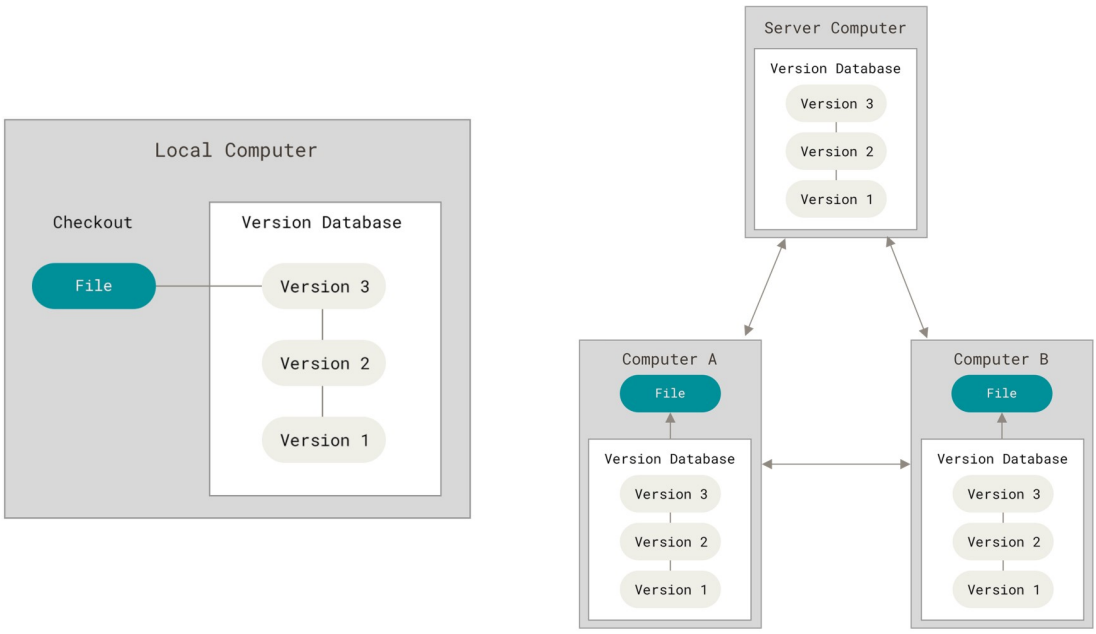

Figura 2.2: Schema delle differenze tra Version Control System centralizzati e distribuiti. Dal *Git Pro book* di Scott Chacon e Ben Straub

2.3 Creative Coders e Version Control

Anche il lavoro dei creative coders, come qualsiasi altro genere di software, può beneficiare di un sistema di version control. Tuttavia, si presenta immediatamente un primo grosso problema: il lavoro creativo non segue lo stesso flusso del codice definito "funzionale", prevede continue iterazioni di *trial and error*, valutazione di molteplici alternative in parallelo e continui ritorni a fasi precedenti. Gli attuali strumenti di version control non sono pensati per queste modalità di lavoro: possono supportarle, ma in modi che richiedono costi in termini di tempo e sforzo creativo che spesso gli artisti non sono disposti ad accettare. Per questo è emerso che spesso preferiscono metodi di version control informali o manuali, come salvare screenshot di ogni sketch o di diverse iterazioni dello stesso sketch, o avere diversi file in cui copiano e incollano il codice per passare a una versione successiva. Diversi studi si sono concentrati sul rapporto tra creative coders e strumenti di versioning, di solito come parte di una trattazione più ampia in merito alla comunità dei creative coders. Di seguito ne vengono presentati quattro tra i più significativi.

2.3.1 Interviste per conoscere la comunità dei creative coders

Verano Merino e Saenz [1] hanno realizzato "The Art of Creating Code-Based Artworks", una presentazione del mondo dei creative coders arricchita dall'intervista a cinque artisti con diversi background, livelli di esperienza e generi di espressione artistica.

La prima domanda, approfondita nella sezione 3.1, riguardava le fonti di ispirazione degli artisti. Andando ad approfondire, veniva chiesto se tra le ispirazioni rientrassero anche altre opere basate sul codice, ma la risposta è stata negativa: "Penso che se vuoi sviluppare uno stile veramente individuale, non dovresti ispirarti troppo al codice di altri o ai tutorial" è stata una delle risposte. In generale, è stato evidenziato che potrebbe interferire con la propria originalità e, inoltre, analizzare e capire il codice di altri potrebbe essere difficile e far perdere tempo. Il fatto che i creative coders preferiscano lavorare da soli al proprio codice fa passare in secondo piano tutte le funzionalità di VCS relative alla gestione della collaborazione di molti autori a un unico progetto. Diventano dunque più interessanti le modalità di gestione della storia e dei cambiamenti del progetto.

Tutti i partecipanti al sondaggio concordano sulla natura esplorativa del creative coding. Ad esempio, viene detto, gli artisti creano continuamente nuovo codice che sperimenta una particolare idea e, una volta soddisfatti dal risultato, questa può diventare la base per una nuova opera, oppure essere integrata in un progetto più grande, o ancora essere conservata in attesa di un possibile utilizzo futuro.

Devono quindi trovare un modo per categorizzare e organizzare i file, così da poter tornare a uno stato precedente o poterli riutilizzare in futuro. Dalle interviste emerge che gli attuali VCS non sono progettati per supportare questa natura esplorativa, e quindi ogni artista è costretto a sviluppare le proprie strategie. Viene inoltre evidenziato che i VCS non vengono utilizzati anche a causa della ripida curva di apprendimento che occorre superare per cominciare a impiegarli. Gli intervistati tornano infine a sottolineare che spesso lo sviluppo del lavoro è fatto di piccole modifiche in punti distanti tra loro, ognuna delle quali non sembra abbastanza importante da giustificare un commit, ma allo stesso tempo rappresentano concetti troppo diversi tra loro per essere raggruppate.

2.3.2 Indagine preliminare per il progetto Spellburst

Tyler, Suzara, Han et al. [4] hanno sviluppato Spellburst, un ambiente di lavoro per creative coders basato su algoritmi di intelligenza artificiale. Durante la fase preliminare del progetto hanno effettuato interviste a diversi creative coders, da cui sono emersi spunti importanti anche riguardo alle pratiche di version control.

I partecipanti al sondaggio descrivono la dissonanza cognitiva tra il processo creativo, esplorativo e spesso portato avanti da cambiamenti inaspettati e non del tutto intenzionali, e i rigidi procedimenti di raggruppamento e minuziosa descrizione dei cambiamenti imposti da sistemi di version control come Git. La maggior parte degli intervistati, dunque, dichiara di aver scelto di non usare sistemi di version control per la creazione di progetti personali. E questo anche nei casi in cui l'intervistato ha familiarità con Git in altri contesti. Le alternative *informali* sono varie, e alcune di esse a loro volta molto creative. Alcuni esempi verranno approfonditi in seguito (sezione 3.2). Il principale problema che accomuna queste soluzioni è la possibile confusione creata dalla separazione tra gli output, il codice, e altri metadati possibilmente utili, e dalla continua duplicazione di file inerenti lo stesso progetto.

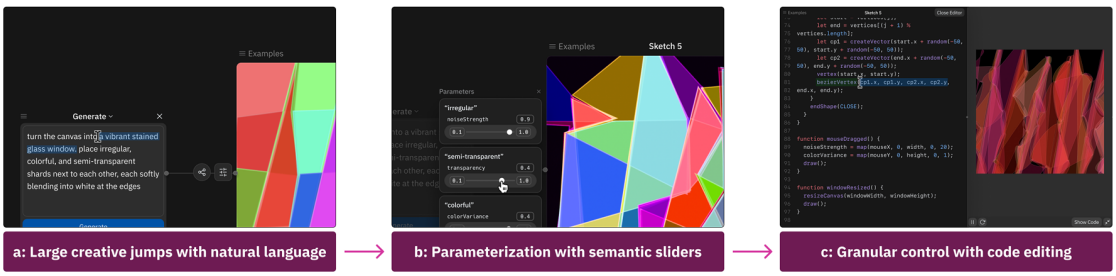

Figura 2.3: Panoramica del funzionamento di Spellburst

2.3.3 Studio sul rapporto tra creativi e version control

Sterman, Nicholas et al. [10] hanno studiato approfonditamente proprio il rapporto tra i creativi e il version control. La loro ricerca non si limita alla comunità dei creative coders, ma il loro campo di ricerca comprendeva creativi di diverse discipline, tra cui registi, performer, e artigiani specializzati nella realizzazione di strumenti musicali. Vengono quindi discussi e confrontati i diversi approcci e strumenti di versioning utilizzati.

Per quanto riguarda il codice, viene riportato che alcuni tra gli intervistati usano VCS disponibili sul mercato, mentre altri, anche se li conoscono o li utilizzano in altri contesti professionali, non ne fanno uso nel contesto artistico. Anche in questo articolo vengono presentate diverse soluzioni *informali*. Vengono evidenziati dei problemi comuni tra chi si affida a un approccio manuale come la maggior facilità di introdurre errori nel flusso di sviluppo del progetto, e la perdita di informazioni chiave causate dal fatto di dimenticarsi quale file contenesse quale specifica informazione e quali cambiamenti, e perché fosse stata creata quella specifica copia. Si passa poi a descrivere modalità più *fisiche*, usate in prevalenza dagli intervistati che si occupano di ambiti *non digitali*, ma anche, come emerge, da alcuni creative coders, per i quali parte del controllo delle versioni consiste in appunti su carta.

Ritornano concetti presenti anche negli studi analizzati in precedenza, come la non linearità del processo creativo: in questo approccio le versioni non vengono viste come una progressione verso un obiettivo ben definito, ma come diverse variazioni su un materiale di partenza, che funzionano come una tavolozza di colori o un set di pennelli da cui poter attingere. In questo contesto, si legge, la definizione di "versione" non è coerente con l'idea alla base di strumenti come Git, in un contesto artistico il *progresso* non significa niente, o comunque non è un concetto tanto utile e rilevante quanto la diversità di opzioni da esplorare.

Un aspetto apprezzato dei sistemi di version control è invece la possibilità di tornare a una versione precedente senza paura di perdere il lavoro svolto o di rovinare il progetto sperimentando nuovi cambiamenti. In questi casi è apprezzata la sensazione di supporto all'esplorazione offerta dai VCS, anche se non vengono intesi allo stesso modo concetti come "corretto" o "funzionante".

2.3.4 Studio sulla piattaforma OpenProcessing

Il quarto lavoro presentato, lo studio di Subbaraman et al. [11], non contiene interviste, ma riporta un'analisi effettuata sull'intera comunità di OpenProcessing.

Processing è un diffuso software per il creative coding basato su Java, è stato reinterpretato per il web con il progetto p5.js, di cui si tratterà più approfonditamente in seguito (sottosezione 4.1.1). OpenProcessing è un sito web, una piattaforma indipendente che permette agli utenti di caricare e condividere progetti creati con

Processing e p5, ed è frequentata da una delle più grandi comunità di creative coders sul web.

Gli autori hanno analizzato come gli utenti di OpenProcessing ricaricano e modificano progetti già presenti sulla piattaforma, un'operazione nota come *remix* (Figura 2.4).

Lo studio individua quattro principali strategie di remix:

- Collezionare sketch senza apportare alcun cambiamento. Nonostante OpenProcessing offra la possibilità di mettere "like" a uno sketch, esistono diversi profili dedicati esclusivamente a collezionare "le migliori opere" di altri utenti.
- Aggiungere commenti. Alcuni progetti vengono ricondivisi senza modificare il codice, ma aggiungendo spiegazioni o traducendo i commenti in altre lingue. In alcuni casi tra i commenti ci sono linee di codice con piccole variazioni, residuo di precedenti sperimentazioni o proposte per nuove variazioni.
- *Tuning* dei parametri. Non vengono cambiate le istruzioni, ma solo il valore di alcuni parametri, vengono così visualizzate tutte le possibili variazioni di output offerte dallo stesso algoritmo.
- Estendere sketch con nuovo codice, spesso ad opera dello stesso autore.

Analizzando la frequenza con cui si presentavano queste strategie, è emerso che il 55,3% degli sketch considerati *remix* conteneva del tuning. Ciò è coerente con l'approccio esplorativo di cui si è già detto molto, e inoltre evidenzia il fatto che spesso questo approccio porta a creare output diversissimi tra loro a partire da un codice che non cambia affatto: questo è un fatto da tenere in considerazione se si vuole progettare un sistema di version control specifico per il creative coding.

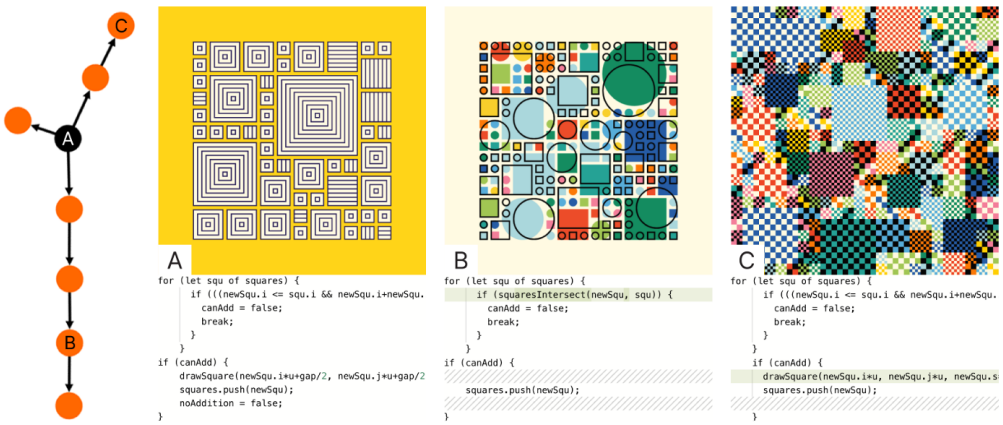

Figura 2.4: Un esempio di remix di uno sketch [11]

Capitolo 3

Progettazione

In questo capitolo vengono riportati i principali passaggi che hanno condotto alla scelta della soluzione da implementare. La sezione 3.1 mostra come l'analisi delle interviste e degli studi preesistenti abbia portato all'identificazione dei *needs* su cui lavorare. La sezione 3.2 discute le risposte a questi *needs* già proposte da altri, mentre la sezione 3.3 presenta le soluzioni proposte da questo progetto.

Dopo la fase di ideazione si procede a immaginare l'applicazione delle idee selezionate, visualizzandole in un primo prototipo, descritto nella sezione 3.4. Il prototipo permette di scegliere le specifiche tecniche e arrivare così, nella sezione 3.5, alla definizione del progetto da realizzare.

3.1 Identificazione dei bisogni

In Human-Computer Interaction, i *needs* rappresentano i requisiti, le aspettative e i desideri degli utenti quando interagiscono con un sistema tecnologico. Possono riferirsi a requisiti funzionali (ho *bisogno* di poter compiere una determinata azione), ma anche a esigenze di usabilità (ho *bisogno* di un'interfaccia chiara, o di poter tornare indietro se commetto un errore), emotive (ho *bisogno* di non sentirmi frustrato mentre eseguo un'operazione complessa che richiede molteplici passaggi, oppure di sentirmi al sicuro navigando su questo sito web), di accessibilità (ho *bisogno* di accedere a questo contenuto pur essendo non vedente) e molte altre.

Dagli studi presentati al Capitolo 2 e da altri sullo stesso tema, emerge che un bisogno dei creative coders è quello di avere a disposizione un sistema di version control adatto alle loro esigenze. Ma occorre capire meglio quali siano, nello specifico, queste esigenze.

3.1.1 Facilità di apprendimento e di utilizzo

Le interviste (sottosezione 2.3.1) sottolineano che uno dei principali ostacoli all'utilizzo dei VCS è la ripida curva di apprendimento. In altre parole, la quantità di concetti nuovi da apprendere prima di poter iniziare ad utilizzare lo strumento è talmente alta da richiedere un impegno che, potendo scegliere se accettarlo o no, l'utente non è disposto a sostenere.

"Per quanto riguarda le ragioni che tengono lontani gli artisti dall'affidarsi a VCS tradizionali, i partecipanti hanno menzionato una ripida curva di apprendimento, e l'utilizzo di questi sistemi richiede una certa esperienza per gestire i conflitti tra le versioni" [1]

Ad esempio, l'utilizzo di un VCS come Git richiede di conoscere concetti come *repository*, *stage*, *commit*, *branch*, *checkout*, *merge*, e i relativi comandi testuali da linea di comando, un altro elemento che può allontanare una fetta di utenti. Non è un caso che gli ambienti di lavoro più moderni, come ad esempio Visual Studio Code (vedi sottosezione 4.1.3) propongano estensioni per utilizzare Git in modo più semplice e visivamente più chiaro.

3.1.2 Gestire le piccole modifiche

Come emerge da varie interviste (sottosezione 2.3.1, sottosezione 2.3.2, sottosezione 2.3.3) e anche dall'analisi effettuata sulla piattaforma OpenProcessing (sottosezione 2.3.4) le piccole modifiche al codice possono avere un grande impatto sull'output ottenuto, e il processo creativo è spesso formato da una serie di piccole modifiche in punti del codice potenzialmente distanti tra loro e senza apparente correlazione logica. Oppure, addirittura, senza modificare affatto il codice, con il *tuning* dei parametri.

Il bisogno può essere individuato in passaggi come questi:

- La seconda ragione che frena gli artisti dall'utilizzo di VCS è che il loro processo di sviluppo è altamente iterativo. Di solito consiste nel cambiare piccole porzioni di codice o fare esperimenti con una serie di valori per i parametri. Di conseguenza, le iterazioni avvengono velocemente, ed effettuare manualmente commit per ogni cambiamento non è pratico o utile. Come commenta Michael: "Ho sempre voluto usare GitHub in modo più aprropriato, ma mi sono accorto che non si adatta al mio stile di lavoro. Per i cambiamenti che apporto a questi sketch, non ne vale davvero la pena." Riferendosi ai commit, Sumeet ha detto: "A volte si tratta di un pezzo di codice molto piccolo. È come dire: 'perché dovrei preoccuparmi che Git controlli questo pezzo di codice nel contesto delle cose più grandi?', giusto?". In questo contesto, Felipe trova auspicabile "Trovare in qualche modo un sistema migliore [per eseguire

il commit dei cambiamenti], come avere un qualche commit automatico ogni cinque secondi." [1]

- Molti partecipanti hanno descritto il loro processo esplorativo come basato sui parametri, in cui l'esplorazione si fonda sulla regolazione di un insieme di variabili, che U9 ha definito come "costanti magiche". U4 organizza il suo codice posizionando tutti i parametri nella parte superiore del file: "Non ho davvero un processo ben strutturato per tracciare le versioni, e tendo a fare una nota mentale o a scrivere i parametri che portano a qualcosa che mi piace... Mi piace raggruppare e mettere tutte le cose che modificherei in un unico posto e tenere da parte tutte le cose che sono abbastanza statiche." [...] Altre volte, le iterazioni sui parametri avvenivano attraverso un processo di tentativi ed errori, che alcuni partecipanti hanno definito "sviluppo guidato dalla casualità". U3 descrive un momento in cui ha scoperto una nuova idea in uno sketch di p5.js: "Ho sbagliato a collegare qualcosa, e alla fine mi sono ritrovata con due oggetti collegati nel mezzo... quindi penso che i piccoli incidenti siano l'ispirazione." Durante tali scoperte, un altro punto di discussione è stato portare in superficie i parametri durante l'esecuzione e il test del codice; questo a volte veniva fatto attraverso dichiarazioni console o piccole funzioni di supporto. U7 ha descritto la creazione di funzioni di "debug" che programmavano tasti di scelta rapida per mostrargli l'intervallo di possibili risultati mentre eseguiva una simulazione di gioco 3D, soprattutto in situazioni in cui era richiesta casualità controllata. U5 ha descritto il ciclo di iterazione su un particolare parametro, l'esecuzione del codice e l'analisi del risultato come un ciclo di feedback: "Se avessi come un piccolo cursore per rendere [questo particolare parametro] più o meno come stavo regolando costantemente... Se ci fosse un modo per rendere quel ciclo di feedback più veloce, sarebbe perfetto." [4]

- Il New Media Artist è un creative coder che lavora professionalmente su opere d'arte digitali e ibride da dodici anni e insegna arte digitale ed elettronica da sette anni. Il New Media Artist considera il codice come un materiale, in modo molto simile a come un artista tradizionale lavora con la pittura o l'argilla. In contrasto con l'approccio "top-down", che punta a un obiettivo specifico, definisce il suo come un approccio "bottom-up": "All'inizio non ho un obiettivo finale... è più o meno come fanno gli artisti quando iniziano il loro lavoro; giocano con la scultura, giocano con l'argilla, la modellano e poi, osservandola, [si dicono] 'Oh, questa è la direzione che voglio seguire' ...quindi [quando programmo] arrivo a pensare 'Oh, voglio creare una funzione di easing,' oppure 'Voglio spostare un punto verso un altro punto e lasciare una traccia.' E una volta che implemento questo... diventano i miei moduli – materiali – da applicare a diversi sketch." Con questo approccio, le versioni

dei moduli non rappresentano progressi verso un obiettivo, ma varianti di un materiale, funzionando come una tavolozza di colori o una selezione di pennelli. Il New Media Artist salva tutte queste versioni come file separati, in modo da potervi accedere in parallelo e passare rapidamente da un'alternativa all'altra. [10]

- Nei nostri dati, vediamo che gli sketch ampiamente remixati presentano molti remix basati sulla regolazione dei parametri. Ad esempio, la catena di remix più lunga su OpenProcessing al momento della nostra raccolta dati era composta da 106 sketch. Lo sketch sorgente consisteva in un volto disegnato con forme semplici, i cui occhi seguono il movimento del mouse. Quasi tutti i remix di questo sketch consistono in altri autori che modificano i colori. Oltre alla regolazione degli sketch altrui, notiamo che i remix di un singolo autore possono essere utilizzati per gestire le variazioni. La Figura 3.1 mostra una serie di sketch di *Trrrrrr*. Dopo aver implementato un algoritmo di base, manipolano vari parametri per generare un insieme di output distinti. [...] Trrrrrr ha remixato lo sketch originale diverse volte, apportando lievi modifiche all'algoritmo di base ogni volta per esplorare i possibili output. [...] abbiamo visto come i valori precedenti delle variabili regolate siano stati archiviati con commenti inline. In altri casi, notiamo che la regolazione può rendere obsoleta la documentazione inline. [Ad esempio, in uno sketch remixato] i valori RGB che definiscono il colore delle linee disegnate sullo schermo sono stati modificati da una tonalità di rosso al nero. Il commento, tuttavia, non è più allineato correttamente. [11]

```
1     /* Versione precedente */
2     stroke(244 , 37 , 37 , 60); // red
3
4     /* Remix successivo */
5     stroke(0, 0, 0); // red
6
```

Se venisse creata una nuova versione a ogni piccola modifica effettuata, oltre all'evidente perdita di tempo, si arriverebbe presto ad avere un numero di versioni non più umanamente gestibile, vanificando così l'utilità del tracciamento delle versioni.

Ma anche raggruppare tutte le piccole modifiche in un unico blocco richiederebbe un grosso sforzo per spiegare cosa caratterizza una data versione, e per tornare a capirlo in seguito, quando la versione deve essere riutilizzata dopo diverso tempo o da qualcun altro.

Collegato a questo bisogno, c'è quello di evitare la frustrazione di salvare manualmente il progetto dopo ogni piccola modifica.

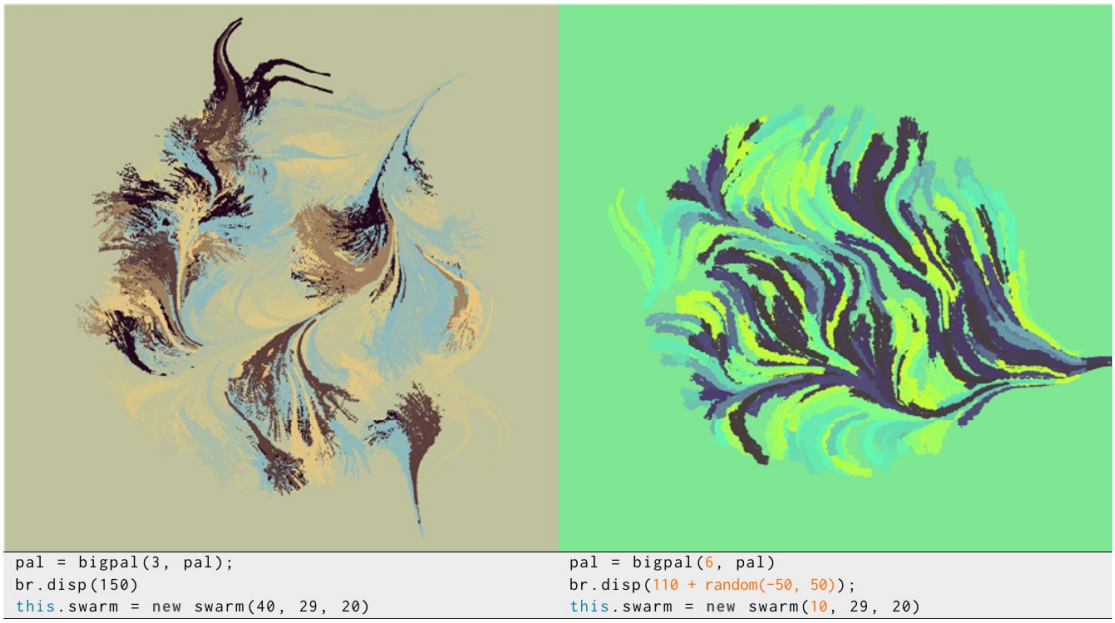

Figura 3.1: Un esempio di *tuning* tratto dall'articolo *Forking a sketch*[11]

3.1.3 Esplorare diverse idee

Tutti i documenti analizzati insistono sul processo esplorativo utilizzato dagli artisti, ma in cosa consiste? Come si può tradurre nella pratica?

Proviamo a immaginare un pedone e una regina sulla scacchiera. Il pedone può rappresentare l'approccio funzionale: ha una direzione precisa in cui sa di dover andare, un obiettivo (raggiungere il fondo della scacchiera per essere promosso a un altro pezzo), e si muove senza poter tornare indietro, passo dopo passo. La regina ha invece la possibilità di esplorare, appunto, ogni direzione e ogni distanza. Ha una scelta enormemente maggiore di mosse possibili e non ha neanche uno scopo ben preciso: potrebbe voler dare scacco al re avversario, mangiare un altro pezzo, o scappare da una minaccia. Se dopo una mossa si accorge che stava prendendo una direzione sbagliata o infruttuosa, può tornare da dove era partita.

Ora, i sistemi di version control sono ottimi per supportare il lavoro del pedone: aiutano a tenere traccia del percorso effettuato e a visualizzare meglio l'obiettivo finale e a segnalare il raggiungimento degli obiettivi intermedi, come un passo avanti di una casella. Sono ottimi anche per gestire l'intera schiera dei pedoni: progetti funzionali complessi che richiedono collaborazione e diversi flussi di lavoro, con obiettivi intermedi diversi, che si supportano a vicenda.

Possono anche gestire un approccio esplorativo. Il problema è che per aiutare la regina a scegliere la prossima mossa sono limitati al campo visivo di un pedone.

"I sistemi di controllo delle versioni spesso rappresentano la storia come una

sequenza di versioni successive. La versione più recente rappresenta lo stato attivo e corretto, in una progressione lineare verso un risultato migliorato. Sebbene il concetto di "branching" consenta l'esplorazione di percorsi alternativi, generalmente esiste un unico ramo principale che viene considerato lo stato attuale e autentico del progetto.

Software Engineer 1 è un ingegnere del software in una grande azienda tecnologica, che lavora professionalmente con il codice da dieci anni. Descrive come utilizza il controllo delle versioni nella sua azienda: "Abbiamo la copia autentica di tutto il codice, e poi le persone possono fare delle deviazioni da essa e successivamente reintegrarle nella copia autentica."

Questo approccio allo sviluppo è una tecnica di processo altamente produttiva quando il creatore o il team persegue un unico obiettivo noto. Tuttavia, per molti professionisti creativi, il paradigma di una progressione lineare verso un unico risultato misurabilmente migliore è in contrasto con il loro processo. Invece di considerare le storie delle versioni come un documento del passato, utilizzano le versioni come una tavolozza di risorse per facilitare un dialogo con i materiali." [10]

Per supportare un approccio esplorativo, dunque, serve:

- Poter visualizzare velocemente l'impatto di una scelta sul resto del progetto.
- Poter considerare simultaneamente e confrontare un ventaglio di possibilità.
- Offrire la sicurezza di sperimentare una strada sapendo di poter tornare al punto di partenza.

3.1.4 Raggruppare le informazioni

Una conseguenza della mancanza di supporto al processo esplorativo, a quanto emerge dalle interviste, è l'utilizzo di soluzioni manuali, informali e a loro volta creative per ottenere gli obiettivi evidenziati in precedenza: gallerie di foto e video per visualizzare simultaneamente e confrontare un ventaglio di possibilità; file testuali, appunti scritti a mano o semplice memoria per ricordarsi le corrispondenze tra il codice e l'output visivo; commit su un sistema di version control o copie multiple dello stesso file per conservare le modifiche nel tempo del codice e accedere alle versioni precedenti. (Alcuni esempi concreti nella sottosezione 3.2.1)

Tutto ciò genera una terribile dispersione dell'informazione e una continua perdita di tempo e di sforzo mentale per orientarsi tra le diverse versioni di un progetto.

Un modo per tenere insieme il codice, il suo output e i commenti dell'autore potrebbe aiutare a semplificare il lavoro.

3.2 Analisi delle soluzioni esistenti

Nei paragrafi precedenti sono state evidenziate le principali problematiche nella gestione del version control per il creative coding. Gli studi analizzati evidenziano come il problema sia stato affrontato e risolto in modi diversi dai diversi artisti, e come diversi ricercatori che hanno studiato l'argomento abbiano proposto linee guida, spunti e possibili soluzioni.

3.2.1 Sistemi informali di controllo delle versioni

"Tim organizza il suo lavoro per temi - si legge nella sezione 3.4 dello studio di Verano Merino e Sáenz [1], che presenta alcuni esempi concreti di strategie utilizzate dagli artisti per il controllo delle versioni - ("Così quando sto trattando uno specifico tema creo, diciamo, una cartella contenitore dove metto tutti gli sketch"). In aggiunta, avere una rappresentazione visiva dell'esecuzione del codice è fondamentale per lui per trovare velocemente gli sketch. A detta sua "Se sto scrivendo uno sketch e mi piace il risultato, lo salvo con un altro nome. Poi lancio sempre una funzione di registrazione video dalla libreria di esportazione video, che crea un'anteprima del mio sketch esattamente nella cartella dove sono tutti i miei sketch."

In modo analogo, per Michael, è fondamentale avere una rappresentazione visuale. La sua strategia, tuttavia, è quella di pubblicare le sue opere su Instagram. In questo modo, quando vuole trovare uno sketch, confronta la data di pubblicazione del post con la data di creazione del file. "Se è qualcosa come un piccolo sketch, ho una cartella di sketch giornalieri, li posto su Instagram e quando ho una nuova idea, probabilmente torno allo sketch e magari cambio alcune variabili e quindi salvo una nuova immagine."

Sumeet, da parte sua, organizza i suoi file per data di creazione. Quando sta lavorando a un progetto, crea una nuova cartella ogni settimana e sposta i file della settimana precedente che intende utilizzare. Inoltre, usa un file esterno per documentare i suoi progressi di apprendimento e sviluppo: "tengo un documento (un notebook di OneNote) ogni settimana in cui inserisco parole chiave e alcune cose imparate durante quella settimana. In più, ho un altro notebook per ogni progetto, e se trovo qualcosa di più generale lo metterò semplicemente in quel documento principale. Quindi ci sono più livelli di documentazione in corso e, se ne sono consapevole, aggiorno semplicemente quei documenti." "

Una prima importante considerazione che possiamo trarre da questi esempi è l'importanza per gli artisti di una rappresentazione visiva. Il loro processo mentale e tutto il loro lavoro passano attraverso immagini ed elementi visivi, e uno strumento pensato per questa categoria di utenti deve necessariamente tenerlo in considerazione. L'altro elemento da notare è la fatica impiegata per associare un record mnemonico associato a un progetto al file con il codice effettivo dello stesso

progetto: sequenze di operazioni come *controllare la data della foto; cercare un file creato lo stesso giorno; verificare che sia il file corretto* oppure *confrontare uno dei diversi diari di progetto con uno dei diversi diari settimanali* hanno un elevato numero di passaggi, che potrebbe essere ridotto.

Anche Angert, Suzara, Han et al. [4] descrivono nella sezione 3.4.4 alcune strategie utilizzate dai partecipanti al loro sondaggio: "Come soluzione alternativa, i partecipanti duplicavano file o frammenti di codice e li commentavano quando non erano in uso. In alternativa, altri tre partecipanti hanno descritto un processo per tracciare le modifiche nella memoria operativa o scrivendo appunti, dicendo qualcosa di simile a "Ricordo semplicemente quello che ho fatto in precedenza": "Spesso apro Notepad e scrivo ciò che è stato positivo [mentre lo eseguivo], e poi me lo ricordo, quindi è sicuramente più un approccio concettuale che un sistema di controllo delle versioni."

Oltre a tracciare le configurazioni del codice che producevano risultati rilevanti, i partecipanti utilizzavano anche pratiche di versioning informale per tenere traccia degli output visivi del codice. I partecipanti tracciavano le modifiche eseguendo e salvando i risultati tramite screenshot e catture video (ad esempio, 'versionFinal.jpg', 'versionFinalFinal.jpg'). "

Da questi esempi emerge la difficoltà di tenere insieme nello stesso file versioni simili del progetto, commentando blocchi di codice o valori alternativi delle variabili. Oltre a ciò, ritorna il problema di affidarsi a un file esterno per tenere traccia dei metadati del progetto, e della duplicazione dei file con nomi poco significativi.

Sterman, Nicholas et al. [10], nelle sezioni 4.1.1 e 4.1.2 dividono le strategie dei creativi intervistati (ricordo che in questo studio non erano coinvolti solo i creative coders) in due categorie: quelle che si basano su un software di VCS e quelle manuali: "Tra i creativi professionisti che lavorano con il codice, alcuni partecipanti utilizzavano sistemi di controllo di versione consolidati e commerciali, come Git o sistemi specifici per aziende, oltre a strumenti come GitHub (Software Engineer 3, Software Engineer 1, Generative Artist). Tuttavia, abbiamo osservato anche alcuni professionisti che, pur avendo usato questi strumenti in ambito professionale, li evitavano nel loro lavoro creativo, preferendo salvare manualmente nuove copie di un file quando apportavano modifiche, o creando strumenti personalizzati (New Media Artist, Creative Coder). Ad esempio, quando il Generative Artist utilizza Git nel suo processo creativo, ha adattato i paradigmi di interazione esistenti di Git al suo flusso di lavoro creativo personale, costruendo una catena di strumenti su misura.

Il versioning manuale, in cui il professionista salva nuove copie di file digitali per tracciare le modifiche, era utilizzato dal New Media Artist per il codice, così come dal Ricercatore sul Comportamento Animale per documenti di testo e fogli di calcolo: "Sono quella persona che ha 8 milioni di versioni diverse di ogni documento Word. Scrivo cose tipo "usami" oppure "no, usa me" o "finale finale finale", o

"finale finale finale versione finale".

Sia il Ricercatore sul Comportamento Animale che il New Media Artist hanno riscontrato difficoltà con l'approccio manuale, in quanto gli errori venivano introdotti più facilmente nel flusso di lavoro e informazioni chiave venivano perse o dimenticate a causa dell'incertezza su quali file contenessero quali modifiche, quale versione fosse utilizzata da un collaboratore, o perché alcune copie fossero state create. "

Da questi racconti ci si può rendere conto che i sistemi di version control esistenti possono essere utilizzati per affrontare il problema, ma devono essere impostati nel modo adatto, cosa che non tutti i creative coders hanno le capacità tecniche per fare.

3.2.2 Spunti da altri studi sull'argomento

Sterman, Nicholas et al. [10], nella sezione 2.4, fanno notare che "i sistemi di version control su larga scala non sono l'unico modo di pensare a come processare l'interazione con i cambiamenti dei dati nel tempo." Su una scala più ridotta, ad esempio, una funzionalità come *undo*, presente praticamente in tutti gli strumenti di scrittura disponibili, permette l'approccio esplorativo per una singola modifica o un piccolo numero. Alcune ricerche hanno studiato l'importanza della funzionalità di *undo*, come quella di Myers et al. [12] nel campo delle applicazioni che permettono di dipingere e quella di Terry et al. [13] nell'ambito della manipolazione di immagini.

Lo studio di Myers et al. propone di integrare sistemi di backtrack più complessi senza parlare esplicitamente di version control, per supportare un approccio più "naturale" al coding esplorativo. Terry et al. suggeriscono di prendere ispirazione da modelli a layer sovrapposti, come quelli usati nelle applicazioni di photo editing, per conservare in un unico file le informazioni relative a versioni successive.

Un altro possibile supporto al processo esplorativo che esula dal campo dei VCS è la possibilità di visualizzare immediatamente l'output del codice scritto. Questa funzionalità, ad esempio, è presente nell'editor web di p5.js, come approfondito nella sottosezione 4.1.1, anche se in quel contesto non è possibile conservare e catalogare le immagini più rilevanti.

Angert, Suzara, Han et al. [4] citano altri esempi di strumenti progettati per aiutare, più o meno direttamente, il processo di version control in un contesto creativo, come sistemi che permettono di esplorare più rapidamente l'esplorazione delle iterazioni, permettendo editing ed esecuzione paralleli [14], ambienti controllati per testare rapidamente e salvare, o scartare, *microversioni* del progetto [15] [16], o ancora strumenti che aiutano a salvare ogni commit come nodo in un grafo, insieme ad annotazioni, screenshot e altre informazioni utili. [17][18][19]

3.3 Soluzioni proposte

Dallo studio dei dati raccolti sono emerse alcune idee che potessero rispondere ai bisogni individuati. Di seguito vengono quindi riportate le linee guida alla base del progetto realizzato.

3.3.1 Integrazione in un ambiente già esistente

Per rispondere al bisogno di facilità nell'apprendimento e nell'utilizzo, si è deciso di non creare una nuova applicazione da zero, con un suo nuovo stile visivo e operativo, magari anche estremamente efficace, ma che per essere utilizzata richiede all'utente di spostarsi in un nuovo ambiente, interiorizzare nuovi meccanismi e gestire la continuità con il lavoro precedente nel passaggio da una piattaforma all'altra.

La soluzione proposta, invece, è quella di estendere un ambiente di lavoro già esistente e già utilizzato da un buon numero di utenti, seguire le sue linee guida nelle modalità di accesso alle nuove funzionalità e nell'aspetto visivo, per minimizzare le nuove informazioni da apprendere prima di utilizzare il nuovo strumento, ed evitare che la fatica dovuta al cambiamento di abitudini scoraggiasse gli utenti.

Sono state individuate due possibili applicazioni da cui partire. Il web editor di p5.js o la versatile piattaforma di programmazione Visual Studio Code. L'analisi delle due alternative e la relativa scelta è trattata nella sottosezione 3.5.1

3.3.2 Version e Variation

La struttura *Tema e variazioni* è una forma di composizione musicale tipica della musica classica. Consiste nell'esposizione di un tema ben identificato e in una successiva serie di variazioni che partono dalla struttura del tema iniziale e ne modificano di volta in volta alcune caratteristiche.

A questa struttura sono ispirati i concetti di *Version* e *Variation*, introdotti per rispondere alla necessità di gestire le piccole modifiche (sottosezione 3.1.2).

Per *Version* si intendono le significative modifiche al codice e alla sua struttura, mentre le *Variation* hanno come "tema" di partenza una data *Version*, non prevedono modifiche al codice, ma solo il *tuning* dei valori di un insieme di variabili o di parametri delle funzioni.

Questo approccio comporta la riduzione del numero di versioni da considerare, e allo stesso tempo permette di visualizzare rapidamente quali output diversi possono essere il risultato di uno stesso codice.

3.3.3 Rappresentazione visuale: gallery o storyline

Come spiegato nella sottosezione 3.2.1, la soluzione proposta deve necessariamente includere un aspetto visuale. La proposta è quella di visualizzare la lista delle versioni come una serie di immagini, ognuna delle quali è l'anteprima dell'output di una versione. Associati all'immagine devono esserci un nome e la possibilità per l'autore di scrivere annotazioni e commenti, nonché altre informazioni rilevanti come la data di creazione e di ultima modifica. Da ogni *Version* deve poi essere possibile accedere alla visualizzazione delle relative *Variation*. Inoltre, l'immagine deve essere strettamente collegata alla possibilità di tornare al codice che l'ha prodotta, rispondendo così anche all'esigenza di raggruppamento dell'informazione.

Sono state proposte due versioni di questa soluzione: *Versions gallery* o *Versions storyline*. La prima prevedeva di disporre le immagini in una griglia ordinata, come in un feed di Instagram o come avviene sulla piattaforma OpenProcessing [20]. La seconda invece evidenziava i rapporti temporali e di parentela tra le *Version*, disponendole come in un albero genealogico.

Alla fine si è deciso di optare per il formato *Versions gallery*, per sottolineare l'equivalenza concettuale delle *Version*, rappresentandole come una tavolozza di possibili opzioni tutte egualmente valide. Per semplificare ancora le situazioni di aumento del numero delle *Version*, è stata aggiunta la possibilità di contrassegnarne alcune come *preferite* e di poter applicare un filtro per visualizzare solo quelle.

3.4 Prototipo

Per cominciare a visualizzare e fissare le modalità di implementazione delle soluzioni proposte, è stato realizzato un primo prototipo.

Si tratta di un *paper prototype*, realizzato a mano su carta. Un prototipo a bassa fedeltà di questo tipo è utile perché permette di concentrarsi sui contenuti, senza farsi distrarre dall'aspetto grafico. Il prototipo mostra quale informazione presenterà ogni parte dell'applicazione, dove sarà disposta e che importanza avrà in relazione alle altre informazioni.

In questo caso specifico, poi, l'aspetto grafico, i colori, i font e lo stile sarebbero derivati dalla tecnologia scelta come punto di partenza; non era quindi necessario in questa fase concentrarsi sul loro impatto nel progetto.

Nell'immagine (Figura 3.2) si può vedere la schermata di partenza, con l'area per scrivere il codice sulla colonna di sinistra e l'area per l'anteprima istantanea su quella di destra.

Nella Figura 3.3 invece c'è la *Variation console*, una dashboard per la gestione delle *Variation* con la possibilità di aggiungere e modificare parametri e altre informazioni associate, e mantenendo un'area per l'anteprima.

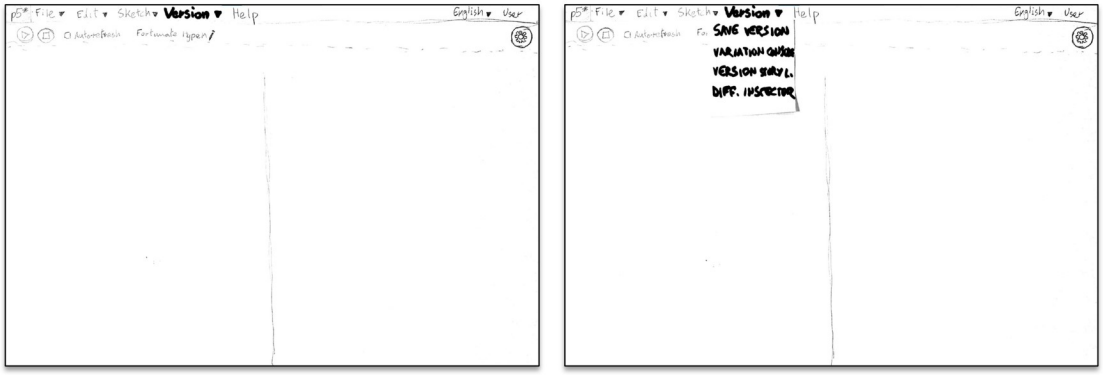

Figura 3.2: La homepage del prototipo

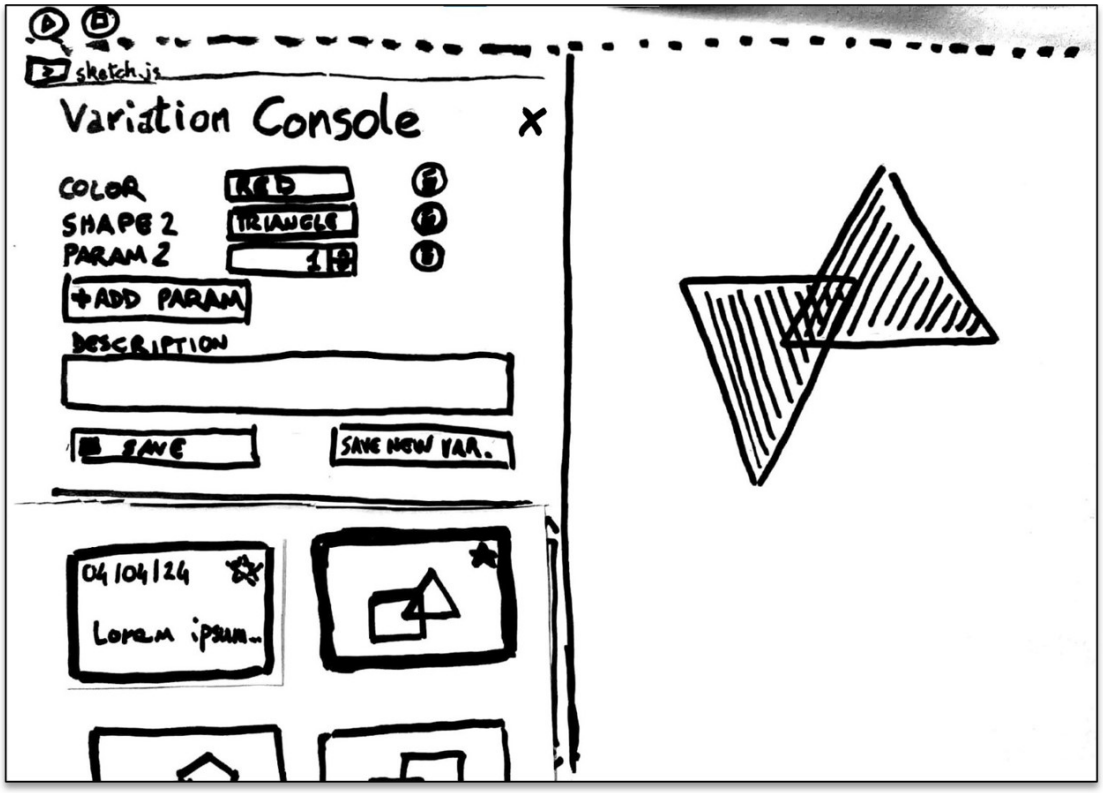

Figura 3.3: Prototipo per la Variation Console

La Figura 3.4 infine rappresenta una proposta per la *Versions storyline*, poi accantonata in favore della *Versions Gallery*.

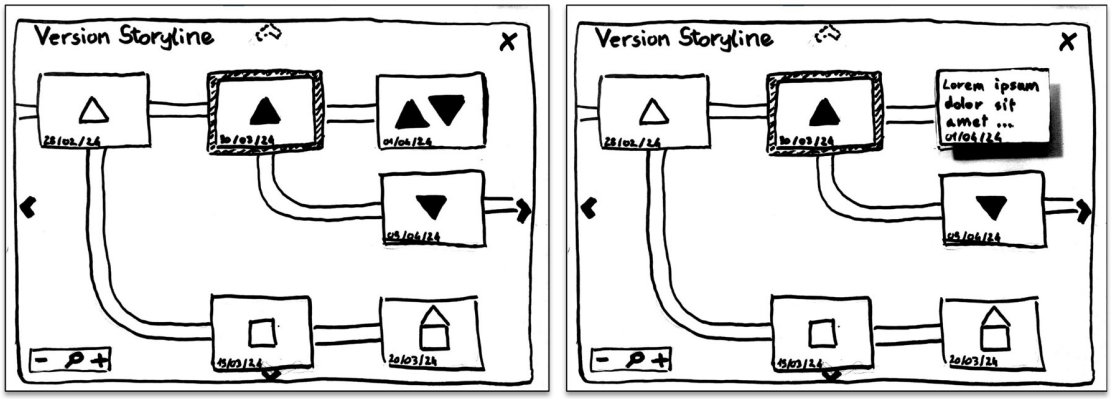

Figura 3.4: Prototipo per la Version Storyline

3.5 Scelte di implementazione

Una volta definito e valutato il *paper prototype* restano da stabilire alcune scelte di implementazione, prima di cominciare a progettare e scrivere il software. La più importante di queste scelte riguardava il sistema da cui partire, con le principali alternative rappresentate dal web editor p5.js e da Visual Studio Code.

3.5.1 Web editor o Visual Studio Code

Entrambe sono già usate da un buon numero di utenti, erano state selezionate anche per questo, quindi il criterio puramente numerico non poteva essere applicato nella decisione finale.

Sono stati analizzati i pro e i contro tecnici delle due soluzioni. Il web editor p5.js presentava i seguenti vantaggi:

- Un ambiente già nelle intenzioni dedicato al creative coding, con la possibilità di visualizzare istantaneamente un'anteprima del risultato e aggiungere facilmente librerie utili.
- La struttura di una moderna applicazione web in JavaScript, mantenuta da una comunità attiva [21] e supportata da una buona documentazione.[22]
- La possibilità, già prevista, di essere integrato con estensioni esterne.

ma presentava anche alcune problematiche:

- La difficoltà a collegarsi al file system dell'utente, che vincola ogni progetto a esistere solo all'interno della piattaforma web.

- La difficoltà di integrazione con Git o con un altro version control system esistente.

Dall'altra parte, anche Visual Studio Code aveva alcuni elementi a favore:

- Il supporto alla realizzazione di estensioni.
- La possibilità di attingere ad altre estensioni per prendere ispirazione o per delegare direttamente alcune funzionalità; in particolare, il supporto fornito dall'estensione di Git.
- La possibilità di interagire con il File System dell'utente.
- La possibilità di effettuare il debug dell'estensione all'interno della stessa piattaforma.

Inoltre, non presentava particolari controindicazioni, se non la necessità di imparare come creare un'estensione per Visual Studio Code, operazione che non avevo mai fatto, ma agevolata da una buona documentazione messa a disposizione da Microsoft. [23]

In conclusione, dunque, la scelta è ricaduta su Visual Studio Code. Con l'accortezza, però, di inserire le nuove funzionalità in una struttura ispirata all'editor di p5.js, che mantenesse quindi, per esempio, l'anteprima istantanea e la colorazione del codice scritto con p5.

3.5.2 Creative Version Control

Il progetto realizzato per questa tesi è Creative Version Control o, più in breve, CVC. Si tratta di un'estensione per Visual Studio Code che offre supporto per la gestione delle versioni di un progetto p5. Si basa su Git, ponendosi come layer intermedio per mascherare alcune funzionalità e presentarle in un modo pensato per le caratteristiche esplorative e visive del creative coding. Le informazioni aggiuntive associate, per esempio, alle *Version*, alle *Variation* e alle immagini sono gestite grazie a un file JSON organizzato come un database di tipo non relazionale.

Capitolo 4

Implementazione

In questo capitolo viene spiegata l'implementazione del progetto Creative Version Control.

Vengono dapprima presentate alcune tecnologie già esistenti che hanno svolto da punto di partenza o che si sono rese necessarie per implementare alcune funzionalità, e che comunque è importante conoscere per comprendere le sezioni successive.

Si passa quindi a un'analisi del codice dell'applicazione, vengono presentate la struttura del progetto e le principali classi e funzioni.

Infine vengono elencate e commentate le funzionalità esposte all'utente.

4.1 Tecnologie utilizzate

4.1.1 p5.js

CVC può supportare il creative coding in qualsiasi forma, ma è pensato in particolare per supportare progetti p5. Permette infatti di configurare automaticamente uno sketch p5, impostando tutti i file e le librerie necessarie.

p5.js è una libreria JavaScript dedicata al Creative Coding. Creata da Lauren Lee McCarthy nel 2013 [24] e rilasciata nel 2014 da Ben Fry e Casey Reas [2], si propone come strumento per imparare facilmente a programmare e produrre arte. [25]

p5.js è basato sui principi chiave di Processing [24][26], un altro tool per il Creative Coding molto diffuso nella comunità, che però si basa sul linguaggio Java. Nasce infatti come "traduzione" di Processing per JavaScript e l'ambiente web, per poi evolversi autonomamente da allora in poi. Come avviene anche in Processing, un progetto di p5.js viene chiamato "sketch", per avvicinarsi anche nel linguaggio utilizzato al mondo dell'arte. Poiché JavaScript è ormai processabile in ogni browser, in p5.js ogni sketch può essere visto come una pagina web [24]. Oltre

alla facilità di disegno e visualizzazione, ciò aggiunge la possibilità di supportare più facilmente anche elementi testuali, audio-video, l'interazione con l'utente e con la webcam, l'integrazione dei componenti con CSS per migliorarne l'aspetto grafico. Tramite una serie di funzioni predefinite, gli utenti possono creare facilmente diversi tipi di forme più o meno complesse e manipolarle. Alle funzionalità di base si aggiungono poi diverse librerie, fornite dai creatori o create dalla community. p5.js offre un web editor per cominciare a programmare facilmente online, con la possibilità di visualizzare immediatamente il risultato del codice prodotto, ma può essere usato anche in locale, come una classica libreria JavaScript. Ad esempio, esistono numerose estensioni di Visual Studio Code che supportano p5, ad esempio evidenziando nell'editor le parole chiave.

CVC si basa su questa seconda modalità, ma si ispira all'editor web nella sua schermata principale, con il codice nel pannello di sinistra e l'anteprima in quello di destra Figura 4.1.

4.1.2 Git

Parlando di controllo delle versioni, non si può non nominare Git: nonostante esistano altri tool come Subversion, CVS, Perforce o ClearCase, è diventato lo strumento di Version Control più utilizzato, soprattutto nelle sue applicazioni online come GitHub [9]. Anche CVC si basa su Git, cercando di rendere un sottoinsieme delle sue funzionalità più accessibili e immediate per le esigenze dei creative coders.

Git è un sistema di version control gratuito e open source, adatto per gestire progetti di ogni dimensione in modo veloce ed efficiente.

Il concetto chiave di Git è il *commit*. Un commit rappresenta il salvataggio di un insieme di modifiche apportate al progetto, ed è univocamente identificato: ciò implica che è possibile riportare lo stato del progetto a un dato commit (questa operazione viene definita *checkout*). Git utilizza un approccio "differenziale": ad ogni commit non viene memorizzato l'intero contenuto dei file che compongono il progetto, ma soltanto le modifiche (linee di codice aggiunte e cancellate) rispetto al commit precedente. L'operazione di commit è preceduta da quella di stage, che permette di "preparare" il commit scegliendo quali modifiche includere e quali, eventualmente, no.

L'altro grande concetto, e la sostanziale differenza rispetto ad altri strumenti di controllo delle versioni, è quello di *branch*. Il flusso di sviluppo di un progetto può non essere lineare e consequenziale ma, a un certo punto, dividersi in diversi rami. Questa possibilità è molto utile soprattutto in caso di divisione del lavoro e contesti collaborativi, oppure in casi di sviluppo continuo, dove un branch rappresenta una versione stabile e distribuibile del prodotto, mentre altri sono dedicati allo sviluppo della versione successiva. Esiste poi la possibilità di ricongiungere due diversi branch, con l'operazione di *merge*. Poiché ogni branch continua a essere composto

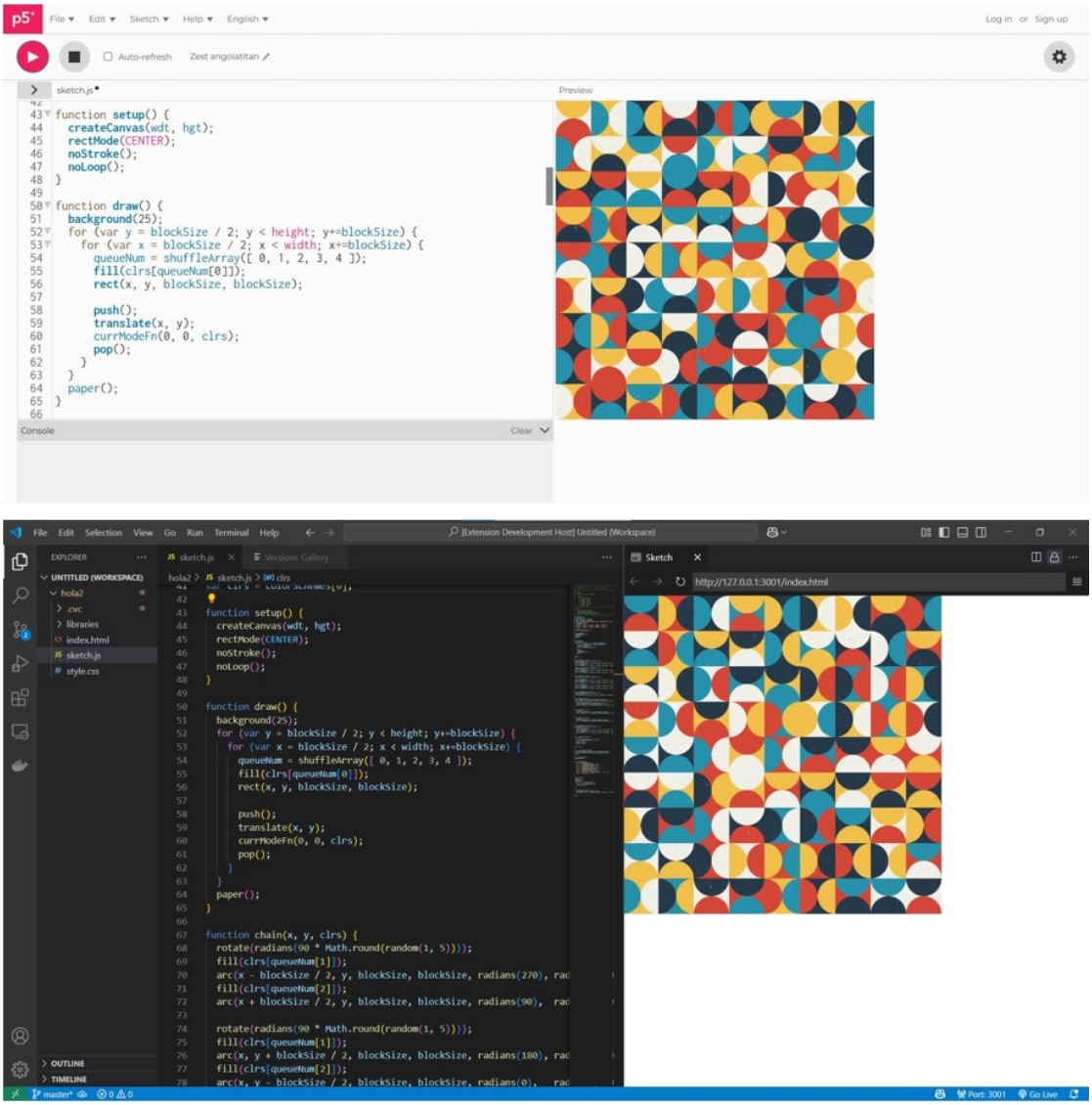

Figura 4.1: Confronto tra il layout dell'editor web di p5 e la schermata principale di CVC in Visual Studio Code

da commit successivi, la stessa operazione di checkout già vista in precedenza permette di spostare il focus del lavoro da un branch all'altro.

Git è un sistema "distribuito": tutte queste operazioni sono svolte localmente nelle diverse istanze del *repository* - così si definisce la cartella principale che contiene un progetto gestito con Git. Ciò comporta un vantaggio in termini di prestazioni rispetto a sistemi centralizzati, in cui tutte le operazioni vengono svolte comunicando con un unico server. Per permettere di gestire la comunicazione da

remoto, Git aggiunge altre operazioni, tra cui le principali sono *clone*, per creare una nuova istanza del repository in un dispositivo locale, copiandolo interamente - ciò offre contemporaneamente una soluzione al problema del "single point of failure" -, *push* per caricare nell'istanza condivisa le modifiche effettuate in locale e *pull* per riversare in locale l'ultima versione dell'istanza condivisa. [9]

4.1.3 Visual Studio Code

CVC è un'estensione per Visual Studio Code: un editor per codice sviluppato e rilasciato da Microsoft, e attualmente il più utilizzato dalla comunità dei programmatori [27]

Non è un editor specifico per un linguaggio: al contrario, cerca di essere il più generico possibile per adattarsi a qualsiasi tipo di linguaggio, framework e tecnologia. Il principale punto di forza di Visual Studio Code è la sua capacità di integrare in un'unica piattaforma tutti gli strumenti di cui il programmatore ha bisogno nel ciclo di scrittura, compilazione e debug del codice. Tutto questo è possibile principalmente grazie al suo sistema di estensioni, alcune prodotte da Visual Studio Code stessa, ma per la maggior parte sviluppate da altri utenti, che offrono una buona esperienza di scrittura di codice per moltissimi linguaggi, una comoda navigazione all'interno del file system dell'utente, un supporto per la fase di debug, di version control, e per l'integrazione con strumenti esterni già esistenti. [28]

Microsoft mette a disposizione API che permettono a chiunque di sviluppare un'estensione per Visual Studio Code, e il processo di sviluppo è supportato all'interno della piattaforma stessa.

Un'estensione può contribuire alla piattaforma in diversi modi. Di seguito sono elencati i principali: [23]

- *Comandi:* aggiungere un nuovo comando alla palette dei comandi; è anche possibile collegarlo a un "action button" da aggiungere in una delle sezioni di VS Code per chiamarlo più rapidamente, o ancora specificare i casi in cui il comando sia raggiungibile o meno.

- *Temi:* personalizzare l'editor con nuovi colori e icone.

- *View:* aggiungere nuove interfacce da visualizzare. Due sono le opzioni principali: *Tree View*, un contenitore di dati e altre view gestito internamente da VS Code per la parte grafica, lo sviluppatore deve soltanto fornire il contenuto che viene strutturato secondo un modello gerarchico, ad albero; oppure *Webview*, in cui tutto il progetto grafico è lasciato allo sviluppatore (che è comunque tenuto a seguire le UX Guidelines di Visual Studio Code). Una webview funziona esattamente come una pagina web, è scritta in HTML e permette di

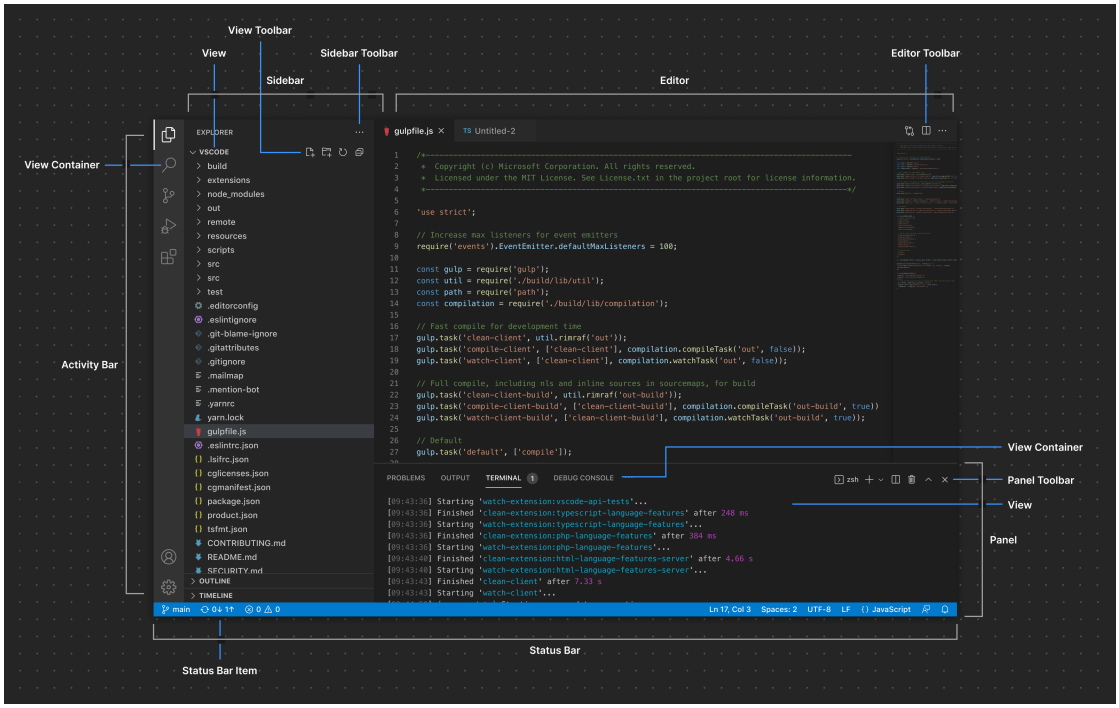

Figura 4.2: Le diverse componenti dell'interfaccia di Visual Studio Code

realizzare componenti più specifiche, non incluse tra quelle offerte dal modello Tree View.

- *Custom Editor:* aggiungere funzionalità all'editor del codice, ad esempio la colorazione delle parole chiave per diversi linguaggi, i suggerimenti o i completamenti automatici dei principali costrutti, o in generale operazioni sul testo.

- *Task Provider:* definire una serie di operazioni che la shell può eseguire automaticamente quando necessario.

- *Source Control:* gestire la provenienza dei file su cui si sta lavorando, le interazioni con il file system locale o remoto e l'integrazione con strumenti di controllo delle versioni.

- *Debug:* aggiungere funzionalità di debug, specifiche per le diverse esigenze.

- *Test:* aggiungere e gestire strumenti e processi per la fase di test.

4.1.4 Live Preview

Alcune estensioni di Visual Studio Code espongono dei punti di accesso utilizzabili da altre estensioni. Ad esempio, CVC per visualizzare l'anteprima istantanea del codice si avvale di Live Preview.

Live Preview [29] è un'estensione fornita da Microsoft che permette a Visual Studio Code di ospitare localmente un server HTML in grado di leggere e renderizzare in tempo reale i file HTML su cui si sta lavorando.

4.1.5 Playwright

Oltre a visualizzare l'anteprima istantanea, CVC è in grado di salvare immagini dello sketch per mostrarle nella galleria delle versioni. Per fare questo si serve di Playwright. [30]

Playwright è una libreria open-source che permette di avviare un browser in modalità *headless*, ovvero senza alcuna interfaccia visibile per l'utente, ed eseguire in esso diverse operazioni.

Il suo utilizzo principale è l'automatizzazione di test per applicazioni web, tuttavia nel contesto di CVC è stata usata per aprire nel browser lo sketch desiderato, individuare il *canvas*, cioè l'area in cui si trova l'opera d'arte, e salvarne uno screenshot.

4.2 Struttura del'applicazione

4.2.1 Struttura di un'estensione per Visual Studio Code

CVC è scritto in linguaggio TypeScript. Si basa quindi su *Node.js* come runtime system e su *Node package manager (npm)* per la gestione delle dipendenze.

All'interno del folder principale, in questo caso `creative-version-control`, un'estensione per Visual Studio Code deve necessariamente avere due file JSON: `package.json` e `package-lock.json`.

`package.json` contiene il nome dell'estensione e le informazioni più rilevanti, la definizione di script e dipendenze, ma soprattutto l'elenco dei "contributi", ovvero di tutti gli elementi - comandi, views, menu... - che l'estensione fornisce a Visual Studio Code. Se un elemento non viene registrato in questo file non verrà riconosciuto da Visual Studio Code al momento dell'esecuzione.

```
1 Alcune linee del file "package.json" di CVC
2 {
3   "name": "cvc",
4   "displayName": "Creative Version Control",
5   "description": "A Version Control System tailored upon the needs
      of Creative Coders!",
```

```
6   "version": "0.0.1",
7   "engines": {
8     "vscode": "^1.93.0"
9   },
10  "extensionDependencies": [
11    "vscode.git"
12  ],
13  "main": "./out/extension.js",
14  "contributes": {
15    "commands": [
16      {
17        "command": "cvc.new_p5",
18        "title": "New p5* Project with CVC"
19      },
20      {
21        "command": "cvc.open_p5",
22        "title": "Open p5* Project with CVC"
23      },
24
25      ...
26
27    ],
28    "menus": {
29      "editor/title": [
30        {
31          "command": "cvc.save",
32          "group": "navigation",
33          "when": "editorFocus"
34        }
35      ],
36      "editor/context": [
37        {
38          "command": "cvc.add_to_var_console",
39          "when": "editor.hasSelection"
40        }
41      ]
42    }
43  },
44
45  ...
46 }
```

`package-lock.json` contiene invece un elenco dettagliato delle dipendenze. Di ognuna sono specificate l'origine, la versione, le versioni di node che le supportano e, dove necessario, la licenza e il codice di integrità.

`creative-version-control` contiene altre tre cartelle: `node_modules`, che contiene i pacchetti di dipendenze gestiti da npm; `out`, che contiene il codice sorgente

compilato; `src` che contiene effettivamente il codice dell'estensione, gli strumenti e le risorse necessarie al suo funzionamento.

All'interno di `src` - e quindi di riflesso anche in `out` dopo la compilazione - un'estensione per Visual Studio Code deve necessariamente contenere un file chiamato `extension.ts`: al suo interno è definita la funzione `activate`, che viene eseguita ogni volta che l'estensione viene attivata, e rappresenta quindi il punto d'ingresso per tutti i contributi forniti dall'estensione.

4.2.2 Registrazione dei comandi

Con la funzione `activate` vengono "registrati", cioè definiti e segnalati a Visual Studio Code, tutti i comandi che l'estensione offre.

La funzione `activate` riceve come unico parametro `context`, di tipo `vscode.ExtensionContext`. Dove `vscode` rappresenta l'insieme di tutte le API messe a disposizione da Visual Studio Code, mentre l'*Extension Context* è una collezione di risorse e funzionalità assegnate all'estensione.

Per registrare un comando si usa la sintassi `vscode.commands.registerCommand("nome-comando", <callback>);` dove per convenzione il nome del comando contiene il nome dell'estensione seguito da un punto e dal nome effettivo del comando (ad esempio `"cvc.save"`) e deve corrispondere al nome di un comando registrato nel file `package.json`, mentre la *callback* rappresenta l'azione che il comando eseguirà: di solito si risolve con la chiamata a un'altra funzione. Infine, per "segnalare" il comando a VS Code viene usata la sintassi `context.subscriptions.push(<comando>);` - per non utilizzare un numero esagerato di variabili al posto di `<comando>` viene inserita l'intera sintassi per la registrazione vista in precedenza.

Per ragioni di leggibilità del codice, in CVC le callback non sono definite all'interno della funzione `activate`, come accade spesso in altre estensioni, soprattutto quelle più piccole, ma in un file separato: `src/commands/commands.ts`. Di seguito sono elencati i principali comandi definiti.

- `"cvc.new_p5"`, associato alla funzione `newProject()` definita in `commands.ts`, permette di inizializzare un nuovo progetto p5 che possa utilizzare CVC per il controllo delle versioni. Dopo aver chiesto all'utente di indicare il nome del progetto e la sua posizione nel file system, il comando crea una nuova cartella in quella posizione e la riempie di tutti i file necessari, inizializza nella stessa cartella un repository di Git e infine apre in Visual Studio Code una visualizzazione simile a quella che si può trovare nel web editor di p5, aprendo il file `sketch.js` nel pannello a sinistra e l'anteprima nel pannello a destra.

- `"cvc.open_p5"`, associato alla funzione `openProject()`, permette di aprire in Visual Studio Code un progetto p5 gestito con CVC già esistente. Dopo aver

fatto selezionare all'utente la cartella desiderata, il comando apre in Visual Studio Code la stessa configurazione descritta nel comando precedente.

- `"cvc.save"`, associato a `saveProject()`, serve per il salvataggio delle *Version*. Offre all'utente due possibilità: sovrascrivere le modifiche sulla Version su cui sta attualmente lavorando, oppure salvarle come nuova Version, con la possibilità di indicarne un nome, una breve descrizione e un messaggio per il commit su Git.

- `"cvc.save_current_variation"` e `"cvc.save_as_new_variation"`, con le rispettive funzioni `saveCurrentVariation()` e `saveAsNewVariation()` gestiscono il processo di salvataggio di una Variation. Con la prima le modifiche vengono salvate sovrascrivendole alla Variation corrente, mentre con la seconda vengono salvate come nuova Variation.

- `"cvc.ver_gallery"`, associata a `showVersionsGallery()`, permette di mostrare all'utente la galleria delle versioni, che verrà descritta in seguito (sezione 4.2.4).

- `"cvc.change_active_version"` e `"cvc.change_active_variation"`, associati alle funzioni `changeActiveVersion()` e `changeActiveVariation()`, permettono di aggiornare lo stato del progetto portandolo alla Version o Variation desiderata.

- `"cvc.var_console"`, associato a `showVariationConsolePanel()`, mostra in VS Code la Console relativa alla Variation attualmente in uso.

- `"cvc.add_to_var_console"`, infine, associato a `addParameterToVariationConsole()`, permette di selezionare il valore di una variabile nel codice e aggiungerlo tra i parametri della Variation. Se non è già visibile, apre la Variation Console.

4.2.3 Gestione del repository Git

La funzione `activate` serve anche per impostare e gestire l'interazione con Git. Al suo interno conserva un riferimento al repository Git del progetto a cui si sta lavorando. Questo riferimento può essere passato ai comandi che di volta in volta vengono chiamati per eseguire le operazioni di commit e checkout necessarie. All'interno di `activate` è definita anche la funzione `initGitRepository()`, che viene utilizzata dal comando `newProject()` per inizializzare correttamente il repository ed effettuare il primo commit. Tutte le operazioni di Git sono messe a disposizione dall'estensione di Git per Visual Studio Code, e sono definite nella cartella `src/git` del progetto.

4.2.4 Webviews

CVC definisce da zero due nuove View. Non viene considerata nel computo la schermata iniziale, che è costruita mettendo insieme componenti già esistenti, come il classico editor di testo di VS Code, e usando componenti fornite da altre estensioni come LivePreview. Le due view sono `Versions Gallery` e `Variation Console`. Entrambe sono definite come *webview*, perché hanno bisogno di componenti diversi da quelli messi a disposizione dalla Tree View di VS Code e di una gestione personalizzata dell'area dello schermo. Per mantenere la coerenza visiva con l'intera piattaforma di Visual Studio Code, sono stati utilizzati i componenti messi a disposizione da `vscode-webview-ui-toolkit`, una libreria per TypeScript che fornisce appunto componenti HTML, come bottoni, aree di testo, tabelle, input di vario genere, che seguono tutte le linee guida per l'interfaccia utente di VS Code. All'interno del folder `src`, la cartella `webview` contiene un file `main.ts` che serve per poter importare nel progetto e utilizzare i componenti di `vscode-webview-ui-toolkit` e alcuni file CSS per definire meglio i dettagli dei componenti.

I file che definiscono le due webview, invece, sono collocati nel folder `panels`.

Versions Gallery

Il file `cvcVersionsGallery.ts` definisce una classe `CvcVersionsGallery`. Nel *costruttore* viene specificato che il pannello avrà come titolo `"Versions Gallery"` e, con la sintassi `vscode.ViewColumn.One`, viene specificato che dovrà apparire nella prima colonna a sinistra dell'area di lavoro di Visual Studio Code, sostituendo quindi l'editor di testo.

La classe `CvcVersionsGallery` ha tra gli attributi la lista delle *Versions* dello sketch su cui si sta lavorando e una lista delle versioni segnalate dall'utente come preferite.

Di seguito nel file viene definito il codice HTML che darà forma alla webview, che è costruita in questo modo:

- Il titolo "Versions Gallery"
- Un input di tipo "checkbox" per filtrare solo le *Version* preferite.
- La "Galleria" vera e propria in cui le anteprime di tutte le versioni, in formato quadrato, sono disposte in una griglia regolare, che può ricordare un *feed* di Instagram o la galleria degli sketch di Open Processing. Quando il cursore passa su una di queste anteprime, vengono mostrati il nome e la data di creazione della *Version*. Nella parte alta dell'anteprima si trovano due bottoni: a sinistra una stella che indica la *Version* attiva, a destra un cuore segnala le *Version* preferite. Entrambi i bottoni rimangono visibili per tutto il tempo

solo quando sono selezionati, altrimenti vengono mostrati soltanto in caso di *hover* sull'area dell'anteprima.

Per gestire le possibili azioni dell'utente viene utilizzato il sistema di messaggi tra interfaccia e applicazione messo a disposizione da VS Code. La funzione `setWebviewMessageListener()` ha quindi lo scopo di intercettare i messaggi emessi dalla webview e innescare le risposte necessarie da parte dell'applicazione.

Con un click su una delle anteprime viene aperto un altro *panel*: si tratta di "Version Details", un pannello che offre una panoramica più approfondita di una singola *Version*. Nella costruzione della webview, l'opzione `vscode.ViewColumn.Two` indica che verrà mostrato nella seconda colonna a partire da sinistra dell'area di lavoro. Anche per "Version Details" vengono definiti un codice HTML e un Listener per i messaggi. Dopo il titolo, il pannello presenta l'anteprima dello sketch in forma completa, non limitata al quadrato della Gallery, alcune informazioni come il nome e un testo di descrizione, le date di creazione e ultima modifica. Nel contesto di "Version Details" il nome e la descrizione possono essere modificati. Dopo tutto ciò, il pannello contiene anche una galleria di tutte le *Variation*, costruita nello stesso stile di quella vista in precedenza.

Variation Console

Come nel caso precedente, anche nel file `VariationConsolePanel` viene definita una classe omonima che contenga tutte le funzionalità necessarie. In questo caso il *panel* viene costruito con l'istruzione `ViewColumn.Beside`, per indicare che la webview sarà visualizzata in una nuova colonna, a destra di quella al momento attiva. Anche questa classe contiene il codice HTML e il Message Listener. Dopo il titolo, il panel mostra due campi di testo modificabili per il nome e la descrizione della *Variation*. Segue quindi l'elenco dei parametri che caratterizzano la *Variation*, ognuno dei quali è identificato dal nome e dalla sua posizione nel file. Accanto a ogni parametro si trova un componente di input, diverso in base al tipo di dato preso in considerazione, che permette di modificarlo, e ancora più a destra un bottone "×" per rimuovere il parametro dall'elenco.

Per concludere, il panel presenta una coppia di bottoni. Il primo, di tipo *primary*, permette di salvare le modifiche effettuate sovrascrivendole nella *Variation* correntemente attiva; il secondo, di tipo *secondary*, consente di salvarle come una nuova *Variation*.

4.2.5 Template

L'ultima cartella da analizzare tra quelle presenti in `src` è quella denominata `template`. Al suo interno si trovano tutti i file necessari per inizializzare un progetto

p5 con CVC, pronti per essere copiati nel file system dell'utente che desidera creare un nuovo sketch. In particolare, il contenuto del folder `template` è il seguente:

- `sketch.js` è il file principale in cui verrà scritto il codice dello sketch. Inizialmente non è vuoto, ma come punto di partenza viene definito un *canvas* di $400px \times 400px$ di colore grigio, come mostrato in Figura 4.3.

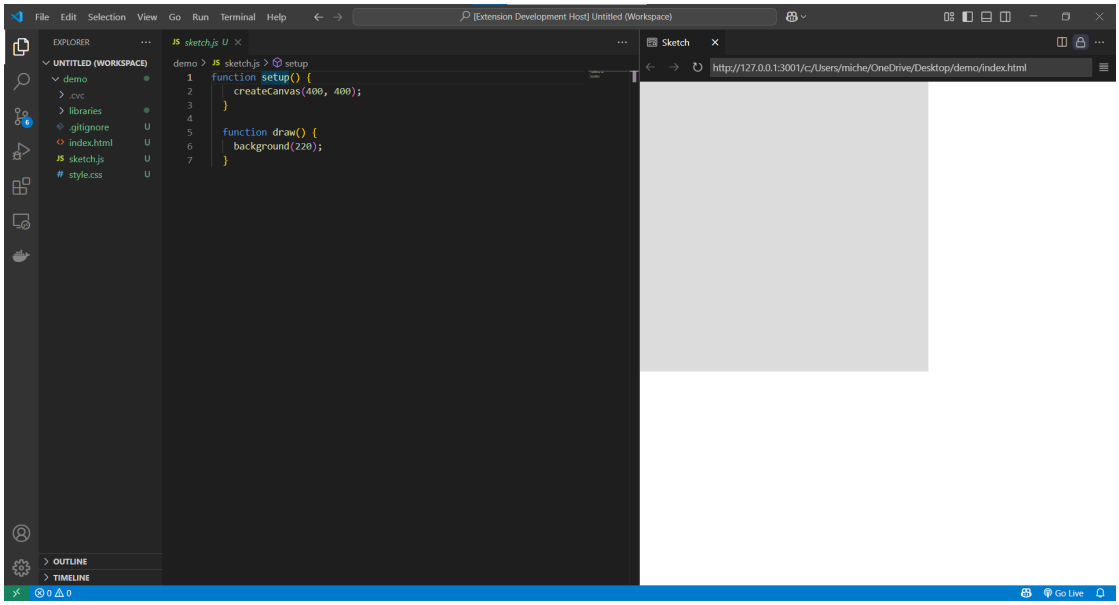

Figura 4.3: Il contenuto del file `sketch.js` all'interno del template iniziale

- `index.html` definisce una pagina HTML che importa le librerie di p5 ed esegue come script il codice di `script.js`. Viene letta da Live Preview per mostrare l'anteprima istantanea.

```
Il contenuto di index.html

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial
   -scale=1.0">

    <title>Sketch</title>

    <link rel="stylesheet" type="text/css" href="style.css">

    <script src="libraries/p5.min.js"></script>
    <script src="libraries/p5.sound.min.js"></script>
```

```
15    </head>
16
17    <body>
18      <script src="sketch.js"></script>
19    </body>
20  </html>
21
```

- `libraries` è un folder che contiene le due librerie base di p5: `p5.min` e `p5.sound.min`.
- La cartella nascosta `.cvc` è alla base del funzionamento dell'estensione. Contiene il file `cvc.json`, di cui si tratta più approfonditamente nel paragrafo successivo, ed è predisposta per contenere tutti i file `png` delle anteprime dello sketch che verranno realizzate.
- Per concludere, il folder contiene anche un file `.gitignore`, predisposto per escludere la cartella `.cvc` dalle operazioni di Git. In questo modo i commit e, soprattutto, i checkout non interferiscono con il file JSON e con le anteprime salvate.

4.2.6 File cvc.json

Il file `cvc.json` contiene tutte le informazioni necessarie per la gestione delle *Version* e delle *Variation* in CVC. Per l'estensione ha la funzione di un leggero database, poiché permette di conservare i dati che hanno bisogno di rimanere persistenti anche quando l'estensione non è attiva. Il file contiene alcune informazioni generali, come il nome del progetto e il suo percorso nel File System, poi una collezione di oggetti `version`, ognuno dei quali può a sua volta contenere una collezione di oggetti `variation`. Le operazioni su questo file sono definite nella cartella `json_utils` del progetto CVC, che a sua volta si appoggia sui file che definiscono le entità `Version` e `Variation`, contenuti nella cartella `models`.

```
1  Un esempio di file cvc.json
2
3  {
4      "projectName": "demo",
5      "projectFolderPath": " ... path ... \\demo",
6      "versions": [
7          {
8              "favorite": false,
9              "active": true,
10             "variations": [
11                 {
```

```
12                      "versionName": "master",
13                      "name": "Default",
14                      "description": "Default variation",
15                      "preview": " ... path ... \\demo\\.cvc\\
    version-master.png",
16                      "creationDate": "2025-03-29T09:22:26.355Z",
17                      "lastUpdateDate": "2025-03-29T09:22:26.355Z",
18                      "favorite": false,
19                      "active": true,
20                      "parameters": []
21                  },
22                  {
23                      "versionName": "master",
24                      "name": "Oceania",
25                      "description": "New palette",
26                      "preview": " ... path ... \\demo\\.cvc\\
    version-master-variation-Oceania.png",
27                      "creationDate": "2025-03-29T09:26:36.750Z",
28                      "lastUpdateDate": "2025-03-29T09:26:36.750Z",
29                      "favorite": false,
30                      "active": false,
31                      "parameters": [
32                          {
33                              "filePath": " ... path ... \\demo\\
    sketch.js",
34                              "name": "color1",
35                              "line": 52,
36                              "character": 40,
37                              "value": "#77C39C",
38                              "text": "#55ABA5",
39                              "type": "string"
40                          },
41
42                          ... other parameters ...
43
44                      ]
45                  },
46
47                  ... other variations ...
48
49              ],
50              "name": "master",
51              "description": "Initial version",
52              "preview": " ... path ... \\demo\\.cvc\\version-master
    .png",
53              "creationDate": "2025-03-29T09:22:26.355Z",
54              "lastUpdateDate": "2025-03-29T09:33:03.216Z"
55          }
56      ]
```

```
57 }
```

4.3 Principali funzionalità implementate

In questa sezione viene mostrato come l'utente può accedere e interagire con le funzionalità di CVC, arricchendo il tutto con screenshot effettuati durante l'esecuzione dell'applicazione.

4.3.1 Pre-requisiti e installazione

Una volta messa in produzione, l'estensione CVC potrà essere scaricata, come ogni altra estensione, dallo store di Visual Studio Code.

Per il momento, per provarla e testarla, basta scaricare il repository da GitLab e aprire la cartella in Visual Studio Code. Bisogna avere l'accortezza di installare tutti i pacchetti dei `node modules` di cui l'applicazione ha bisogno, e di inizializzare Playwright, lanciando questi comandi da un terminale aperto all'interno della cartella del progetto:

```
1 npm i
2 npx playwright install
```

Queste istruzioni possono essere automatizzate nella distribuzione definitiva.

VS Code offre la possibilità di effettuare il debug direttamente al suo interno: verrà aperta una seconda istanza dell'applicazione che funziona come se l'estensione da testare fosse già installata e attiva.

Per funzionare, CVC ha bisogno che sia attiva l'estensione di Git per Visual Studio Code. Non è neanche da installare, perché già presente di default nella piattaforma, ma deve essere attivata.

Viene poi suggerito di installare altre due estensioni:

- LivePreview, senza la quale non si avrà l'anteprima istantanea.
- Una delle molte estensioni per p5, che consentirà la colorazione del codice nell'editor di testo secondo le regole di p5 e, in alcuni casi, suggerimenti e auto-completamenti.

L'assenza di queste estensioni non impedisce il funzionamento di CVC, ma ne limita alcuni aspetti dell'esperienza.

Infine, in Visual Studio Code è possibile attivare la funzionalità di salvataggio automatico di ogni modifica apportata ai file su cui si sta lavorando. Questa

opzione potrebbe essere d'aiuto agli utenti che chiedevano un modo per salvare automaticamente a intervalli di tempo predefiniti: in questo modo per salvare una *Version* o una *Variation* c'è sempre bisogno di un'azione volontaria dell'utente, ma il lavoro svolto non rischia di essere perso. Per attivare questa funzionalità basta aprire il menu `File` e selezionare l'opzione `Auto Save`

4.3.2 Come raggiungere i comandi

Come avviene per molte altre estensioni, il modo per accedere alle funzionalità di CVC è la *Palette dei comandi* (Figura 4.4). Per chi utilizza abitualmente Visual Studio Code si tratta di un'operazione classica, che viene effettuata in molte altre occasioni. Per chi la utilizza per la prima volta, invece, si tratta di un'unica combinazione di tasti o di due click del mouse. Ci sono infatti due modi per accedere ai comandi:

- Con la combinazione `Ctrl + Shift + P`

- Con un click sulla barra di ricerca posizionata centralmente in alto, e poi con un secondo click sull'opzione `Show and Run Commands`, la seconda dell'elenco che viene presentato.

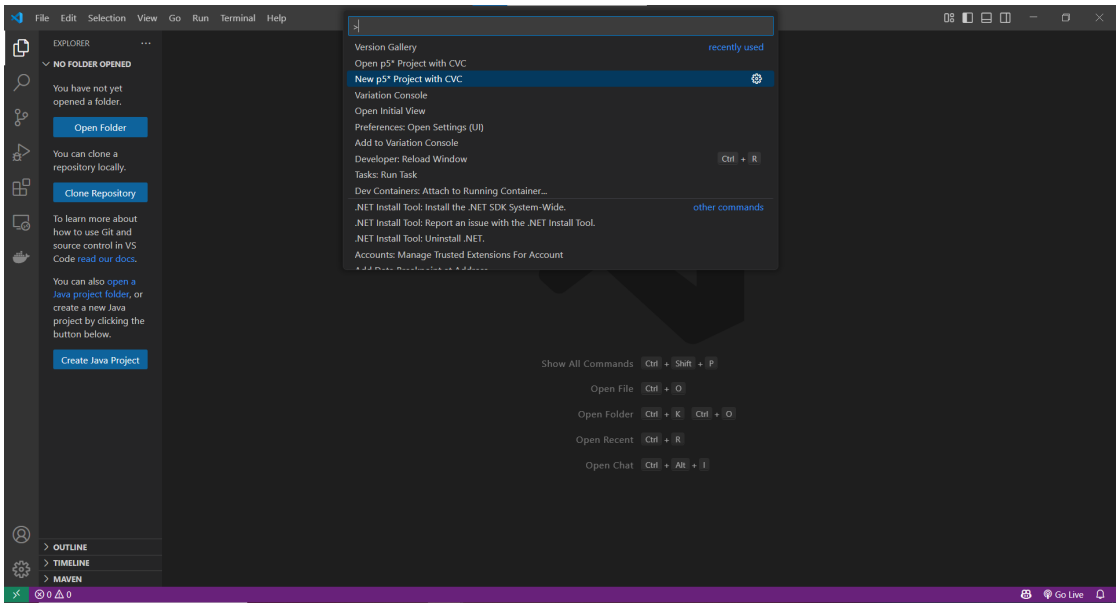

Figura 4.4: La *Palette dei comandi* in Visual Studio Code

4.3.3 Creare un nuovo progetto

Una volta raggiunti i comandi, per creare un nuovo progetto con CVC basta scegliere il comando `"New p5* project with CVC"`.

Nella posizione in cui si trovava la lista dei comandi verrà visualizzato un input testuale in cui Visual Studio Code chiede di inserire il nome del nuovo progetto, quindi si aprirà una finestra di dialogo con il File System in cui l'utente potrà scegliere dove collocare la nuova cartella, infine verrà costruita la schermata iniziale con il codice dello sketch sulla sinistra e l'anteprima sulla destra, come mostrato in Figura 4.3

Se invece si vuole aprire un progetto già creato in precedenza, basta scegliere il comando `"Open p5* project with CVC"` e scegliere dalla finestra di dialogo con il File System la cartella del progetto desiderato.

4.3.4 Salvataggio di una Version

Per salvare una *Version* è stato aggiunto un bottone con la classica icona per il salvataggio nella parte alta dell'editor (Figura 4.5).

Un menu a tendina permette di scegliere tra due opzioni: salvare le modifiche rimanendo all'interno della *Version* corrente, oppure creare una nuova *Version* (Figura 4.6).

L'applicazione chiederà di inserire, facoltativamente, un nome e una breve descrizione per la nuova *Version*, mentre in entrambi i casi darà l'opportunità di scrivere un messaggio per il commit su Git.

4.3.5 Gestione di una Variation

Per gestire una *Variation* è stato creato lo strumento della `Variation Console`, che può essere raggiunto in due modi:

- selezionando il valore di un parametro nell'editor del codice e cliccando con il tasto destro del mouse, tra le opzioni del menu si può scegliere `"Add to Variation Console"`, come in Figura 4.7. Verrà aperta la `Variation Console` della *Variation* attualmente in uso, con l'aggiunta del parametro selezionato;
- con un click sull'anteprima della *Variation* desiderata, tra quelle mostrate nel pannello dei dettagli della *Version* scelta.

La `Variation Console`, che è possibile vedere in Figura 4.8, permette di:

- Inserire o modificare il nome dato alla *Variation* e la sua descrizione.
- Visualizzare la lista dei parametri che caratterizzano la *Variation*, aggiungerli e rimuoverli.

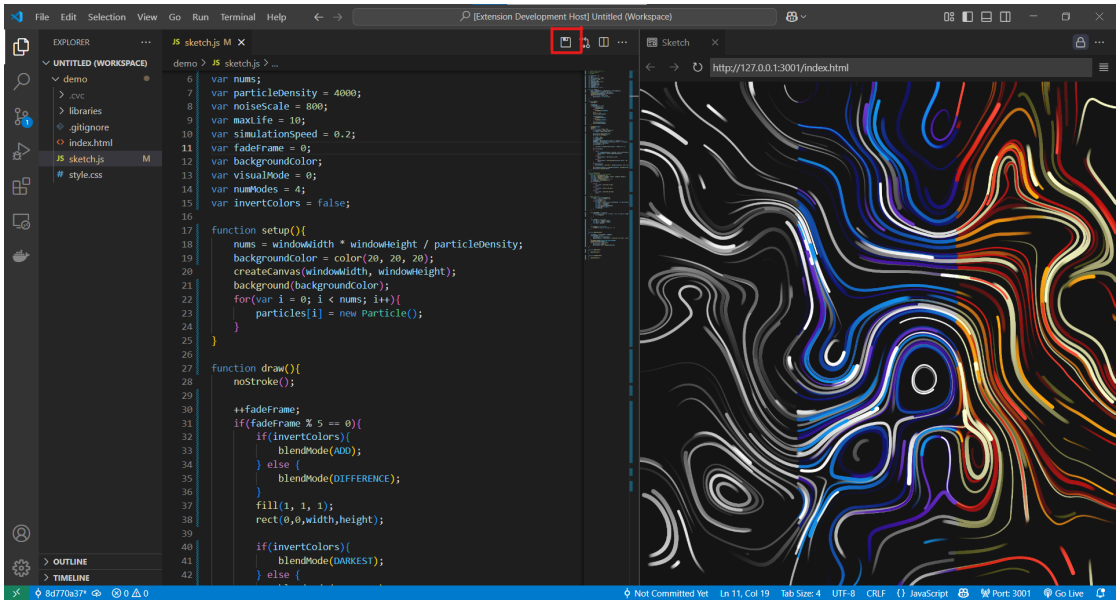

Figura 4.5: Il *Button* per il salvataggio di una *Version*

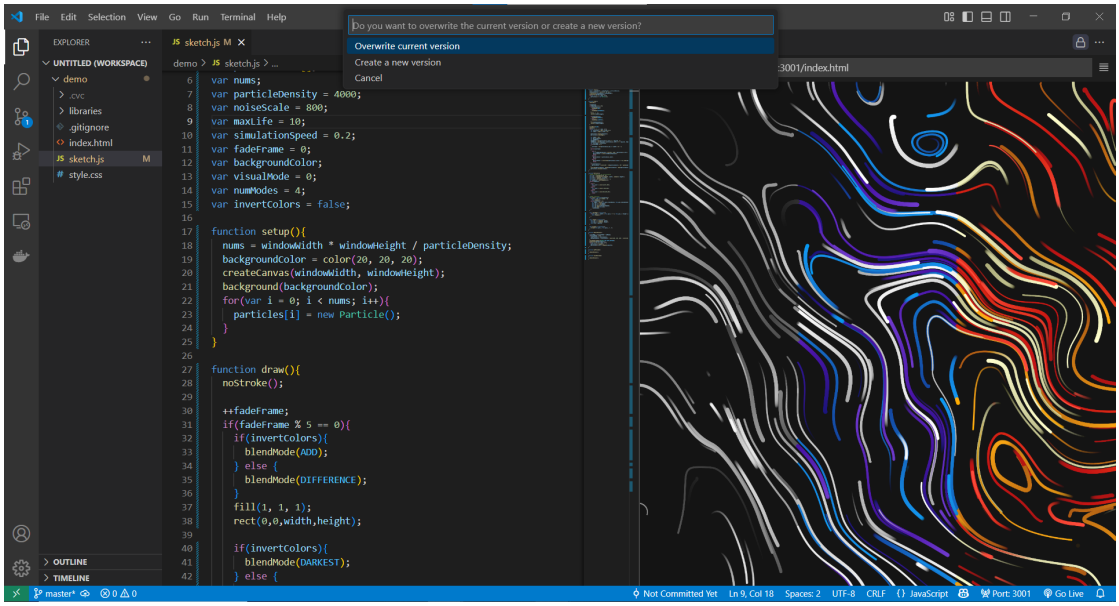

Figura 4.6: Le opzioni per il salvataggio di una *Version*

- Modificare il valore di uno dei parametri, e vedere in tempo reale come cambia l'anteprima dello sketch.

- Salvare le modifiche apportate come una nuova *Variation* o sovrascriverle alla

Variation in uso.

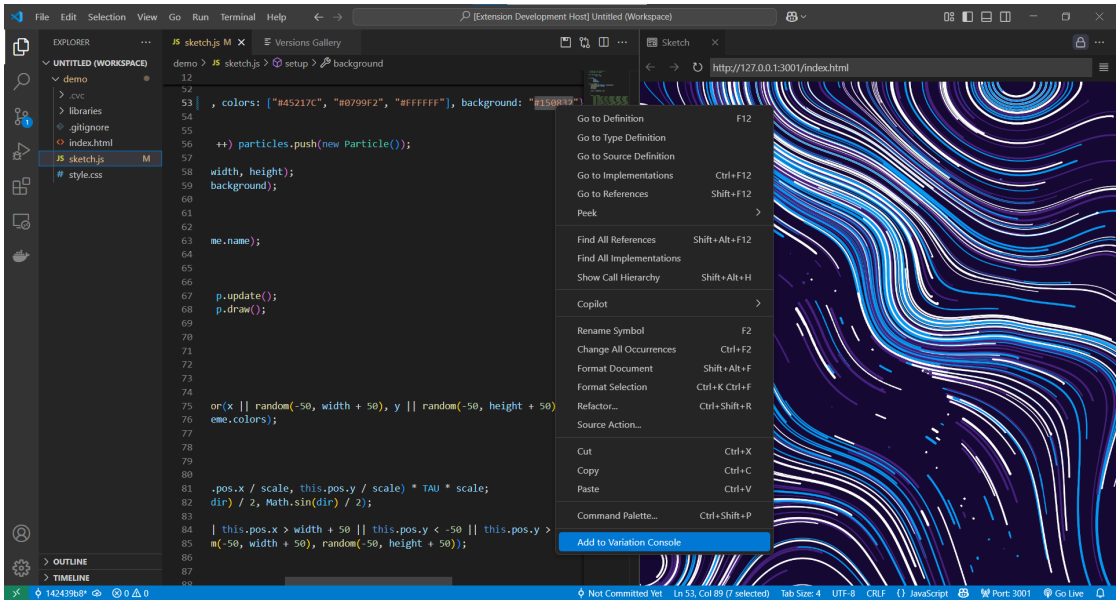

Figura 4.7: L'opzione per aggiungere un valore alla Variation Console

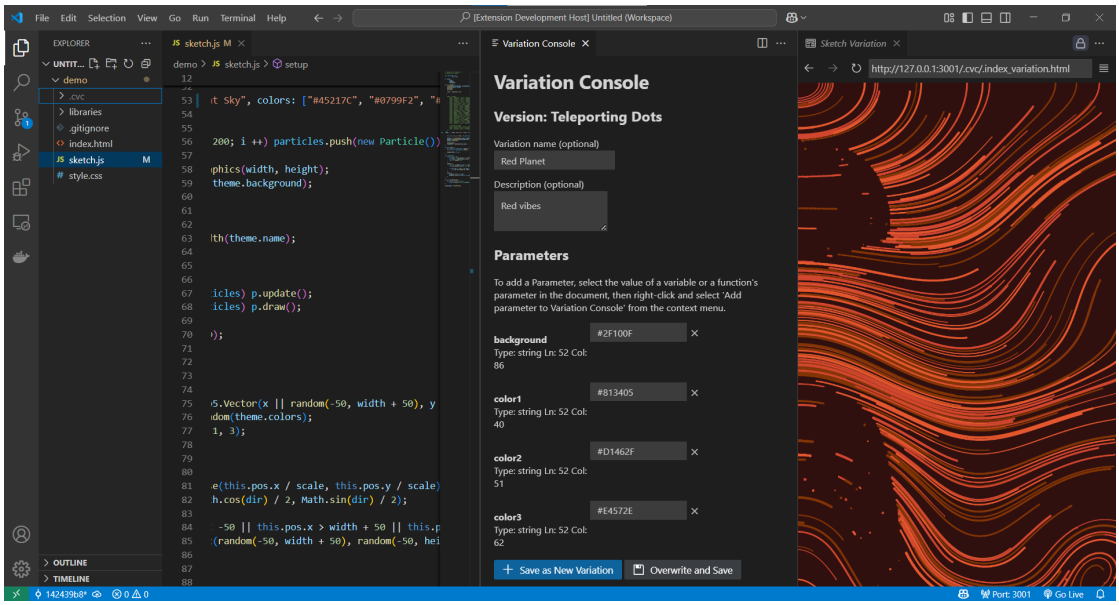

Figura 4.8: La Variation Console

Quando una nuova *Variation* viene creata, i valori dei parametri selezionati

vengono salvati nel file `cvc.json`, mentre non viene toccato il codice presente in *sketch.js* o in altri file del progetto.

Per rendere una data *Variation* quella principale, cioè per rendere i suoi valori quelli predefiniti nel codice della *Version* a cui si riferisce, basta selezionarla come attiva con il bottone della *stella* nella schermata dei `Version Details`.

Il sistema si occuperà di aggiornare le altre *Variation*, aggiungendo parametri che riportino i nuovi valori del codice a quelli con cui erano state create.

4.3.6 Versions Gallery

Si può accedere alla Versions Gallery attraverso la solita *Palette dei comandi*, scegliendo quello chiamato `"Versions Gallery"`. La Gallery viene aperta come mostrato in Figura 4.9.

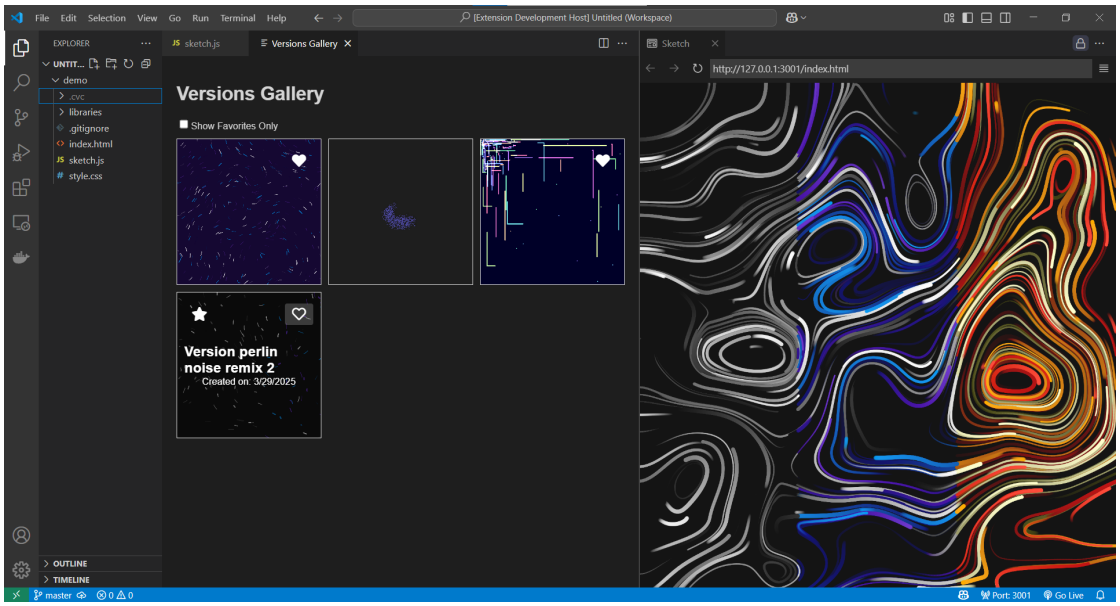

Figura 4.9: La Versions Gallery

Da questa schermata è possibile eseguire una serie di operazioni.

- Cambiare la *Version* attiva, cliccando sull'icona della stella in alto a sinistra
- Aggiungere una *Version* alla lista di quelle preferite, cliccando sull'icona del cuore in alto a destra dell'anteprima, oppure toglierla nello stesso modo
- Scegliere di visualizzare solo le *Version* preferite
- Visualizzare maggiori dettagli di una *Version*, cliccando sulla sua anteprima. I dettagli sono visualizzati come in Figura 4.10. Da qui è possibile modificare

nome e descrizione della *Version*, e visualizzare la galleria delle sue *Variation*. Nella *Variations Gallery* le opzioni per rendere una *Variation* attiva o preferita funzionano in modo analogo a quanto visto per le *Version*

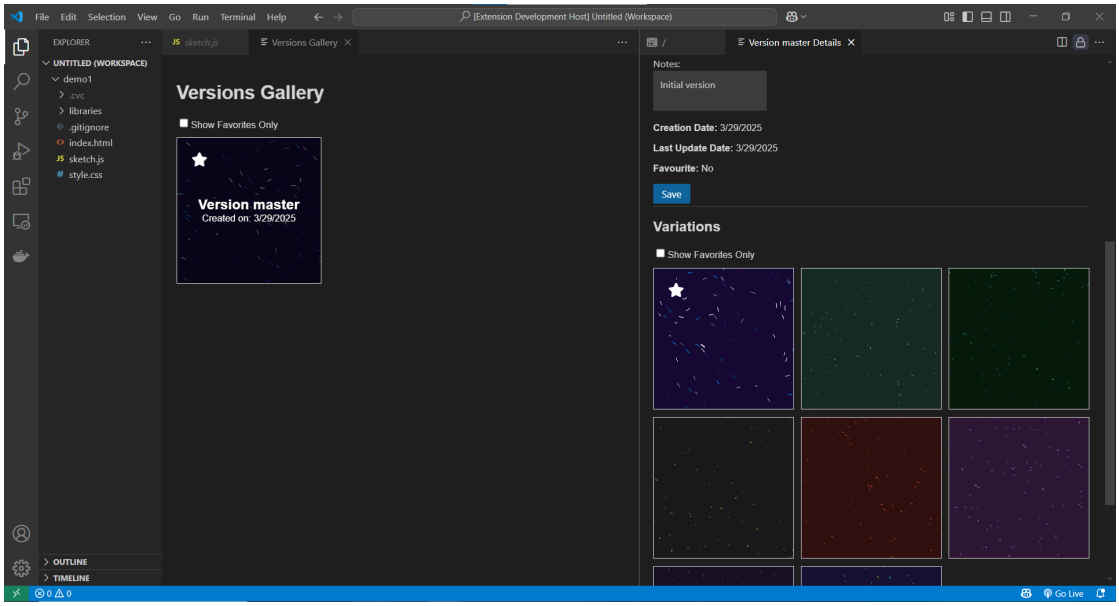

Figura 4.10: Il pannello che mostra i dettagli di una *Version* e la galleria delle sue *Variation*

Capitolo 5

Valutazione

Dopo aver realizzato una prima versione funzionante dell'applicazione, il processo di human-centered design prevede una fase di valutazione del prodotto. Dopodiché si torna a correggere le problematiche riscontrate e a sviluppare eventuali nuove idee emerse nel confronto con gli utilizzatori. Si ripete poi questo ciclo finché l'applicazione non è pronta per essere distribuita, ma anche in quel caso, auspicabilmente, il processo di raccolta di feedback dagli utenti e relativo adeguamento del prodotto va avanti.

Per ragioni organizzative e di tempistiche, questa tesi non contiene i risultati della fase di valutazione. Nei paragrafi seguenti verranno tuttavia presentate diverse possibilità di ottenere feedback su un'applicazione in sviluppo, e verranno proposte alcune linee guida per una valutazione di CVC.

5.1 Possibili modalità di valutazione

In questo contesto, la valutazione non riguarda i test funzionali, quelli automatizzati che verificano il funzionamento del software in ogni singola funzione, nell'integrazione tra tutte le componenti, nella gestione dei casi limite e nelle prestazioni a fronte di carichi di lavoro elevati. Tutto questo fa parte del processo di sviluppo del software, ed è importante tenerne conto durante la lavorazione, ma una volta terminata una fase di sviluppo bisogna anche considerare l'interazione con l'utente *umano*, che non è possibile simulare in modo automatico.

I test di usabilità si possono dividere in due macro-categorie in base al tipo di utenti coinvolti nella valutazione. Possono essere infatti effettuati sottoponendo il test all'utente *target*, cioè la categoria di utilizzatore per cui l'applicazione è pensata - in questo caso, ad esempio, un creative coder che programma utilizzando p5 - oppure a un esperto in usabilità. Da un lato l'*usability expert* ha meno conoscenze pratiche in merito al contesto del prodotto, ma dall'altro ha esperienza nel riconoscere le

principali problematiche di usabilità, che sono trasversali e ritrovabili in ogni tipo di applicazione che abbia un'interfaccia con l'utente, utilizzando per esempio le dieci euristiche di Nielsen [31][32] o altre classificazioni dello stesso tipo.

Quando l'applicazione viene distribuita, è poi possibile ottenere feedback e indicazioni usando come tester tutti gli utenti effettivi del software: raccogliendo dati sull'utilizzo, infatti, si possono vedere quali funzionalità sono usate di più, quali sono molto ricercate e quindi potrebbero essere rese più accessibili oppure, dall'altra parte, quali non vengono utilizzate, per approfondire la questione e capire se erano funzionalità inutili e non più necessarie, oppure presentano problemi di usabilità che ne ostacolano la fruizione.

5.2 Pianificazione di un test di usabilità

Un test di usabilità comprende le fasi di pianificazione, esecuzione e analisi.

Durante la fase di esecuzione un facilitatore propone ai partecipanti una serie di *task*, mentre una o più persone osservano e prendono nota dei successi e degli errori, dei dubbi e delle difficoltà che affrontano. Insieme ai *task*, di solito viene proposto anche un questionario.

Durante la fase di analisi, i dati raccolti dall'osservazione e dai questionari vengono classificati e analizzati, per tradurli in modifiche concrete da apportare all'applicazione o in idee per successivi sviluppi.

La fase di pianificazione serve per rendere più efficaci le due successive, e preparare il terreno per raccogliere informazioni nel miglior modo possibile. In particolare, in questa fase vanno prese decisioni su tutti gli aspetti del test menzionati poco sopra: tra le altre cose bisogna definire:

- I partecipanti al test, come tipologia di utente e come quantità.
- Chi svolgerà il ruolo di facilitatore, come lo svolgerà, e chi avrà invece il ruolo di osservatore.
- Quali *task* verranno proposti all'intervistato e quali metriche verranno utilizzate per valutarli.
- Il questionario da sottoporre durante il test.

5.2.1 I partecipanti

I partecipanti, come già detto, possono essere utenti esperti nel dominio dell'applicazione o nel campo dell'usabilità. Possono essere scelti con diversi livelli di esperienza o di età, o differenziarsi per altre competenze specifiche, oppure essere

omogenei in una o più di queste caratteristiche, se si vuole puntare a una fetta di utilizzatori più precisa.

Per quanto riguarda il numero, alcune ricerche, come quella di Nielsen [33], mostrano che con cinque partecipanti si riesce a individuare la maggior parte dei problemi. Aumentandone il numero, cresce anche la probabilità che i problemi evidenziati dagli ulteriori tester siano gli stessi già trovati in precedenza. Per questo motivo, questo tipo di test viene di solito effettuato con un numero limitato di partecipanti, e ciò influisce positivamente anche sulla complessità dell'analisi dei dati.

Insieme ai partecipanti, va anche definito l'ambiente in cui il test sarà svolto. In particolare, nei casi in cui l'applicazione non è ancora stata distribuita, il test può essere svolto nell'ambiente di sviluppo - sul computer degli sviluppatori, ma già pronto per funzionare - oppure nell'ambiente di lavoro dell'utente, e in questo caso c'è da stabilire se la fase di installazione e inizializzazione del prodotto sia parte dei compiti da valutare.

A volte è utile registrare un video del test. Il partecipante va perciò informato e gli va fatto firmare un modulo in cui accetta il trattamento dei dati ai fini della ricerca.

5.2.2 Il facilitatore

Il ruolo del facilitatore è quello di introdurre e condurre il test, e di essere di supporto all'intervistato in caso di problemi tecnici. Il suo compito è anche quello di ricordare all'intervistato che l'oggetto della valutazione non sono le sue abilità, ma il prodotto che sta utilizzando, e quindi esplicitare eventuali incertezze è perfino utile alla valutazione.

Ci sono diverse modalità per proporre un *task* al partecipante. La più semplice è introdurre il compito da svolgere e osservare come questo viene eseguito, senza interferire. Ci sono però dei casi in cui si vogliono ottenere più informazioni dall'utente, e a questo scopo due sono le tecniche più utilizzate: *think-aloud*, al partecipante viene chiesto di "pensare ad alta voce" per evidenziare i passaggi logici che lo portano a prendere le sue scelte, oppure la modalità *cooperative*, in cui utente e facilitatore dialogano e si fanno domande a vicenda per raggiungere insieme l'obiettivo prefissato. Entrambe le modalità escludono il tempo impiegato per eseguire il *task* dalle metriche utilizzabili per valutarlo, ma con il *think-aloud* si vuole capire se l'intervistato vede il sistema nello stesso modo di chi l'ha progettato o se alcune interfacce non sono chiare, mentre con una modalità *cooperative* il partecipante è invitato a esprimere pareri e impressioni sulla sua esperienza che probabilmente non direbbe se gli venissero chiesti esplicitamente.

Durante la fase di pianificazione vengono dunque definite le modalità di esecuzione di ogni *task*, e viene scritto un copione per il facilitatore: da una parte questo

lo aiuta a mantenere il flusso del test, dall'altra cerca di favorire la replicabilità del test, rendendolo quanto più possibile identico per tutti i partecipanti.

Di solito il facilitatore è una persona esterna al gruppo che ha sviluppato il software, per evitare che influenzi l'intervistato con indicazioni e suggerimenti, anche in maniera involontaria. Gli sviluppatori, invece, di solito sono tra gli osservatori.

5.2.3 Task, metriche e questionari

I *task* sono obiettivi che il partecipante al test deve completare compiendo una serie di azioni all'interno del sistema da valutare. Spesso per introdurre meglio il compito, il *task* viene inserito all'interno di uno scenario di utilizzo concreto. Solitamente, il numero dei *task* è compreso tra cinque e dieci.

Insieme al *task* sono definite le metriche con cui può essere valutato. Possono variare dalla semplice alternativa "completato/non completato" al conteggio degli errori commessi durante il completamento, oppure a una misurazione del tempo impiegato, o ancora a una valutazione chiesta al partecipante stesso.

I questionari possono essere sottoposti all'intervistato durante il test - dopo ogni *task* o dopo un gruppo di *task* - o alla fine. Possono riguardare le funzionalità specifiche dell'applicazione oggetto della valutazione, approfondendo da un lato le impressioni sul prodotto, dall'altro quanto le informazioni che volevano essere veicolate siano state comprese, o una serie di domande più generiche riguardanti l'usabilità. Per questo secondo tipo, esistono diversi questionari standard, uno dei più diffusi è il *SUS* [34] che prevede dieci domande con risposte che vanno da "completamente in disaccordo" a "completamente d'accordo".

5.3 Esempi di task e metriche

Di seguito vengono elencati alcuni esempi di *task* che potrebbero essere utilizzati per una valutazione di CVC. Il contesto considerato è quello di sviluppo, con l'estensione già installata e pronta a funzionare.

1. Partendo da un'istanza di Visual Studio Code vuota, senza file aperti, creare un nuovo progetto p5 con CVC. Per questo *task* potrebbe essere valutato il tempo, ma potrebbe essere interessante anche sentirne il *think aloud* per capire come viene percepita la principale modalità di accesso alle funzionalità di CVC.

2. È stato preparato un progetto con un codice più esteso, e che contiene già un certo numero di *Version*. L'utente dovrà aprire questo progetto e controllare il numero di *Version* presenti. Dal momento che questo *task* è composto da due sotto-obiettivi, può essere valutato per il raggiungimento di ognuno di essi. In aggiunta, può essere utile misurare il tempo impiegato.

3. Modificare il codice dello sketch e salvare una nuova *Version* del progetto. Siccome il test non riguarda le abilità creative dell'intervistato, può essere utile preparare in anticipo un blocco di codice da aggiungere al progetto. In questo caso il tempo impiegato non è rilevante, è più interessante capire come ragiona l'intervistato con una modalità *think aloud* o *cooperative*.

4. Riportare lo sketch alla *Version* creata in una specifica data. Per questo task è utile misurare il tempo impiegato, e prendere nota di eventuali tentativi errati o incertezze.

5. Tornando al codice, aggiungere una serie di parametri a una nuova *Variation* e salvarla. Anche in questo caso, non importa tanto il tempo impiegato, quanto osservare come interagisce con uno dei nuovi concetti introdotti da CVC, attraverso una modalità *think aloud* o *cooperative*.

6. Recuperare la *Variation* appena creata, contrassegnarla come preferita e aggiungere un commento alla sua descrizione. In quest'ultimo caso la valutazione può essere basata sul raggiungimento dei tre sotto-obiettivi, sui rallentamenti e gli errori avvenuti durante il processo, e sul tempo impiegato.

In un questionario finale possono essere inserite domande in merito all'efficacia della visualizzazione compatta delle informazioni. Mentre per quanto riguarda il supporto al processo creativo, probabilmente solo un utilizzo più prolungato nel tempo può offrire feedback degni di nota.

Capitolo 6

Conclusioni

L'obiettivo della tesi, e cioè la realizzazione di uno strumento per il version control creato appositamente per il creative coding, è stato completato.

Certo, Creative Version Control ha ancora bisogno di essere testato e valutato, di essere messo alla prova dell'utilizzo nel lavoro giornaliero degli artisti, e sicuramente si tratta di un punto di partenza che può essere migliorato ed espanso. Tuttavia, questa tesi è stata utile anche per evidenziare le peculiarità del metodo di lavoro dei creative coders, raccogliere le testimonianze più significative, e offrire una base di partenza anche a chi vorrà proporre soluzioni diverse.

La fase di ricerca iniziale ha mostrato che la comunità scientifica, in particolare nell'ambito di ricerca sull'*human-computer interaction*, recentemente si è rivolta spesso al mondo del creative coding, e il motivo risiede proprio nella scoperta di un modo di relazionarsi ai sistemi informatici diverso sia da chi li usa per lavoro, sia da chi ne è estraneo e li utilizza mediati da interfacce più accessibili per necessità.

Il processo creativo ed esplorativo tipico dell'arte viene citato in tutti gli studi sul creative coding, ma sono ancora pochi quelli che provano a scomporlo, analizzarlo e comprenderlo. L'auspicio è che i *needs* individuati in questa tesi possano essere uno dei punti di partenza per ulteriori approfondimenti su questo tema, sia per migliorare la nostra comprensione del rapporto tra l'uomo e la tecnologia, sia per offrire agli utenti strumenti sempre migliori e alla portata di tutti.

Il progetto sviluppato non ha la velleità di ridefinire l'approccio al version control, mettendosi in contrapposizione agli strumenti già esistenti. Al contrario, è costruito con Git, uno dei VCS più diffusi, efficienti e collaudati sul mercato. Quello che si vuole dimostrare è che, grazie a un'interfaccia appositamente progettata, questi strumenti possono essere resi accessibili anche ad altre categorie di utilizzatori che in precedenza ne stavano lontani.

Dal punto di vista personale, l'esperienza di questa tesi è stata l'occasione per scoprire e approfondire come sono fatti e come funzionano strumenti che avevo già utilizzato in passato, come i version control system, e Git in particolare, ma

soprattutto l'ecosistema di Visual Studio Code e delle sue estensioni. In questa tesi vengono presentate e analizzate tutte le tecnologie utilizzate, e per ognuna viene citata la documentazione ufficiale, perché capire come sono pensati e costruiti gli strumenti che si usano aiuta ad utilizzarli meglio, e aiuta l'ingegnere ad ampliare il proprio bagaglio di possibili soluzioni ai problemi che potrà affrontare.

6.1 Sviluppi futuri

Oltre a inserirsi nella scia degli studi sulla comunità dei creative coders - questa tesi non ha prodotto dati nuovi, ma ha provato a contribuire organizzando e analizzando quelli già esistenti sul rapporto tra creative coding e version control - il progetto realizzato, CVC, ha sicuramente margini per potenziali sviluppi futuri, miglioramenti da apportare alle funzionalità esistenti e aggiunta di nuove.

Alcuni spunti potrebbero arrivare dai risultati di una valutazione, che potrebbero indurre a ripensare alcuni meccanismi o a migliorare alcune scelte grafiche.

Altre idee, invece, sono già emerse durante le fasi di progettazione e realizzazione. Possono essere raggruppate in tre categorie: miglioramento di funzionalità esistenti, aggiunta di nuove funzionalità e estensione del prodotto a contesti diversi da p5.

Un esempio per il miglioramento degli elementi già esistenti è la posizione della `Variation Console`. Al momento viene presentata in un *panel* nello spazio di lavoro principale, affiancato a quello dell'editor e dell'anteprima dell'output (Figura 4.8). Questo, soprattutto in caso di schermi più piccoli, può creare un sovraffollamento e un'eccessiva suddivisione dello spazio di lavoro principale, perciò si potrebbe pensare di spostarla nel pannello inferiore - quello orizzontale dove di solito si trovano la shell dei comandi e la console di debug (Figura 4.2) - oppure nella *sidebar* secondaria, quella che appare a destra. In entrambi i casi si dovrebbe ripensare leggermente la sua composizione per adattarla a una visualizzazione orizzontale nel primo caso, o sempre verticale ma più stretta nel secondo.

Un altro possibile miglioramento potrebbe essere la possibilità di invocare i principali comandi senza passare ogni volta dalla *Command Palette* di Visual Studio Code, con bottoni accessibili direttamente dalla schermata principale.

6.1.1 Aggiunta di funzionalità

Le funzionalità qui presentate sono state considerate in fase di progettazione o durante la realizzazione. Alcune facevano anche parte del prototipo, ma per vari motivi si è deciso di dare maggior priorità a quelle presentate nel Capitolo 4, e accantonare (per il momento) queste.

- Si è già parlato della *Version Storyline* (sottosezione 3.3.3, Figura 3.4), che può avere la sua utilità nel rappresentare l'evoluzione di un progetto nel corso

del tempo e il percorso che ha condotto ad ottenere un certo sketch. Potrebbe essere implementata come visualizzazione alternativa della *Versions Gallery*, lasciando all'utente la scelta su quale delle due visualizzare e cambiare in base alle necessità del momento.

- Una funzionalità presa in considerazione all'inizio della fase di progettazione, e finita anche nel prototipo (Figura 6.1), è quella di uno strumento in grado di visualizzare le differenze (nel codice) tra due versioni dello stesso sketch. L'idea è stata accantonata quando si è deciso di sviluppare il progetto come estensione di Visual Studio Code basata su Git, perché Git, e in particolare la sua estensione per Visual Studio Code, offre già uno strumento molto simile. Quello che potrebbe fare una prossima versione di CVC è offrire la possibilità di chiamare questo strumento partendo dalle *Version* o dalle *Variation* anziché dai *commit* di Git, e/o associarlo a una rappresentazione visiva dell'impatto delle differenze sull'output generato.

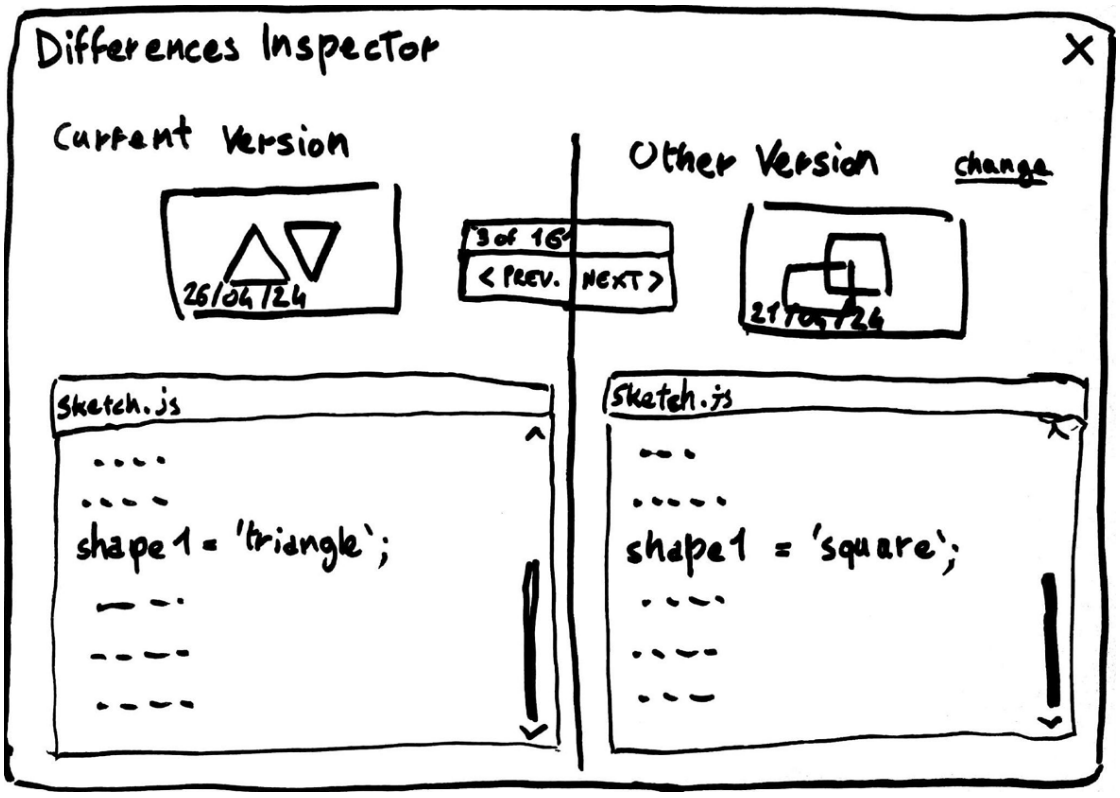

Figura 6.1: Il *Differences Inspector Tool* nel prototipo del progetto.

- Si potrebbero differenziare maggiormente i parametri della *Variation Console*, aggiungendo modalità di input specifiche per i diversi tipi di dato. Ad esempio,

la rappresentazione dei colori meriterebbe uno studio a parte, per capire come collegare tra loro e gestire tutte le modalità con cui JavaScript, p5, e le singole librerie permettono di gestire i colori e offrire una modalità di inserimento e modifica standard e funzionale. Oppure, i dati numerici che possono essere compresi in un range delimitato potrebbero essere rappresentati con uno slider su cui scorrere tra il valore minimo e il massimo senza dover digitare il numero ogni volta.

Potrebbe essere utile anche uno strumento che analizzi il codice dello sketch e proponga una lista di variabili e parametri tra cui scegliere quelli da aggiungere a una *Variation*.

- Al momento è possibile associare CVC a un progetto solo creandone uno nuovo, che quindi sia integrato con l'applicazione dall'inizio. Una funzionalità può essere quella di aggiungere CVC a un progetto già esistente. A livello teorico basterebbe inserire la cartella `.cvc`, il file `.gitignore` e inizializzare il repository Git, a livello pratico, tuttavia, bisognerebbe anche gestire l'eventuale presenza di un altro repository Git nel progetto o nomi e posizioni dei file possibilmente diverse da quelle che CVC si aspetta.

- Il tema della collaborazione tra diversi programmatori era stato messo da parte nelle fasi preliminari della ricerca (sottosezione 2.3.1), ma il fatto di aver implementato il progetto utilizzando Git rende possibile una sua estensione in questo senso, collegandolo ad esempio a GitHub. La possibilità di lavorare insieme a uno stesso progetto potrebbe addirittura incoraggiare la collaborazione tra i creative coders, che forse al momento è limitata anche per la carenza di strumenti adatti.

6.1.2 Oltre p5

Per come si presenta ora, Creative Version Control è un'estensione strettamente legata a p5. Tuttavia, la maggior parte delle sue meccaniche di funzionamento non è vincolata a uno specifico linguaggio di programmazione. Al contrario, le sue funzionalità sono progettate pensando ai bisogni dei creative coders in generale, e non in modo specifico per chi lavora con p5.

CVC ha dunque la possibilità di essere estesa anche ad altri contesti di lavoro, e questo è anche uno dei motivi per cui si è scelto di implementarla all'interno di una piattaforma versatile come Visual Studio Code.

Gli aspetti principali da ritoccare per ottenere questo risultato sarebbero soltanto due: l'inizializzazione del progetto e la visualizzazione delle anteprime.

Per quanto riguarda il primo punto, ci sono due possibili strade: aggiungere processi di inizializzazione dedicati a ogni diverso ambiente di lavoro, o renderlo

il più generico possibile in modo che un solo procedimento sia adatto a un ampio ventaglio di possibilità.

Il secondo punto è più delicato, perché non tutti i linguaggi permettono una visualizzazione istantanea del risultato come JavaScript. Si dovrebbero quindi adattare le strategie di visualizzazione per i linguaggi per cui è possibile, e offrire un supporto diverso dove invece non lo è, ad esempio aggiungendo la possibilità di caricare manualmente le anteprime da associare alle *Version* e alle *Variation*.

Bibliografia

[1] Mauricio Verano Merino e Juan Pablo Sáenz. «The Art of Creating Code-Based Artworks». In: *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems (CHI EA '23)*. Hamburg, Germany. ACM, New York, NY, USA, apr. 2023. URL: https://doi.org/10.1145/3544549.3585743 (cit. alle pp. 4, 8, 10, 16, 17, 21).

[2] Giacomo Vitali. «P5+: Uno strumento per migliorare l'esperienza degli artisti nel mondo del creative coding». Tesi di laurea mag. Politecnico di Torino, A.a. 2022/2023 (cit. alle pp. 4, 8, 29).

[3] *Ergonomics of human-system interaction — Part 210: Human-centred design for interactive systems*. ISO 9241-210:2019(en). (Visitato il giorno 20/03/2025) (cit. a p. 5).

[4] Tyler Angert, Miroslav Ivan Suzara, Jenny Han, Christopher Lawrence Pondoc e Hariharan Subramonyam. «Spellburst: A Node-based Interface for Exploratory Creative Coding with Natural Language Prompts». In: *The 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. San Francisco, CA, USA. ACM, New York, NY, USA, ott. 2023. URL: https://doi.org/10.1145/3586183.3606719 (cit. alle pp. 8, 11, 17, 22, 23).

[5] Nazatul Nurlisa Zolkifli, Amir Ngah e Aziz Deraman. «Version Control System: A Review». In: *3rd International Conference on Computer Science and Computational Intelligence 2018*. Kuala Nerus, Terengganu, Malaysia, ago. 2018. URL: //www.sciencedirect.com/science/article/pii/S1877050918314819 (cit. a p. 8).

[6] Nayan B. Ruparelia. «The history of version control». In: *ACM SIGSOFT Software Engineering Notes* 35 (gen. 2010), pp. 5–9. URL: https://dl.acm.org/doi/abs/10.1145/1668862.1668876 (cit. a p. 9).

[7] Stefan Otte. *Version Control Systems*. Computer Systems e Telematics, Institute of Computer Science, Freie Universitat Berlin, Germany, 2009. URL: https://www.mi.fu-berlin.de/inf/groups/ag-tech/teaching/2008-

09_WS/S_19565_Proseminar_Technische_Informatik/otte09version.pdf (cit. a p. 9).

[8] Brian de Alwis e Jonathan Sillito. «Why are software projects moving from centralized to decentralized version control systems?» In: *ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. Vancouver, BC, Canada, gen. 2009, pp. 36–39. URL: https://ieeexplore.ieee.org/document/5071408 (cit. a p. 9).

[9] *Git - Documentation*. URL: https://git-scm.com/doc (visitato il giorno 11/03/2025) (cit. alle pp. 9, 30, 32).

[10] Sarah Sterman, Molly Jane Nicholas e Eric Paulos. «Towards Creative Version Control». In: *Proc. ACM Hum.-Comput. Interact. 6, CSCW2, Article 33* (nov. 2022). URL: https://doi.org/10.1145/3555756 (cit. alle pp. 12, 18, 20, 22, 23).

[11] Blair Subbaraman, Shenna Shim e Nadya Peek. «Forking a Sketch: How the OpenProcessing Community Uses Remixing to Collect, Annotate, Tune, and Extend Creative Code». In: *Designing Interactive Systems Conference (DIS '23)*. Pittsburgh, PA, USA. ACM, New York, NY, USA, lug. 2023. URL: https://doi.org/10.1145/3563657.3595969 (cit. alle pp. 12, 13, 18, 19).

[12] Brad A. Myers, Ashley Lai, Tam Minh Le, YoungSeok Yoon, Andrew Faulring e Joel Brandt. «Selective Undo Support for Painting Applications». In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. Seoul, Republic of Korea, Association for Computing Machinery, New York, NY, USA, 2015, pp. 4227–4236. URL: https://doi.org/10.1145/2702123.2702543 (cit. a p. 23).

[13] Michael Terry e Elizabeth D. Mynatt. «Recognizing Creative Needs in User Interface Design». In: *Proceedings of the 4th Conference on Creativity & Cognition (Loughborough, UK) (C&C '02)*. Association for Computing Machinery, NewYork, NY, USA, 2002, pp. 38–44. URL: https://doi.org/10.1145/581710.581718 (cit. a p. 23).

[14] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang e Scott R. Klemmer. «Design as exploration: creating interface alternatives through parallel authoring and runtime tuning». In: *Proceedings of the 21st annual ACM symposium on User interface software and technology*. Ott. 2008, pp. 91–100. URL: https://doi.org/10.1145/1449715.1449732 (cit. a p. 23).

[15] Mary Beth Kery, Amber Horvath e Brad A. Myers. «Variolite: Supporting Exploratory Programming by Data Scientists». In: 10 (mag. 2017), pp. 1265–1276. URL: https://doi.org/10.1145/3025453.3025626 (cit. a p. 23).

[16] Hiroaki Mikami, Daisuke Sakamoto e Takeo Igarashi. «Micro-Versioning Tool to Support Experimentation in Exploratory Programming». In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (Denver, Colorado, USA) (CHI '17)*. Association for Computing Machinery, New York, NY, USA, mag. 2017, pp. 6208–6219. URL: https://doi.org/10.1145/3025453.3025597 (cit. a p. 23).

[17] Cameron Burgess, Dan Lockton, Maayan Albert e Daniel Cardoso Llach. «Stamper: An Artboard-Oriented Creative Coding Environment». In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems (Honolulu, HI, USA) (CHI EA '20)*. Association for Computing Machinery, New York, NY, USA, 2020, pp. 1–9. URL: https://doi.org/10.1145/3334480.3382994 (cit. a p. 23).

[18] Lingdong Huang. *Srcsnap: Screenshot-driven version tracking*. 2022. URL: https:%20//github.com/LingDong-/srcsnap (cit. a p. 23).

[19] Eric Rawn, Jingyi Li, Eric Paulos e Sarah Chasins. «Understanding Version Control as Material Interaction with Quickpose». In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Mag. 2023, pp. 1–18. URL: https://doi.org/10.1145/3544548.3581394 (cit. a p. 23).

[20] *OpenProcessing, browse sketches*. URL: https://openprocessing.org/browse (cit. a p. 25).

[21] *p5 community*. URL: https://p5js.org/community/ (cit. a p. 27).

[22] *p5.js - Contributors guidelines*. URL: https://p5js.org/contribute/contributor_guidelines/ (cit. a p. 27).

[23] *Documentation for Visual Studio Code Extension*. URL: https://code.visualstudio.com/api/extension-guides/overview (cit. alle pp. 28, 32).

[24] *Il codice di p5.js su GitHub*. URL: https://github.com/processing/p5.js (cit. a p. 29).

[25] *Homepage di p5js.org*. URL: https://p5js.org (cit. a p. 29).

[26] *Homepage di Processing.org*. URL: https://processing.org/ (cit. a p. 29).

[27] *Dati dall'ultimo Stack Overflow Survey*. URL: https://survey.stackoverflow.co/2024/technology#1-integrated-development-environment (cit. a p. 32).

[28] *Il codice di Visual Studio Code su GitHub*. URL: https://github.com/microsoft/vscode (cit. a p. 32).

[29] *Live Preview*. URL: https://marketplace.visualstudio.com/items?itemName=ms-vscode.live-server (cit. a p. 34).

[30] *Playwright*. URL: https://playwright.dev/ (cit. a p. 34).

[31] Jakob Nielsen. *Ten Usability Heuristics*. 2005. URL: https://pdfs.semanticscholar.org/5f03/b251093aee730ab9772db2e1a8a7eb8522cb.pdf. (cit. a p. 52).

[32] Jakob Nielsen. «Enhancing the explanatory power of usability heuristics». In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (Boston, Massachusetts, USA) (CHI '94)*. Association for Computing Machinery, New York, NY, USA, apr. 1994, pp. 152–158. URL: https://doi.org/10.1145/191666.191729 (cit. a p. 52).

[33] Jakob Nielsen. *Why You Only Need to Test with 5 Users*. Mar. 2000. URL: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5users/ (cit. a p. 53).

[34] John Brooke. «SUS: A Retrospective». In: *JUS, Journal of Usability Studies* 8 (2 feb. 2003), pp. 29–40. URL: https://uxpajournal.org/wp-content/uploads/sites/7/pdf/JUS_Brooke_February_2013.pdf (cit. a p. 54).