POLITECNICO DI TORINO

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

Tesi di Laurea Magistrale

Extension Development:
Sviluppo di un'estensione per programmi Adobe in tecnologia CEP

Relatore
Giovanni Malnati
Correlatore
Matteo Ruffinengo

Candidato
Marco Oggero

ANNO ACCADEMICO 2018-2019

Ottobre 2019

Abstract

Il lavoro di tesi svolto vuole portare a compimento un percorso, in cui si analizzerà innanzitutto una figura professionale relativamente nuova, quella del *Technical Director*, trovandone i punti di contatto con il profilo dell'ingegnere del Cinema, nell'intenzione di far convergere e sintetizzare tutte le competenze acquisite durante un percorso accademico così variegato nella proposta di materie di studio. Da questa prima tappa si entrerà quindi nel merito di *una* possibile applicazione delle competenze di un *technical director* (e dunque, anche di un ingegnere del Cinema): a seguito di un'analisi condotta nel contesto di produzione del film *Milkoffee*, cortometraggio d'animazione in tecnica mista, ci si è trovati di fronte un problema per il quale andava ricercata una soluzione. Questo problema si configurava come un collo di bottiglia nel flusso di lavoro tra i due software Adobe adoperati nella produzione, vale a dire Animate e After Effects, impiegati nella realizzazione di animazioni *frame-by-frame* in grafica *vettoriale*. Nello specifico, questo rallentamento nella produzione era dovuto alla mancata interoperabilità dei due programmi. Questa ricerca porterà dunque alla realizzazione di un'*estensione CEP* per i suddetti programmi capace di superarne le carenze in comunicazione.

Durante la trattazione verranno anche analizzati molti degli strumenti a disposizione del *Technical Director* in ambito di *sviluppo di componenti integrativi* per altri software, approfondendo successivamente il caso specifico degli strumenti offerti da Adobe: verranno perciò esaminate le diverse tecnologie, distinguendo le definizioni di *scripts*, *plug-ins* ed *estensioni*.

A partire da queste ultime, essendo la tecnologia prescelta allo svolgimento del progetto pratico della tesi, si svilupperà un'analisi sulla struttura fondante di questa tipologia di software integrativi, il *CEP* (*Common Extensibility Platform*), del quale se ne esibirà l'*architettura* e i *componenti fondamentali*. Si proseguirà quindi con l'esposizione delle *API* dei due software protagonisti della ricerca, Animate e After Effects, evidenziandone tratti comuni e differenze nell'organizzazione dei propri *DOM* e della *grafica vettoriale*.

Una volta preparato il campo *teorico* della tesi, si procederà allo sviluppo *pratico* dell'estensione: questo si configurerà come una *guida* passo per passo nella gestione degli elementi fondamentali per una qualsiasi estensione, nell'intenzione di creare delle nozioni *manualistiche* su questo argomento, delle quali è stata riconosciuta una scarsità diffusa nel panorama web. Da questa trattazione *generale* sullo sviluppo di un'estensione, si entrerà quindi nella *specificità* di *questa* estensione, fornendo descrizioni dettagliate sugli algoritmi e l'organizzazione dei dati trattati da essa.

Infine, la trattazione verrà conclusa dall'esposizione dei risultati dei test effettuati sull'impiego dell'estensione, al fine di trarre dei dati obiettivi sull'efficacia di quest'ultima relativamente al problema proposto.

Indice

Indice delle Figure

Introduzione

Motivazioni

Le motivazioni alla base della presente tesi nascono nel quadro di una riflessione su una sintesi personale (o per lo meno un suo tentativo) delle competenze, nonché interessi, di un Ingegnere del Cinema e dei Mezzi di Comunicazione. Con questo non si vuole assolutamente intendere di voler dare una definizione oggettiva o assoluta della suddetta professione: quanto verrà scritto e descritto nel presente lavoro ha valore puramente personale nelle sue parti più discorsive e *astratte*, nell'intenzione, innanzitutto, di mettere per iscritto quanto ragionato nel percorso di auto analisi avvenuto durante l'iter accademico; in secondo luogo, nella speranza o presunzione di fornire uno spunto di riflessione anche a dei potenziali lettori, i quali decideranno da sé quale valore attribuire a queste considerazioni.

Chiaramente queste motivazioni non sarebbero sufficienti, per me, ad intraprendere un percorso di ricerca per una tesi, né una relazione basata sulle suddette motivazioni rappresenterebbe una degna o quantomeno accettabile conclusione di un percorso accademico prettamente improntato alla pratica (in quanto ingegneristico) se, appunto, l'intero discorso non fosse preambolo di un progetto pratico. Così dicendo però, ci tengo a precisare che il progetto qui presentato non è assolutamente un argomento di convenienza, affrontato perché richiesto dalla prassi nella stesura di una tesi, bensì anch'esso frutto della riflessione di cui sopra. Tuttavia, si potrebbe anche dire che la

volontà di sintetizzare la mia idea di professione di ingegnere del Cinema sia nata dalla prospettiva del progetto pratico. Oppure, più semplicemente, mi piace pensare che le due cose siano complementari: a una definizione di ruolo è corrisposta una declinazione pratica e viceversa.

Ultima ma non ultima, per chi potesse considerare banale o di dubbia utilità quanto detto finora, non può mancare una motivazione di obiettiva necessità di questa tesi, dalla quale è nata la specificità del progetto pratico. Mi riferisco qui alla circostanza nel cui contesto è stata proposta l'idea della ricerca, ovvero durante la produzione del cortometraggio di animazione *Milkoffee*, progetto inizialmente di natura accademica poi esteso a progetto personale. Premessi i miei interessi preesistenti a tutto quanto approfondito nella tesi, la problematica affrontata durante la produzione del cortometraggio è stata però la proverbiale scintilla che ha innescato la stesura di questo documento. Passerò ora alla descrizione del problema sopracitato, la cui soluzione ha segnato l'obiettivo ultimo della tesi.

Obiettivo

Si può parlare dell'obiettivo di questa ricerca entrando nel merito della problematica affrontata per il cortometraggio, se problematica si può definire. Mi sono così espresso perché la situazione proposta non rappresenterebbe un problema in sé e per sé, ma lo diventa nel momento in cui il processo di cui si fa protagonista si debba replicare più e più volte, dilatando drasticamente i tempi di produzione di un qualsiasi prodotto audiovisivo. Nello specifico, sto parlando di *esportazione di file video.* Lo scopo finale della tesi, a livello progettuale, è quello di sviluppare un nuovo componente nella *pipeline* di animazione 2D con l'accoppiata di app Adobe Animate-After Effects, software prescelti per la maggior parte della produzione di *Milkoffee*; in una pipeline ideale di questi due software, un animatore non dovrebbe vedersi costretto ad esportare e importare l'animazione da un software all'altro come file video: ciò infatti implica un notevole spreco di risorse in termini di tempo e memoria della macchina. Le due applicazioni hanno il potenziale di interfacciarsi in maniera più fluida e dinamica, potendo entrambe sfruttare la grafica vettoriale: il progetto si concretizzerà quindi in ciò che definisco un *renderless exporter*, ovvero un componente integrativo che permetta il trasferimento immediato dei dati vettoriali da una app all'altra senza che ci sia il bisogno di rendering intermedio. Le specificità e criticità della pipeline e dei due programmi, nonché il processo che ha portato allo sviluppo del componente che ne colmi il *gap*, verranno analizzati più avanti nella trattazione, nei capitoli preposti ai suddetti argomenti.

Metodo

La metodologia adottata nel processo di sviluppo si rifà allo *user-centered design*, oltretutto oserei dire in una posizione privilegiata: infatti mi sono ritrovato a ricoprire sia il ruolo di *designer* nonché di *user* finale del prodotto. Questo aspetto si riconcilia perfettamente con quella figura professionale *sintetica* che mi appresto ad analizzare nel corso della tesi: in quanto utente ho analizzato, insieme al team di produzione del cortometraggio, quali fossero i punti deboli della pipeline di produzione e quali potrebbero essere state le soluzioni, avendo quindi un punto di vista molto più efficace rispetto ad un esterno. Ciononostante, come sviluppatore ho comunque dovuto tener conto anche dell'*individualità* del progetto, non dimenticando quindi di confrontarmi anche con potenziali utenti futuri, esterni a questa specifica produzione filmica. Se quindi la prima domanda da porsi è stata "Di cosa *ho* bisogno?" dopo le prime fasi di sviluppo ho adottato un punto di vista più ampio, ponendomi la domanda "Di cosa *c'è* bisogno?".

Organizzazione dei Contenuti

Il testo è composto di 4 parti: nella prima affronto quanto anticipato all'inizio di questa introduzione, ovvero la descrizione di questa figura professionale da me individuata attraverso l'analisi degli attori e dei trend nell'industria dell'intrattenimento. Le tre parti successive sono tutte incentrate sul progetto pratico della tesi: nella seconda si descriverà la *ricerca*, ovvero un'indagine sugli strumenti a disposizione nella produzione del cortometraggio, le loro differenze e punti di forza, quindi una successiva indagine sulle tecnologie utilizzabili per sopperire alle loro mancanze e ottimizzare la pipeline; nella terza si tratterà dello *sviluppo* vero e proprio, con descrizione degli algoritmi utilizzati ed esempi di codice scritto; infine nella quarta verranno presentati i *risultati* di test effettuati sul campo con animatori professionisti, traendo le considerazioni su un'eventuale effettivo miglioramento della pipeline di questi programmi.

Capitolo 1

Software: il cuore dell'industria dell'intrattenimento

Quanto verrà affermato adesso potrà sembrare banale, ma non si può evitare di puntualizzare quanto l'avvento del digitale abbia stravolto ogni aspetto dell'industria dell'intrattenimento, dalla produzione alla distribuzione, perfino nell'ideazione dei prodotti, creando di fatto nuovi settori e all'occorrenza anche nuovi ruoli professionali: ne è un esempio lampante l'industria videoludica, impensabile prima dell'avvento dei computer. Nuovi strumenti e nuovi *medium* infatti suggerivano la possibilità di creare nuove modalità di fruizione dei prodotti di intrattenimento, influenzandone anche il contenuto [1]. I computer hanno sostituito la quasi totalità dei supporti e strumenti precedentemente utilizzati: la pellicola con i pixel, carta e matite con tavole grafiche, la moviola con l'editing non-lineare, e così via. È da questo assunto fondamentale che si può quindi affermare quanto l'industria del software sia alla base di ogni declinazione dell'intrattenimento, sia esso cinema, animazione, videogiochi, TV fino ad arrivare, volendo, all'editoria con libri e fumetti.

In questo capitolo affronterò quanto anticipato all'inizio dell'introduzione, ovvero cercherò di delineare quella figura professionale di cui l'ingegnere del Cinema potrebbe essere una declinazione, attraverso una personale elaborazione delle affermazioni sopracitate: partendo dal concetto del software come *cuore* dell'industria dell'intrattenimento, elencherò brevemente i protagonisti del panorama attuale, ovvero le aziende leader del settore, delle diverse *filosofie* di distribuzione e modelli di business, e di come il modello dell'*open source* abbia aiutato a plasmare la figura del *Technical Director* (abbr. *TD*), ruolo sempre più presente e fondamentale in ogni produzione, dalle majors alle piccole produzioni. Da questa trattazione di carattere generale, arriverò quindi alla specificità del mio caso, esponendo nel particolare le risorse offerte da Adobe per quanto riguarda lo sviluppo e l'estensione indipendente dei propri programmi, trattando infine delle necessità nel contesto della produzione del cortometraggio *Milkoffee* che mi hanno spinto a intraprendere questa ricerca.

1.1 Il panorama attuale: i leader e l'*open source*

Come si può intuire dal titolo del capitolo, tutto parte dal software: oggi il computer è diventato lo strumento primario dei creativi; ma, ovviamente, non sono gli artisti a sviluppare i propri programmi (o per lo meno non lo erano agli inizi della rivoluzione digitale). Dalle prime fasi del passaggio analogico-digitale ad oggi, varie aziende si sono rese protagoniste, introducendo sempre crescenti novità e affermandosi più volte leader del settore: *Adobe, Autodesk, Foundry, Pixar, Side Effects, Sony* sono solo alcuni dei nomi delle aziende produttrici dei programmi che nel mercato attuale rappresentano gli standard industriali. Va detto però che queste aziende nascono in un contesto successivo all'originale idea di *open source*, in cui si era già ampiamente consolidato un sistema economico fondato sulla *proprietà* dei prodotti, basando quindi il proprio modello di business sulla vendita di licenze dei software, e quindi limitandone o in alcuni casi addirittura evitandone la condivisione e diffusione. Questi programmi erano quindi estremamente settoriali, destinati a pochi o pochissimi (come nel caso di Pixar, il cui sviluppo software e quasi esclusivamente destinato ad uso interno nella produzione dei propri film). Generalmente c'era una netta distinzione tra sviluppatore ed utente finale, il quale non aveva possibilità di personalizzare il proprio flusso di lavoro se non tramite richiesta esplicita all'azienda produttrice stessa. Il modello *open source* però non è mai tramontato e questo, unendosi al cambio di paradigma economico del nuovo millennio, ovvero dalla vendita di *prodotti* all'erogazione di *servizi*, ha segnato la nascita di nuovi protagonisti, fautori di nuovi modelli di business e/o della pura condivisione col mondo. È questo il caso di *Blackmagic Design* (*BMD*) e della *Blender Foundation*: la prima è sviluppatrice del programma di *video-editing* e *color-grading DaVinci Resolve* [2], distribuito gratuitamente, il quale sta rapidamente soppiantando la concorrenza nel montaggio video, ed è già lo standard *de facto* nel color grading (BMD basa il proprio business non tanto sul software ma sulla vendita di attrezzatura audio-video, che il

software va semplicemente a completare; mantenere il software aperto a tutti ne ha segnato l'ampio successo); la seconda è la sviluppatrice del programma di modellazione, animazione e simulazione fisica *Blender* [3], in circolazione ormai da più di 25 anni, e che vanta ormai una *community* sempre crescente anche in medie-grosse produzioni. Il mercato è ormai saturo di surrogati e alternative *open source*, spesso più che validi se non superiori alle soluzioni proprietarie, quindi le aziende leader non possono più basarsi esclusivamente sulla *fidelizzazione* dei clienti più affezionati (i quali comunque potrebbero cambiare idea in qualsiasi momento ed in ogni caso, sul lungo periodo, verrebbero sostituiti da nuove generazioni abituate all'*open source*), ma devono attuare dei cambiamenti e nuove strategie. La prima e più semplice delle strategie sta proprio nel concetto di *community* sopracitato: quanto più il programma diventa *comune* (sia nel senso di *popolare* che di *comunitario, collettivo*), tanto più esso si può affermare come leader o, per lo meno, un valido competitor [4]. Tutte le aziende, in un modo o nell'altro, hanno quindi iniziato ad *aprirsi*, e non senza difficoltà, non potendo rinunciare dall'oggi al domani al proprio modello di business, ma offrendo almeno come base la possibilità di *personalizzare* i propri software. La recente nascita nell'agosto 2018 dell'*ASWF – Academy Software Foundation*, ente istituito con lo scopo di riunire le aziende promotrici di software *open source*, è un'ulteriore conferma di questa *tendenza all'apertura*, dato anche il sempre crescente impiego di software liberi anche agli alti livelli dell'industria [5].

Secondo questa filosofia di *creazione di comunità*, Adobe rappresenta il caso più peculiare tra le aziende, essendo stata capace di creare una *community* vasta per non dire invidiabile tra le produttrici di software proprietari: già leader in diversi settori della grafica e web development con i propri programmi di punta, quali Photoshop, Illustrator, Flash e altri, è riuscita ad adattarsi alle nuove tendenze, cambiando il proprio modello di business dal 2013 passando come servizio in cloud; già questo ha contribuito ad un ampliamento sensibile della propria *community*. Non solo, tra tutte le aziende è forse quella più attiva nell'apertura dei propri programmi [6], offrendo un'ampia gamma di soluzioni nella personalizzazione dei prodotti, soluzioni che verranno analizzate in seguito nel prossimo capitolo.

Ad ogni modo, è in questo panorama di *apertura* che la figura del *Technical Director* inizia a prosperare; in quasi 40 anni di trasformazioni infatti, non è solo cambiata la concezione di questi programmi, ma anche degli utenti stessi: l'utente non è più solo un fruitore confinato alla fine della catena di produzione, ma è molto più coinvolto nello sviluppo del software, ne è più interessato. E in alcuni casi ne viene così coinvolto da prendere egli stesso l'iniziativa, piegando il programma alle sue esigenze, e questo perché le aziende, avendo *aperto* i propri prodotti alle *community*, hanno fornito i mezzi per farlo. È così che nasce un *ibrido* tra l'artista e lo sviluppatore: il *Technical Director*.

1.2 La figura emergente del *Technical Director*

Entriamo ora nel merito del ruolo del *Technical Director* (TD). Come appena sopracitato, esso è il risultato della convergenza della figura dello sviluppatore con quella dell'utilizzatore finale dei programmi [7]. Ora, ad una pluralità di programmi e mansioni deriva una pluralità di possibili TD: per l'animazione, per la modellazione, per i VFX, per la pipeline di produzione, …. Ciò che accomuna ogni tipo di TD però è il *problem solving*: è quella figura che, all'occorrenza di un problema di produzione o di una necessità altrimenti impossibile con gli strumenti attualmente a disposizione, trova la soluzione più adatta, soluzione che prevede, il più delle volte, lo sviluppo di componenti software integrativi, alle volte interi nuovi programmi (in tal caso col supporto del dipartimento R&D della produzione). D'altra parte, il TD non può essere uno sviluppatore puro, in quanto deve essere ben conscio dei programmi su cui va a intervenire e dei processi in cui opera: è quindi egli stesso, oltre che un tecnico, un artista del proprio settore (animatore, VFX artist, ecc.…).

Negli ultimi anni si registra una crescente incidenza di TD, richiesti non solo nelle grandi produzioni, ma anche in quelle più piccole e indipendenti, soprattutto all'estero, dove l'industria dei VFX, settore di punta per un TD, è molto più sviluppata; in Italia il ruolo specifico del TD è poco riconosciuto, più per mancanza di occasioni che per altro: esso viene richiesto soprattutto in grandi produzioni, dove, per possibilità di budget, è possibile introdurre innovazioni nella catena di produzione, ed è proprio nel carattere innovativo del proprio lavoro che prospera il TD; in piccole produzioni invece, laddove si necessita di un intervento tecnico, ci si affida a sviluppatori puri o alle eventuali competenze in programmazione di specialisti di produzione che però non si definirebbero TD, e tutto questo quando si decide di affrontare il problema di petto, dove con gli strumenti classici lo si dovrebbe aggirare in altro modo. Ciononostante, una seppur minima consapevolezza c'è, ed è in crescita: sempre più professionisti apprendono in autonomia competenze di programmazione, e sempre più produzioni, anche piccole, ricercano figure che vadano oltre lo sviluppatore puro, una figura più coinvolgibile nella produzione, una figura che possa ricoprire un ruolo, se non fisso, almeno a lungo termine. Questa crescente consapevolezza è indice di quanto gli strumenti software si stiano ormai *interiorizzando* nella mente dei professionisti (e questo, lo ribadisco, è possibile anche grazie all'*apertura* delle aziende), e non sarebbe ingenuo pensare che in un futuro le competenze tecniche saranno quasi un presupposto imprescindibile del lavoro. Ad ogni modo, attualmente il TD rimane una novità, di cui è riconosciuta l'importanza già partendo dal titolo: è infatti *direttore* nel senso più genuino del termine, non da intendersi come *capo* o *dirigente* (nonostante spesso il reparto in cui opera faccia riferimento a lui, ricoprendo un ruolo senior), ma come colui che *traccia la direzione*, una guida o volendo un *consulente* al quale rivolgersi per affrontare un problema e organizzarne la soluzione; è infatti anche l'organizzazione un'importante mansione del TD, in quanto a lui è affidata anche la definizione delle linee guida di produzione nella loro forma più pratica, ovvero nella gestione dei file e delle memorie delle macchine; in tal senso diviene anche un insegnante, in quanto sta a lui istruire gli artisti una volta adottata una certa soluzione e

organizzazione, fino allo spiegare il funzionamento dei componenti software da lui creati, ovviamente.

Parlando di *ibridazione* di ruoli, non si può fare a meno di includere nel discorso il panorama videoludico: è mia ferma convinzione infatti che tutte queste figure *ibride* debbano anche solo in minima parte rendere omaggio all'innovazione professionale che ha rappresentato la professione del *game developer*, che ritengo la più completa dal punto di vista sia tecnico che creativo; chiaramente oggi anche questo ruolo non è che un termine generico che caratterizza un ampio ventaglio di specialisti (modellatori, animatori, programmatori,…) i quali però, nella maggior parte dei casi, mantengono una base di competenze generaliste in tutti i settori di produzione dell'industria videoludica. Il *game developer* propriamente detto infatti altro non è in origine che uno sviluppatore facente parte di produzioni indipendenti, composte da un numero esiguo di elementi se non addirittura da singoli, dovendo quindi ricoprire la (quasi) totalità delle mansioni nella creazione del videogioco (si potrebbe quasi dire quindi che il *game developer* subisce una trasformazione inversa rispetto al *technical director*, ovvero da generalista trova un settore di specializzazione).

1.3 Strumenti a disposizione del *Technical Director*: uno sguardo

A questo punto si può finalmente parlare dell'aspetto pratico di un TD, ovvero delle tecnologie a sua disposizione. Da quanto detto finora dovrebbe essere chiaro che si sta parlando in massima parte di *programmazione*: ora, non è obiettivo di questa tesi dare una panoramica completa di tutte le possibilità offerte da ogni singolo software, proprietario o *open source*, ma almeno un breve sguardo dei *linguaggi di programmazione* più comuni e i programmi più standard nell'industria. Oltretutto quanto verrà riportato è frutto di una ricerca preliminare mirata all'individuazione dello strumento che più si adattava alla situazione presentatasi nel cortometraggio *Milkoffee*, ed è quindi, intrinsecamente, non approfondito più di quanto non servisse ai fini del progetto pratico. Poiché è nei programmi Adobe che è stata riconosciuta l'adeguata validità ai fini di produzione del cortometraggio, la ricerca su questi ultimi è stata chiaramente approfondita, e verrà affrontata nel dettaglio nel prossimo capitolo.

Precedentemente ho citato la possibilità che un TD possa ritrovarsi a sviluppare un intero nuovo software ma ribadisco però che in questo caso un lavoro del genere richiederebbe un ingente impiego di risorse sia umane che finanziarie: non è questo infatti compito primario del TD, che lavora più a stretto contatto con gli artisti durante lo svolgimento della preproduzione e produzione, ma del dipartimento *R&D* (*Research and Development*) della compagnia, qualora possa permetterselo. In questo caso il TD è sì presente nelle fasi di sviluppo, anche a livello pratico, ma si trova in una posizione a metà strada tra gli artisti, utenti finali del software, e i *R&D engineers*, professionisti molto più coinvolti nello sviluppo vero e proprio. Oltretutto lo sviluppo di un nuovo software viene

approntato ben prima dell'inizio della produzione ma anche della preproduzione di un nuovo prodotto d'intrattenimento, in quanto chiaramente ne provocherebbe bruschi arresti altrimenti. Ne è nuovamente esempio la Pixar con i suoi ormai celebri cortometraggi: questi infatti molto spesso, oltre ad avere un chiaro valore artistico, sono dei test per una nuova tecnologia da utilizzare nel film successivo [8].

Ora, parlando concretamente, l'operato di un TD consisterà per lo più nella creazione di componenti software integrativi che a seconda del contesto assumono diverse denominazioni: plug-in, add-on, estensioni, framework o, più semplicemente, *script*; tutti queste denominazioni infatti hanno quasi sempre in comune il fatto di essere sviluppate con *linguaggi di scripting*, comuni o proprietari; questa però non è comunque una regola in quanto, ad esempio, in ambito Adobe, plug-in e script indicano due componenti differenti, il primo dei quali sviluppato in C++, un linguaggio compilato. Sta qui infatti la sostanziale differenza tra un linguaggio di scripting da qualsiasi altro linguaggio: esso è infatti un linguaggio *interpretato*, ovvero non soggetto a compilazione da parte del software che lo esegue, denominato in questo frangente *interprete*. L'interprete quindi, a differenza di un *compilatore*, che ha il solo compito di *tradurre* senza eseguire, esegue lo script direttamente in codice sorgente, traducendolo sul momento in linguaggio macchina.

Il linguaggio di scripting più utilizzato nell'ambito della direzione tecnica è sicuramente *python*: questo è un linguaggio di ampio uso comune, che trova utilizzo praticamente in ogni ambito e settore di qualsiasi industria, data la sua versatilità. Non è un caso che negli ultimi anni sia stato progressivamente adottato dalla maggior parte delle aziende leader: ad oggi infatti è integrato nelle applicazioni Autodesk Maya e 3ds Max, in Nuke di Foundry, Houdini di Side Effects e molti altri. È inoltre presente anche in Blender, il quale è scritto in gran parte proprio in python. Merita oltretutto far presente che queste aziende abbiano sviluppato originariamente dei propri linguaggi proprietari per lo *scripting*, com'è il caso di MEL (Maya Embedded Language) o MaxScript rispettivamente per Maya e 3ds Max di Autodesk; ciononostante, progressivamente si stanno tutte uniformando all'utilizzo di python, a testimonianza dell'esistenza di una tendenza ad un'apertura ad un più ampio bacino d'utenza e alla volontà di creazione di *community* [9].

Un caso particolare di linguaggio di scripting utilizzato in questo tipo di programmi è *javascript*: questo linguaggio è infatti prerogativa per lo sviluppo di applicazioni web, destinate quindi ad essere utilizzate tramite un browser. C'è infatti un sostanziale legame tra i software capaci di utilizzare javascript e le tecnologie web, che verrà affrontato nel prossimo capitolo, in quanto Adobe è una delle aziende ad avere adottato questo differente paradigma per le proprie capacità di scripting. Oltre alla sopracitata Adobe, sono degni di menzione anche i *game engine* Unity e Unreal: negli ultimi anni si è infatti assistito ad un boom nell'utilizzi di motori di gioco nell'industria cinematografica e dei VFX [10].

Chiaramente anche i linguaggi compilati trovano ampio utilizzo nelle pipeline di produzione di un prodotto d'intrattenimento, non solo nella scrittura di software

standalone. Quasi sempre infatti, in questo ambiente, si tende a indicare con plug-in un componente software scritto in un linguaggio compilato, molto spesso C++. Tra gli altri linguaggi utilizzati troviamo C, C# e Java.

Categoria a sé stante di linguaggi invece, molto settoriali ma comprensibilmente largamente utilizzati in questa industria, sono i linguaggi di *shading*, ovvero i linguaggi utilizzati nella definizione di *shaders*. Questi sono componenti fondamentali nella computer grafica 3D e 2D, in quanto alla base della definizione delle proprietà e caratteristiche dei materiali impiegati in fase di rendering. Tra questi linguaggi il più conosciuto è sicuramente GLSL, ovvero *OpenGL Shading Language* (colloquialmente chiamato *glslang*), in quanto, come suggerisce il nome, linguaggio proprio delle OpenGL, le API standard nella computer grafica (e, in questo senso, standard d'industria). Subito dopo trovano un enorme impiego linguaggi più specializzati, in quanto di derivazione proprietaria: RenderMan SL, utilizzato nell'omonimo motore di rendering proprietario di Pixar; VEX, impiegato in Houdini; Open Shading Language o OSL, creato da Sony per essere impiegato nel proprio motore di rendering Arnold, e successivamente anche adottato da Blender per il proprio motore di rendering fotorealistico Cycles, da V-ray e da Octane.

1.4 Il caso Adobe

Prendiamo ora in esame il caso di Adobe, studiato approfonditamente ai fini di questa ricerca. Adobe è con tutta probabilità il leader indiscusso di tutto il panorama nell'industria dell'intrattenimento: è la scelta primaria di un'innumerevole quantità di piccole e medie realtà indipendenti, dal *videomaking*, alla grafica, dall'illustrazione al web design, alle volte anche nelle majors; la quasi totalità dei professionisti sono andati incontro almeno una volta nella vita all'utilizzo di un programma Adobe, che è spesso requisito di base nel lavoro. Per imporsi come leader, Adobe ha negli anni adottato svariate strategie, comprendenti anche l'acquisizione diretta delle aziende concorrenti, ma con tutta probabilità è stata soprattutto la cura nel coltivare la propria *community* ciò che ha maggiormente influenzato la diffusione capillare dei propri prodotti. Adobe infatti, ad oggi offre la più ampia gamma di soluzioni e supporti per la propria *community*, da un vasto assortimento di documentazione (gratuita) anche per chi non fosse cliente, fino ad arrivare a consulenti privati per aziende e studi che facciano uso dei software nei propri prodotti (il *solution consultant*, questo il termine ufficiale, è in un certo modo assimilabile alla figura del TD). A questo vanno unite le politiche commerciali, trasformatesi negli anni, nel tentativo di facilitare la domanda nella distribuzione sia da un punto di vista pratico (Adobe si è adattata al cambio generazionale di paradigma commerciale da *venditore di prodotti* a *distributore di servizi* con il passaggio da *Creative Suite* a *Creative Cloud*) sia da un punto di vista finanziario, con prezzi agevolati (soprattutto per le licenze studenti) e la possibilità di personalizzare i pacchetti di prodotti.

Tornando al supporto offerto da Adobe, spicca tra tutti il forum ufficiale, il cui sito è stato recentemente ristrutturato e ribattezzato proprio *Adobe Support Community* [11], a testimonianza dell'importanza per l'azienda nella cura e creazione di *community* attorno ai propri prodotti. Qui gli utenti possono confrontarsi e richiedere aiuto ad utenti più esperti, molto spesso ricevendo risposta proprio dai *solution consultant* ufficiali Adobe (io stesso ho avuto modo di ricevere assistenza dalla *community*, tra gli altri anche da un consulente ufficiale), oppure semplicemente condividere le proprie idee e soluzioni. In secondo luogo, ma non per importanza, le risorse web: sono numerose infatti le pagine dedicate alla documentazione dei prodotti, in ogni loro aspetto, dall'utilizzo più semplice allo sviluppo per essi. Riporto qui di seguito i due siti da me ritenuti i più utili, in quanto aggregano la maggioranza di documentazione e strumenti: ad ogni prodotto è dedicata una sezione nell'*Adobe Help Center* [12], in cui è possibile consultare una versione digitalizzata e indicizzata dei manuali ufficiali, oltre a numerosi tutorial. Ma è forse il sito *Adobe I/O* [13] a rappresentare, almeno nei fini di questa tesi, quello più importante; sono qui infatti raccolti tutti gli SDK (*Software Development Kit*), le API (*Application Programming Interface*) e tutte le risorse *open source* offerte dall'azienda. Ed è da qui quindi che bisogna partire per lo sviluppo di qualsiasi componente integrabile nell'ecosistema Adobe: nel prossimo capitolo darò una panoramica di tutti gli strumenti a disposizione per lo sviluppo per Adobe, descrivendone le differenze già brevemente anticipate in precedenza, entrando infine nel merito della scelta utilizzata. Di seguito invece viene trattata la casistica presentatasi nell'ambito della produzione del cortometraggio *Milkoffee* che ha portato al concepimento di questa ricerca.

1.5 Com'è nata la necessità della ricerca: il progetto *Milkoffee*

Milkoffee è un cortometraggio d'animazione in tecnica mista nato all'interno del percorso di studi della magistrale di Ingegneria del Cinema e dei Mezzi di Comunicazione, inizialmente come consegna finale per il progetto di laboratorio del corso di Effetti Speciali. La consegna era piuttosto libera: produrre un video di qualsiasi natura, narrativo, musicale, sperimentale, alla cui base dovevano essere applicate le nozioni apprese durante il corso. Nel mio gruppo fu scelto di produrre un video di animazione: dopo una prima fase di scambio di idee, fu deciso di produrre un cortometraggio basato su un racconto breve facente parte della raccolta *Delitti Esemplari*, dello scrittore spagnolo Max Aub, della durata approssimativa di 3'. La storia, ambientata in un diner americano in mezzo ad un deserto, vede due uomini seduti l'uno di fronte all'altro, il primo intento a mescolare un bicchiere di caffelatte (da cui il termine, inventato, *Milkoffee*, contrazione di *milk and coffee*); il rumore provocato dal cucchiaino nel gesto di mescolare turba così profondamente il secondo uomo da gettarlo in un profondo abisso di disagio e follia, culminante nell'omicidio. Ora, il software di principale utilizzo durante il corso era *After Effects*, software di estrema flessibilità in molte fasi di videomaking, potendolo usare anche in modo molto basilare per montaggio e codifica, ma specialmente impiegato nella realizzazione di *VFX*, nel *compositing* e nella *motion graphics*; quest'ultima ricade nella

più ampia definizione di *animazione*, essendone una categoria; viene infatti spesso usato come sinonimo di *animazione 2D* anche se i due termini hanno sfumature leggermente diverse: entrambe indicano un tipo di animazione basato su *keyframes*, ovvero un'animazione *interpolata* da un computer dopo aver definito dei *pose chiave* dell'oggetto da animare, ma la prima è utilizzata spesso in relazione a prodotti di natura più *grafica*, alle volte commerciale, mentre il secondo con un'accezione più *narrativa*, basata su dei personaggi (*2D character animation*). In questo senso si distingue parecchio dall'*animazione tradizionale*, in cui ogni fotogramma è disegnato a mano, ma trova chiaramente similitudine con l'animazione 3D, in quanto anch'essa basata su *keyframes* ma in uno spazio 3D. La *motion graphics*, e in generale qualsiasi elemento grafico bidimensionale animabile per interpolazione, non è inoltre necessariamente vincolata allo spazio 2D in quanto i suddetti elementi possono essere su più piani bidimensionali disseminati in uno spazio 3D, definendo così un tipo di ambiente definito *2.5D* (ovvero l'impiego di elementi 2D in uno spazio 3D). Dopo la dovuta digressione, è qui quindi che si ritorna a *Milkoffee*: le intenzioni iniziali erano infatti quelle sviluppare l'animazione secondo una logica 2.5D, essendo noi fino a quel momento abituati per lo più alla *motion graphics*. Dopo i primi mesi spesi per lo più nella definizione dell'estetica degli elementi grafici, la produzione è rallentata a causa di crescenti dubbi sul fatto che un'animazione in *motion graphics* potesse essere effettivamente la più adatta a rappresentare una storia del genere, fino all'arrestarsi completamente per un semestre per via della partenza di alcuni membri del gruppo per l'esperienza estera dell'*Erasmus*. Questi 6 mesi sono stati comunque una preziosa pausa, per così dire, *formativa*: al rientro dei membri mancanti, i quali nel frattempo hanno potuto studiare proprio all'estero nuove tecniche di animazione, la situazione si sbloccò con la decisione di includere nel cortometraggio animazione tradizionale e 3D, la prima per l'animazione dei personaggi, la seconda nella modellazione e animazione degli ambienti. Per l'animazione tradizionale vennero quindi inclusi nella *pipeline* di produzione i software *Photoshop* e *Animate*. Il primo veniva utilizzato nella definizione preliminare delle scene (*rough animation*) ed è in tal senso di scarso interesse per questa ricerca, mentre il secondo veniva utilizzato nell'ultimazione di quanto fatto in Photoshop, ed è letteralmente il secondo protagonista di quanto verrà descritto nei prossimi capitoli. Sorvolerò su tutto quanto concerne l'animazione e la modellazione degli ambienti 3D in quanto di relativo interesse ai fini della tesi. Ma tornando ad Animate: questo è l'erede di *Flash*, software utilizzato per la più varia creazione di contenuti, dalle animazioni vettoriali, allo sviluppo di applicazioni web, mobile e videogiochi e in un certo senso definibile, senza esagerazione, il *padre* del web 2.0, in quanto permise la creazione di siti e applicativi web impensabili prima dell'avvento di HTML5 e JavaScript. Ad oggi è caduto in crescente disuso, e vedrà nel 2020 l'ufficiale annuncio di fine vita. Ciononostante, anche dopo il passaggio ad HTML5, *Flash* ha continuato ad essere utilizzato e apprezzato dagli animatori, cambiando così aspetto e nome e diventando *Animate*, tuttora distribuito da Adobe. Come anticipato, Animate è principalmente utilizzato per animazione in grafica vettoriale, e in minima parte per lo sviluppo web/mobile (nonostante ne abbia le capacità). Ed è proprio nell'utilizzo come strumento per animazione tradizionale che ha trovato spazio nella pipeline di produzione di *Milkoffee*. After Effects è quindi passato da unico strumento di produzione a strumento di finalizzazione, in cui approntare il *compositing* di quanto veniva prodotto negli altri software e disporre degli effetti che avrebbero contribuito

all'estetica finale del cortometraggio. Ma sta proprio nel principio del *compositing* che si sono intraviste le prime ombre di debolezza nell'impiego congiunto di questi due software: per la corretta composizione delle scene, ogni livello creato su Animate andava, chiaramente, importato separatamente su After Effects. Vien da sé quindi che per scene complesse, con animazioni nell'ordine delle decine di livelli, l'esportazione dei livelli separati implichi un notevole spreco di risorse in termini di tempo e memoria della macchina. Ci siamo quindi domandati come potesse essere ottimizzata la pipeline e la soluzione ci è arrivata da un'*estensione* molto popolare in After Effects, *Overlord*: questa opera contemporaneamente su After Effects e *Illustrator*, altro programma di punta di Adobe per la grafica vettoriale, e permette il trasferimento dei dati vettoriali da una app all'altra senza che ci sia bisogno di esportazione di file. After Effects ha infatti delle basilari capacità nella grafica vettoriale, ma più che sufficienti per l'animazione. L'esistenza di *Overlord* ha per lo meno suggerito la reale possibilità di scambio di dati vettoriali tra due app Adobe. Adesso andava trovata una soluzione analoga per Animate e After Effects, soluzione di non banale realizzazione rispetto alla maggioranza delle estensioni esistenti sul panorama Adobe, data l'integrazione più difficoltosa di Animate nell'economia del Cloud rispetto qualsiasi altro programma Adobe: Animate (forse per caratteristiche *ereditate* dalle precedenti versioni edite da *Macromedia*, software house sviluppatrice originale di *Flash*) non è mai stato completamente adattato al resto della suite di programmi Adobe, i quali godono di una estrema flessibilità nell'importazione dei file nativi l'uno nell'altro (in After Effects è infatti possibile importare, ad esempio, file .psd o file .ai, rispettivamente i *file di progetto* di Photoshop e Illustrator, senza difficoltà, cosa non possibile con i file di progetto Animate .fla). Negli ultimi anni in After Effects è stata inserita la possibilità di importare file .swf (derivante da *Shockwave Flash,* più comunemente chiamato Flash o *swiff*) il formato di *export* di progetti Animate, proprietario di Adobe; una volta esportato in questo formato però, il file perde di *flessibilità*, non potendo più essere suddiviso nei livelli componenti il progetto (cosa possibile, ad esempio, con i file .psd). Molto recentemente (mesi dopo l'inizio dello sviluppo del progetto pratico di questa tesi), Adobe ha approntato nell'ultima versione di After Effects un espediente per aggirare questo problema; è infatti ora possibile *importare* file .fla in un progetto After Effects, anche se non in modo diretto come avviene per gli altri tipi di file degli altri programmi: una volta indicato il .fla da importare, After Effects automatizza la conversione ed esportazione di tutti i livelli componenti il progetto .fla in singoli file .swf, importando successivamente, sempre in modo automatico, i suddetti file generati nel medesimo ordine. Questa soluzione è però a mio modesto parere ancora poco efficiente, in quanto, se è vero che abbatte i tempi di esportazione e importazione, appesantisce l'economia generale del progetto, dovendo generare dei file, sprecando memoria sulla macchina, e spesso in modo non indifferente per file .fla aventi numerosi livelli. Ad ogni modo, l'arrivo di questa nuova funzionalità in After Effects ha confermato l'effettiva esistenza del problema anche all'infuori del caso particolare della produzione *Milkoffee*, e quindi la necessità di una nuova soluzione, possibilmente più efficiente di quella proposta da Adobe.

Come già affermato con Illustrator e After Effects, anche quest'ultimo e Animate hanno il potenziale di interfacciarsi molto più solidamente, andava solo trovata la maniera di estrapolare e lavorare i dati vettoriali. Con il potenziale in una mano e la necessità nell'altra, è iniziato il lavoro di ricerca riportato di seguito nel prossimo capitolo.

Capitolo 2

La Ricerca

In questo capitolo verranno descritte nel dettaglio tutte le alternative di sviluppo offerte da Adobe accennate nel capitolo precedente, esponendone differenze e punti di forza, indicando oltretutto l'alternativa scelta per il progetto in questione. Dopodiché verranno analizzate le API specifiche dei due programmi protagonisti del progetto, After Effects e Animate, sottolineandone le differenze soprattutto dal punto di vista dell'impiego della grafica vettoriale. Infine, verrà fatta una breve panoramica sugli strumenti utilizzabili, ponendo l'accento su quelli effettivamente scelti per lo sviluppo.

2.1 Sviluppo per Adobe: panoramica sulle alternative

Prima di entrare nel merito del paragrafo, bisogna fare una doverosa premessa: quanto verrà descritto in seguito, per quanto abbia indubbio carattere generale nell'ecosistema Adobe, ha forti connotazioni specifiche per i due programmi in esame, e quindi non sempre quanto viene detto ha il medesimo esatto valore per After Effects e Animate, o

per After Effects e Photoshop e così via. Indicherò quindi, qualora ce ne fossero, le differenti sfumature riguardanti le due app in esame, e cercherò anche, quando possibile, di riportare quelle degli altri programmi Adobe, entro i limiti di approfondimento concessi, essendo qui materia di ricerca nient'altro che After Effects e Animate. Invito quindi di approfondire personalmente gli altri programmi nelle documentazioni ufficiali Adobe riportate in precedenza, semmai quanto scritto in questa tesi non risultasse sufficientemente esaustivo.

Ciò detto, si può iniziare a parlare delle alternative di sviluppo suddividendole in due macro-categorie: *scripts* e *plug-ins*. La prima, sostanziale differenza tra le due, come già anticipato, sta nel linguaggio in cui queste vengono sviluppate: C/C++ per i plug-ins ed *ExtendScript* o, solo per Animate, *Flash JavaScript* per gli scripts, due dialetti derivati di ECMAScript (specifica di linguaggio standardizzata, origine di JavaScript, di cui è un'implementazione), creati appositamente da Adobe; in quanto dialetti, hanno comunque la medesima struttura (o *grammatica*, per così dire) del classico JavaScript ma con l'integrazione di funzioni specifiche per la manipolazione dei DOM (*Document Object Model*) delle app Adobe. Nel caso particolare di After Effects inoltre, a mio parere il software con la più alta capacità di personalizzazione, la categoria di script può essere declinata in varie definizioni, alle volte con sostanziali differenze, ma più spesso risultando in massima parte in espressioni *gergali* del settore, non distinguendosi particolarmente da comuni scripts (vedasi più avanti la definizione di *super script*).

Ovviamente le differenze non si fermano ai linguaggi utilizzati per lo sviluppo, ma invito il proseguimento della lettura per l'approfondimento.

2.1.1 Scripts

Partiamo dalla definizione più generica che si possa dare per uno script in contesto Adobe: uno script non fa altro che automatizzare delle procedure utilizzando ciò che il software *sa già fare*, senza aggiungere nuove funzionalità vere e proprie. Tradotto in termini più tecnici, uno script, a differenza di un plug-in, *non può disegnare pixel*, non direttamente almeno. Deve infatti affidarsi alle funzionalità già presenti nel programma per portare a termine i propri compiti. Questo potrebbe sembrare riduttivo in termini pratici, ma così non è: uno script infatti può essere utilizzato per una moltitudine di operazioni, sia da un punto di vista gestionale che creativo, in quanto può abbattere i tempi di lavoro automatizzando azioni ripetitive e/o lunghe, oppure combinare le funzionalità del programma creando effetti completamente nuovi e altrimenti impossibili utilizzando l'interfaccia nativa del software. Uno script ha infatti accesso all'API del programma, potendone così governare l'*Object Model* fin nelle sue più piccole componenti.

Tutti i programmi del *Creative Cloud* hanno la possibilità di eseguire scripts, chiaramente ognuno con la sua API specifica. Il linguaggio di riferimento è *ExtendScript* (la cui estensione file è .jsx), una derivazione o, volendo, *espansione* di JavaScript comprendente le API Adobe; costituisce un caso a sé stante Animate, unico a non essersi

adattato, probabilmente per questioni ereditarie, ad ExtendScript: esso infatti utilizza un diverso linguaggio, denominato semplicemente *Flash JavaScript* (la cui estensione file è .jsfl). Oltretutto in Animate gli scripts assumono la denominazione di *Comandi* (*Commands*), laddove negli altri programmi la nomenclatura rimane ugualmente per tutti *Scripts*.

In After Effects lo scripting assume diverse forme e denominazioni che vale la pena approfondire: la prima da far presente è uno strumento molto comune tra gli utilizzatori del software, talmente comune da esulare dal puro campo dello sviluppo software ed essere una competenza propria di professionisti anche meno tecnici (ma comunque più esperti dell'utente medio). Va inoltre detto che quanto riportato successivamente non rientra in tutto e per tutto nella definizione di script in senso stretto (inteso nel contesto Adobe), ma data la radice comune dell'argomento in questione, e in mancanza di un altro spazio in questa tesi consono alla trattazione del suddetto, trovo appropriato riportarlo qui di seguito. Sto parlando delle *espressioni* (*expressions*), strumento, come sopraindicato, tanto comune quanto potente nell'utilizzo di After Effects. Propriamente, le espressioni sono *snippet* di codice utilizzati per manipolare i valori di un singolo parametro o proprietà di un livello nella timeline di After Effects, definendone un *comportamento*. In quanto tali, sono molto meno complessi di uno script e soprattutto sono tempo-dipendenti: essi vengono infatti eseguiti e valutati in *runtime* con l'avanzare del conteggio dei frame. Per queste ragioni possono essere accomunati agli scripts, in quanto scritti in un linguaggio basato su JavaScript (denominato, in modo non troppo ufficiale, semplicemente *After Effects Expression Language*) e interpretati da un motore JavaScript V8 integrato, ma se ne discostano nelle premesse applicative e nei limiti di usabilità. Ad ogni modo trovano largo utilizzo e utilità, e non hanno necessità di particolari strumenti per essere scritte, potendolo fare direttamente nella timeline di After Effects (all'interno della proprietà interessata).

Inoltre, After Effects offre una variegata gamma di *expression controllers*, particolari effetti nei quali è possibile definire un valore di qualsiasi natura, sia essa numerica (*Slider Control*), angolare (*Angle Control*), booleana (*Checkbox Control*), un codice colore (*Color Control*), un punto spaziale (*2D/3D Point Control*) o il riferimento a un altro livello della composizione (*Layer Control*). Un'espressione, invece di esprimere un valore direttamente nel proprio codice, può essere connessa a un *controller*: così facendo, quest'ultimo può *pilotare* l'espressione attraverso i propri valori. E potendo variare nel tempo i valori definiti dai *controllers* (essendo degli effetti è possibile definire dei fotogrammi chiave anche su di essi), ecco che viene ad aggiungersi un nuovo grado di complessità ai comportamenti definibili dalle espressioni. È possibile visualizzare i concetti finora espressi in Figura 2.1.

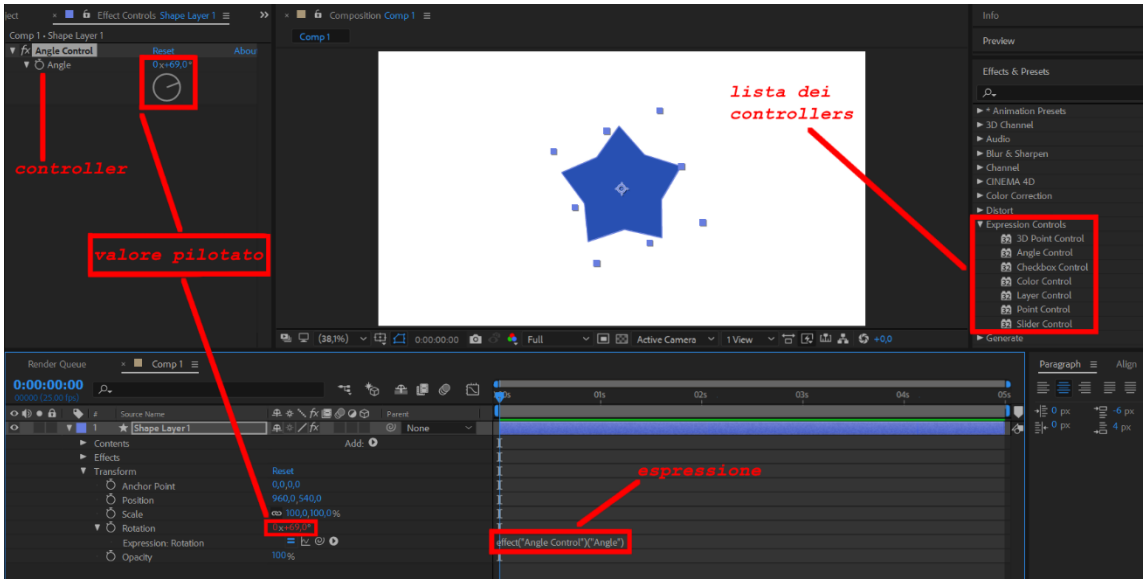

Figura 2.1 - visione dell'interfaccia di After Effects in cui la proprietà "rotation" di uno Shape Layer viene pilotata da un Angle Control; questo comportamento viene definito dall'espressione presente in timeline. Da notare infine la lista completa dei controllers disponibili nel programma (riquadro destro)

Sempre tramite l'impiego di *controllers* è possibile definire un'altra categoria di pseudo-script, e il termine *pseudo* non è stato utilizzato a caso in quanto questo tipo di costrutti vengono appunto denominati *Pseudo Effetti* (*Pseudo Effects*). In realtà uno pseudo effetto altro non è che la definizione alternativa di un *custom controller*, ovvero il risultato del *raggruppamento* di più *controllers* in un'unica struttura organizzata; la creazione di uno pseudo effetto permette di aumentare sensibilmente il livello d'interazione che i *controllers* possono offrire alle espressioni che andranno a pilotare (oltre che un conseguente miglioramento nello spazio di lavoro dovuto all'ordinamento dei suddetti *controllers*, come rappresentato in Figura 2.2).

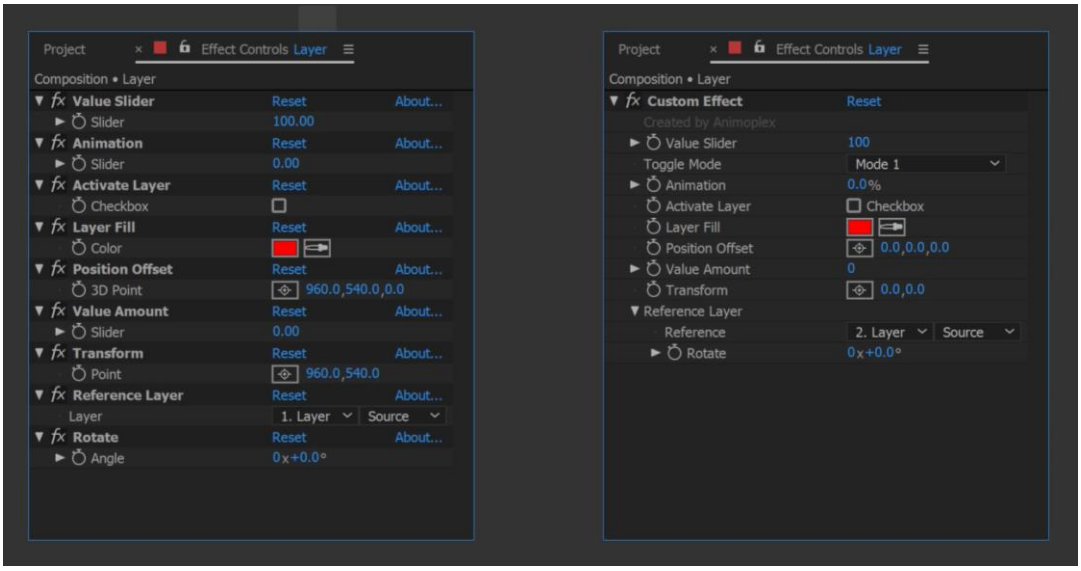

Figura 2.2 - confronto tra l'impiego di controllers liberi (sinistra) e l'organizzazione degli stessi in uno pseudo effetto (destra)

La costruzione di uno pseudo effetto non richiede obbligatoriamente l'impiego di particolari strumenti, infatti è definibile manualmente in un file *XML* fornito di default

nell'installazione di After Effects, il file *PresetEffects.xml*; la struttura definita nel file rifletterà la tipologia nonché l'ordine dei *controllers* che costituiranno lo pseudo effetto (ed è oltretutto possibile dare un nome al proprio *custom controller*). Il percorso del suddetto file è il seguente: *C:\Program Files\Adobe\Adobe After Effects CC [version]\Support Files\PresetEffects.xml*. Chiaramente un approccio del genere richiede, seppur in minima parte, delle conoscenze di programmazione in linguaggio XML; avendone la possibilità però, è possibile un metodo alternativo nella creazione di pseudo effetti, ossia grazie all'estensione *Pseudo Effect Maker*, con la quale è possibile, tramite una semplice interfaccia grafica, modificare il suddetto file XML senza modifica diretta del codice.

ExtendScript ha tra le sue funzioni una limitata capacità di definire un'interfaccia grafica. Qualora uno script facesse uso di queste librerie, e potendo quindi visualizzare un'interfaccia, se collocato nella giusta cartella di installazione (*C:\Program Files\Adobe\Adobe After Effects CC [version]\Support Files\Scripts\ScriptUI Panels*) diverrebbe uno *ScriptUI*, con l'effetto di poter essere visualizzato come finestra mobile e agganciabile nell'interfaccia generale del programma.

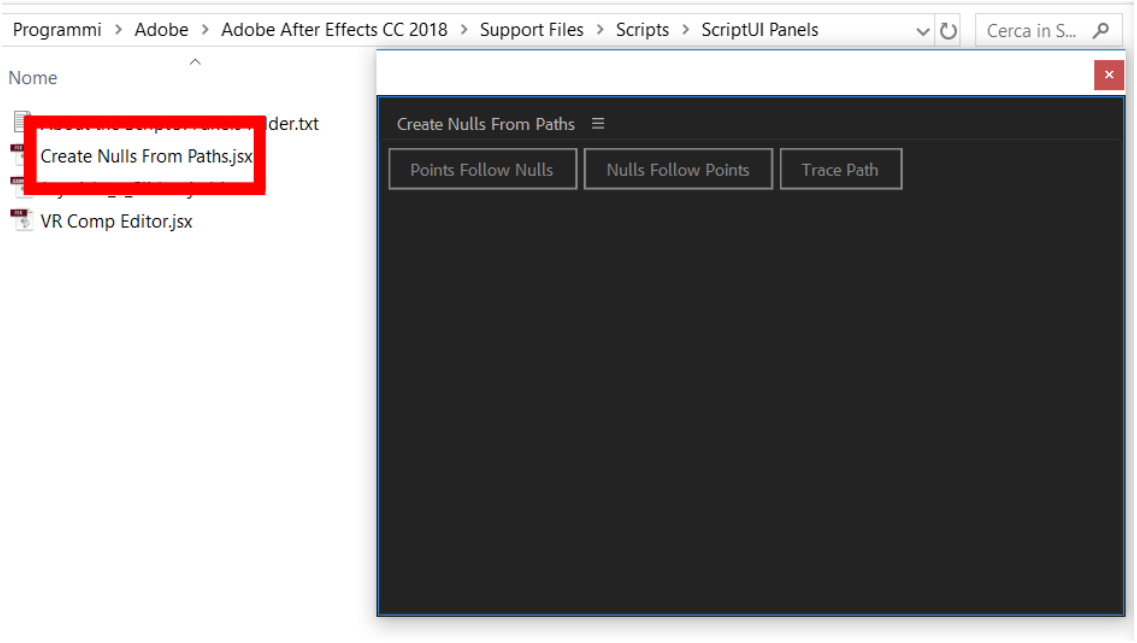

Figura 2.3 - la cartella degli ScriptUI con relativo esempio di script renderizzato

L'ultima tipologia di script qui presentata è in realtà totalmente un'espressione gergale del settore, essendo questi in tutto e per tutto dei classici script o, il più delle volte, degli ScriptUI: i *Super Scripts*. Quando uno script raggiunge un certo livello di complessità, permettendo funzionalità al limite dell'incredibile per uno script, quasi a celebrarne il risultato raggiunto, riceve il titolo di *Super Script*, termine coniato dalla *community* ma che non trova alcuna distinzione formale dal semplice termine *Script*. In termini di funzionalità, un super script si potrebbe quasi avvicinare ad un plug-in: vedasi ad esempio *Newton*, che permette la simulazione di un motore fisico (*physics engine*) in After Effects, o *Duik Bassel*, che implementa un sistema di *rigging* molto avanzato, diventato tra l'altro uno standard nella *character animation 2D*.

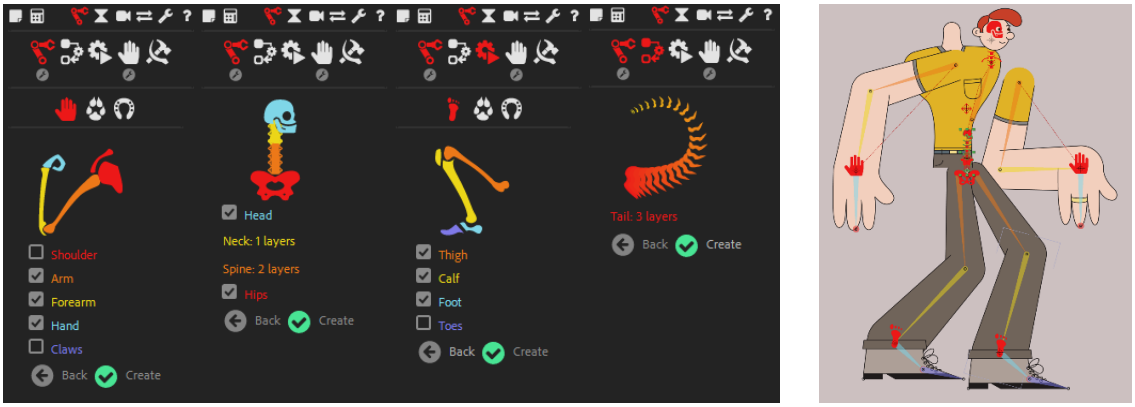

Figura 2.4 - interfaccia di Duik Bassel con relativo esempio di character rig. Duik ha rappresentato una vera rivoluzione per la character animation 2D, introducendo un sistema di rigging estremamente avanzato rispetto a quanto After Effects possa offrire nativamente

2.1.2 Plug-ins

Ribaltando quanto detto per gli script, un plug-in *può disegnare pixel* e viene comunemente definito come quel componente che *va oltre* ciò che il software sa fare. In effetti, essendo sviluppato con codice esterno (C/C++), è a conti fatti un programma *parallelo* che vive all'interno dell'app. Il primo modo per distinguere un plug-in da un qualsiasi altro componente è il fatto di trovarlo tra gli *effetti* di After Effects. Ma utilizzando un'altra chiave di lettura, potrebbe stupire in realtà scoprire che *tutti gli effetti sono plug-ins*. Andando ad esplorare nelle cartelle delle risorse del programma, si potrà infatti verificare che gli effetti standard, quelli già presenti alla prima installazione, e i *plug-ins* di terze parti hanno la medesima estensione *.aex*.

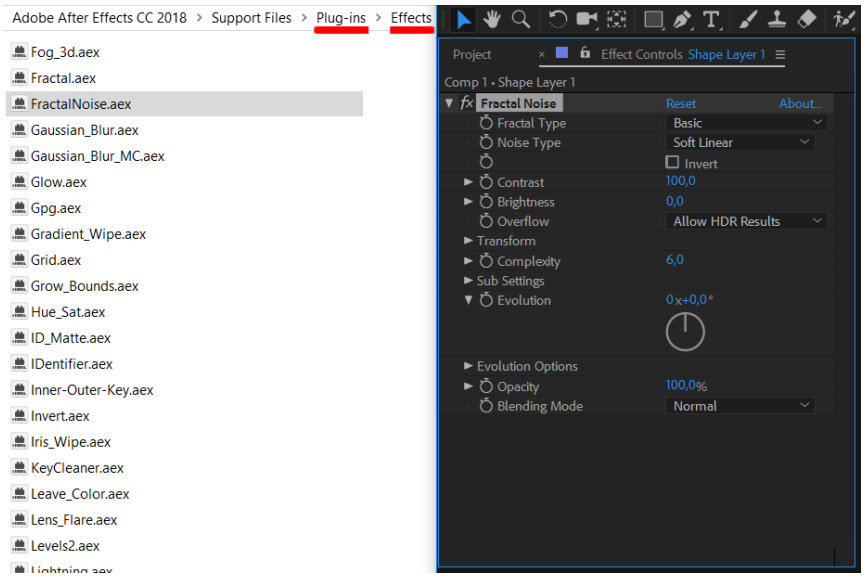

Figura 2.5 - tutti gli effetti di After Effects di fatto sono plug-ins

Non è quindi propriamente corretta la classica definizione fornita dalla *community* di un plug-in sul fatto di essere quel componente capace di *andare oltre* rispetto le capacità del software, quanto in realtà essere un componente che *inserisce* (letteralmente *plugs in*) una nuova funzionalità, ed è piuttosto la totalità dei plug-ins a definire ciò che il software sa fare. Detto questo però c'è da dire che esistono plug-ins di estrema complessità, che sono obiettivamente dei software autonomi rispetto After Effects, ed è questo il caso ad esempio di *Element 3D,* di *Trapcode Particular* o dei plug-in ufficiali di *mocha* e *Cinema 4D,* che sono in tutto e per tutto delle versioni ridotte dei software di origine, aventi la propria interfaccia separata da After Effects. Ad ogni buon conto, a prescindere dalla complessità di un plug-in, essi rappresentano *l'ultima frontiera* dello sviluppatore in termini pratici, nonché *l'ultima spiaggia* in termini concettuali: data l'elevata difficoltà di esecuzione nella creazione di un plug-in infatti, questa dovrebbe essere presa in considerazione solo dopo aver vagliato tutte le alternative possibili, e solo allora tentare l'impresa del suo sviluppo. Adobe, nel sito *Adobe I/O*, fornisce gratuitamente l'SDK per il *plug-in development*, comprendente di manuale. Ciononostante, la scarsità di materiale d'apprendimento, oltre la sopracitata documentazione ufficiale, testimonia quanto lo sviluppo di plug-ins rimanga comunque un campo ancora relativamente inesplorato dalla *community*, rimanendo prerogativa di un settore professionale molto specializzato. Sono infatti molto pochi gli studi e le software house che possono vantare una proposta di plug-ins, tra gli altri i premiati *Videocopilot* (studio sviluppatore, tra gli altri, del già citato *Element 3D* e *Optical Flare*) e *Red Giant* (sviluppatori del sopracitato *Trapcode*, *Magic Bullet*, *Universe* e molti altri).

2.1.3 La terza scelta: le Estensioni

Dalla distinzione tra script e plug-in presentata è stato subito chiaro, ai fini del progetto di questa ricerca, che si sarebbe intrapreso la via di sviluppo di uno script: le funzionalità delle due app infatti potenzialmente già permettevano la possibilità di eseguire quanto richiesto dalla soluzione prospettata; non era quindi necessario inventarsi una nuova funzionalità *ex novo*, scartando così l'alternativa del plug-in. Ciononostante, è stato subito abbastanza chiaro anche che nello sviluppo dello script si sarebbero affrontati due problemi non indifferenti: la *comunicazione inter-app* e l'*integrazione delle due API in linguaggi differenti* (JSX per After Effects e JSFL per Animate). Un semplice script infatti non permette una comunicazione diretta tra due app, ed una comunicazione diretta presuppone che si possano utilizzare indifferentemente i due linguaggi dei due software. Ed è nella ricerca per la risoluzione di queste due problematiche che ci si imbatte inesorabilmente nel concetto di *Estensione,* che nell'ambiente R&D Adobe viene spesso presentata come una *terza* categoria di componenti aggiuntivi (oltre ai già esaminati script e plug-in). In realtà, se dovessimo classificarli esclusivamente per il linguaggio utilizzato nello sviluppo delle funzionalità di base, queste cadrebbero inevitabilmente nella categoria degli scripts, in quanto anch'esse scritte in ExtendScript o Flash JavaScript alle loro fondamenta. Ma c'è un fattore, già accennato precedentemente, che fa sì che le estensioni assumano un proprio carattere originale, crescendo in complessità di sviluppo

e funzionalità offerte, distinguendosi così da script e plug-in e piazzandosi a metà strada tra le due categorie: l'impiego di *tecnologia web*. Le estensioni si fondano infatti sul *CEP – Common Extensibility Platform*, piattaforma che permette l'integrazione della più comune tecnologia web, HTML, CSS e JavaScript all'interno dei programmi Adobe: dal 2014 infatti le app Adobe implementano il *CEF – Chromium Embedded Framework*, un framework basato su *Chromium*, il web browser *open source* alla base di Google Chrome, potendo così renderizzare pannelli basati su HTML5 e interpretare JavaScript tramite il motore V8. In pratica grazie al CEP è possibile costruire delle vere e proprie *web app* che vivono all'interno dei programmi Adobe, un po' come i plug-ins rappresentano delle *app desktop* interne. È facile intuire come l'impiego della tecnologia web permetta la creazione di componenti integrativi molto più ricchi e potenti rispetto ai semplici scripts, sia dal punto di vista grafico che funzionale.

Dopo questa dovuta introduzione del CEP, e tornando alle problematiche di cui sopra (*comunicazione* e *integrazione*), il CEP ne rappresenta la soluzione, compensandole: nella piattaforma sono infatti presenti due API JavaScript, *CSInterface.js* e *Vulcan.js*, appositamente pensate, tra le altre cose, per l'integrazione dei linguaggi proprietari JSX e JSFL, i quali rimangono comunque essenziali per accedere alle funzionalità del programma su cui è installata l'estensione, e la comunicazione tra due o più app Adobe. S'invita a proseguire oltre nella lettura per una trattazione più approfondita riguardante il CEF, il CEP e le sue API.

2.2 CEF & CEP

Volendo utilizzare una metafora automobilistica, il CEP rappresenterebbe la totalità della *macchina*, il CEF ne rappresenterebbe il *motore* e infine le API, ciò che permette di *interfacciarsi* col motore e il resto della macchina, sarebbero il *cruscotto*, inteso come totalità della *strumentazione di controllo* del veicolo. Con questa metafora in mente, cerchiamo ora di mettere maggior ordine nella terminologia finora esposta: la *Common Extensibility Platform* è così chiamata poiché *piattaforma*, ovvero *base* software, su cui sviluppare le *estensioni*; *comune* perché *condivisa* da tutte le app, da intendersi sia come *condivisione di funzionalità* che di *uguaglianza nei processi di sviluppo*. È quindi la sovrastruttura che governa l'ecosistema delle estensioni Adobe. In termini operativi però ciò che fa *muovere* la macchina CEP è il *Chromium Embedded Framework*, il motore: questo è un *framework*, *open source*, pensato per permettere l'*integrazione* del web browser *Chromium*, anch'esso *open source*, all'interno di applicazioni terze. Questo permette l'implementazione di funzionalità di *web browsing* nelle suddette applicazioni, la possibilità di utilizzare HTML5 e CSS per la creazione di interfacce grafiche e soprattutto l'utilizzo di JavaScript per la definizione delle funzioni, grazie all'*engine* JavaScript V8 integrato in Chromium. Inoltre, tramite il CEF è permesso utilizzare qualsiasi tecnologia possa essere impiegata nell'ecosistema web, primo fra tutte *Node.js*, ma anche librerie come *jQuery* o frameworks come *Angular*…potenzialmente non esiste un limite pratico con ciò che è possibile creare con il CEP: se esiste sul web può essere integrabile su Adobe.

Bisogna però tenere ben presente che le capacità di scripting in JavaScript offerte dal CEP non possono sostituirsi alle API proprietarie di Adobe e i relativi linguaggi, le quali risultano ancora l'unico strumento per accedere alle funzionalità *core* dei programmi. Oltretutto, ritengo importante a questo punto portare alla luce la particolarità strutturale dell'architettura di un pannello HTML di una estensione CEP: l'interpretazione del linguaggio di scripting, per quanto quelli proprietari siano una derivazione di JavaScript, non è affidata ad un unico motore JavaScript; bensì, nell'utilizzo di un'estensione CEP convivono due motori separati, uno dedicato all'interpretazione dell'API proprietaria dell'app (l'*host tool*) e un altro impiegato nell'interpretazione degli script interni al pannello HTML, ovvero il suddetto motore JavaScript V8 implementato nel CEF. Questa architettura si può riassumere come in Figura 2.6:

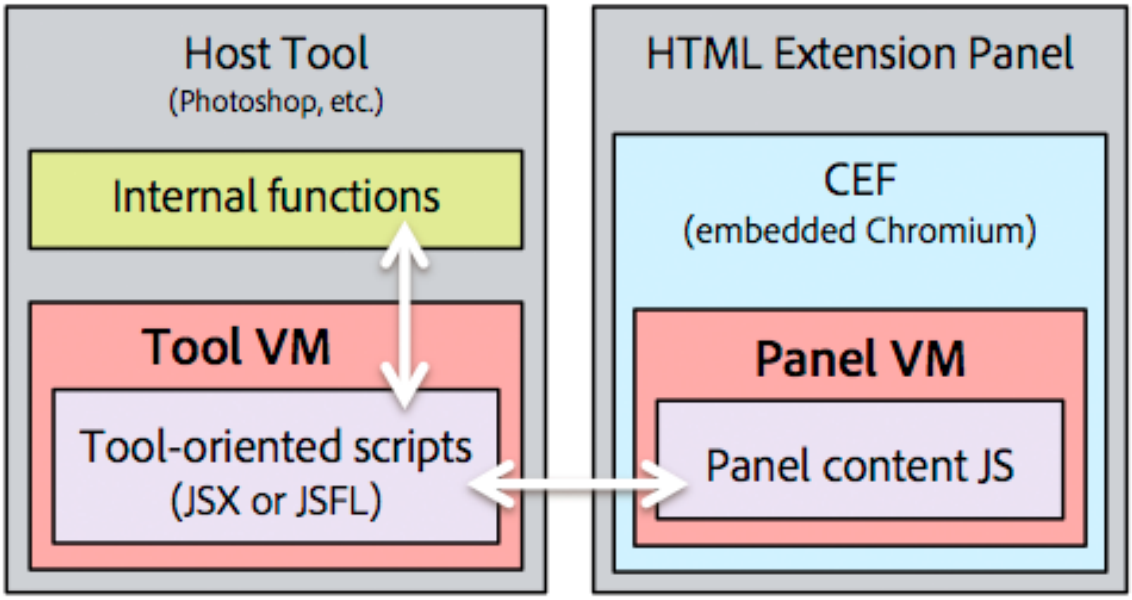

Figura 2.6 – schematizzazione dell'architettura CEP, con la separazione delle VM

Il *Tool VM* e il *Panel VM* (dove VM sta per *Virtual Machine*) sono rispettivamente il motore per il linguaggio proprietario dell'app *host* e il motore V8 di Chromium e da qui in avanti si utilizzerà questa nomenclatura per indicarli. Ora, la freccia orizzontale rappresenta la necessaria comunicazione che deve avvenire tra i due ambienti per il corretto funzionamento dell'estensione. La risoluzione di questo collegamento arriva dalla prima delle due API proposte nel CEP, la *CSInterface.js*, il cui metodo *evalScript()* rappresenta proprio quel *ponte* fondamentale tra le due VM. Oltre a questa sostanziale funzionalità, la *CSInterface* offre una serie di metodi dedicata alla comunicazione *intra-app*, ovvero la gestione degli eventi interni al programma, nonché per il recupero di varie informazioni, tra le quali informazioni sull'*host tool* (l'app Adobe attualmente in uso), sul sistema operativo in uso e suo percorso nel computer e sul CEP stesso.

La seconda API, *Vulcan.js*, permette la comunicazione *inter-app*, ovvero tra due o più app Adobe sul quale possa essere installata la stessa estensione, oltre che comprendere un'utile serie di metodi dedicati al recupero delle informazioni su tutte le app installate, potendole anche *lanciare* all'occorrenza. Il funzionamento dei suddetti metodi è gestito tramite l'impiego di *specificatori identificativi*, uno per ogni app e dei quali se ne dà una

tabella riassuntiva nella Figura 2.74, insieme ai numeri di versioni corrispondenti alle varie edizioni dei software.

Application	Host ID	CC Version	CC 2014 Version	CC 2015 Version	CC 2015 Dot Version	CC 2017 Version	CC 2018 Version	CC 2019 Version
Photoshop	PHSP/PHXS	14 (CEP 4)	15 (CEP 5)	16 (CEP 6)	17.0.2 (CEP 7)	18 (CEP 7)	19 (CEP 8)	20 (CEP 9)
InDesign	IDSN	9 (CEP 4)	10 (CEP 5)	11 (CEP 6)	-	12 (CEP 7)	13 (CEP 8)	14 (CEP 9)
InCopy	AICY	9 (CEP 4)	10 (CEP 5)	11 (CEP 6)	-	12 (CEP 7)	13 (CEP 8)	14 (CEP 9)
Illustrator	ILST	17 (CEP 4)	18 (CEP 5)	19 (CEP 6)	20 (CEP 7)	21 (CEP 7)	22 (CEP 8)	23 (CEP 9)
Premiere Pro	PPRO	7 (CEP 4)	8	9	10.3 (CEP 6)	11 (CEP 6)	12 (CEP 8)	23 (CEP 9)
Prelude	PRLD	2 (CEP 4)	3	4	5.0.1 (CEP 6)	6 (CEP 7)	7 (CEP 8)	8 (CEP 9)
After Effects	AEFT	12	13	13.5	13.8.1 (CEP 6)	14 (CEP 6)	15 (CEP 8)	16 (CEP 9)
Animate (Flash Pro)	FLPR	13	14 (CEP 5)	15 (CEP 6)	15.2 (CEP 6.1)	16 (CEP 6.1)	18 (CEP 8)	19 (CEP 9)
Audition	AUDT	6	7	8	9.2.1 (CEP 6)	10 (CEP 6)	11	12 (CEP 9)
Dreamweaver	DRWV	13 (CEP 4)	15 (CEP 5)	16 (CEP 6)	-	17 (CEP 6.1)	18 (CEP 8)	19 (CEP 9)
Muse	MUSE	7.4	-	2015	-	2017	2018	-
Bridge	KBRG	6	-	6.3.1	-	-	8 (CEP 8)	9 (CEP 9)
Rush	RUSH	-	-	-	-	-	-	1 (CEP 9)

Figura 2.7 – tabella riassuntiva dei codici delle app e delle versioni CEP

Il corretto funzionamento del sistema di messaggistica per la comunicazione *cross-platform* è garantito grazie all'impiego del *singleton VulcanInterface*. Di seguito viene elencata la sequenza di procedure disposte al *Vulcan Messaging*:

1) *Registrazione Listener - VulcanInterface.addMessageListener(type, callback):* viene predisposto il sistema (la *VulcanInterface*) alla ricezione di un certo *tipo* di messaggio e alla sua conseguente *gestione*; *type* rappresenta l'*intestazione* del messaggio, mentre *callback* è la funzione approntata alla gestione del messaggio una volta ricevuto: in essa, qualora il messaggio trasportasse un *carico utile* di informazioni, il metodo *VulcanInterface.getPayload(vulcanMessage)*, descritto al punto 5), deve necessariamente essere presente. Volendo utilizzare una *metafora postale*, il *Message Listener* rappresenta la *buca delle lettere* o la *casella di posta* il cui *indirizzo* è indicato da *type*.

2) *Creazione Messaggio – (new) VulcanMessage(type):* la classe *VulcanMessage* implementa gli oggetti-messaggio utilizzati per impacchettare il *carico utile*, le informazioni da spedire. Metaforicamente rappresenta la *busta* della lettera da inviare all'*indirizzo "type"*.
3) *Scrittura Messaggio – VulcanMessagge.setPayload(string):* scrive il *carico utile* da *imbustare* nel messaggio da spedire in forma di stringa. È la *lettera*.
4) *Spedizione Messaggio: VulcanInterface.dispatchMessage(msg):* il messaggio, il quale ricordiamo ha nella sua intestazione l'*indirizzo* di consegna, viene spedito. Questo verrà captato dal *listener* corrispondente all'indirizzo *type* del messaggio Vulcan.
5) *Gestione Messaggio – VulcanInterface.getPayload(msg):* il messaggio, ricevuto dal *listener*, viene aperto e ne viene letto il *carico utile* (che ricordiamo, è in forma di stringa); questo passaggio, benché presentato per ultimo, è conseguente al punto 1), in quanto di norma il metodo di gestione del messaggio va approntato durante la definizione del *listener* e della sua *callback function*. Va inoltre fatto notare che in *Vulcan.js* è presente un metodo *getPayload* sia per la *VulcanInterface* sia per la classe *VulcanMessage*; ma siccome il primo, nella sua risoluzione, richiama il secondo, è buona norma utilizzare sempre questa (il primo), poiché preferibile utilizzare sempre la *VulcanInterface* quando possibile.

Quindi, sintetizzando: l'API *CSInterface.js* è praticamente *obbligatoria* ed è utilizzata a livello *interno* all'app *host*, ovvero per il recupero delle informazioni interne dell'estensione e/o dell'*host tool* e per la gestione degli eventi, anch'essi interni (*CSEvents*), oltre che per il vitale compito di connessione tra le due VM coinvolte nel funzionamento dell'estensione. L'API *Vulcan.js* è invece utilizzata a livello *esterno* all'app *host*, ovvero per il recupero delle informazioni su *tutti* gli *host tools* installati sulla macchina locale e per la gestione della messaggistica tra app (*VulcanMessages*).

2.3 Animate & After Effects: API a confronto

Analizzata la tecnologia a disposizione per la *forma* dell'estensione, passiamo ora alla tecnologia disponibile per il *contenuto* della suddetta: si parla qui delle API dei due programmi, che d'ora in avanti verranno indicati tramite il nome dei linguaggi che li caratterizzano, ovvero *ExtendScript* e *JSFL* (acronimo per *Flash JavaScript*). In particolare, si prenderanno in esame le capacità per la grafica *vettoriale*, confrontandone le differenze nell'ottica di trovare dei tratti *comuni* nel *modello a oggetti* delle due API, in modo da agevolarne lo scambio dati.

Prima di addentrarsi nel vivo dell'analisi, va però introdotto un termine che sta alla base di tutta quanta la ricerca qui proposta sulla grafica vettoriale, vale a dire *Shape*. Con questo termine inglese (letteralmente *forma, figura, sagoma*) si vuole qui indicare una *geometria 2D di natura vettoriale* (un po' come il termine *Mesh* per la grafica 3D).

2.3.1 ExtendScript (After Effects)

Iniziamo ora l'analisi a partire da ExtendScript e il suo DOM, del quale se ne dà uno schema riassuntivo nella Figura 2.8 di seguito.

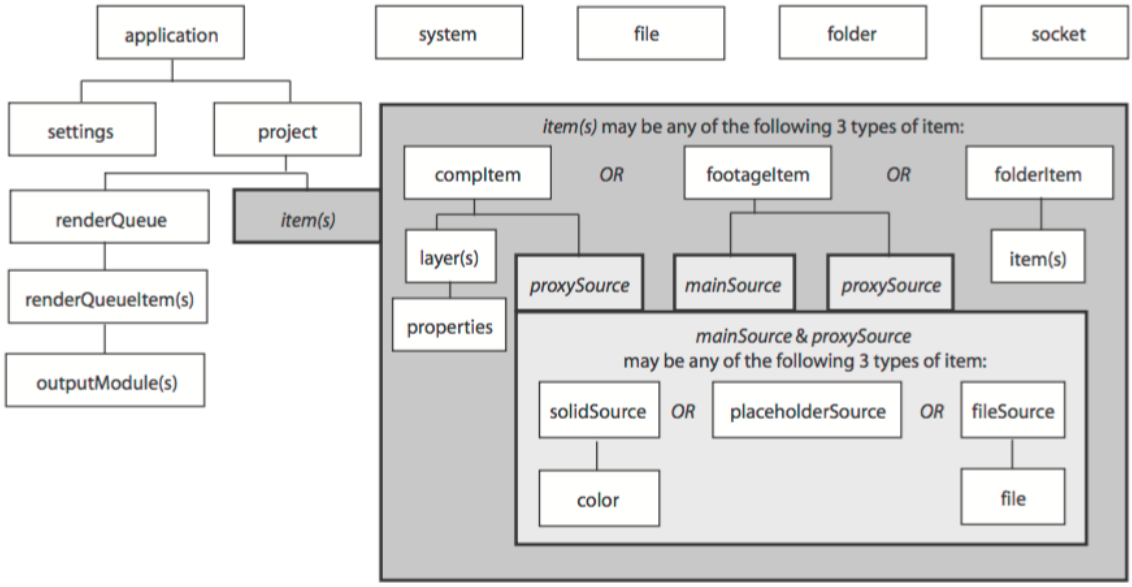

Figura 2.8 – DOM di After Effects

Nello schema non è presente l'ulteriore suddivisione dell'oggetto *Layer* nelle sue sottoclassi, per le quali se ne dà qui una lista schematizzata:

Layer Object:

1) *AVLayer Object:*
   - *ShapeLayer Object;*
   - *TextLayer Object;*
2) *CameraLayer Object;*
3) *LightLayer Object;*

È lo *ShapeLayer* a rappresentare l'oggetto di maggior interesse, in quanto la tipologia di *layer* After Effects coinvolto nella grafica vettoriale.

Nonostante la mancanza di alcune sottoclassi nello schema, la struttura completa del DOM di After Effects non è molto più complessa di quanto presentato; ExtendScript risulta infatti un'API molto *maneggevole* per non dire semplice: una volta navigato il DOM *verso il basso*, ovvero verso le sue componenti più operative, ci si renderà conto che la stragrande maggioranza delle funzionalità è affidata all'oggetto *Property*, il quale rappresenta una generica proprietà di un livello. L'accesso alle proprietà specifiche del livello, spesso differenti per tipi di *layers* diversi, è possibile tramite l'utilizzo di *match names* per i quali è possibile consultare una lista nel manuale di scripting per After Effects

[14]. La modalità di espressione dei *match names* è molto versatile, con la possibilità di essere scritti in uno svariato numero di modi equivalenti. Se ne dà un esempio con la proprietà *position* del *Transform* di un qualsiasi livello:

- `layer.transform.property("position");`
- `layer.transform.property("Position");`
- `layer.transform.property("ADBE Position");`
- `layer.transform.property(1);`
- `layer.transform("position");`
- `layer.transform("Position");`
- `layer.transform("ADBE Position");`
- `layer.transform(1);`
- `layer.transform.position;`
- `layer.transform.Position;`

Come si è detto, tutti questi metodi sono equivalenti, quindi l'indicazione esplicita *property()* non risulta nemmeno obbligatoria.

Torniamo ora all'oggetto *ShapeLayer*: a livello logico questo non è altro che un *contenitore* atto a raccogliere le vere strutture dati delle geometrie vettoriali, dette *Path*, che possono essere di 4 tipi, *Rectangle Path, Ellipse Path, Polystar Path* e *Path*; i primi tre sono delle forme *primitive* definibili da semplici parametri numerici quali, ad esempio, *centro* e *assi* nel caso dell'*Ellipse Path*; qualora invece si volesse creare una forma personalizzata, viene utilizzata la quarta struttura dati *Path,* la quale descrive la geometria nella forma di *curve di Bézier cubiche* (ovvero *con 4 punti di controllo*). Le informazioni della curva vengono incapsulate in un oggetto *Shape*, che è il valore da attribuire alla proprietà *path* della struttura *Path*. Le informazioni contenute in un oggetto *Shape* sono in forma relativamente semplice: un *array* di vertici espressi nella forma *[x,y]*, al quale, eventualmente, si associano altri due array, *inTangents* e *outTangents*, rappresentanti le tangenti rispettivamente *entranti* e *uscenti* corrispondenti al singolo vertice (graficamente rappresentano gli *handles* della curva di Bézier), espressi in coordinate relative rispetto alle coordinate assolute del vertice. La corrispondenza di vertice, sua tangente entrante e sua tangente uscente è data dalla posizione nei rispettivi array: a uguale indice corrisponde correlazione dei tre dati. Il sistema di riferimento del piano, inoltre, pone l'origine al centro della finestra dell'interfaccia, sezionandola in 4 regioni; i vertici avranno quindi i seguenti valori possibili, partendo dalla regione in alto a sinistra in senso orario: *[-x, -y], [+x, -y], [+x, +y], [-x, +y]* (Figura 2.9).

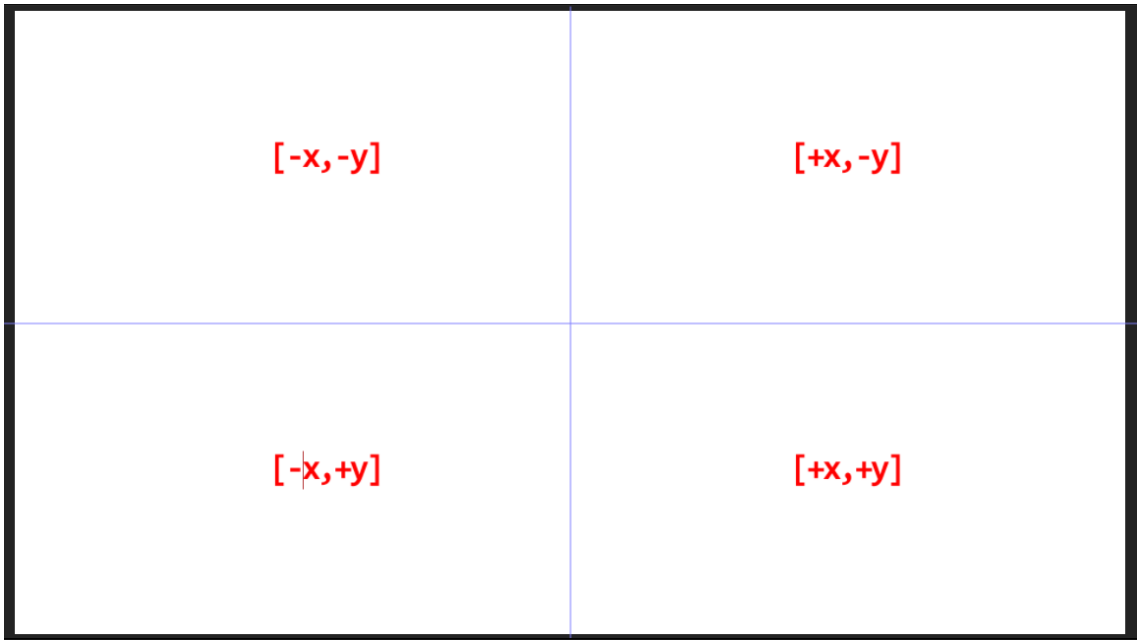

Figura 2.9 – sistema di riferimento del piano di After Effects

Uno *ShapeLayer* può contenere più *Path* (di qualsivoglia tipo), inoltre è possibile organizzarli raggruppandoli in *Shape Group*; ogni gruppo è da considerarsi una sorta di sub-*ShapeLayer*, o forse è più corretto dire che è lo *ShapeLayer* a rappresentare il *Gruppo Radice*, com'è testimoniato dai loro match names. A seguire è presentato uno schema gerarchico dei match names, con relativo esempio annotato dell'interfaccia del programma in cui vengono associati i match names alle relative strutture dati. Per non dilungarsi troppo in elenchi, verranno esposti solo quelli ritenuti essenziali per la comprensione della logica gerarchica. Una volta colta, risulterà facile ricavare da sé le relazioni delle restanti strutture dati nella gerarchia, con i rispettivi match names dall'elenco ufficiale [15].

```
ShapeLayer = 'ADBE Vector Layer'
└Contents = 'ADBE Root Vectors Group'
  ├Group = 'ADBE Vector Group'
  │  └"SubContents" = 'ADBE Vectors Group'
  │     ├PathName = 'ADBE Vector Shape - Group'
  │     │  └Path = 'ADBE Vector Shape'
  │     ├Stroke = 'ADBE Vector Graphic – Stroke'
  │     └Fill = 'ADBE Vector Graphic – Fill'
  └Rectangle = 'ADBE Vector Shape – Rect'
    └…
```

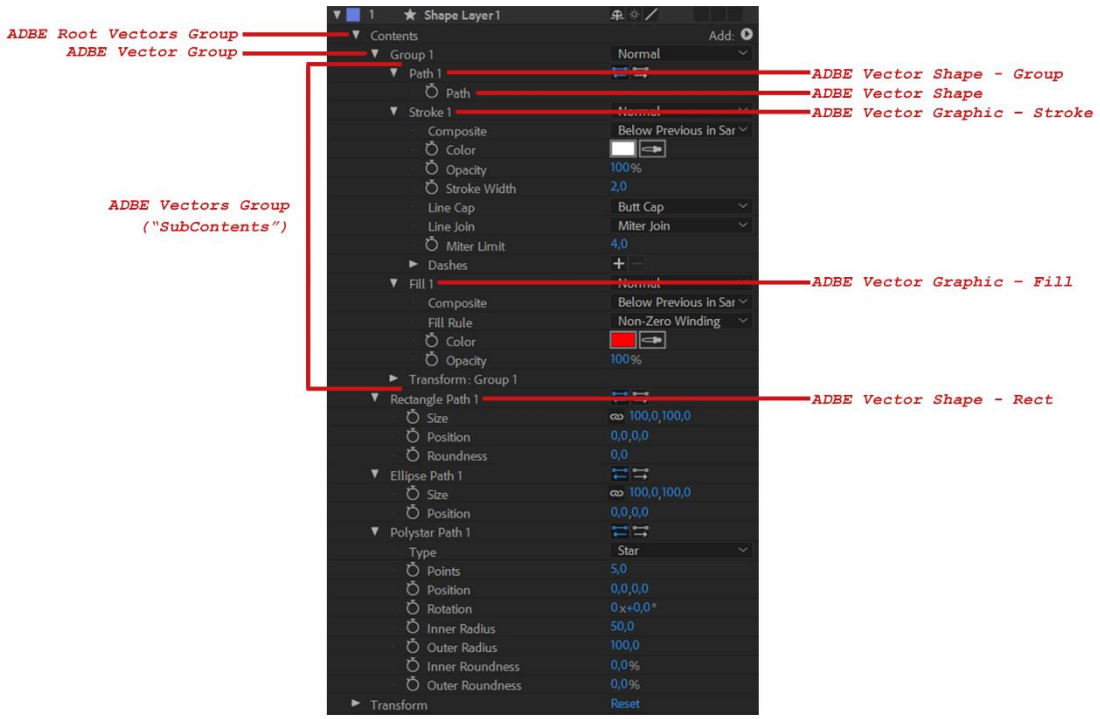

Figura 2.10 – (sopra) schematizzazione della gerarchia di uno Shape Layer; (sotto) interfaccia di uno Shape Layer con corrispondenza delle proprietà con i rispettivi "match names"

L'elemento virgolettato *"SubContents"*, il vero elemento *contenitore* di uno *Shape Group*, è in realtà invisibile nell'interfaccia del programma, ciononostante risulta di estrema importanza in sede di scripting, in quanto non sarebbe appunto sufficiente il solo *Shape Group* nel raggruppamento delle *Shape*. Come si può vedere nell'esempio, i gruppi non hanno solo importanza nell'ordinamento delle *Shapes* ma anche nell'assegnazione degli oggetti *Stroke* e *Fill*, rispettivamente il *contorno* e il *riempimento* della *Shape*. In realtà non sarebbe obbligatorio l'inserimento di questi oggetti all'interno di un gruppo, ma lo diventa nel momento in cui si voglia un'assegnazione *univoca* dei suddetti ad una determinata *Shape*. In caso contrario infatti, un oggetto *Stoke* e/o *Fill* vanno a influenzare tutti le *Shapes* presenti, a seconda di dove vengono inseriti, nei *Contents* dello *ShapeLayer* o di un suo gruppo secondo una logica *bottom up* (ovvero tutti gli elementi *sopra* lo *Stroke*/*Fill* ne sono influenzati).

Viene ora dato un esempio di codice ExtendScript in cui è possibile notare molti dei concetti espressi finora sulla disposizione del modello a oggetti di After Effects

```
//retrieve layerCollection from active Comp
var compLayers = app.project.activeItem.layers;

//add ShapeLayer to Comp
var shapeLayer = compLayers.addShape();

//ShapeLayer's Contents
var shapeLyrContents = shapeLayer.property('ADBE Root Vectors Group');

//add Group
var shapeGroup = shapeLyrContents.addProperty('ADBE Vector Group');

//Group's Contents or ShapeLayer's "SubContents"
```

```
var shapeGrpContents = shapeGroup.addProperty('ADBE Vectors Group');

//add Path
shapeGrpContents.addProperty('ADBE Vector Shape - Group');

//retrieve path property
var pathPrprty = shapeGrpContents.property(1).property('ADBE Vector Shape');

//initialize Shape Object
var myShape = new Shape();
myShape.vertices = [[300,50],[200,150],[300,250],[400,150]];
myShape.inTangents = [[55.23,0],[0,-55.23],[-55.23,0],[0,55.23]];
myShape.outTangents = [[-55.23,0],[0,55.23],[55.23,0],[0,-55.23]];
myShape.closed = true;

//assign Shape to path property
pathPrprty.setValue(myShape);
```

Si noti nell'esempio l'importanza dell'aggiunta esplicita della proprietà assegnata alla variabile `ShapeGrpContents`, che rappresenta appunto l'elemento precedentemente indicato come *"SubContents"*. Inoltre, si prenda nota della modalità di dichiarazione di un oggetto `Shape()` e sua assegnazione come valore alla proprietà `pathProperty`. Nel caso di una curva aperta (`myShape.closed = false`) il primo elemento `inTangents` e l'ultimo `outTangents` sono ignorati.

2.3.2 Flash JavaScript (Animate)

Nel caso di Animate, utilizzando principalmente grafica vettoriale, l'API è molto più ricca sia nelle strutture dati che nei metodi offerti per la loro gestione. In Figura 2.11 viene rappresentato il DOM di JSFL in una schematizzazione gerarchica

```
01 Top-Level Functions and Methods
01 FLfile object
01 flash object (fl)
  02 compilerErrors object
  02 componentsPanel object
  02 Document object (fl.documents array)
    03 Filter object
    03 Matrix object
    03 Fill object
    03 Stroke object
    03 library object
      04 Item object (library.items array)
      04 BitmapItem object(subclass of Item object)
      04 folderItem object (subclass of Item object)
      04 fontItem object (subclass of Item object)
      04 SoundItem object (subclass of Item object)
      04 SymbolItem object (subclass of Item object)
      04 VideoItem object (subclass of Item object)
    03 Timeline object (document.timelines array)
      04 Layer object (timeline.layers array)
        05 Frame object (layer.frames array)
          06 Element object (frame.elements array)
            07 Matrix object (element.matrix)
          06 Instance object (abstract class, subclass of Element object)
          06 BitmapInstance object (subclass of Instance object)
          06 CompiledClipInstance object (subclass of Instance object)
          06 ComponentInstance object (subclass of SymbolInstance object)
            07 Parameter object (componentInstance.parameters array)
          06 SymbolInstance object (subclass of Instance object)
          06 Text object (subclass of Element object)
            07 TextRun object (text.textRuns array)
              08 TextAttrs object (textRun.textAttrs array)
          06 Shape object (subclass of Element object)
            07 Oval object
            07 Rectangle object
            07 Contour object (shape.contours array)
              08 HalfEdge object
                09 Vertex object
                09 Edge object
            07 Edge object (shape.edges array)
              08 HalfEdge object
                09 Vertex object
                09 Edge object
            07 Vertex object(shape.vertices array)
              08 HalfEdge object
                09 Vertex object
                09 Edge object
        05 Parameter object (screen.parameters array)
  02 drawingLayer object
    03 Path object
      04 Contour object
  02 Math object
  02 outputPanel object
  02 presetPanel object
    03 presetItem object(presetPanel.items array)
  02 swfPanel object
  02 Tools object (fl.tools array)
    03 ToolObj object (tools.toolObjs array)
  02 XMLUI object
```

Figura 2.11 – DOM di Animate

L'oggetto *flash* a capo della gerarchia è un residuato della vecchia nomenclatura del programma (*Flash Pro*). Oggi è sostituito dall'oggetto *animate*, anche se in realtà è possibile utilizzare entrambi i termini alternativamente come sinonimi. È inoltre possibile riferirsi ad essi con le abbreviazioni `fl` e `an` in sede di scripting.

Andiamo subito all'oggetto di principale interesse per la ricerca, lo *Shape Object*: innanzitutto definiamone il percorso *verso l'alto* nel DOM, così da identificare le modalità di accesso ai dati di una singola *Shape* partendo dall'oggetto *animate*, dopodiché potremo addentrarci all'interno di un oggetto *Shape*, navigando il DOM *vero il basso* e scoprirne i dati utili ai fini del progetto.

Come si può vedere dallo schema, uno *Shape Object* è una sottoclasse dell'oggetto *Element*; un *Element Object* rappresenta un qualsiasi elemento presente *in stage* (l'inquadratura), sia esso di tipo *text* (*Text Object*), *instance* (*Instance Object*, classe astratta per rappresentare un qualsiasi elemento *istanziato* dalla libreria del documento FLA corrente; gli oggetti presenti nella libreria sono di tipo *Item Object*) o il già citato *shape*. I primi due tipi, nonostante rientrino negli interessi di sviluppo futuri del progetto, non verranno presi in considerazione in sede di questa trattazione.

Ora, la totalità degli elementi *in stage* identifica un *fotogramma* dell'animazione, il quale è rappresentato nel DOM dal *Frame Object*. Uno o più fotogrammi sequenziali vengono contenuti in un *livello*, equivalente ad un *Layer Object* nel DOM. Uno o più livelli formano una *timeline*, ovvero un *Timeline Object*, contenuta necessariamente in un *documento* (*Document Object*) che è infine stato creato dall'oggetto *animate*. In altri termini, partendo dall'oggetto *animate*, questo non è nient'altro che un array di oggetti *Document*, rappresentante i FLA file attualmente aperti, ognuno dei quali rappresenta un array di oggetti *Timeline*, ognuno dei quali è un array di oggetti *Layer*, ognuno dei quali è un array di oggetti *Frame*, ognuno dei quali è un array di oggetti *Element*, ognuno dei quali può essere uno dei tipi sopra elencati. Quindi, esprimendo questo concetto in forma di codice, questo è quello che risulta per il primo elemento del primo fotogramma del primo livello nella timeline corrente del documento corrente:

```
var shape = an.getDocumentDOM().getTimeline().layers[0].frames[0].elements[0];
```

Finalmente si può adesso affrontare lo *Shape Object*; sorvoliamo gli oggetti *Rectangle* e *Oval*, di scarso interesse: queste sono infatti figure primitive definite parametricamente, in maniera praticamente identica ai *Rectangle Path* e *Oval Path* di After Effects. Come si può notare dallo schema, ci sono 4 strutture che concorrono alla definizione di una *Shape*, tutte legate tra loro: il *Contour Object,* l'*Edge Object,* l'*HalfEdge Object* e il *Vertex Object*. A queste poi si aggiungono due pseudo-strutture, così etichettate poiché non facenti parte formalmente del DOM: il primo è un oggetto ufficialmente senza tipo (è infatti formalmente un oggetto di tipo *Object*), risultato di un metodo dell'oggetto *Shape*; data però l'importanza di questo oggetto ai fini del progetto di questa ricerca, gli verrà dato il nome informale di *CubicPoint Object*. La seconda pseudo-struttura è anch'essa il risultato di un metodo, stavolta dell'oggetto *Edge*, che ha però la denominazione formale di *Point Object*.

Prima di entrare nello specifico della descrizione degli oggetti appena citati, è opportuno introdurre le due *modalità di disegno* delle *Shape* in Animate, le quali giocano un ruolo molto importante nella comprensione di queste strutture dati, nonché nel *conteggio* del numero di elementi di tipo *shape* immagazzinati nell'array *elements* dell'oggetto *Frame*: la *merge drawing mode* e la *object drawing mode*. È possibile distinguere in quale

modalità sia disegnato un oggetto *Shape* dal modo in cui questo viene visualizzato nell'interfaccia di Animate: se in *merge mode* si vedrà sovrapposto su di esso un motivo *a puntini*, se in *object mode* sarà possibile vedere i *punti di controllo* della rappresentazione della *Shape* come *curva di Bézier cubica* (Figura 2.12). Troviamo qui infatti il primo punto di contatto tra le due API: le *Shapes* possono essere rappresentate come curve di Bézier cubiche in entrambi i programmi. Ora, se rapportiamo queste due modalità di disegno alla dimensione dell'API, potrebbe venire in mente che uno *Shape Object* abbia al suo interno i dati per la sua rappresentazione come curva di Beziér se e solo se in modalità *object* ma non è così: le due modalità infatti non sono esclusive, ma *convivono* contemporaneamente nello *Shape Object* in ogni istante. Una *Shape* avrà quindi in ogni caso i dati per la sua rappresentazione come curva di Bézier cubica, anche se in *merge mode*.

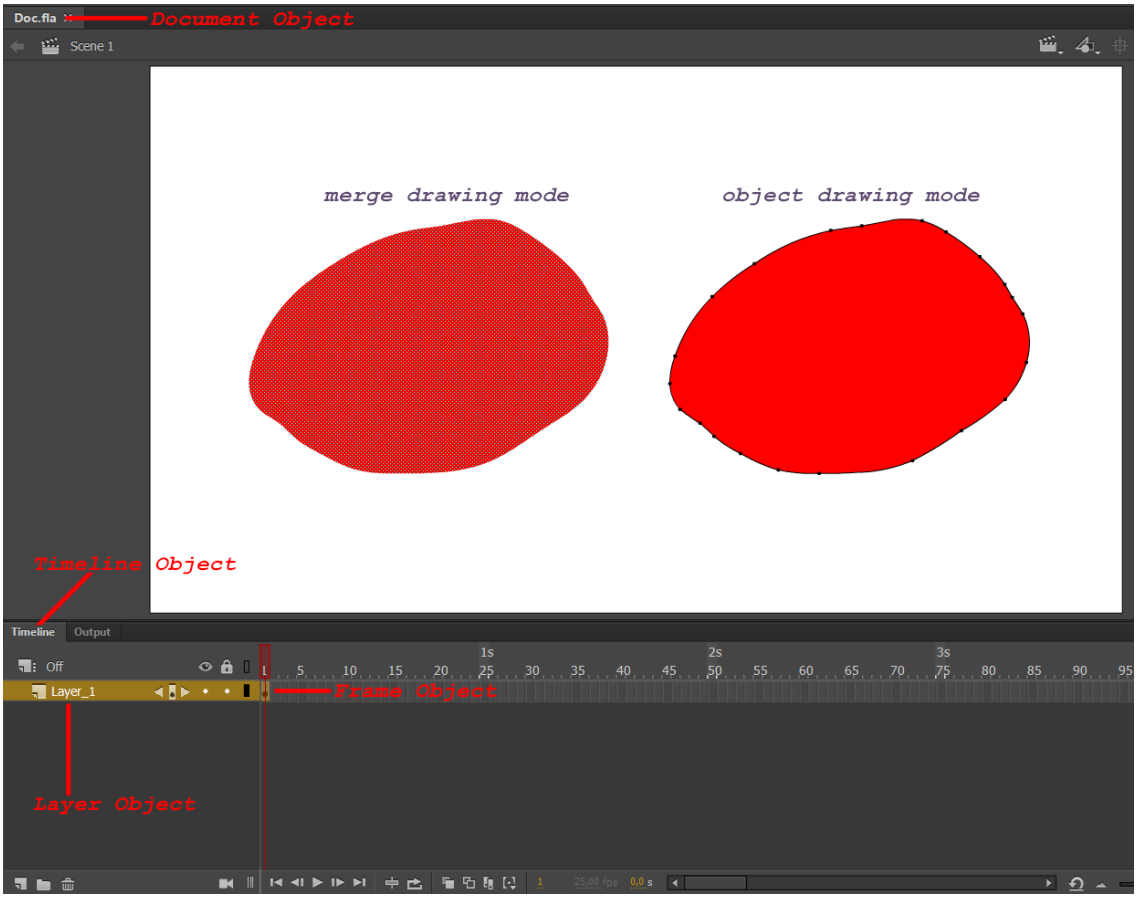

Figura 2.12 – differenza di visualizzazione delle modalità di disegno

La differenza nelle due modalità di disegno è infatti più a livello funzionale, ovvero nel modo in cui queste interagiscono col resto degli elementi *in stage*: la *merge mode* infatti è chiamata così proprio perché *fonde* (in inglese *merge*) tutte le forme disegnate in questa modalità, qualora queste si sovrapponessero; è quindi, in un certo modo, una modalità *distruttiva*, poiché di fatto le porzioni della *shape*, che in sovrapposizione ad un'altra sottostanno a quest'ultima, vengono *cancellate*; il risultato di questa operazione di cancellazione fa sì che tutte le *shapes* in questa modalità giacciano sullo stesso piano. In *object mode* questo non avviene, bensì le *shapes* mantengono una propria indipendenza l'una dall'altra; se sovrapposte in *object mode*, le *shapes* vengono impilate una sull'altra su più piani, senza così incappare alla cancellazione delle porzioni sovrapposte.

Possiamo ora tornare alla trattazione degli oggetti del DOM precedentemente introdotti, a partire dal più *piccolo*: il *Vertex Object*; questo è la parte della struttura dati delle *shapes* che contiene le informazioni delle coordinate e rappresenta, come suggerisce il nome, un *vertice* della geometria. Come si può notare dallo schema del DOM, l'oggetto *Vertex* è al livello più basso nella gerarchia, ed è alla base di tutte le altre strutture dati dello *Shape Object*. Tenendo presente quanto è stato detto sull'*object drawing mode*, e cioè sul fatto che in questa modalità le *shapes* vengano visualizzate come curve di Bézier *cubiche*, verrebbe da pensare che i vertici rappresentati dall'oggetto *Vertex* siano di natura *cubica.* Ora, quanto verrà detto in seguito sull'oggetto *Vertex* è di vitale importanza nell'ottica del progetto, in quanto, nonostante le premesse, si discosta grandemente da ExtendScript, imponendo quindi di prendere provvedimenti adeguati al mantenimento di una dimensione comune alle due API. Infatti, contro-intuitivamente rispetto quanto è possibile supporre dall'interfaccia del programma, l'intera struttura dati dell'oggetto *Shape*, e quindi i primi 4 oggetti sopracitati, si fondano su *curve di Beziér quadratiche* (ovvero *con 3 punti di controllo*), di cui il *Vertex Object* è componente fondamentale. Esso infatti rappresenta uno dei due punti agli estremi di un *segmento quadratico* sulla *shape* (viene escluso il punto intermedio). "Uno dei due" poiché in un segmento il vertice di testa può essere anche considerato come il vertice di coda del segmento precedente, così come il vertice di coda può essere considerato il vertice di testa del segmento successivo.

L'introduzione del concetto di *segmento* ci porta alla trattazione delle due strutture dati successive, l'*Edge Object* e l'*HalfEdge Object*, strettamente collegati. Anzi, letteralmente collegati, in quanto, come si può intuire dal nome, un oggetto *Edge* è l'unione di due oggetti *HalfEdge*. Fondamentalmente, entrambe queste strutture dati rappresentano un *segmento quadratico*, inteso come tratto d'unione dei due vertici estremi di una *curva quadratica*, ma a seconda del caso trasportano informazioni differenti. Per capire di cosa si sta parlando bisogna innanzitutto far presente che Animate adotta una struttura logica denominata *Doubly Connected Edge List data structure (DCEL)*, alternativamente chiamata, per l'appunto, *Half-Edge data structure*; in questa struttura logica ogni segmento (*edge*) è composto da due *mezzi-segmenti* (*half-edge*), rappresentanti i due *versi* del segmento e aventi direzioni opposte. In una curva chiusa, la direzione del mezzo-segmento dà uno dei due possibili sensi di percorrenza del contorno della geometria, orario o antiorario. Ogni *half-edge* è connessa contemporaneamente a quella successiva, precedente e opposta, quindi è sempre possibile navigare una *shape* in entrambi i sensi in qualsiasi momento. In Figura 2.13 viene fornita una rappresentazione grafica di quanto espresso finora.

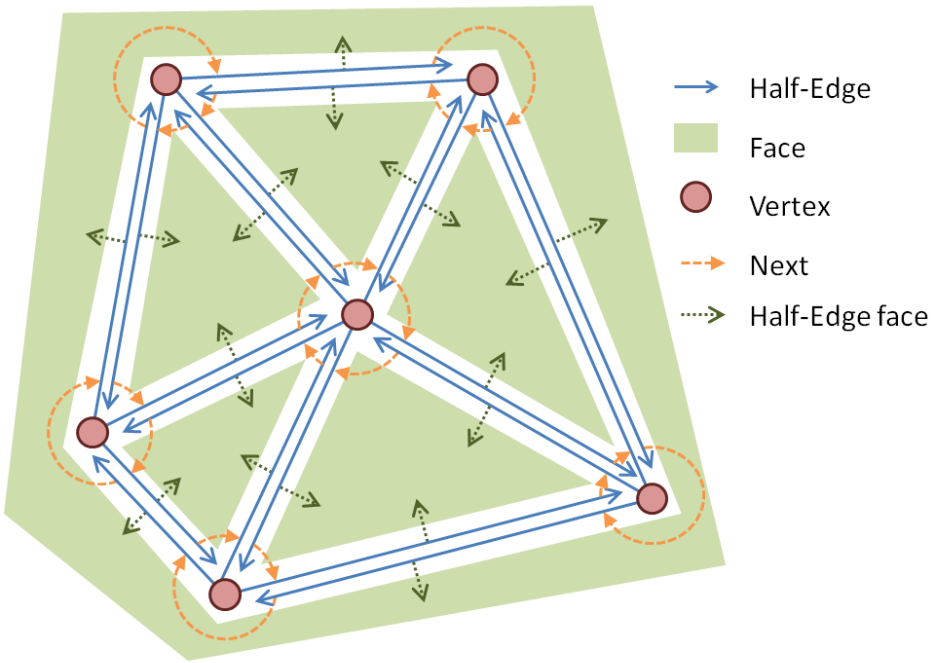

Figura 2.13 – Doubly Connected Edge List

Nella Figura 2.14 è invece rappresentata la connessione di un'*half-edge* col resto della geometria, con un'astrazione dei metodi che grosso modo si riscontrano in ogni programma che implementi la struttura *DCEL*, Animate compreso.

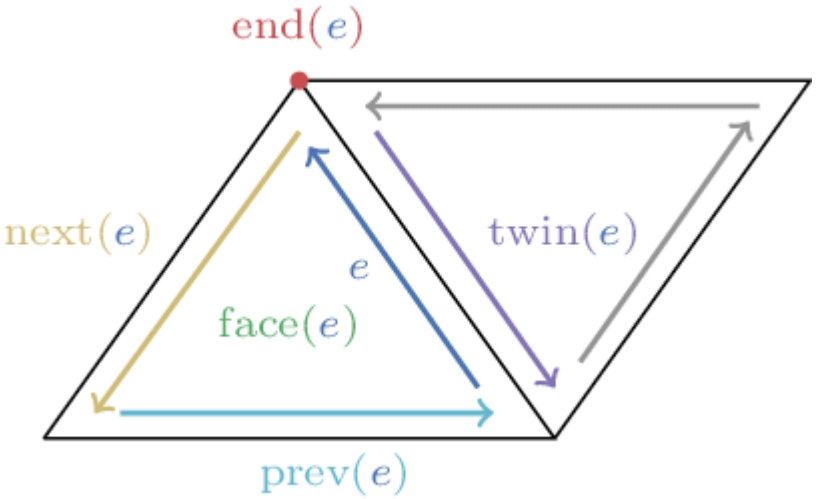

Figura 2.14 – relazioni di un "half-edge" nella struttura DCEL

Riassumendo, entrambe le strutture dati rappresentano, nella loro base logica, la congiunzione dei due vertici di una curva quadratica, ma come si è detto trasportano questa informazione in modo differente: l'*HalfEdge Object*, in virtù del fatto di trasportare intrinsecamente l'informazione riguardante il senso di percorrenza di un contorno, contiene *un* solo vertice del segmento, quello alla testa del mezzo-segmento; per ricavare l'altro vertice basterà recuperare il vertice contenuto nell'oggetto *HalfEdge* opposto, il cui vertice di testa rappresenta proprio il vertice di coda del mezzo-segmento attuale. L'*Edge Object*, poiché unione di due oggetti *HalfEdge gemelli*, non solo contiene

entrambi i vertici ma trasporta l'intera informazione sulla curva quadratica, potendo accedere a tutti e tre i punti di controllo: tramite un metodo dell'oggetto infatti, esprimendo un indice che può assumere i valori 0, 1 o 2, è possibile ricavare un oggetto di tipo *Point*, una delle due pseudo-strutture precedentemente introdotte, che a seconda dell'indice rappresenta il punto iniziale, intermedio o finale della curva di Bézier quadratica.

Il senso di percorrenza del *contorno* di una *shape* dato dalla direzione di un oggetto *HalfEdge* anticipata precedentemente introduce la quarta e ultima struttura dati formalmente riconosciuta dal DOM di JSFL, il *Contour Object*. Nella sua definizione più generale, un *contorno* rappresenta un *percorso chiuso di half-edges*. Attenzione però che questa definizione non sta a indicare una *curva chiusa*, bensì al fatto che la struttura logica DCEL porta all'inevitabile concatenazione dei mezzi-segmenti in percorsi chiusi; questo vuol dire che, partendo dal vertice di un mezzo-segmento e assecondandone il verso di percorrenza senza mai cambiare, si tornerà certamente al vertice di partenza sullo stesso mezzo-segmento, ma solo dopo aver navigato *tutta* la figura. In altri termini, si può dire che qualsiasi curva, aperta o chiusa che sia, consiste in un percorso chiuso di *half-edges*, ciò che cambia è il numero di contorni della curva: nel caso di una curva chiusa infatti sono possibili contemporaneamente due percorsi chiusi, uno per ogni senso di percorrenza, orario e antiorario, per un totale di due contorni, i quali si rivolgono rispettivamente all'area esterna e all'area interna della curva chiusa; in una curva aperta invece è possibile (e sufficiente) un unico percorso di *half-edges* per rappresentare la figura; il suddetto percorso non ha un vero e proprio senso di percorrenza e non si rivolge quindi ad alcuna area (interna o esterna che sia), condizione questa denominata *"no area"*.

Con la trattazione dell'oggetto *Contour* si esauriscono gli oggetti dedicati alla strutturata rappresentazione quadratica di una *shape*. È stato fatto notare però che questa non rappresenta un metodo sufficientemente adeguato ai fini del progetto, poiché in contrasto con l'utilizzo di curve di Bézier cubiche nell'API di After Effects. È su questo scenario che arriva in aiuto l'ultima struttura dati, nonché pseudo-struttura, introdotta all'inizio del paragrafo, quella da me nominata *CubicPoint Object*. Come è già stato detto, questo oggetto è in realtà il risultato di un metodo dell'oggetto *Shape*: esprimendo un indice indicante in *segmento cubico* della *shape*, si avrà come valore di ritorno un array di 4 oggetti *CubicPoint*, ovvero i 4 punti di controllo del segmento cubico. Questa capacità dell'oggetto *Shape* di poter ricavare i punti cubici, unità alla presenza di un attributo indicante il numero dei segmenti cubici della *shape*, farebbe pensare che sia sufficiente questa struttura dati per estrapolare le informazioni vettoriali più adatte ad essere inviate ad After Effects, rendendo le altre strutture dati superflue, ma così non è, o per meglio dire non lo è nella maggior parte dei casi. Arrivati a questo punto della trattazione bisogna infatti portare alla luce un difetto, se così si può chiamare, riguardanti le due modalità di disegno precedentemente affrontate; innanzitutto ricordiamo che un oggetto *Shape* è un'implementazione di un oggetto *Element*, quindi che la totalità degli elementi in un dato *frame* vengono immagazzinati in un array, l'attributo *elements* dell'oggetto *Frame*. L'ordine degli elementi nell'array non è casuale, ma rispecchia l'ordine di sovrapposizione dei *piani* degli elementi *in stage*. Ora ricordiamo quanto si è detto sulle modalità di sovrapposizione a seconda della modalità di disegno: in *merge mode* non

esiste una sovrapposizione vera e propria, in quanto tutte le *shape* in questa modalità giacciono sullo stesso piano. Alla luce di tutto ciò si può delineare il difetto di cui sopra ovvero che *tutte le shapes in merge mode sono considerate come un unico elemento*. Queste infatti, proprio perché per loro natura appartengono tutte ad un solo piano, sono tutte immagazzinate alla prima posizione dell'array *elements*. Questo vuol dire che, nel caso di numerose geometrie in *merge mode*, queste sarebbero considerate come un unico oggetto *Shape*, rendendo impossibile la distinzione degli oggetti *CubicPoint* dell'una o dell'altra geometria. Inoltre, il programma permette la conversione di più geometrie in *merge mode* in un unico *drawing object*, facendo sì che anche la modalità *object drawing* soffra in un certo modo di questo difetto. Gli unici casi in cui la sola struttura dati *Shape*, con relativi oggetti *CubicPoint*, risulta sufficiente alla descrizione univoca delle geometrie in forma cubica sono quelli in cui ogni singola *shape* rappresenta un *drawing object* indipendente, ammettendo al massimo un'unica geometria in *merge mode*; ma questo vorrebbe dire basarsi sul solo buon senso degli utenti, il che non è auspicabile per il corretto sviluppo del progetto.

Ora che anche l'oggetto *CubicPoint* è stato affrontato, viene presentata in Figura 2.15 la visualizzazione delle 3 strutture dati che concorrono nella definizione dei diversi tipi di punti e vertici. Da notare come il segmento, che in forma cubica consta di soli 4 punti, ne presenti molti di più del tipo *Vertex* e *Point*. Notare inoltre come un oggetto *Vertex* coincida sempre con un oggetto *Point*, a riconferma della loro identica natura di punti agli estremi di un segmento quadratico. Notare infine i punti di tipo *CubicPoint* e *Point* all'esterno del segmento, rispettivamente i punti di controllo centrali del segmento in forma cubica e dei segmenti in forma quadratica.

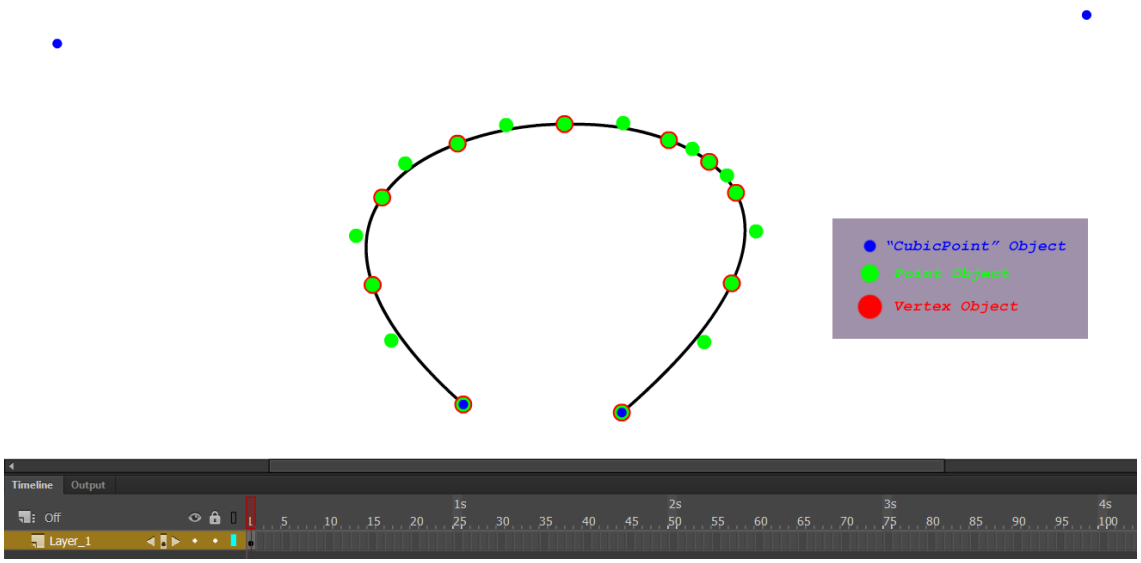

Figura 2.15 – rappresentazione delle 3 diverse strutture che concorrono nella definizione di una shape

Da tutto questo risulta chiaro che le strutture dati presentate siano tutte necessarie, ognuna a suo modo, alla corretta estrapolazione dei dati vettoriali, che ricordiamo dovranno essere nella forma di curva di Bézier cubica, affinché risultino adatti ad ExtendScript. A questo proposito però, va chiarito un ultimo dettaglio riguardo la disposizione dei dati

delle curve nelle due API: finora infatti si è parlato delle curve di Bézier cubiche in termini di *4 punti di controllo*; ciononostante, nella trattazione dell'API di After Effects si è detto che, data l'organizzazione dei dati della curva in tre array (*vertices, inTangents* e *outTangents*) ad ogni vertice corrisponde un punto *inTangent* e un punto *outTangent*, per un totale di *3 punti*; ebbene questa espressione dei dati, benché più semplice e intuitiva, non è formalmente la più corretta: ExtendScript riferisce la strutturazione dei dati in corrispondenza dei *vertici* della curva, mentre JSFL, maggiormente in linea con la rappresentazione teoricamente più corretta, la riferisce in corrispondenza del *segmento* della curva. I *4 punti di controllo* di una curva cubica infatti rappresentano *due vertici*, ovvero il primo e il quarto punto, e i punti *outTangent* (secondo punto) e *inTangent* (terzo punto) rispettivamente del primo e del quarto punto; ed è infatti in questa forma che vengono disposti i dati degli oggetti *CubicPoint*, ovvero in array di dimensione 4. Se volessimo quindi riferire i dati di una curva cubica in JSFL ad un singolo vertice come in ExtendScript, dovremmo prendere in considerazione *due segmenti adiacenti alla volta*: ad un vertice corrisponderebbe quindi come *outTangent* il secondo punto del proprio segmento e come *inTangent* il terzo punto del segmento precedente; va oltretutto fatto notare che, in questa disposizione, il primo punto del segmento attuale e il quarto del segmento precedente coincidono.

2.4 Gli strumenti

In conclusione di questo capitolo, viene data una panoramica degli strumenti, IDE e Text Editor, presi in considerazione per lo sviluppo del progetto.

Il primo strumento nel quale ci si imbatte inevitabilmente, poiché unico strumento ufficialmente riconosciuto da Adobe per lo sviluppo di estensioni, è *Extension Builder 3*: questi non è altro che un componente integrativo da installare su *Eclipse* per abilitare e facilitare la scrittura di codice JavaScript, HTML e CSS e accedere alle cartelle del CEP installate sulla macchina, rendendolo di fatto *un'estensione di Eclipse*. Extension Builder rappresenterebbe di fatto lo strumento migliore per lo sviluppo, offrendo, tra le altre cose, l'inizializzazione automatica dei file di progetto e il fatto di non necessitare l'abilitazione della *modalità debug* sulla macchina (vista più avanti); degno di nota è l'inizializzazione del *manifest file* (affrontato nel capitolo successivo) con il semplice impiego di un'interfaccia grafica (Figura 2.16).

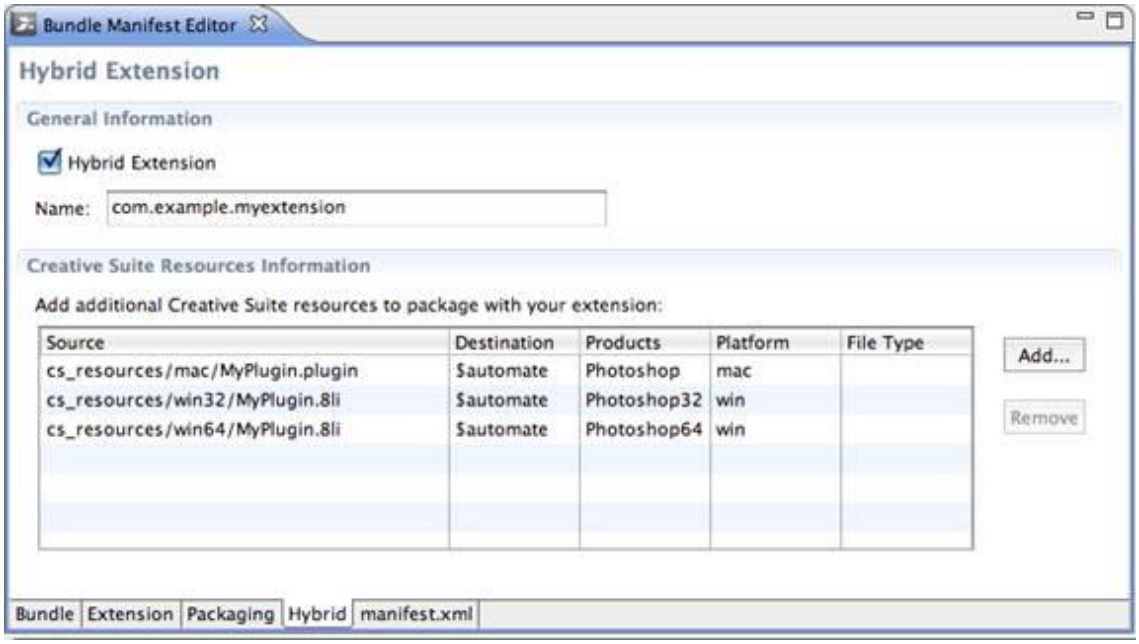

Figura 2.16 - Manifest editor

Purtroppo, Adobe non aggiorna più Extension Builder da parecchi anni ormai, rendendolo incompatibile, e quindi obsoleto, dalla sesta versione del CEP in poi, con grave perdita di un valido strumento di sviluppo.

Presa coscienza dell'ormai verificata inadeguatezza di Extension Builder, sono stati presi in considerazione *Brackets* e *Sublime Text*: questi sono entrambi Text Editors *open source*, il primo dei quali sviluppato dalla stessa Adobe; ciò che li rende dei validi candidati allo sviluppo è che anche in questo viene data la possibilità di installare dei plugin che automatizzino l'inizializzazione dei file di progetto di un'estensione.

In quanto Text Editors però, Brackets e Sublime Text non avevano lo stesso potenziale offerto da un IDE, ed è per questo che sono stati scartati in favore di *WebStorm*. WebStorm è un IDE rivolto allo sviluppo web, proprietario dell'azienda di sviluppo software JetBrains. Visti tutti i vantaggi derivanti dall'utilizzo di un IDE, e data anche la preesistente familiarità col programma, WebStorm è stato scelto come ambiente di sviluppo finale, nonostante la mancanza di un esplicito supporto per un progetto indirizzato ad una piattaforma molto specifica come il CEP, anche se è da tenere da conto la presenza del supporto per JSFL. La possibilità di scaricare uno *scheletro* di progetto dalla *repository* ufficiale Adobe per lo sviluppo CEP [16] ha sopperito alla mancanza di supporti da parte di JetBrains.

Va però detto che WebStorm è stato scelto come strumento di *finalizzazione* dell'intera estensione. Per la scrittura e il *testing* delle singole funzionalità nelle API dei due software *host* ci si è affidati agli strumenti ufficiali Adobe: per lo scripting in ambiente JSFL, Animate offre già al suo interno un IDE di facile impiego, scelto soprattutto per l'indubbio vantaggio di avere l'ambiente di sviluppo integrato nel software di utilizzo finale. Per ExtendScript ci si è invece affidati ad un *tool* esterno, sempre di proprietà Adobe, *ExtendScript Toolkit* che, nonostante essere un software esterno, è chiaramente il

più integrato con le API dei programmi Adobe, avendo oltretutto le capacità di *debugging* di più rapido utilizzo rispetto qualsiasi altro ambiente di sviluppo.

Capitolo 3

Lo Sviluppo

In questo capitolo si entrerà finalmente nel merito dello sviluppo di un'estensione CEP, riferendosi nello specifico, chiaramente, all'estensione oggetto del progetto pratico di questa tesi. Ad ogni modo si cercherà di descrivere i vari passaggi nel modo più accurato ed esaustivo possibile, con una prospettiva *manualistica* nelle definizioni di carattere più generale, così da poter offrire, forse, un più ampio respiro nei fini di questa ricerca (nonché il suo futuribile impiego in altri progetti).

3.1 Fasi preliminari: abilitazione del debug e *scheletro* di un'estensione CEP

Un'estensione normalmente è installata tramite un file *ZXP* (estensione *.zxp*). Questo è, in parole povere, un formato speciale di file archivio (come uno *zip*), ovviamente proprietario di Adobe, contenente, oltre che l'estensione, un *signing certificate* (letteralmente *certificato di firma*). La trattazione dei file ZXP e i *signing certificate* verranno trattati verso la fine di questo capitolo, nel paragrafo dedicato al *packaging* dell'estensione, ma basti sapere per il momento che è impossibile far avviare correttamente un'estensione senza il suddetto certificato. Siccome però quest'ultimo non viene approntato se non alla fine della creazione di un'estensione, bisogna chiaramente

trovare un'altra maniera per avviare un'estensione in piena fase di sviluppo. Bisogna abilitare la *modalità debug*: questa modalità, attivata direttamente sulla macchina (PC o Mac), permette di bypassare la necessità di un certificato in quanto permette di far *saltare* la verifica di validità della firma. Su Windows la modalità debug si abilita accedendo direttamente al registro di sistema utilizzando lo strumento *regedit* (Windows key + R: regedit) e navigando alla seguente locazione:

```
regedit > HKEY_CURRENT_USER/Software/Adobe/CSXS.8
```

dopodiché andrà aggiunto un nuovo valore di tipo *string* col nome *PlayerDebugMode* e assegnargli valore *1*.

Su Mac la modalità debug può essere abilitata da terminale digitando

```
defaults write com.adobe.CSXS.8 PlayerDebugMode 1
```

Da notare la sottolineatura del numero *8*: questo rappresenta il numero della versione CEP per la quale si sta abilitando la modalità debug, da sostituire all'occorrenza con la versione desiderata. Nel caso del numero 8 vuol dire che si sta abilitando un'estensione per la versione dei programmi *CC 2018*. Nel caso di Windows è possibile prendere nota di tutte le versioni disponibili (da 4 a 9) dal registro di sistema. È possibile prendere visione della corrispondenza tra versione del *Creative Cloud* e versione del CEP alla Figura 2.7 già presentata nel capitolo precedente.

Ora che la modalità debug è stata abilitata, è possibile iniziare a progettare l'estensione. Innanzitutto, va approntato il progetto, inteso come l'insieme di cartelle e sottocartelle che costituiranno lo *scheletro* dell'estensione. La struttura del progetto è in realtà molto flessibile, non esistendo un particolare disposizione obbligatoria dei file e delle cartelle, eccezion fatta per una sola cartella che deve essere necessariamente presente, la cartella *CSXS*: questa è la cartella che ospiterà il file *manifest.xml*, anch'esso obbligatorio. In Figura 3.1 viene presentata una divisione di file e cartelle di progetto che si potrebbe definire in un certo senso *ottimale*; in ogni caso, si ribadisce, questa non costituisce carattere obbligatorio, se non per la presenza della cartella CSXS.

Figura 3.1 – divisione ideale delle cartelle in un progetto CEP

In questa divisione si può notare la separazione delle due *Virtual Machine* (*Panel* e *Tool*) già citata nel capitolo precedente. Ma passiamo in rassegna le tre cartelle:

- **CSXS**: questa è la cartella obbligatoria per qualsiasi estensione. In essa non è contenuto nient'altro che il *manifest.xml*, la cui funzione verrà trattata nel paragrafo successivo. Si coglie inoltre qui l'occasione di spiegare il perché della nominazione *"CSXS"*, già incontrata precedentemente durante l'abilitazione della modalità debug: questa è la sigla per *Creative Suite eXtensible Services* ed è un residuato della precedente iterazione di tutta la struttura al quale il CEP si è oggi sostituito. Il mantenimento di questo nome è dovuto appunto al fatto che il CEP sia stato costruito *sopra* il CSXS.
- **Client**: questa è la cartella per il codice *front-end*, ovvero tutti i file per l'interfaccia grafica nonché suo funzionamento; è in altre parole la cartella che verrà eseguita dal *Panel VM*. È qui inoltre che vengono posizionati i due file JavaScript usati per accedere alle API del CEP, *CSInterface* e *Vulcan* (quest'ultimo non presente in Figura 3.1).
- **Host**: la cartella delle funzionalità *core* dell'estensione, scritte nei linguaggi proprietari Adobe ExtendScript e JSFL (quest'ultimo non presente in Figura 3.1). Anche in questo caso si può altresì interpretare questa divisione in virtù del motore che ne eseguirà il codice, ovvero il *Tool VM*.

Si può senza difficoltà predisporre manualmente un progetto seguendo il modello presentato qui sopra, o alternativamente scaricare uno *scheletro* già inizializzato dalla già citata *repository* ufficiale del CEP o dalla grande quantità di *repositories* d'esempio create dalla *community*. In particolare, è da citare il *template* d'esempio [17] offerto dal *developer* di *Adobe Japan* Andy Hall, il cui blog [18] si è rivelato una risorsa fondamentale per la riuscita di questa tesi.

Poiché l'estensione che ci si appresta a sviluppare non è stata installata tramite un file ZXP (il quale, tramite un apposito *installer*, posizionerebbe automaticamente il progetto nella giusta locazione di sistema), le suddette cartelle andranno posizionate manualmente nell' *extensions folder* della macchina: installandole in questa cartella, l'estensione apparirà nella lista di estensioni disponibili per il programma Adobe desiderato (se il *manifest* dell'estensione in questione ne prevede l'abilitazione per un programma specifico, ma ci arriveremo nel paragrafo seguente). In realtà esistono due *extensions folder* disponibili, in quanto è possibile installare un'estensione a livello *user*, abilitando così un singolo utente del sistema, o a livello *system*, abilitando così l'intero sistema. Su Windows i percorsi specifici per queste due cartelle sono:

- `User-level:`
  `C:\Users\<USERNAME>\AppData\Roaming\Adobe\CEP\extensions`
- `System-level:`
  `C:\Program Files (x86)\Common Files\Adobe\CEP\extensions`

Su Mac:

- `User-level:`
  `~/Library/Application Support/Adobe/CEP/extensions`
- `System-level:`

```
/Library/Application Support/Adobe/CEP/extensions
```

Per riassumere quanto discusso nel paragrafo, viene data un'immagine esplicativa in Figura 3.2 per percorso e struttura (alternativa rispetto al modello precedentemente presentato per ribadirne la flessibilità) di un'estensione *user-level* d'esempio, predisposta tramite il *template* di Andy Hall.

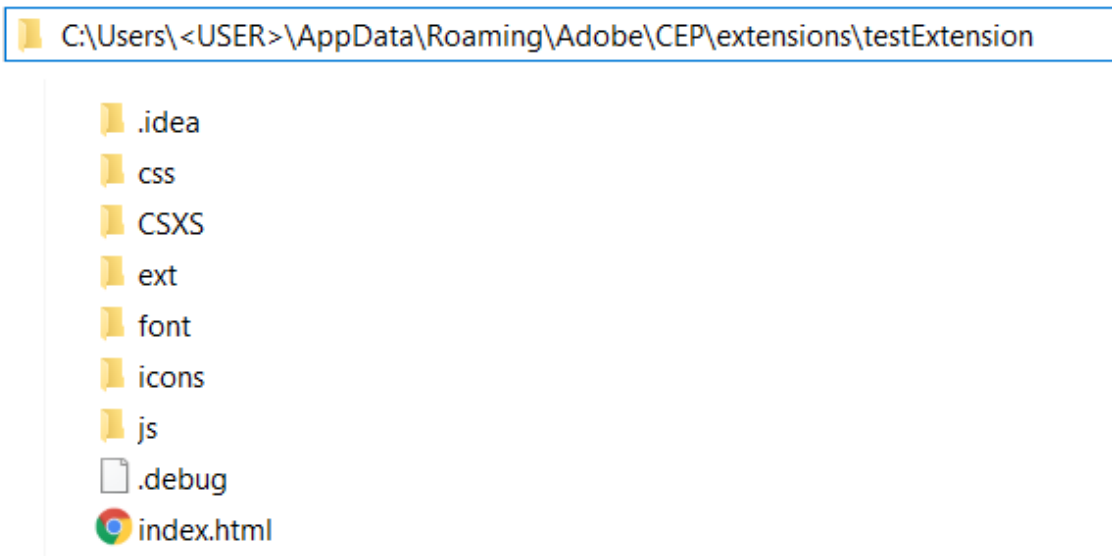

Figura 3.2 – percorso e struttura di un'estensione CEP

In questa struttura la cartella *ext* corrisponde alla cartella *host* del modello precedente. Contrariamente a quanto fatto con la cartella *client* del modello, non sono stati invece unificati in un'unica locazione i componenti *front-end*, ovvero CSS (cartella *css*), HTML (*index.html*), JavaScript (cartella *js*, contenente anche i due file per le API CEP) e gli *asset* per i *font* e le *icone*. Di particolare interesse è invece il file *.debug*, in quanto abilita il *debug remoto*, vale a dire la possibilità di *debuggare l'estensione esattamente come se fosse una web app tramite una console Chromium*. Quest'ultima è in tutto e per tutto un equivalente dello strumento *inspector* di Chrome, tant'è che è possibile accedervi proprio tramite il browser. Innanzitutto, si prenda visione della struttura di un file *.debug* generico:

```
<?xml version="1.0" encoding="UTF-8"?>
<ExtensionList>
	<Extension Id="com.example.testextension.panel">
		<HostList>
			<Host Name="PHXS" Port="8080"/>
			<Host Name="ILST" Port="8080"/>
			<Host Name="FLPR" Port="8080"/>
			<Host Name="AEFT" Port="8080"/>
			<Host Name="PPRO" Port="8080"/>
			<Host Name="PRLD" Port="8080"/>
			<Host Name="IDSN" Port="8080"/>
			<Host Name="AICY" Port="8080"/>
		</HostList>
	</Extension>
</ExtensionList>
```

Come si può vedere, un file *.debug* consiste essenzialmente in una lista di *host tools* (i programmi Adobe), ognuno identificato con il proprio *id*, il cui elenco è stato presentato nel capitolo precedente (Figura 2.7). Ad ogni *host* viene poi associata una *porta* tramite la quale accedere al debugger; può essere utilizzata un'unica porta per più applicazioni. Una volta predisposto il file *.debug*, sarà sufficiente avviare l'estensione da debuggare (eventualmente riavviando il programma nel caso fosse già aperto mentre si scriveva il file); per accedere alla console si dovrà semplicemente avviare una finestra Chrome e digitare, nella barra di ricerca, `localhost`, specificando la porta inserita nel file, in questo caso `localhost:8080`. Questo è ciò che si presenterà affiancando la finestra del programma (Animate nell'esempio) con la finestra della console:

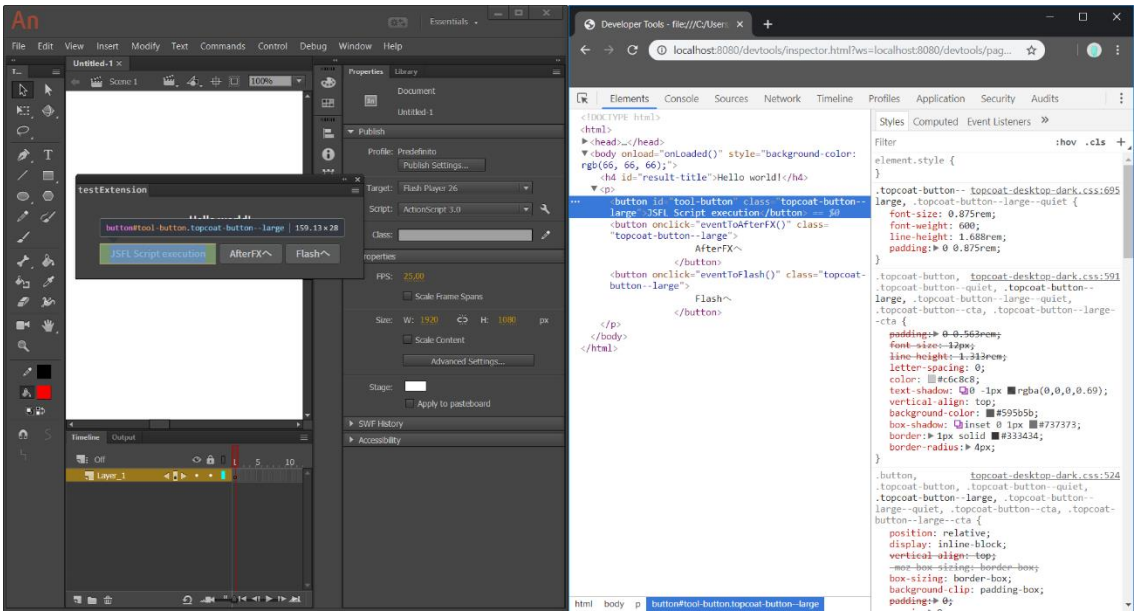

Figura 3.3 – debugging HTML di un'estensione CEP, con affiancamento delle finestre di Animate e inspector di Chrome

È innegabile la similarità di questa configurazione con quanto si presenti nel debug HTML di una qualsiasi web app.

Come si può intuire dalla struttura del file *.debug* inoltre, è possibile definire un unico file per più estensioni (un *bundle*, specificato nel prossimo paragrafo), fatto testimoniato dal tag `<ExtensionList>`. Infine, in conclusione dell'analisi sul *debug remoto*, nonché dell'intero paragrafo, va specificata l'importanza di far coincidere il tag `<Extension Id>` con l'id specificato nel file *manifest*, che altrimenti impedirebbe il corretto avvio della console. Ed essendo ormai necessaria una dovuta spiegazione sull'argomento, passiamo dunque alla trattazione del *manifest*.

3.2 Manifest

I file *manifest* generalmente sono file contenenti i metadati di un'applicazione. Normalmente sono scritti con un linguaggio di markup, il più comune dei quali è XML. Il *manifest* di un'estensione CEP è esattamente questo: un file XML contenente le informazioni di configurazione dell'estensione, necessari per la sua corretta integrazione

nel programma in fase di avviamento. Ad ogni avvio di programma infatti, il CEP verifica in tutte le cartelle presenti nel *extensions folder* innanzitutto la presenza della cartella CSXS. Se presente, ne analizza il *manifest* per controllare in quale applicazione è possibile integrare l'estensione e verifica la validità delle restanti informazioni, prima tra tutte l'*id dell'estensione*. Errori o incongruenze nel file *manifest* possono comportare errori nell'avviamento dell'estensione o la sua totale assenza nel programma. Ma andiamo a descrivere nel dettaglio un file *manifest* analizzando le sue componenti più importanti. Un *manifest* generico avrà una struttura simile:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ExtensionManifest
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  ExtensionBundleId="com.example.testextension"
                  ExtensionBundleName="testExtension"
                  ExtensionBundleVersion="0.1.0"
                  Version="8.0">
    <Author>Sam Name</Author>
    <ExtensionList>
        <Extension Id="com.example.testextension.panel"
                  Version="0.1"/>
    </ExtensionList>
    <ExecutionEnvironment>
        <HostList>
            <Host Name="FLPR" Version="14.0"/>
            <Host Name="AEFT" Version="13.0"/>
        </HostList>
        <LocaleList>
            <Locale Code="All"/>
        </LocaleList>
        <RequiredRuntimeList>
            <RequiredRuntime Name="CSXS" Version="5.0"/>
        </RequiredRuntimeList>
    </ExecutionEnvironment>
    <DispatchInfoList>
        <Extension Id="com.example.testextension.panel">
            <DispatchInfo>
                <Resources>
                    <MainPath>./index.html</MainPath>
       <!--ALTERNATIVE 1 FOR JSFL ONLY-->
                    <ScriptPath>./ext/tool.jsfl</ScriptPath>
       <!--ALTERNATIVE 2 FOR JSX ONLY-->
                    <ScriptPath>./ext/tool.jsx</ScriptPath>
                </Resources>
                <UI>
                    <Type>Panel</Type>
                    <Menu>testExtension</Menu>
                    <Geometry>
                        <Size>
                            <Height>110</Height>
                            <Width>420</Width>
                        </Size>
                        <MaxSize>
                            <Height>110</Height>
                            <Width>500</Width>
                        </MaxSize>
                        <MinSize>
                            <Height>110</Height>
                            <Width>390</Width>
                        </MinSize>
```

```
                    </Geometry>
                    <Icons>
                        <Icon Type="Normal"        >./icons/icon_light.png</Icon>
                        <Icon Type="RollOver"   >./icons/icon_light_over.png</Icon>
                        <Icon Type="DarkNormal"     >./icons/icon_dark.png</Icon>
                        <Icon Type="DarkRollOver">./icons/icon_dark_over.png</Icon>
                    </Icons>
                </UI>
            </DispatchInfo>
        </Extension>
    </DispatchInfoList>
</ExtensionManifest>
```

La prima cosa da far notare è che in realtà un *manifest* descrive un *bundle* di estensioni, ovvero una raccolta che può includere più estensioni. Il tag `<ExtensionManifest>` è la radice di tutto il *manifest*, l'elemento che contiene tutte le altre informazioni; al suo interno sono indicate le informazioni del *bundle*:

- `ExtensionBundleId:` un *id* unico per identificare il *bundle* espresso nella forma *com.my.extension.bundle.id*. L'unicità degli *id* è di vitale importanza poiché nel caso in cui più *bundle* condividessero lo stesso *id* si incapperebbe nell'impossibilità di avviarli.
- `ExtensionBundleName:` il nome del *bundle*
- `ExtensionBundleVerison:` l'attuale versione del *bundle* (non esiste una regola fissa nella numerazione ma tipicamente le versioni in *beta testing* sono numerate nella forma *0.x*. La prima *release* ufficiale è quindi a partire da *1.0*)
- `Version:` la versione del CEP utilizzata, in questo caso il CEP 8

Si passa poi all'elenco delle singole estensioni facenti parte del *bundle* col tag `<ExtensionList>`: qui ogni estensione è dichiarata con un proprio *id* e numero versione, spesso nella forma *bundleId* + *.panel* ma di nuovo non esiste una regola fissa, anche se il *bundleId* è usato sempre come prefisso.

Successivamente vengono dichiarati gli ambienti di esecuzione, ovvero le app autorizzate ad integrare il *bundle* di estensioni in fase di avvio. Gli ambienti vengono espressi nel tag `<ExecutionEnvironment>` all'interno di una `<HostList>:` la lista è espressa, similmente al file *.debug*, in un tag `<Host>`, indicando l'*id* dell'app e la versione, nei campi *Name* e *Version*. È possibile esprimere inoltre un *intervallo di versioni delle app* in cui far funzionare l'estensione, esprimendolo nella forma *[n, m]*. Se espressa come singolo numero, l'estensione funzionerà da quella versione dell'app in avanti. Il tag `<RequiredRuntime>` nell'omonima lista invece indica nuovamente le versioni del CEP autorizzate a gestire l'estensione.

L'ultimo elemento è forse il più importate, poiché detiene le informazioni *operative* del *bundle*, ovvero i dati da *spedire* (in inglese *dispatch*) al CEP affinché costruisca l'interfaccia; viene quindi definito un elemento `<DispatchInfo>` per ogni estensione del *bundle* in un'omonima lista; ognuno di questi elementi ne contiene essenzialmente altri due:

- `<Resources>:` vengono qui immagazzinate le *risorse* dell'estensione, generalmente il file HTML che fungerà da interfaccia principale; inoltre è buona regola dichiarare qui gli script per le *core functionalities* delle API dei programmi per il quale si sta attivando l'estensione. Da notare però i due commenti sovrastanti gli `<ScriptPath>`; i due tag infatti rappresentano due *alternative*, benché la dichiarazione di entrambi non comporti alcun tipo di errore nel caricamento dell'estensione, ma provocherà spiacevoli imprevisti qualora si tentasse di richiamare una funzione dichiarata nel file presentato come *secondo*: il CEP prenderà infatti in considerazione solo il *primo* file dichiarato in questa maniera. Il modo per utilizzare due o più file (indipendentemente JSFL o JSX) verrà affrontato in un paragrafo successivo.
- `<UI>:` nell'elemento *UI* viene definito il *tipo* di estensione (`<Type>`), il *nome* da visualizzare sull'interfaccia (`<Menu>`), e le *dimensioni* da far assumere alla finestra (`<Geometry>`), potendo anche specificare l'estensione minima (`<MinSize>`) e massima (`<MaxSize>`) consentita. Quattro sono i tipi possibili di un'estensione: *Panel*, il tipo più comune, si comporta come qualsiasi altro pannello dell'applicazione, può essere sganciato e agganciato in qualsiasi punto dell'interfaccia del programma. *ModalDialog*, apre una finestra di *dialog* e costringe l'utente ad interagire con essa, facendo assumere priorità all'estensione rispetto al programma *host*; la priorità torna al programma solo alla conclusione dell'interazione con l'estensione o alla sua chiusura. *Modeless*, simile alla precedente, ma non costringe l'interazione con la finestra di *dialog*. *Custom*, per *estensioni invisibili*; queste non verranno trattate durante il corso di questa ricerca, ma basti sapere che, come suggerisce il nome, un'estensione invisibile rimane nascosta all'utente durante l'intero suo ciclo di vita.

Ora che l'estensione comprende un *manifest*, verrà sicuramente accettata dal CEP e apparirà nella lista *Extensions* delle applicazioni per le quali è stata registrata.

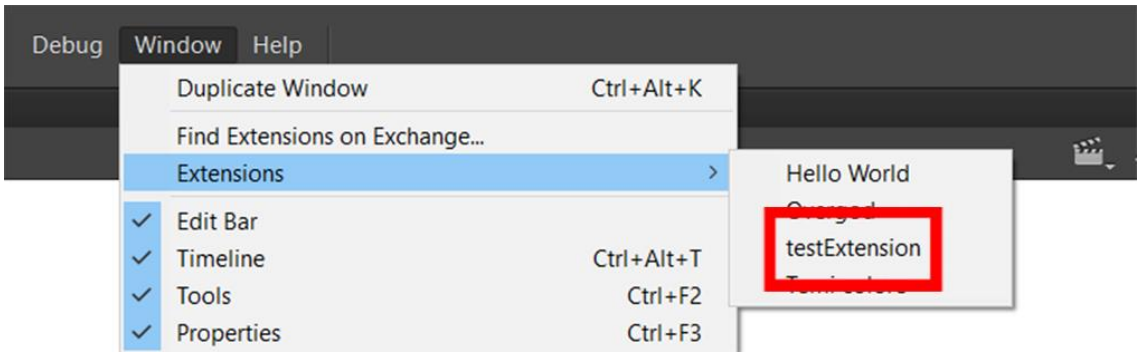

Figura 3.4 – lista Extensions in un programma Adobe (Animate)

Ora che le fasi preliminari sono concluse, si può quindi procedere allo sviluppo vero e proprio, a partire dall'*interfaccia grafica* dell'estensione.

3.3 Interfaccia Grafica di un'estensione: HTML e CSS

Si vuole ribadire qui ciò che è già stato ampiamente introdotto nel capitolo precedente: visto l'impiego del CEF per il rendering degli elementi HTML e dei fogli di stile CSS, il processo di sviluppo per una *UI* di un'estensione CEP è esattamente identico a qualsiasi altro progetto per il web.

Le possibilità di personalizzazione dell'aspetto grafico offerte dal CEP vanno ben oltre quelle offerte dal semplice impiego dell'API del programma (come ad esempio gli *ScriptUI*, molto limitati nell'aspetto grafico), potendo creare estensioni che quasi non hanno niente a che fare con l'interfaccia del software di appartenenza (Figura 3.5), fino a poter addirittura includere intere pagine web preesistenti che possano interagire con l'applicazione (Figura 3.6).

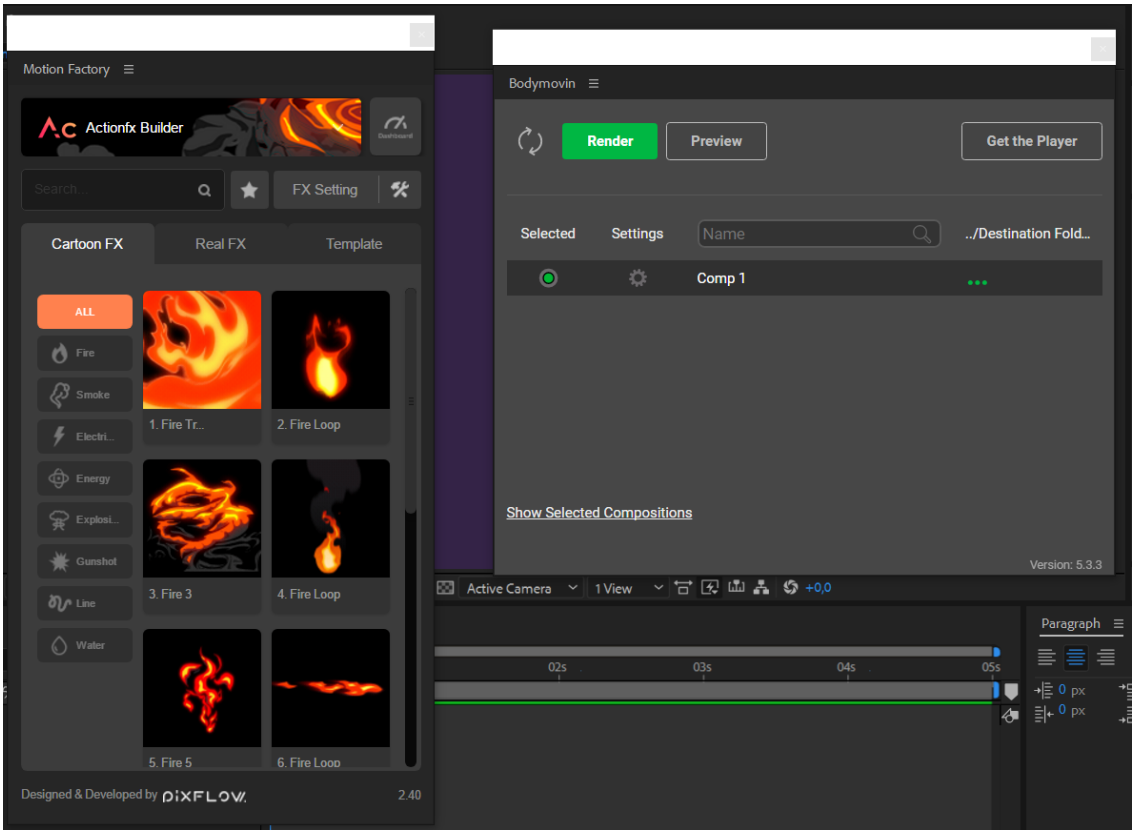

Figura 3.5 – le estensioni "Motion Factory" e "Bodymovin" per After Effects offrono delle GUI notevolmente originali, che ben si discostano dall'interfaccia di After Effects stesso.

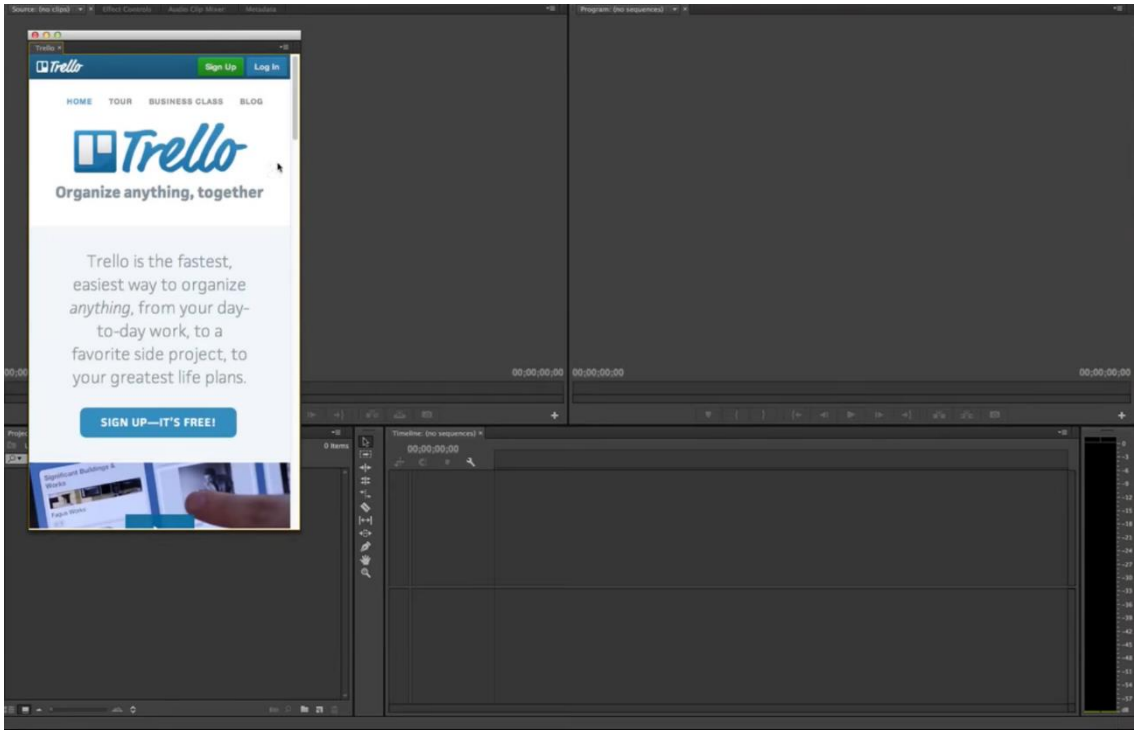

Figura 3.6 – l'homepage di Trello inclusa in un'estensione per Premiere Pro.

Nonostante questo immenso potenziale creativo, per il progetto di questa tesi si è deciso un approccio estremamente minimalista per la strutturazione dell'interfaccia, almeno per questa prima sua versione: un unico bottone. Dietro questa scelta si cela la volontà di incoraggiare un utilizzo dell'estensione molto più *dinamico*, proponendo un flusso di lavoro *fluido* e in continuo movimento: sarà l'estensione stessa a suggerire il proprio utilizzo all'utente, il quale, in mancanza di altri componenti visivi, dovrà solo *preoccuparsi di cliccare*.

Viene quindi qui presentato il file *index.html* dell'estensione, il cui tag `<head>` presenta comunque interesse di carattere genarle nella sua disposizione:

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>myExtension</title>
        <link rel="stylesheet" type="text/css" href="css/topcoat-desktop-
dark.css" id="topcoat-style" >
        <link rel="stylesheet" type="text/css" href="css/main-dark.css" id="main-
style" >
        <script src="js/CSInterface-4.0.0.js"></script>
        <script src="js/Vulcan-5.0.0.js"></script>
        <script src="js/main.js"></script>
    </head>
    <body onload="onLoaded()">
        <p>
            <button id="tool-button" class="topcoat-button--large">
                Script execution
            </button>
```

```
        </p>

    </body>
</html>
```

Il suddetto file genera quindi la seguente interfaccia:

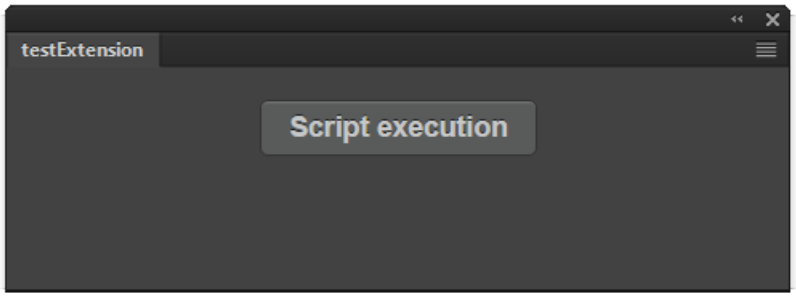

Figura 3.7 – rendering dell'interfaccia HTML di un'estensione CEP

La prima considerazione sull'tag `<head>` è sui tag `<link>`: questi sono i riferimenti ai fogli di stile che il file HTML impiegherà nella costruzione dell'interfaccia; vista l'intenzionale semplicità della *UI* dell'estensione, sono stati lasciati i CSS presenti nel *template* del progetto di partenza. La seconda considerazione è sui tag `<script>`: come si ricorderà, nel file *manifest.xml* sono stati dichiarati gli scripts per le funzionalità *core* dell'estensione; nell'*index.html* invece vengono dichiarati gli script (in linguaggio JavaScript puro) impiegati dal CEF per la gestione dell'interfaccia. Da far notare è la fondamentale presenza dei due file *CSInterface.js* e *Vulcan.js*, necessari per l'utilizzo delle API del CEP.

Qui di seguito è presentato il codice JavaScript del file *main.js* dichiarato nell'*index.html*, vale a dire il codice principale ordinato ad amministrare il pannello; nello specifico questo è il codice utilizzato per l'impostazione dell'aspetto grafico e l'inizializzazione delle funzioni, ovvero la funzione `onLoaded()` nel `<body>` dell'HTML.

```
var cs = new CSInterface();
var app = cs.hostEnvironment.appName;

function onLoaded() {
    // Initialize theme
    setAppTheme(null);
    // Register for events when the theme has changed
    cs.addEventListener( CSInterface.THEME_COLOR_CHANGED_EVENT, setAppTheme );
    // Initial of button to execute tool side script
    initToolScript();
    initCrossToolEvents();
}


function setAppTheme(e) {
    // In this example, the event object (e) is unnecessary.
    // Get theme information from CSInterface:
    var hostEnv = window.__adobe_cep__.getHostEnvironment();
    var skinInfo = JSON.parse(hostEnv).appSkinInfo;
    var color = skinInfo.panelBackgroundColor.color;
    var avg = (color.red+color.blue+color.green) / 3;
```

```
    // Apply CSS depending on whether bg color average exceeds 128:
    var type = (avg > 128) ? "light" : "dark";
    document.getElementById("topcoat-style").href = "css/topcoat-desktop-
" + type + ".css";
    document.getElementById("main-style").href = "css/main-" + type + ".css";
    // Fit panel bg color to tool:
    var rgb = "rgb(" +
        Math.round(color.red)   + "," +
        Math.round(color.green) + "," +
        Math.round(color.blue)  + ")";
    document.body.style.backgroundColor = rgb;
}
```

`initToolScript()` e `initCrossToolEvents()` sono i metodi dedicati alla sopracitata inizializzazione delle funzioni dell'estensione, ma verranno affrontati nel prossimo paragrafo, dedicato alla comunicazione tra *Virtual Machines* e alla comunicazione inter-app.

3.4 Comunicazione interna ed esterna all'estensione: *CSinterface* e *Vulcan*

In questo paragrafo verrà analizzato il resto del file *main.js* introdotto nel precedentemente. Nello specifico si reintrodurranno le due API *CSInterface* e *Vulcan* già ampiamente discusse, analizzando l'impiego che ne viene fatto in questo progetto.

Prima di iniziare è opportuno però tenere a mente un assunto fondamentale della comunicazione, sia interna che esterna, in un'estensione: questa è veicolata sempre e comunque da *stringhe di testo*. I valori che verranno scambiati nelle varie fasi della comunicazione saranno quindi sempre delle variabili di tipo *string*. Inoltre, anticipando quanto verrà detto nel paragrafo dedicato alle *core functions*, queste stringhe non sono altro che i dati vettoriali estratti da Animate, adattati in formato *JSON* tramite una funzione *stringify*.

Senza indugio, iniziamo dunque ad analizzare la funzione `initToolScript()`:

```
function initToolScript() {

    var but = document.getElementById('tool-button');

    var extPath = cs.getSystemPath(SystemPath.EXTENSION);
    if (app=="FLPR") {
        var jsflPath = extPath + "/ext/tool.jsfl";
        jsflPath = encodeURI( "file:///" + jsflPath );
        cs.evalScript( 'fl.runScript( "' +jsflPath+ '")' );
    } else {
        var jsxPath = extPath + "/ext/tool.jsx";
        cs.evalScript( '$.evalFile( "' +jsxPath+ '")' );
    }

    if (app=="FLPR") {
        but.onclick = function () {
            if(!VulcanInterface.isAppInstalled("aftereffects")) {
                cs.evalScript( 'alert("Ae is not installed!")' );
                return;
            }
            if(!VulcanInterface.isAppRunning("aftereffects")) {
```

```
                VulcanInterface.launchApp("aftereffects", true);
                return;
            }
            cs.evalScript('AnExtractData()', function(result) {
                sendToAe(result);
            })
        }
    }
}
```

Innanzitutto, si prenda in esame la prima metà della funzione e si ricordi quanto spiegato nel paragrafo precedente sul tag `<ScriptPath>` del file *manifest*: il CEP prende in considerazione solo la *prima* dichiarazione di un file script dal *manifest*. Questo metodo è quindi adatto per progetti che prevedano un unico file (JSFL o JSX) contenente tutte le funzioni *core*. In un progetto come quello qui presentato però è necessaria la presenza di almeno *due* file, vista la presenza di due API aventi linguaggi diversi. Le prime righe di codice della funzione `initToolScript()` sopperiscono a questa problematica; in generale questo metodo rappresenta un'alternativa alla dichiarazione di un (unico) file di script nel *manifest*. Ma vediamolo nel dettaglio: la chiave di tutto il codice sta nel metodo `evalScript` della variabile *cs*, che altro non è che un'istanza della *CSInterface* precedentemente dichiarata. La descrizione formale del metodo `evalScript` è quella di *valutare* (in inglese *evaluate*) uno script JavaScript che può utilizzare il DOM di un'applicazione *host*. Volendo astrarre il significato di questa definizione, ciò che in pratica fa il metodo `evalScript` è *consegnare* alla VM dell'applicazione *host* uno script nel proprio linguaggio. Oppure, come nel caso presentato, dei file: entrambe le API *host* hanno infatti a loro volta delle funzioni *eval*, rispettivamente `fl.runScript()` per Animate e `$.evalFile()` per After Effects; questi metodi vengono utilizzati per eseguire codice JSFL/JSX caricandolo da un file esterno; concatenando il metodo `evalScript` col metodo *eval* dell'API corrispondente è quindi possibile *caricare codice JSFL/JSX da un file esterno attraverso la CSInterface*. Per amor di chiarezza, ecco riassunti i 3 metodi con relative sintassi e precisazioni:

- `cs.evalScript(script [, callback])`: metodo della *CSInterface* chiamato dal *Panel VM*, passa una *stringa* di codice all'*Host VM*. Il valore *script* è appunto la stringa contenente il codice; può inoltre essere definita una funzione *callback* da eseguire successivamente all'esecuzione dello script valutato; eventualmente, può anche essere assegnato come parametro della funzione *callback* il risultato dello *script* (se esiste);
- `fl.runScript(scriptPath)`: metodo dell'API JSFL (quindi interno all'*Host VM*), prende come parametro un *percorso* in cui trovare il file JSFL da valutare;
- `$.evalFile(scriptPath)`: metodo dell'API ExtendScript (quindi interno all'*Host VM*), prende come parametro un *percorso* in cui trovare il file JSX da valutare;

Ora che i due file di script sono stati caricati, è possibile richiamarne le funzioni dinamicamente semplicemente dichiarandone il nome in un `evalScript`, che è esattamente ciò che succede nella seconda metà della funzione `initToolScript()`: dopo aver verificato di trovarsi nell'*host* corretto tramite l'identificatore *app* (dichiarato precedentemente) viene assegnata una funzione *onclick* all'unico bottone; questa, prima di valutare la funzione

del file JSFL, esegue un'ulteriore verifica sullo stato di After Effects utilizzando due metodi offerti da *Vulcan*: nel primo caso viene verificata l'installazione dell'app, avvertendo eventualmente l'utente della sua mancanza sulla macchina; nel secondo caso verifica se l'app è attualmente in uso e se in caso contrario provvede a *lanciarla*; da notare la differente annotazione degli identificatori utilizzati da *Vulcan* rispetto a quelli di *CSInterface* (per esempio nel caso di After Effects rispettivamente *aftereffects* e *AEFT*); ad ogni modo, al verificarsi di uno dei due casi, la funzione si interrompe, altrimenti richiama la funzione `AnExtractData()` dal file JSFL precedentemente caricato tramite `evalScript`; infine nella funzione *callback* di quest'ultimo `evalScript`, in cui ricordiamo viene passato come parametro il risultato della funzione appena valutata, viene chiamata la funzione `sendToAe` per inviare ad After Effects i dati appena estratti da Animate.

Prima di approfondire quest'ultima funzione, è necessario analizzare l'altra funzione d'inizializzazione precedentemente introdotta, ovvero `initCrossToolEvents()`, essenziale per comprendere il funzionamento `di sendToAe`. Ecco quindi il contenuto della funzione:

```
function initCrossToolEvents() {
    var type = "afterfx.msg";
    VulcanInterface.addMessageListener(
        VulcanMessage.TYPE_PREFIX + type,
        function(message) {
            if(app=="AEFT") {
                var data = VulcanInterface.getPayload(message);
                cs.evalScript('AeHandleData(' + data + ')');
            }
            VulcanInterface.launchApp("aftereffects", true);
        }
    );
}
```

Viene qui messo in pratica quanto spiegato sul funzionamento della messaggistica gestita da *Vulcan*: in questa funzione viene infatti registrato il *listener* predisposto alla ricezione dei messaggi indirizzati ad After Effects; per prima cosa viene creato il *tipo* di messaggio, equivalente all'*indirizzo* di consegna del messaggio; dopo questa azione sono subito da notare due cose: la prima è l'utilizzo del già citato *singleton VulcanInterface*, il quale deve essere l'*unica* istanza della classe *Vulcan* affinché il servizio funzioni correttamente; la seconda è la presenza di un *prefisso* obbligatorio da anteporre al *tipo* appena creato. Una volta fissato l'*indirizzo*, viene predisposta la funzione *callback* per la gestione dei messaggi in entrata: per prima cosa viene verificato che ci si trovi in ambiente After Effects con l'identificatore *app*; viene quindi estratto il *carico utile* del messaggio (i dati vettoriali precedentemente estratti da Animate), il quale viene successivamente affidato, tramite `evalScript`, alla funzione `AeHandleData()` del file JSX, precedentemente caricato nella funzione `initToolScript()`; infine After Effects viene nuovamente *lanciato* per essere messo in primo piano.

Passiamo ora a `sendToAe`, il quale, come si può già prevedere arrivati a questo punto della trattazione, è predisposto all'invio dei messaggi rivolti ad After Effects. Eccone il codice:

```
function sendToAe(data) {
    var msg = new VulcanMessage (VulcanMessage.TYPE_PREFIX + "afterfx.msg");
    msg.setPayload(data);
```

```
    VulcanInterface.dispatchMessage(msg);
}
```

Ora il quadro delle procedure per la messaggistica *Vulcan* è completo: viene creato un nuovo messaggio avente lo stesso *indirizzo* registrato nel *listener*, dopodiché ne viene impostato il *carico utile* (di nuovo, i dati vettoriali estratti da Animate); infine viene spedito il messaggio. Il metodo `dispatchMessage` farà sì che il *listener* registrato si attivi per ricevere il messaggio qui creato, gestendolo con la funzione *callback*.

Con questo paragrafo si conclude lo sviluppo dell'interfaccia dell'estensione: nello stato attuale questa consiste in un componente perfettamente integrato nei programmi e potenzialmente in grado di farli comunicare; ciononostante al momento non si può dire che sia capace di fare effettivamente qualcosa. A questo punto quindi non rimane che entrare nel merito delle *core functions*, ovvero le funzioni `AnExtractData()` e `AeHandleData()` precedentemente menzionate.

3.5 *Core Functions*: JSFL & JSX

L'esposizione delle due funzioni avverrà separatamente nei sotto paragrafi successivi. È però importante enunciare le regole di base che si sono imposte nel procedimento delle suddette funzioni; si sta qui parlando delle modalità in cui i dati vettoriali vengono trattati e immagazzinati per la spedizione: è già stato detto infatti che la comunicazione tra le due app avviene esclusivamente per mezzo di *stringhe di testo*; ora, è chiaramente impensabile di trattare la mole di informazioni che costituiscono i dati vettoriali come semplici stringhe; d'altra parte sarebbe inefficiente utilizzare direttamente le classi di oggetti offerte dall'API del programma, in quanto la memorizzazione di tutti i dati offerti da questi modelli risulterebbe in molta informazione superflua, con conseguente spreco di memoria, e tutto questo senza considerare la difficoltà nell'adattare i modelli dell'una e dell'altra API, radicalmente differenti, come è già stato ampiamente discusso in precedenza.

La soluzione proposta è stata quindi quella di sviluppare un *modello a oggetti* completamente nuovo, facilmente adottabile da entrambe le API e contenente lo stretto indispensabili delle informazioni vettoriali; questo oggetto JavaScript sarà quindi annotato nel formato *JSON*, indispensabile per lo scambio dati tra le due app: il metodo `stringify` infatti rappresenta la risposta al primo problema, potendo convertire grandi quantità di dati incapsulate in un oggetto JavaScript in un'unica stringa.

Il modello risultante, nato dal tentativo di ibridare il meglio dei modelli proposti dalle due API, è il seguente:

```
{
  "layers": [
    {
      "shapes": [
        {                                   //<- AnShape Object
          "frames": [
            {                               //<- AnFrame Object
```

```
              "vertices": [“[x,y]”, …],
              "inTangents": [“[x,y]”, …],
              "outTangents": [“[x,y]”, …],
              "closed": boolean,
              "startFarme": number,
              "endFrame": number,
              "holes": [
                {                           //<- Hole Object
                  "vertices": [“[x,y]”, …],
                  "inTagnents": [“[x,y]”, …],
                  "outTangents": [“[x,y]”, …]
                },
                …
              ]
            },
            …
          ],
          "strokes": [
            {                               //<- AnStroke Object
              "color": string,
              "opacity": number,
              "width": number,
              "startFrame": number
            },
            …
          ],
          "fills": [
            {                               //<- AnFill Object
              "color": string,
              "opacity": number,
              "startFrame": number
            },
            …
          ]
        },
        …
      ]
    },
    …
  ]
}
```

La presenza dei puntini di sospensione (“…”) all’interno degli array sta a indicare che la lunghezza dell’array in questione non è nota a priori; in ogni proprietà è stato invece segnato il *tipo* del valore da assegnare a quella proprietà; nel caso delle proprietà *vertices, inTangents* e *outTangents* invece la formula *“[x,y]”* sta a indicare l’espressione di coordinate in una *stringa*. Mi rendo però conto che una notazione del genere possa generare confusione. Andiamo quindi ad analizzarla scomponendone le proprietà:

La proprietà *layers* è un *array* rappresentante i *livelli* di Animate *selezionati* per il trasferimento dati; è infatti volontà del progetto lasciare all’utente la scelta di quale livello o livelli trasferire da Animate ad After Effects.

Ogni elemento di *layers* è un *oggetto* che altro non è che la *collezione* di un tipo di oggetti da me nominati *AnShape* per non essere confusi con gli *Shape Object* dell’API JSFL. Questa *collezione* è stata appunto nominata *shapes*, l’unica proprietà presente negli elementi dell’array *layers*; questa disposizione è però del tutto superflua in termini pratici,

avendo per lo più fini espositivi. In fase di scripting infatti, ogni oggetto *collezione* dell'array *layers* sarà semplicemente implementato come un *array*, facendo diventare *layers* un *array di arrays*. Ad ogni modo, almeno ai fini di questa analisi, verrà comunque usata la nomenclatura *shapes* per riferirsi all'array di oggetti *AnShape*. E a tal proposito, viene esposto di seguito la funzione *costruttore* dell'oggetto *AnShape*:

```
function AnShape() {
    this.frames = [];
    this.strokes = [];
    this.fills = [];
    this.addNewFrame = function(frame) {
        this.frames.push(frame);
    };
    this.addNewFill = function(fill) {
        this.fills.push(fill);
    };
    this.addNewStroke = function(stroke) {
        this.strokes.push(stroke);
    };
}
```

Ogni oggetto *AnShape* ha come sue proprietà tre array, i cui elementi sono di altrettanti nuovi tipi di oggetti: *frames* è un *array* di oggetti *AnFrame*, *strokes* un *array* di oggetti *AnStroke* e *fills* un *array* di oggetti *AnFill*; questi elementi rappresentano le varie forme e colori di contorno/riempimento che una *shape* assume in ogni istante; da notare inoltre che anche in questo caso è stata utilizzata una nominazione alternativa col prefisso *"An"* per distinguere questi oggetti con quelli già presenti nell'API JSFL.

Di seguito il costruttore dell'oggetto *AnFrame*

```
function AnFrame(verts, inTangs, outTangs, closed, stFr, ndFr) {
    this.vertices = "[" + verts + "]";
    this.inTangents = "[" + inTangs + "]";
    this.outTangents = "[" + outTangs + "]";
    this.closed = closed;
    this.startFrame = stFr;
    this.endFrame = ndFr;
    this.addHoles = function(holes) {
        this.holes = holes;
    };
}
```

I valori *verts, inTangs* e *outTangs* assegnati rispettivamente alle proprietà *vertices, inTangents* e *outTangents* sono *arrays* di *stringhe* di coordinate nella forma *"[x,y]"*, rappresentanti i punti di controllo della curva di Bézier cubica che costituisce la *shape*; di queste coordinate, solo quelle di *verts* sono espresse in termini assoluti, mentre *inTangs* e *outTangs* sono coordinate relative a quelle di *verts*. La proprietà *closed* è un *booleano* per distinguere tra geometrie chiuse e aperte; *startFrame* e *endFrame* i valori *numerici* del *timecode* di inizio e fine del quadro.

Come si può notare dalla struttura del costruttore, la proprietà *holes* è facoltativa, venendo creata all'occorrenza da un metodo interno; questa è un *array* di oggetti *Hole*, la rappresentazione JavaScript di buchi all'interno di geometrie chiuse (non esistendo un corrispettivo in JSFL è stata utilizzata una nominazione standard). Questo il costruttore:

```
function Hole(verts, inTangs, outTangs) {
    this.vertices = "[" + verts + "]";
    this.inTangents = "[" + inTangs + "]";
    this.outTangents = "[" + outTangs + "]";
}
```

I valori delle proprietà sono espressi in maniera identica a quelle dell'oggetto *AnFrame*.

Infine, i costruttori degli oggetti *AnStroke* e *AnFill*

```
function AnStroke(color, opacity, width, stFr) {
    this.color = color;
    this.opacity = opacity;
    this.width = width;
    this.startFrame = stFr;
}
```

```
function AnFill(color, opacity, stFr) {
    this.color = color;
    this.opacity = opacity;
    this.startFrame = stFr;
}
```

In entrambi, *color* è una *stringa* rappresentante il codice esadecimale del colore, *opacity* un valore *numerico* per la percentuale di trasparenza del colore e *startFrame* il valore *numerico* di inizio del quadro; la proprietà *width* di *AnStroke* è invece lo spessore della linea di contorno.

Quindi, volendo sintetizzare tutto quanto appena detto per riassumere la gerarchia della notazione del modello:

- *layers* è l'*array* dei *livelli selezionati* in Animate;
- ogni *livello* è a sua volta un *array* di elementi *AnShape* (*shapes*);
- ogni elemento di *shapes*, ovvero ogni *AnShape*, ha al suo interno tre *arrays*, *frames*, *strokes* e *fills*, rispettivamente di oggetti *AnFrame*, *AnStroke* e *AnFill*;
- ogni elemento di *frames*, ovvero ogni *AnFrame*, è costituito di tre *arrays* di coordinate nella forma *"[x,y]"* utilizzati per la costruzione della curva di Bézier cubica che costituisce la *shape*; oltre questi tre arrays, ne possiede eventualmente un quarto, *holes*, costituito di oggetti *Hole*;
- ogni elemento di *holes*, ovvero ogni *Hole*, è costituito di tre *arrays* di coordinate nella forma *"[x,y]"*, aventi la stessa funzione degli omologhi della classe *AnFrame*;

Volendo utilizzare una scrittura compatta di questa gerarchia, in modo simile a quella presentata nel capitolo dell'API JSFL

```
layers.shapes.frames
```

e confrontandola proprio con quella JSFL

```
layers.frames.shapes(=elements)
```

si può notare l'inversione degli elementi di tipo *frame* con quelli di tipo *shape* (che ricordiamo essere una sottoclasse di *element*). Questo scambio è dovuto al fatto che una gerarchia del primo tipo risulta più adeguata al trasferimento dei dati in After Effects, poiché ne rispecchia più fedelmente il DOM.

Nel caso non sia stata ancora colta implicitamente, a questo punto della discussione colgo l'occasione per evidenziare la sostanziale differenza *concettuale*, più che strutturale, che queste due gerarchie comportano, differenza questa che altro non è che la base della distinzione tra *animazione 2D* basata su *keyframes* e *animazione frame-by-frame*: anteporre la classe *shape* rispetto *frame* nella gerarchia significa infatti *unificare* in un unico oggetto una *sequenza di forme*, le quali inevitabilmente assumono il valore di *trasformazioni* del suddetto oggetto; questo oggetto quindi, nonostante le trasformazioni, mantiene una sua *identità*; questo è esattamente ciò che succede nell'*animazione tramite keyframes*: ogni *keyframe* è il valore di una proprietà di un oggetto, fissato in un dato istante; passando da *keyframe* a *keyframe*, questo oggetto si trasforma nel tempo. Non è invece ciò che avviene nel secondo caso: anteporre la classe *frame* rispetto *shape* nella gerarchia infatti ha l'effetto di *spezzare* quella che altrimenti sarebbe una *continuità* di trasformazioni in tanti istanti indipendenti, ognuno dei quali contiene una certa *quantità di forme*, ognuna delle quali non ha relazioni con alcuna forma contenuta in istanti successivi o precedenti; in altre parole è ciò che accade nell'*animazione farme-by-frame*: ogni fotogramma rappresenta un disegno *indipendente*, il cui contenuto può anche avere relazioni di dipendenza con altri fotogrammi, ma solo nella mente dell'animatore, poiché per disposizione strutturale il programma non può assumere la suddetta dipendenza in forme contenute in fotogrammi differenti. Si cercherà di spiegare meglio quanto appena scritto con un esempio pratico estremamente semplificato, riassumibile come in Figura 3.8:

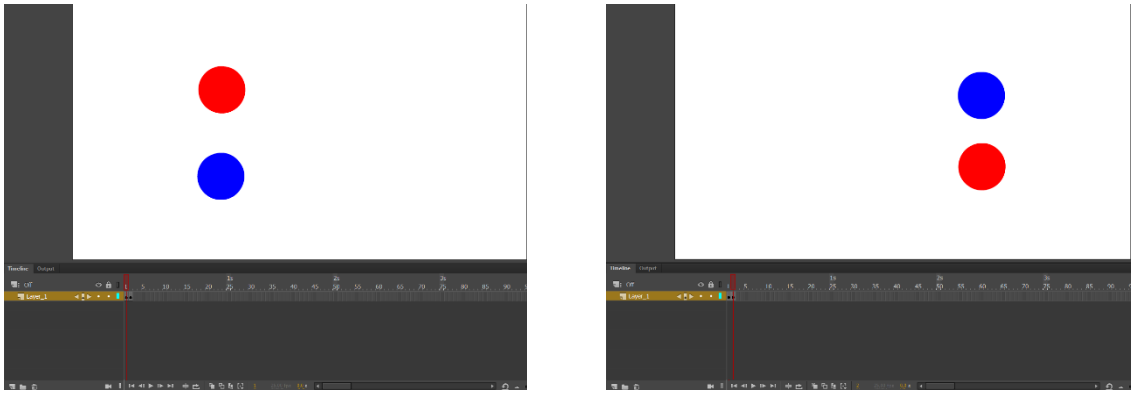

Figura 3.8 – indipendenza dei frame in animazione tradizionale

Si prendano i due oggetti in Figura 3.8, presentati in due fotogrammi adiacenti di un'animazione *frame-by-frame*: un essere umano potrebbe tranquillamente assumere il movimento del pallino rosso in diagonale, dall'alto in basso, da sinistra a destra, e stessa cosa si potrebbe dire del pallino blu; in altre parole, per un essere umano è percepibile l'*identità* dei due oggetti in diversi istanti di tempo. Questo Animate, per sua conformazione strutturale, non può assumerlo a priori. Tutto ciò che può constatare è la presenza di due oggetti in due fotogrammi adiacenti.

È quindi importante sottolineare che l'assunzione della gerarchia `layers.shapes.frames` in un programma per animazione tradizionale, avente quindi la gerarchia opposta, costituisce in un certo modo un limite, poiché nel trasferimento dei dati non può essere garantito il mantenimento del senso di continuità delle forme percepibile dall'occhio umano in un'animazione tradizionale, continuità che ci si auspicherebbe fosse mantenuta nella sequenza di *keyframes* che risulterebbe nell'importazione dei dati di Animate su After Effects. Nel caso questo costituisse un problema, sarà compito dell'animatore organizzare i *layers* di Animate per mantenere il livello di continuità desiderato (ad esempio dedicando un intero *layer* ad una sola *shape*, così facendo ogni *frame* avrà una sola forma e la continuità sarà inevitabile).

Va precisato che, in effetti, per ovviare a questo limite era stata avanzato un differente modello a oggetti. Questo prevedeva comunque una gerarchia del tipo `layers.shapes.frames`, sempre per adattarsi al DOM di After Effects, ma un approccio diverso nella definizione dell'oggetto *AnShape*: questo avrebbe avuto un unico valore *fill* o *stroke*, rispettivamente se fosse stata una geometria chiusa o aperta; questi valori sarebbero serviti, oltre che al loro utilizzo in sé e per sé, a identificare la stessa *shape* attraverso i diversi *frame*. Questo approccio però comportava nuovi problemi: innanzitutto ad un dato numero di colori diversi doveva corrispondere lo stesso numero di *shapes* su After Effects; questo voleva dire, ad esempio, che in un'animazione di 25 fotogrammi (un secondo a 25 fps) con una sola *shape* per ogni fotogramma ma con colori diversi l'una rispetto all'altra, sarebbero corrisposte su After Effects 25 *shapes* separate, con un notevole spreco di risorse; se poi questo primo problema potrebbe sembrare per alcuni accettabile, il secondo rappresentava un limite insormontabile. Si utilizzerà nuovamente un esempio pratico per rappresentare il suddetto problema:

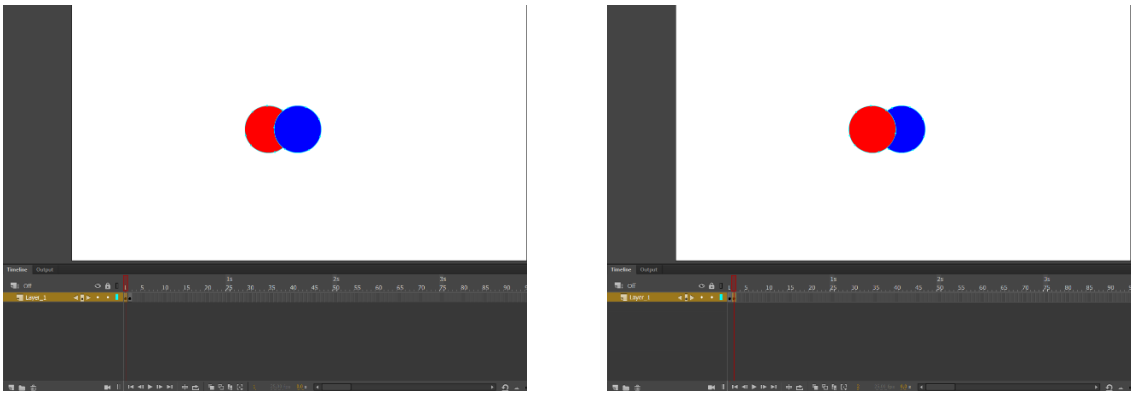

Figura 3.9 – indipendenza dell'ordine di sovrapposizione degli oggetti in fotogrammi differenti

Si prendano nuovamente due oggetti come in Figura 3.9: i due pallini sono stati disegnati in *drawing object mode*; come si ricorderà, gli elementi disegnati in questa modalità trovano spazio nell'array `elements` del DOM di Animate nell'esatto ordine in cui sono sovrapposti. Nell'esempio in Figura 3.9 i due pallini si scambiano di posizione da un fotogramma all'altro; ora si immagini di trasferire i dati di questa animazione su After Effects con il secondo modello dati proposto: essendoci solo due colori, verranno generate due *AnShape* aventi come indicatore identificativo rispettivamente il colore rosso e il colore blu; queste due *AnShape* verranno importate su After Effects come due proprietà *Path* in uno *ShapeLayer* ed è qui che nasce il problema: una volta fissati i *Path* in un certo ordine, questi non possono cambiare di posizione dinamicamente, perdendo

così lo scambio di posti dei due pallini che avviene nell'animazione su Animate. L'unica maniera per ovviare a questo problema è pensare a questa situazione, più che un'inversione di posizione nell'ordine di sovrapposizione, come un'inversione dei colori degli elementi, che è infatti ciò che avviene con il primo modello proposto. Avendo quindi preso visione dei pro e dei contro di ogni modello, si è optato per il male minore, se così si può definire, adottando così il primo modello.

Si conclude qui il discorso preliminare sull'esposizione delle *core functions*, al quale possono far seguito le analisi individuali delle suddette.

3.5.1 Animate *Core Function*: AnExtractData

Va premesso che l'intero meccanismo della funzione *core* di Animate ha un alto livello di suddivisione dei compiti in più metodi interdipendenti, come potrà essere testimoniato in seguito. Ciò detto, iniziamo subito con il contenuto della funzione `AnExtractData()`:

```
function AnExtractData() {
    an.getDocumentDOM().selectNone();

  var selectedLayers = [];
  var selectedLayersIndexes = an.getDocumentDOM().getTimeline().getSelectedLayers();
  var allLayers = an.getDocumentDOM().getTimeline().layers;

    for(var n = 0; n<selectedLayersIndexes.length; n++) {
        selection = selectedLayersIndexes[n];
        shapes = [];
        var frames = allLayers[selection].frames;
        for(var i=0; i<frames.length; i++) {
            if(i==frames[i].startFrame) {
                currFrameStart = frames[i].startFrame;
                currFrameEnd = frames[i].startFrame + frames[i].duration;
                frameAnalysis(frames[i]);
            }
        }
        selectedLayers.push(shapes);
    }
    return stringify(selectedLayers);
}
```

L'importanza qui sta nel riconoscere il modello a oggetti precedentemente introdotto per l'immagazzinamento dei dati: la variabile più importante, l'oggetto *radice* che conterrà tutti i dati, è l'array *selectedLayers*, equivalente al *layers* del modello; Animate non offre un metodo diretto per risalire ai livelli selezionati nell'interfaccia, tutt'al più il metodo `getSelectedLayers()` dell'oggetto *Timeline* che permette di recuperare i soli *indici* nell'array `layers` del DOM. Una volta recuperati i veri oggetti *Layer* selezionati, si passa all'analisi di ogni oggetto *Frame* nell'array `frames` di ogni livello nella selezione, ed è qui che inizia il primo livello di compartimentazione delle funzioni: l'analisi del *frame* è infatti affidata al metodo `frameAnalysis`, al quale arriveremo tra poco. Al momento basti sapere che il compito di questo metodo è *riempire* la variabile globale `shapes` delle geometrie presenti in quel dato fotogramma; `shapes` è un array,

equivalente allo *shapes* del modello, e viene reinizializzato ad ogni iterazione di un nuovo `selectedLayer`. Una volta *pieno*, `shapes` viene inserito in `selectedLayers` che viene poi convertito in stringa e inviato al *main.js* per la spedizione in After Effects.

Passiamo quindi al metodo `frameAnalysis(frame)`:

```
function frameAnalysis(frame) {
    for(var i = 0; i<frame.elements.length; i++) {
        el = frame.elements[i];
        switch(el.elementType) {
            case "shape":
                cubicPointsSegmentation(el);
                shapeAnalysis(el);
                break;
            case "text":
                break;
            case "instance":
                break;
        }
    }
}
```

In questo metodo avviene un'ulteriore suddivisione degli oggetti: viene infatti estratto, per ogni fotogramma, l'array `elements`, contenente tutti gli oggetti *Element*, e quindi anche gli oggetti *Shape*. Oltretutto, in questa versione dell'estensione gli elementi di tipo *"shape"* sono gli unici presi in considerazione, com'è testimoniato dallo `switch`. Una volta riconosciuto un *Element* come uno *Shape*, avvengono due azioni: la prima è la *segmentazione dei punti cubici* dell'oggetto *Shape*; si ricorderà infatti che, siccome la pseudo-struttura *CubicPoint* è presente solo a livello *Shape*, i punti cubici non sono suddivisi per ogni contorno di *shape* in *merge drawing mode* e/o per *drawing objects* compositi di più contorni; `cubicPointsSegmentation` provvede quindi a questa suddivisione, i cui risultati saranno utili per il riconoscimento del *verso* dei contorni. La seconda azione e l'*analisi della shape*, in cui verranno estratti i dati vettoriali veri e propri. Di seguito il contenuto dei due metodi:

```
function cubicPointsSegmentation(element) {
    elPoints = [];
    var index = -1;
    for (var i = 0; i < element.numCubicSegments; i++) {
        var cubicPoints = element.getCubicSegmentPoints(i);
        var point0 = "[" + cubicPoints[0].x + "," + cubicPoints[0].y + "]";
        var point3 = "[" + cubicPoints[3].x + "," + cubicPoints[3].y + "]";
        if(index ==-1 || elPoints[index].indexOf(point0)==-1) {
            index++;
            elPoints[index] = [];
            elPoints[index].push(point0);
        }
        if(elPoints[index].indexOf(point3)==-1)
            elPoints[index].push(point3);
        else
            elPoints[index].push("closed");
    }
}
```

```
function shapeAnalysis(element) {
    for(var i = 0; i<element.contours.length; i++) {
        contour = element.contours[i];
```

```
        if(isCorrect(contour, element)) {
            if(shapes.length>0) {
                var index = -1;
                for(var j = 0; j<shapes.length; j++) {
                    shape = shapes[j];
                    if(shape.frames[shape.frames.length-
1].startFrame<currFrameStart) {
                        index = shapes.indexOf(shape);
                        break;
                    }
                    else
                        continue;
                }
                if(index!=-1)
                    extractContourData(contour, shapes[index], element);
                else {
                    var anShape = new AnShape();
                    extractContourData(contour, anShape, element);
                    shapes.push(anShape);
                }
            }
            else {
                var anShape = new AnShape();
                extractContourData(contour, anShape, element);
                shapes.push(anShape);
            }
        }
    }
}
```

Si sorvolerà sulla spiegazione della prima in quanto, avendo ben chiaro in mente quanto detto nei capitoli precedenti sulla trattazione dei punti cubici e della pseudo-struttura *CubicPoint*, risulta piuttosto auto esplicativa.

Nella seconda invece avviene un procedimento importante, degno di essere approfondito: innanzitutto viene ulteriormente suddivisa la struttura dati, estraendo per ogni oggetto *Shape* l'array `contours` degli oggetti *Contour*. Dopodiché viene eseguita un'analisi per ogni *Contour* tramite il metodo `isCorrect`: questa verifica è di estrema importanza in quanto serve a discernere i contorni che caratterizzano delle *shapes* vere e proprie dai contorni di buchi nelle geometrie; questi ultimi infatti vanno analizzati in modo differente in un metodo a parte. Verificata la correttezza del *Contour* si decide se questo possa essere immagazzinato in un'*AnShape* esistente o in una nuova creata all'occorrenza; la decisione avviene confrontando le *AnShape* già presenti nell'array *shapes*, attraverso il confronto tra la proprietà `startFrame` dell'*AnShape* corrente con la variabile globale `currStartFrame`: nel caso il confronto verificasse che l'*AnShape* attuale non abbia ancora valori assegnati per il fotogramma corrente, verrebbe utilizzata per l'immagazzinamento dei dati, in caso contario si passa all'*AnShape* successiva fino alla fine di *shapes*. Nel caso in cui tutte le *AnShape* abbiano già un valore per il fotogramma corrente, oppure nel caso in cui *shapes* sia ancora vuoto, viene creata una nuova *AnShape*. Una volta risolta questa decisione, l'*AnShape* designata viene utilizzata per *l'estrazione dei dati del contorno* con il metodo `extractContourData`.

Prima di analizzare il funzionamento del metodo `isCorrect`, è bene esporre cos'è che contraddistingue una *shape* rispetto un *buco*, innanzitutto dando una definizione generica di quest'ultimo: *un buco è una curva chiusa, interna ad un contorno chiuso, che*

circoscrive un'area di assenza di colore di riempimento. Con questa descrizione è chiaro che una curva aperta risulti *sempre* un contorno corretto, in quanto non può costituire un buco per definizione. Per riconoscere se una curva chiusa rappresenti un buco invece bisogna avvalersi del sopracitato *colore di riempimento*, ovvero l'oggetto *Fill* associato ad ogni oggetto *Contour*. Si ricordi inoltre che ad ogni curva chiusa corrispondono due oggetti *Contour gemelli*, rivolti in direzioni opposte l'uno rispetto all'altro, ovvero *interno* ed *esterno*; questa caratteristica è riassunta nella proprietà `interior` degli oggetti *Contour*, un valore booleano indicante se il contorno racchiude un'area (ed è quindi rivolto verso l'interno) o no (ed è quindi rivolto verso l'esterno). In un *shape* con colore di riempimento, l'oggetto *Fill* è correlato al contorno interno della curva chiusa; al contorno esterno invece non è corrisposto alcun *Fill*. In un buco invece accade l'esatto contrario: essendo una curva chiusa, anche in questo caso sono presenti due contorni gemelli rivolti uno all'esterno e uno all'interno, ma con la situazione degli oggetti *Fill* invertita; al contorno interno infatti non è correlato alcun *Fill*, mentre a quello esterno è associato lo stesso *Fill* corrispondente al contorno interno della curva chiusa entro la quale il buco si trova. Viene fornita la Figura 3.10 per riassumere questa distinzione dei vari contorni in una *shape* avente un buco al suo interno.

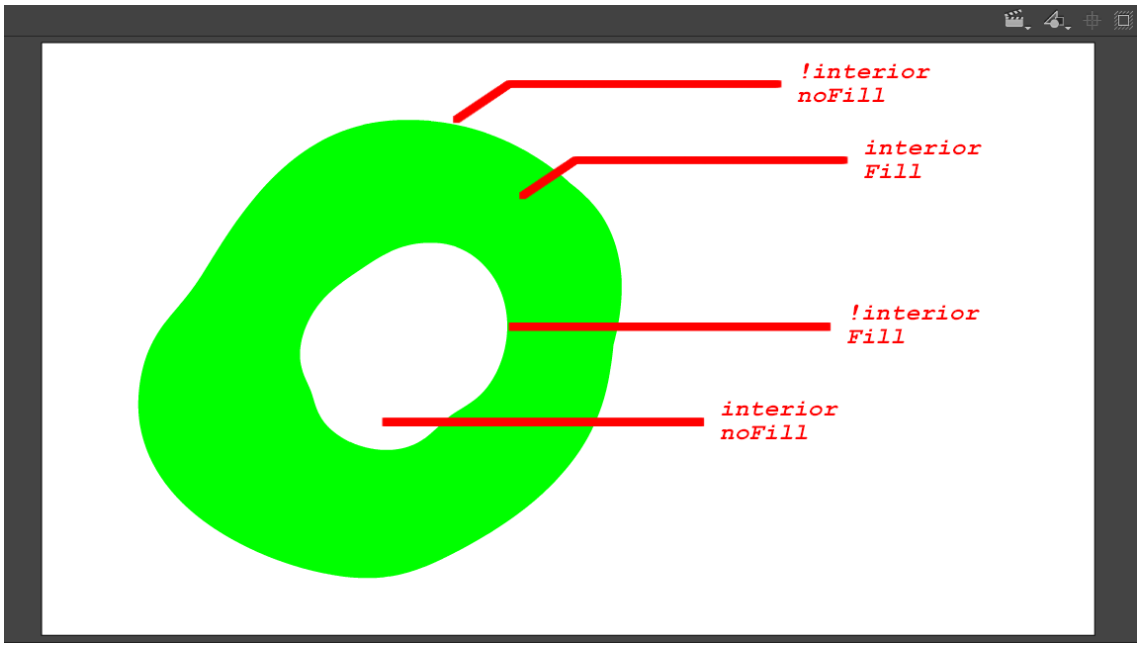

Figura 3.10 – i 4 oggetti Contour di una shape bucata, con relativa disposizione delle proprietà "interior" e "fill"

Verrebbe da pensare quindi che la chiave per distinguere un contorno chiuso corretto da un buco sia il fatto che esso racchiuda un'area (`interior==true`) ed abbia un *Fill* definito; questo però escluderebbe le *shapes* chiuse aventi solo uno *stroke* a definirle, poiché, se è vero che possano racchiudere un'area, non hanno un *Fill* definito né nel contorno interno né all'esterno (Figura 3.11).

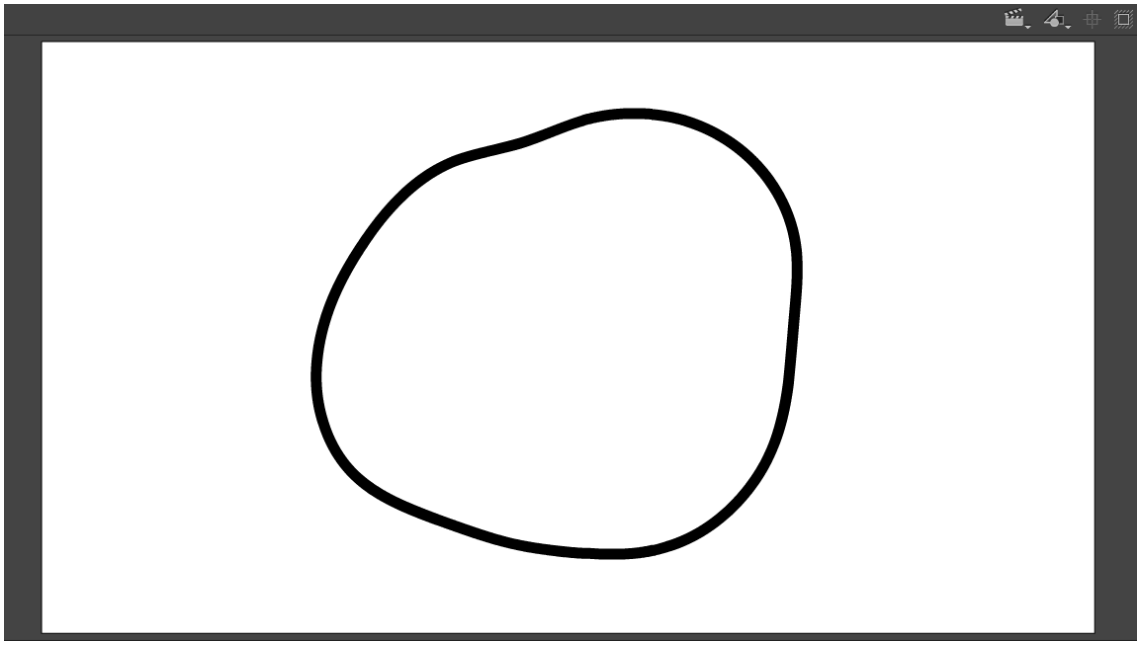

Figura 3.11 – una shape costituita dal solo stroke non possiede un fill né nel suo contorno esterno né in quello interno

Allora, invertendo il ragionamento e considerando l'esterno di un contorno, ci si accorge invece che a fare la differenza è il fatto che l'esterno di un buco abbia sempre un *Fill* definito; quindi un contorno corretto è contraddistinto dal fatto di racchiudere un'area e avere il proprio gemello senza alcun *Fill* definito. Ma anche in questo caso non si andrebbe incontro alla situazione presentata in Figura 3.12, visibilmente corretta ma scartata dalla definizione appena data.

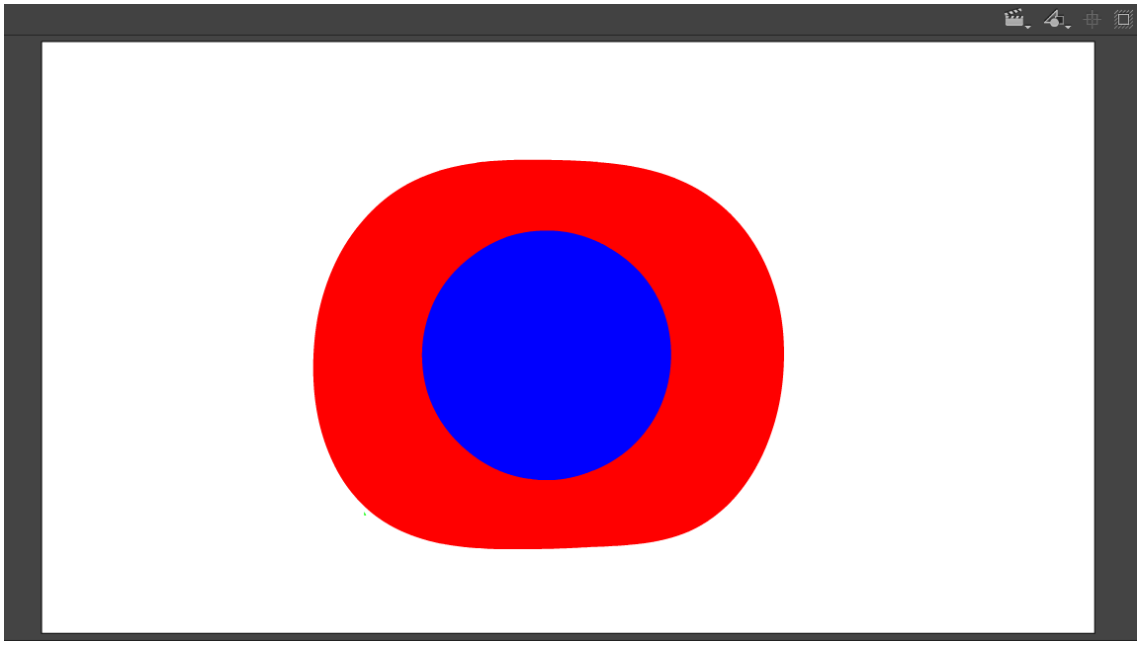

Figura 3.12 – una shape interna ad un'altra shape avrebbe contemporaneamente un fill all'interno ed uno all'esterno

La *shape* blu infatti si trova nella condizione di avere contemporaneamente il contorno interno ed esterno con un *Fill* definito (interno blu ed esterno rosso). Sarà quindi questo il distinguo nel caso si presentasse questa situazione.

Possiamo quindi finalmente riassumere quanto appena detto nel metodo `isCorrect`:

```
function isCorrect(contour, element) {
    if(!isClosed(contour))
        return true;
    if(contour.interior && (contour.fill.style!="noFill" || getTwin(contour, element
).fill.style=="noFill"))
        return true;
    else
        return false;
}
```

```
function getTwin(contour, element) {
    var baseContour = getContourVertices(contour).sort();
    for(var i = 0; i<element.contours.length; i++) {
        cntr = element.contours[i];
        if(cntr!=contour) {
            var otherContour = getContourVertices(cntr).sort();
            if(stringify(baseContour)==stringify(otherContour))
                return cntr;
        }
    }
}
```

`getTwin` è il metodo utilizzato per il recupero dei contorni gemelli, semplicemente confrontando l'insieme di vertici che li compongono con la loro controparte.

Dopo questa digressione possiamo tornare al metodo `extractContourData` lasciato in sospeso, di cui se ne dà il codice qui di seguito:

```
function extractContourData(contour, shape, el) {
    var contourVertices = getContourVertices(contour);
    filterPoints(contourVertices, getElementCubicPoints(el));
    var tangents = getContourTangents(contourVertices, el, isClosed(contour));
    if(isClosed(contour)) {
        var contourInTangents = tangents.inT;
        var contourOutTangents = tangents.outT;
    }
    else {
        var contourInTangents = tangents.outT;
        var contourOutTangents = tangents.inT;
    }
    var closed = isClosed(contour);

    var frame = new AnFrame(spaceConversion(contourVertices), contourInTangents, con
tourOutTangents, closed, currFrameStart, currFrameEnd);

    var shapeLastFill = shape.fills[shape.fills.length-1];
    if(contour.fill.style!="noFill"){
        frame.addHoles(holesDetection(contour, el));
        var ColOp = decompHexColor(contour.fill.color);
        if(shapeLastFill==undefined || shapeLastFill.color!=ColOp.color || shapeLast
Fill.opacity!=ColOp.opacity) {
            var fill = new AnFill(ColOp.color, ColOp.opacity, currFrameStart);
            shape.addNewFill(fill);
        }
    }
    else {
        if(shapeLastFill==undefined || shapeLastFill.color!=contour.fill.style)
            shape.addNewFill({color:contour.fill.style, startFrame:currFrameStart});

    }
```

```
    var currStroke = contour.getHalfEdge().getEdge().stroke;
    var shapeLastStroke = shape.strokes[shape.strokes.length-1];
    if(currStroke.style!="noStroke") {
        var ColOp = decompHexColor(currStroke.color);
        if(shapeLastStroke==undefined || shapeLastStroke.color!=ColOp.color || shape
LastStroke.opacity!=ColOp.opacity
        || shapeLastStroke.width!=currStroke.thickness) {
            var stroke = new AnStroke(ColOp.color, ColOp.opacity, currStroke.thickne
ss, currFrameStart);
            shape.addNewStroke(stroke);
        }
    }
    else {
        if(shapeLastStroke==undefined || shapeLastStroke.color!=currStroke.style)
            shape.addNewStroke({color:currStroke.style, startFrame:currFrameStart});

    }
    shape.addNewFrame(frame);
}
```

Questo è con tutta probabilità il cuore funzionale di `AnExtractData()`. Qui viene popolato l'*AnShape* precedentemente designata al contenimento dati. In particolare, verranno creati tre nuovi oggetti, un *AnFrame*, un *AnFill* e un *AnStroke*, che andranno ad aggiungersi agli array *frames*, *fills* e *strokes* della suddetta *AnShape*. Vediamo quindi la creazione di questi tre oggetti separatamente:

L'*AnFrame* per essere creato ha bisogno delle informazioni vettoriali della *shape* e di inizio e fine del quadro; questi ultimi due valori sono presi direttamente dalle variabili globali `currFrameStart` e `currFrameEnd`. Per le informazioni vettoriali della *shape* invece bisogna procedere per gradi, a partire dai soli *vertici*: vengono quindi estratti i vertici del contorno con il metodo `getContourVertices` qui presentato:

```
function getContourVertices(contour) {
    var he = contour.getHalfEdge();
    var iStart = he.id;
    var id = 0;
    var contourVertices = [];
    while (id != iStart) {
        if (he.getNext().getEdge().id == he.getOppositeHalfEdge().getEdge().id) {
            contourVertices.push("[" + he.getVertex().x + "," + he.getVertex().y + "
]");
            break;
        }
        contourVertices.push("[" + he.getVertex().x + "," + he.getVertex().y + "]");

        he = he.getNext();
        id = he.id;
    }

    if (!isClosed(contour)) {
        he = contour.getHalfEdge();
        id = 0;
        while (id != iStart) {
            if (he.getPrev().getEdge().id == he.getOppositeHalfEdge().getEdge().id)
{
                contourVertices.unshift("[" + he.getPrev().getVertex().x + "," + he.
getPrev().getVertex().y + "]");
                break;
            }
            he = he.getPrev();
```

```
            id = he.id;
            contourVertices.unshift("[" + he.getVertex().x + "," + he.getVertex().y
+ "]");
        }
    }
    return contourVertices;
}
```

Si può apprezzare in questo metodo come la struttura *DCEL* venga ampiamente sfruttata per la navigazione del contorno e per la distinzione di quest'ultimo in curva aperta o chiusa: partendo dalla stringa identificativa della prima *half-edge* sul contorno (`id`, proprietà dell'oggetto *HalfEdge*), tramite il `while` si passa da un *half-edge* all'altro continuando a collezionarne i vertici. Due sono le condizioni che possono interrompere il loop: la prima è ritrovarsi all'*half-edge* di partenza, nel qual caso vorrà dire di trovarsi di fronte una curva chiusa; la seconda è l'eventualità che le *half-edge*s successiva e opposta a quella attuale coincidano; questa condizione si verifica solo in presenza di una curva aperta, rendendo inutile il proseguimento *in avanti* dell'analisi del contorno. Questo però vuol dire dover collezionare i vertici *precedenti* l'*half-edge* di partenza, ed è infatti ciò che accade nella seconda metà del metodo, in una modalità simile alla prima metà. Una volta collezionati tutti i vertici, questi vengono utilizzati come valore di ritorno del metodo.

Riordiamo però che questi vertici, essendo presi da oggetti *HalfEdge*, sono punti di controllo di curve di Bézier *quadratiche*, ma come è già stato spiegato, un punto *cubico* è sempre anche un punto *quadratico*; questo vuol dire che l'array di punti *quadratici* appena estratto va *filtrato* per ricavarne i punti *cubici*: è questo il caso della funzione `filterPoints`:

```
function filterPoints(array, filter) {
    var isCubic;
    var i = 0;
    while(i<array.length) {
        for(var n = 0; n<filter.length; n++) {
            point = filter[n];
            if(!checkTolerance(array[i],point)) {
                isCubic = false;
                continue;
            }
            else {
                isCubic = true;
                break;
            }
        }
        if(!isCubic)
            array.splice(i, 1);
        else
            i++;
    }
}
```

L'array `filter` altro non è che la collezione di tutti i punti cubici della *shape*, che come è già stato detto non ha modo di segmentarli a seconda dei contorni. Il funzionamento del metodo è semplice: ad ogni vertice presente in `array` viene fatta una verifica su `filter` per trovare una corrispondenza; questa è verificata tramite il metodo `checkTolerance`,

un funzione speciale creata dopo aver verificato empiricamente che non sempre il valore numerico delle coordinate di un punto quadratico e il suo corrispettivo cubico coincidono perfettamente, rendendo insufficiente il semplice confronto con il simbolo "=="; con il metodo `checkTolerance` si aggira il problema introducendo nel confronto una tolleranza ad un errore del 10%, sufficiente a verificare l'identità tra i due punti senza il dubbio che siano troppo diversi. Ad ogni modo, in caso di avvenuta corrispondenza il vertice è mantenuto nell'array `array`, altrimenti viene eliminato.

Una volta filtrati i vertici della *shape*, si può procedere a trovarne le *inTangents* e *outTangents* con il metodo `getContourTangents`:

```
function getContourTangents(contourVertices, element, closed) {
    var contourInTangents = [];
    var contourOutTangents = [];
    for(var i=0; i<contourVertices.length; i++) {
        for(var j=0; j<element.numCubicSegments; j++) {
            var cubicPoints = element.getCubicSegmentPoints(j);
            var point0 = "[" + cubicPoints[0].x + "," + cubicPoints[0].y + "]";
            var point3 = "[" + cubicPoints[3].x + "," + cubicPoints[3].y + "]";
            if(checkTolerance(contourVertices[i],point0))
                contourInTangents.push("[" + (cubicPoints[1].x - cubicPoints[0].x) +
 "," + (cubicPoints[1].y - cubicPoints[0].y) + "]");
            if(checkTolerance(contourVertices[i],point3))
                contourOutTangents.push("[" + (cubicPoints[2].x - cubicPoints[3].x)
+ "," + (cubicPoints[2].y - cubicPoints[3].y) + "]");
        }
    }
    if(!closed) {
        contourInTangents.push("[0,0]");
        contourOutTangents.unshift("[0,0]");
    }
    return {inT:contourInTangents,outT:contourOutTangents};
}
```

È piuttosto semplice intuire il funzionamento del metodo avendo chiaro in mente quanto spiegato sui punti di controllo di una curva di Bézier cubica. Ciò che è importante notare è che i valori sono immagazzinati come *coordinate relative* rispetto i vertici. Il valore di ritorno del metodo invece è un oggetto contenente i due array delle tangenti.

Tornando al metodo `extractContourData`, si noti la verifica successiva all'estrazione delle tangenti: a seconda che si abbia una curva chiusa o aperta, le tangenti vengono immagazzinate normalmente o invertite nell'ordine. Questo è dovuto ad un'analisi empirica in cui si è potuto constatare che nel caso di una *shape* aperta, su After Effects le tangenti risultassero invertite rispetto a quanto assegnato in fase di memorizzazione su Animate.

Viene infine assegnata la variabile booleana `closed` tramite il metodo `isClosed`, già ampiamente utilizzato durante gli esempi di funzioni precedenti, e di cui si dà finalmente il codice:

```
function isClosed(contour) {
    var he = contour.getHalfEdge();
    var iStart = he.id;
    var id = 0;
    while (id != iStart) {
        if (he.getNext().getEdge().id == he.getOppositeHalfEdge().getEdge().id)
```

```
            return false;
        he = he.getNext();
        id = he.id;
        if (id == iStart)
            return true;
    }
}
```

Questi non è altro che una versione semplificata del metodo `getContourVertices`.

Ora si hanno tutti gli elementi per creare l'oggetto *AnFrame* (previa conversione delle coordinate dei vertici dallo spazio di Animate allo spazio di After Effects con il metodo `spaceConversion`, che adopera un semplice cambio del sistema di riferimento), o per lo meno quelli obbligatori. Non è ancora stata verificata infatti la presenza di buchi nella geometria, ma non essendo obbligatori nel costruttore di *AnFrame*, questa può essere approntata successivamente, ovvero nella fase di costruzione dell'oggetto *AnFill*. È infatti importante notare che, in caso di assenza dell'oggetto *Fill*, anche la ricerca di buchi nella *shape* risulterebbe superflua, per definizione stessa di un buco. In caso di effettiva presenza di un *Fill* invece, oltre che l'avvio della costruzione di un oggetto *AnFill*, che affronteremo tra poco, viene avviata anche la ricerca di buchi nella *shape*. Questo il codice dell'algoritmo di `holesDetection`:

```
function holesDetection(contour, element) {
    var holes = [];
    var baseBounds = getBoundingBox(contour);
    for(var i = 0; i<element.contours.length; i++) {
        hole = element.contours[i];
        if(hole.orientation==1 && hole.fill.color==contour.fill.color && hole!==cont
our) {
            var holeBounds = getBoundingBox(hole);
            if(holeBounds.left>baseBounds.left && holeBounds.top>baseBounds.top && h
oleBounds.right<baseBounds.right && holeBounds.bottom<baseBounds.bottom)
                holes.push(extractHoleData(hole, element));
        }
    }
    return holes;
}
```

I fattori fondamentali di questo metodo sono due: il primo è il riconoscimento di un contorno come buco, già affrontato precedentemente ma che qui assume una sfumatura leggermente diversa; il secondo è, verificato lo stato di buco in un contorno, l'effettiva assegnazione del buco al contorno preso in esame. Per il primo problema si ricordi quanto detto in precedenza, ovvero che un buco può essere riconosciuto come un contorno chiuso rivolto verso l'interno (`interior==true`) ma senza un *Fill* definito, oppure come contorno chiuso rivolto verso l'esterno (`interior==false`) e avente un *Fill* identico a quello della *shape* in cui si trova; nel metodo in questione quest'ultima definizione trova miglior utilizzo per due motivi, il primo dei quali è la presenza del *Fill*, che rappresenta un ulteriore elemento di verifica per l'assegnazione del buco alla *shape*. Il secondo è dovuto al fatto di dover considerare il contorno esterno: come si noterà infatti, più che la proprietà `interior` è stata utilizzata `orientation`, sempre dell'oggetto *Contour*. È possibile verificare empiricamente che esista sempre una corrispondenza tra il valore `interior==false` e il valore `orientation==1`, corrispondente al senso *orario*; questo, oltre a rappresentare una condizione più forte nella verifica (vengono infatti

scartate direttamente le curve aperte, velocizzando la verifica), garantisce l'interpretazione del contorno come buco una volta trasferito su After Effects: il programma infatti buca le *shapes* a fronte di due contorni sovrapposti aventi sensi opposti, e siccome i contorni immagazzinati negli array di *AnFrame* sono sempre *antiorario* (poiché `interior==true`), questa condizione è sempre garantita.

Dopo aver verificato queste due condizioni, ovvero orientamento verso l'esterno e presenza di un *Fill*, viene appurato che il contorno in questione non coincida con la *shape* di riferimento; superate tutte le verifiche, si può indubbiamente asserire che il contorno in questione sia un buco. Ora va comprovato che questo sia assegnabile alla *shape* in esame; questa verifica va eseguita confrontando *bounding boxes* dei due contorni: se il *bounding box* del buco sarà completamente contenuto nel *bounding box* della *shape*, sarà possibile affermare univocamente che il buco appartiene alla *shape*. Purtroppo, JSFL non offre un metodo per il recupero del *bounding box* di un contorno, al che ne è stato scritto uno *ad hoc*; questo si compone di tre funzioni interdipendenti che sfruttano *l'algoritmo di de Casteljau*, utilizzando le pseudo strutture *Point* degli oggetti *Edge* del contorno per eseguire i calcoli. Qui di seguito il codice delle tre funzioni:

```
function getBoundingBox(contour) {
    var he = contour.getHalfEdge();
    var iStart = he.id;
    var id = 0;
    left = top = Number.POSITIVE_INFINITY;
    right = bottom = Number.NEGATIVE_INFINITY;
    while (id != iStart) {
        e = he.getEdge();
        edgeAnalysis(e);
        he = he.getNext();
        id = he.id;
    }
    return {left:left,top:top,right:right,bottom:bottom};
}

function edgeAnalysis(edge) {
    step = 1/40;
    p0x = edge.getControl(0).x;
    p0y = edge.getControl(0).y;
    p1x = edge.getControl(1).x;
    p1y = edge.getControl(1).y;
    p2x = edge.getControl(2).x;
    p2y = edge.getControl(2).y;
    for(d = 0; d<1.001; d+=step) {
        left = Math.min(left, quadBez(p0x, p1x, p2x, d));
        top = Math.min(top, quadBez(p0y, p1y, p2y, d));
        right = Math.max(right, quadBez(p0x, p1x, p2x, d));
        bottom = Math.max(bottom, quadBez(p0y, p1y, p2y, d));
    }
}

function quadBez(p0, p1, p2, t) {
    return Math.pow((1.0 - t), 2) * p0 + 2 * t * (1.0 - t) * p1 + Math.pow(t, 2) * p
2;
}
```

Come si può notare, i lati del *bounding box* altro non sono i valori del punto minore su ascisse e ordinate rispettivamente per il lato sinistro e superiore, mentre sono i valori del punto maggiore su ascisse e ordinate rispettivamente per il lato destro e inferiore; dati i

tre punti di controllo della curva di Bézier quadratica disegnata da un oggetto *Edge*, l'algoritmo ricorsivo di de Casteljau (rappresentato dalla funzione `edgeAnalysis`) è utilizzato per calcolare i punti intermedi della curva da confrontare all'avanzare dell'algoritmo coi valori dei lati del *bounding box*, da aggiornare all'occorrenza.

Ora che finalmente è stato verificato lo stato di buco e l'appartenenza alla *shape*, non resta che estrarre i dati con il metodo `extractHoleData`:

```
function extractHoleData(hole, el) {
    var holeVertices = getContourVertices(hole);
    filterPoints(holeVertices, getElementCubicPoints(el));
    var tangents = getContourTangents(holeVertices, el, true);

    var holeInTangents=checkDirection(holeVertices)? tangents.outT : tangents.inT;
    var holeOutTangents=checkDirection(holeVertices)? tangents.inT : tangents.outT;

    var anHole = new Hole(spaceConversion(holeVertices), holeInTangents, holeOutTang
ents);
    return anHole;
}
```

Questo, per ovvie ragioni, è molto simile alla prima parte del metodo `extractContourData`, con l'aggiunta però della *verifica della direzione*: sempre in maniera empirica infatti, è stato verificato che, a seconda che il buco sia stato fatto direttamente circoscrivendo un'area vuota con lo strumento *Pennello* oppure con lo strumento *Gomma*, questi presenta gli array delle tangenti in uno o in un altro ordine; va quindi verificato che la direzione dei punti appena immagazzinati nell'array `holeVertices` sia coerente con quella dell'array contenente la *segmentazione dei punti cubici* effettuata agli inizi di questo paragrafo. La verifica è così effettuata:

```
function checkDirection(contourVertices) {
    for(var i = 0; i<elPoints.length; i++) {
        for(var j = 0; j<elPoints[i].length; j++) {
            if(checkTolerance(contourVertices[0], elPoints[i][j])) {
                if(j==elPoints[i].length-1 || elPoints[i][j+1]=="closed")
                    return checkTolerance(contourVertices[1], elPoints[i][0]);
                else
                    return checkTolerance(contourVertices[1], elPoints[i][j+1]);
            }
        }
    }
}
```

Passiamo ora alla creazione dell'oggetto *AnFill*: questi ha bisogno di tre elementi nel costruttore, *color*, *opacity* e *startFrame*; quest'ultimo è nuovamente preso dalla variabile globale `currStartFrame`. Gli altri due elementi invece sono ricavati dalla scomposizione del codice colore esadecimale dedotto dalla proprietà `color` dell'oggetto *Fill* del contorno. Ciò che è importante però in fase di costruzione di un oggetto *AnFill* è la verifica dell'ultimo elemento dell'array *fills* dell'oggetto *AnShape*: gli oggetti *AnFill* infatti non vengono creati indistintamente ad ogni fotogramma come gli *AnFrame*, ma solo all'occorrenza nel caso sia avvenuto un cambiamento rispetto l'ultimo elemento dell'array. Questo approccio migliora l'efficienza dell'algoritmo, rimuovendo le eventuali ridondanze di oggetti *AnFill* identici in posizioni contigue dell'array.

Ora non rimane altro che affrontare l'oggetto *AnStroke*, per il quale non rimane però molto da dire: questo ha un costruttore molto simile ad *AnFill* e viene oltretutto inizializzato in modo pressoché identico. Con quest'ultimo pezzo della funzione `extractContourData` si conclude la trattazione della *core function* di Animate. E ora che l'oggetto `selectedLayers` è stato convertito in stringa e spedito, si può passare alla *core function* lato After Effects.

3.5.2 After Effects *Core Function*: AeHandleData

Data la già discussa semplicità dell'API ExtendScript rispetto JSFL, la funzione *core* di After Effects risulterà relativamente meno complessa di quella di Animate. A questo si aggiunga anche il grande lavoro di modellazione della struttura dati precedentemente affrontato che semplifica ulteriormente la complessità del codice. Ma vediamo subito il contenuto di `AeHandleData`:

```
function AeHandleData(layers) {
    for (var i = layers.length-1; i>=0; i--) {
        createLayer(layers[i]);
    }
}
```

Come si può vedere, questo consiste esclusivamente in un loop per lavorare separatamente i livelli selezionati da Animate tramite la funzione `createLayer`, vero cuore pulsante della *core function*. Si potrà notare la particolarità di estrarre i livelli dall'array `layers` (che ricordiamo essere il `selectedLayers` spedito da Animate) partendo dall'ultimo: questo è dovuto alla modalità in cui After Effects aggiunge nuovi livelli nella Composizione, ovvero impilando il nuovo livello sopra il primo della colonna; siccome nell'array i livelli sono già memorizzati nell'ordine corretto (il primo elemento è quello più in alto nella pila), e tenendo conto della modalità di inserimento livelli di After Effects, bisogna quindi partire dall'ultimo per arrivare fino al primo.

Qui di seguito viene mostrato il contenuto della funzione `createLayer`:

```
function createLayer(layer) {
    var frameDur = app.project.activeItem.frameDuration;
    var shapeLayer = app.project.activeItem.layers.addShape();
    var shapeLyrContents = shapeLayer.property("ADBE Root Vectors Group");
    for (var i = layer.length-1; i>=0; i--) {
        var shape = layer[i];
        var shapeGroup = shapeLyrContents.addProperty('ADBE Vector Group');
        var shapeGrpContents = shapeGroup.addProperty('ADBE Vectors Group');
        var maxPaths = 0;
        for(var j = 0; j<shape.frames.length; j++) {
            if(shape.frames[j].holes!=undefined && shape.frames[j].holes.length>maxP
aths)
                maxPaths = shape.frames[j].holes.length;
        }
        for(var j = 0; j<=maxPaths; j++) {
            shapeGrpContents.addProperty('ADBE Vector Shape - Group');
        }
        for(var j = 0; j<shape.frames.length; j++) {
            var frame = shape.frames[j];
```

```
            var pathPrprty = shapeGrpContents.property(1).property('ADBE Vector Shap
e');
            var shapePath = new Shape();
            if(lastFrameEnd!=undefined && frame.startFrame!=lastFrameEnd)
                wipeOut(shapeGrpContents, maxPaths, lastFrameEnd*frameDur);
            if(j==0 && frame.startFrame>0) {
                shapePath.vertices = [[0,0]];
                shapePath.inTangents = [[0,0]];
                shapePath.outTangents = [[0,0]];
                setHoldKeyframe(pathPrprty, 0, shapePath);
            }
            shapePath.vertices = JSON.parse(frame.vertices);
            shapePath.inTangents = JSON.parse(frame.inTangents);
            shapePath.outTangents = JSON.parse(frame.outTangents);
            shapePath.closed = frame.closed;
            setHoldKeyframe(pathPrprty, frame.startFrame*frameDur, shapePath);
            if(shape.frames[j].holes!=undefined) {
                for(var k = 0; k<maxPaths; k++) {
                    pathPrprty = shapeGrpContents.property(2+k).property('ADBE Vecto
r Shape');
                    if(frame.holes[k]!=undefined) {
                        shapePath.vertices = JSON.parse(frame.holes[k].vertices);
                        shapePath.inTangents = JSON.parse(frame.holes[k].inTangents)
;
                        shapePath.outTangents = JSON.parse(frame.holes[k].outTangent
s);
                        shapePath.closed = true;
                        setHoldKeyframe(pathPrprty, frame.startFrame*frameDur, shape
Path);
                    }
                    else {
                        shapePath.vertices = [[0,0]];
                        shapePath.inTangents = [[0,0]];
                        shapePath.outTangents = [[0,0]];
                        setHoldKeyframe(pathPrprty, frame.startFrame*frameDur, shape
Path);
                    }
                }
            }
            else {
                for(var k = 0; k<maxPaths; k++) {
                    pathPrprty = shapeGrpContents.property(2+k).property('ADBE Vecto
r Shape');
                    shapePath.vertices = [[0,0]];
                    shapePath.inTangents = [[0,0]];
                    shapePath.outTangents = [[0,0]];
                    setHoldKeyframe(pathPrprty, frame.startFrame*frameDur, shapePath
);
                }
            }
            if(j==shape.frames.length-1)
                wipeOut(shapeGrpContents, maxPaths, frame.endFrame*frameDur);
            var lastFrameEnd = frame.endFrame;
        }
        if(shape.strokes.length>1 || (shape.strokes.length==1 && shape.strokes[0].co
lor!="noStroke")) {
            shapeGrpContents.addProperty('ADBE Vector Graphic - Stroke');
            var strokePrprty = shapeGrpContents.property(shapeGrpContents.numPropert
ies);
            for(var j = 0; j<shape.strokes.length; j++) {
                var stroke = shape.strokes[j];
                var strokeTime = stroke.startFrame*frameDur;
                if(j==0 && stroke.startFrame>0)
                    setHoldKeyframe(strokePrprty.opacity, 0, 0);
                if(stroke.color!="noStroke") {
```

```
                    setHoldKeyframe(strokePrprty.color, strokeTime, hexToColor(strok
e.color));
                    setHoldKeyframe(strokePrprty.opacity, strokeTime, stroke.opacity
);
                    setHoldKeyframe(strokePrprty.strokeWidth, strokeTime, stroke.wid
th);
                }
                else
                    setHoldKeyframe(strokePrprty.opacity, strokeTime, 0);
            }
        }
        if(shape.fills.length>1 || (shape.fills.length==1 && shape.fills[0].color!="
noFill")) {
            shapeGrpContents.addProperty('ADBE Vector Graphic - Fill');
            var fillPrprty = shapeGrpContents.property(shapeGrpContents.numPropertie
s);
            for(var j = 0; j<shape.fills.length; j++) {
                var fill = shape.fills[j];
                var fillTime = fill.startFrame * frameDur;
                if (j == 0 && fill.startFrame > 0)
                    setHoldKeyframe(fillPrprty.opacity, 0, 0);
                if(fill.color!="noFill") {
                    setHoldKeyframe(fillPrprty.color, fillTime, hexToColor(fill.colo
r));
                    setHoldKeyframe(fillPrprty.opacity, fillTime, fill.opacity);
                }
                else
                    setHoldKeyframe(fillPrprty.opacity, fillTime, 0);
            }
        }
    }
}
```

A prima vista potrebbe sembrare una funzione particolarmente complicata; avendo però ben chiaro in mente il DOM di After Effects ci si renderà conto che non è così, poiché gran parte del codice è dedicato alla dichiarazione e creazione delle proprietà di un livello *ShapeLayer* tramite *match names*. Per il resto la funzione del metodo può essere riassunta in *spacchettare e trattare le proprietà contenute nelle AnShape che compongono un livello*. Partiamo dall'array `frames`: la difficoltà qui è conoscere e gestire il numero di proprietà *Path* che andranno a comporre la *shape*; ricordiamo infatti che ogni oggetto *AnFrame* è sì composto da un'unica *shape* (al quale corrisponde una proprietà *Path*) ma può avere un numero arbitrario di buchi che vanno ad aumentare il numero di proprietà *Path* da aggiungere; siccome in After Effects non è possibile aggiungere proprietà *Path* dinamicamente, questo numero va conosciuto prima di iniziare a *riempire* i *Path*. Data la disposizione dei dati in `frames`, si può dire che questo numero è uguale alla somma dell'unica *shape* propriamente detta dell'*AnShape* più il valore massimo tra le lunghezze degli array `holes` di ogni oggetto *AnFrame*, valore questo rappresentato da `maxPaths`. Una volta fissato questo valore è possibile iniziare a popolare uno *Shape Group* (contenitore dei dati di una singola *AnShape*) di tutte le proprietà *Path* necessarie. Ora però queste proprietà vanno gestite in modo consono; infatti non si devono dimenticare le peculiarità derivanti dalle diverse gerarchie dati utilizzate dai due programmi e quella adottata dall'estensione. Per aiutarsi a spiegare ciò che si vuole esprimere in questo frangente verrà fornito un esempio di un'animazione ipotetica: si immagini su Animate un'animazione composta di 3 fotogrammi; nel primo è presente una *shape*, il secondo è vuoto, nel terzo è nuovamente presente una *shape*. Ora, spedendo questi dati su After

Effects con il modello adottato, si avranno solo i dati riguardanti i fotogrammi i cui è presente una *shape*. Importando questi dati nel programma è vero che questi due fotogrammi verranno fedelmente tradotti in *keyframes*, ma senza adeguate misure l'animazione non sarà corretta poiché, data l'interpolazione dei *keyframes*, il fotogramma centrale non sarà vuoto. Questo vuoto va quindi *dedotto* dalle informazioni ricevute ed esplicitamente comunicato ad After Effects. Questo stesso discorso vale anche nel caso in cui varino i numeri dei buchi presenti su una *shape* da un fotogramma all'altro. Queste operazioni di *creazione del vuoto* sono particolarmente evidenti tramite l'impiego della funzione `wipeOut`, che letteralmente *spazza via* ogni *Path* presente in scena:

```
function wipeOut(propertyGroup, maxP, time) {
    var nullPath = new Shape();
    nullPath.vertices = [[0,0]];
    nullPath.inTangents = [[0,0]];
    nullPath.outTangents = [[0,0]];
    for(var i = 0; i<=maxP; i++) {
        var pathPrprty = propertyGroup.property(1+i).property('ADBE Vector Shape');

        setHoldKeyframe(pathPrprty, time, nullPath);
    }
}
```

Avendo introdotto nell'argomento l'*interpolazione dei keyframes* va fatto notare che per definizione stessa di animazione *frame-by-frame* non dovrebbe esistere una vera interpolazione. Per trasferire correttamente un'animazione da Animate ad After Effects quindi è necessario che ogni *keyframe* su quest'ultimo sia di tipo *hold*, che è esattamente lo scopo della funzione `setHoldKeyframe`:

```
function setHoldKeyframe(property, time, value) {
    var keyIndex = property.addKey(time);
    property.setValueAtKey(keyIndex, value);
    property.setInterpolationTypeAtKey(keyIndex, KeyframeInterpolationType.HOLD);
}
```

Arrivati a questo punto è facile constatare come il trattamento degli array `strokes` e `fills` delle *AnShape* segua una procedura simile, se non più semplice, a quella dell'array `frames`. Si può quindi dire conclusa la trattazione delle *core functions* dell'estensione, nonché l'intero sviluppo della stessa. Adesso si può procedere al *packaging* dell'estensione per la sua distribuzione.

3.6 *Packaging & Signing*: ZXPSignCMD

Ora che l'estensione è conclusa si può pensare alla distribuzione. Chiaramente è impensabile di richiedere ad altri utenti l'installazione manuale dell'estensione, poiché questa, oltre una conoscenza dell'organizzazione delle cartelle dei programmi Adobe, richiederebbe l'attivazione del *debug mode* sulla macchina, procedura che per molti potrebbe risultare complicata. L'estensione andrà quindi *confenzionata* in un formato più adatto a un'installazione più semplice e diretta, ovvero il formato *ZXP*, già introdotto in precedenza. Questo è in pratica un file archivio che andrà a contenere, oltre che cartelle

e risorse dell'estensione, un *signing certificate* (*certificato di firma*), il cui scopo è quello di *verificare l'integrità del pacchetto dal momento del suo confezionamento*, ovvero verifica che lo ZXP non sia stato alterato dal momento della sua creazione né abbia superato la data di scadenza [19].

Per generare il file ZXP, Adobe rende disponibile un pratico strumento scaricabile dalla repository ufficiale del CEP, lo *ZXPSignCMD*: questo non è altro che un eseguibile da utilizzare da linea di comando. Per procedere al confezionamento, innanzitutto vanno spostate in una nuova posizione sia la cartella contenente l'estensione che l'eseguibile *ZXPSignCMD.exe* (è infatti sconsigliato effettuare il *packaging* direttamente nella cartella *extensions*). Fatti ciò, va innanzitutto generato il *certificato di firma*, utilizzando la seguente istruzione da linea di comando:

```
ZXPSignCmd -selfSignedCert <countryCode> <stateOrProvince> <organization> <commonName>
<password> <outputPath.p12> [options]
```

in Figura 3.13 sono segnate le corrispondenze dei parametri richiesti dall'istruzione

countryCode *stateOrProvince* *organization* *commonName*	The certificate identifying information.	
password	The password for the new certificate.	
outputPath.p12	The path and file name for the new certificate.	
options	-locality <*code*>	If supplied, the locale code to associate with this certificate.
	-orgUnit <*name*>	If supplied, an organizational unit to associate with this certificate.
	-email <*addr*>	If supplied, an email address to associate with this certificate.
	-validityDays <*num*>	If supplied, a number of days from the current date-time that this certificate remains valid.

Figura 3.13 – legenda dei parametri per l'istruzione "selfSignedCert"

Questo genererà un *self-signed certificate* utilizzabile per il *packaging* vero e proprio. Questa l'istruzione da eseguire da linea di comando:

```
ZXPSignCmd -sign <inputDir> <outputZxp> <p12> <p12Password> [options]
```

e la legenda

inputDir	The path to the folder containing the source files to package.	
outputZxp	The path and file name for the ZXP package.	
p12	The signing certificate;	
p12Password	The password for the certificate.	
options	`-tsa <timestampURL>`	The timestamp server. For example: `https://timestamp.geotrust.com/tsa`

Figura 3.14 - legenda dei parametri per l'istruzione "sign"

Qui di seguito vengono dati degli esempi pratici di scrittura per le istruzioni di *self-signing*

```
./ZXPSignCmd -selfSignedCert US NY MyCompany MyCommonName abc123 MyCert.p12
```

e *packaging*

```
./ZXPSignCmd -sign myExtProject myExtension.zxp MyCert.p12 abc123
```

Ora il file ZXP è pronto alla distribuzione; per l'installazione sarà sufficiente utilizzare uno dei tanti *installer* gratuiti disponibili sul web, tra i quali *Extension Manager*, l'*installer* ufficiale Adobe [20], o *ZXP Installer* della piattaforma *aescripts.com* [21], e-commerce dedicato alla compravendita di estensioni, scripts e plug-ins per programmi Adobe (soprattutto After Effects, com'è intuibile dal nome del sito).

Nel prossimo capitolo verranno analizzati i risultati dell'impiego dell'estensione in un'effettiva pipeline di animazione, traendo le conclusioni su un eventuale miglioramento della stessa.

Capitolo 4

Risultati sul campo

Con questo capitolo si concluderà l'intero ciclo di creazione di un'estensione, nonché la tesi stessa. Siamo dunque arrivati al momento di verificare l'effettiva validità dell'estensione come soluzione proposta relativamente al problema identificato durante la produzione del cortometraggio *Milkoffee*, di cui però è stata riconosciuta un'importanza di carattere generale e non specifica di questa particolare produzione. Per chiarezza si riepilogherà nel primo paragrafo del capitolo in cosa consista il suddetto problema; successivamente verranno avanzate le modalità dei test proposti ai partecipanti; a questo farà seguito l'esposizione dei risultati con relative considerazioni finali.

4.1 Il Problema

Il problema dal quale è nata la necessità di questa ricerca si può riassumere nella *mancata compatibilità di Animate con il resto della suite di programmi Adobe, in particolare con After Effects*. Ciò di cui si sta parlando è la *mancanza di flessibilità* nell'utilizzo dei file di progetto di Animate (file *FLA*) all'interno di After Effects; la flessibilità a cui si fa

riferimento sarebbe rappresentata dalla possibilità di importare direttamente il file FLA in After Effects e avere la facoltà di dividere il progetto nei livelli che lo compongono. Se è pur vero che almeno la prima parte del problema trovi una soluzione grazie all'esportazione del progetto in formato SWF, tramite il quale è possibile mantenere le capacità vettoriali del file FLA, è anche vero che questo rappresenti un passaggio in più nel flusso di lavoro, senza considerare la necessità di memoria della macchina per l'esportazione del file. E oltretutto sarebbe appunto una soluzione per la sola prima parte del problema: la divisione dei livelli rimane insoluta.

Adobe ha comunque recentemente rilasciato, nell'ultima versione di After Effects (CC 2019), un aggiornamento per aggirare questo problema; questa soluzione consiste in una *finta* importazione del file FLA: selezionando il file di progetto da importare, After Effects automatizza l'esportazione di ogni livello del progetto in file SWF separati, collocati in una cartella adiacente al file di progetto di After Effects. Come è già stato discusso però, questa rappresenta una soluzione valida ma non del tutto efficiente, in quanto se da un lato ottimizza il flusso di lavoro, dall'altro va a diminuire la memoria disponibile sulla macchina. A questo vanno ad aggiungersi tutti i problemi relativi alla condizione di avere una soluzione *statica*, ovvero incapace di aggiornarsi in caso di modifiche al progetto: se un livello venisse modificato, non ci sarebbe modo di aggiornare il file corrispondente su After Effects se non riesportandolo.

La soluzione qui proposta consiste quindi nello sfruttare appieno le capacità vettoriali di entrambi i programmi per abbattere la richiesta di risorse sia in termini di tempo che di memoria della macchina. Inoltre rappresenterebbe una soluzione *dinamica*, al contrario di quella ufficiale Adobe: si avrebbe un interfacciamento dei due programmi molto più diretto e solido, con la possibilità di aggiornare i livelli da un programma all'altro senza sforzo né spreco di risorse (è pur vero che questa capacità non sia stata approfondita più di tanto durante la trattazione, essendo questa una prima versione del progetto, ma non è difficile intuire le reali potenzialità di questa soluzione dopo tutto quanto approfondito nei precedenti capitoli).

4.2 Modalità dei test

Essendo stata pensata primariamente per migliorare il *workflow*, la verifica dell'efficienza dell'estensione va affrontata innanzitutto *confrontando lo scarto di tempo* nell'eseguire la stessa azione con o senza il suo impiego. L'azione in questione è proprio quella di esportare separatamente i livelli di un progetto Animate. Chiaramente, per l'oggettività dei dati, il progetto e il suo contenuto dovrà essere lo stesso in entrambe le fasi del test (con e senza estensione). Questo test è quindi stato ripetuto per 7 utenti partecipanti. In un unico caso tra i 7 proposti inoltre, è stato possibile confrontare l'impiego dell'estensione con la soluzione proposta da Adobe. In questo caso, oltre che sul confronto dei tempi d'esecuzione dell'azione proposta, sono state tratte delle considerazioni sull'*impiego della memoria* e sulla *possibilità di modifica* del progetto FLA iniziale in caso di esportazione già effettuata.

Vanno inoltre precisate quali sono le corrispondenze di inizio e fine procedura nelle due fasi di test: per l'esportazione senza estensione, poiché per esportare ogni livello separatamente è necessario *nascondere* volta per volta i livelli non interessati tramite abilitazione dell'*hide layer* (icona dell'occhio presente sulla *timeline*), è stato considerato come inizio del test il clic sulla prima icona *hide layer* per l'esportazione del primo livello; è stata invece considerata come fine del test l'ultima importazione nella timeline di After Effects di un livello esportato da Animate. Per l'esportazione con estensione invece è stato considerato come inizio del test il clic sul primo livello selezionato per l'esportazione, mentre è stata considerata come fine la conclusione dell'importazione dei dati vettoriali.

4.3 Risultati

In Figura 4.1 sono presentati i risultati dei test effettuati con i 7 partecipanti:

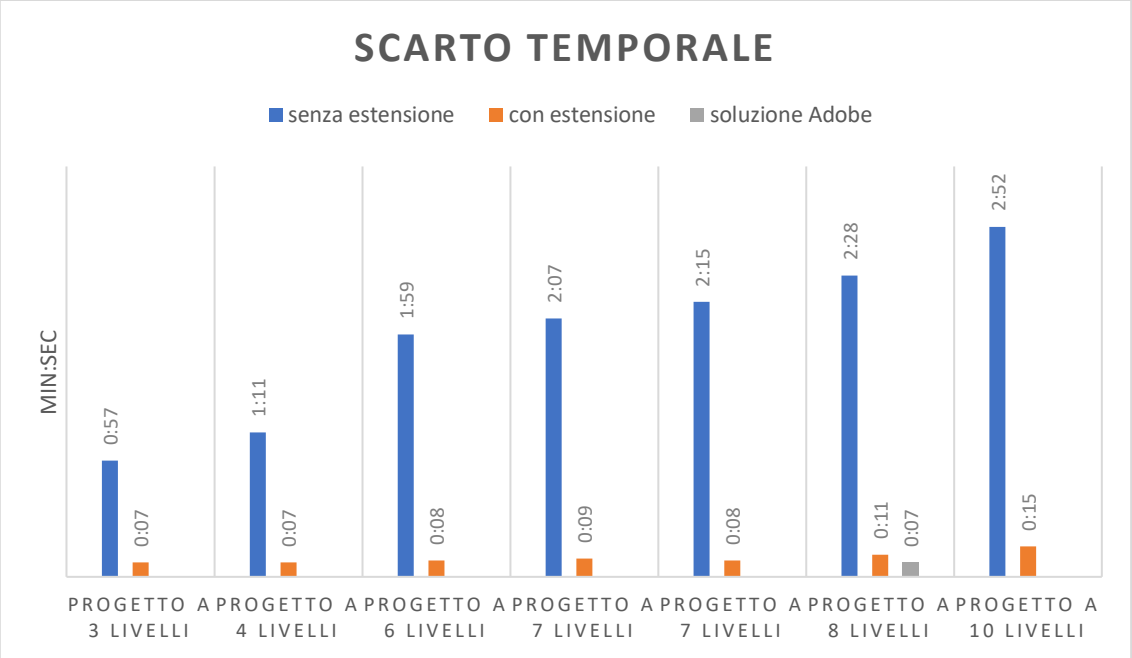

Figura 4.1 - scarti temporali nell'esecuzione di export dei livelli separati di un progetto Animate con e senza estensione. Da notare nel sesto test la presenza del tempo preso con la soluzione Adobe

Se già non fosse stata abbastanza esplicita nei suoi supposti di base, dai risultati risulta evidente la superiorità dell'estensione rispetto la procedura manuale. Da notare la grande differenza nei tempi di fasi uguali in test diversi; questo è dovuto, oltre che alla manualità soggettiva con il software e quindi avente un certo errore umano intrinseco, anche alla quantità di livelli di esportare: aumentando i livelli da nascondere volta per volta, va ad aumentare esponenzialmente anche il tempo di esportazione complessivo. I tempi di esportazione con estensione invece subiscono delle fluttuazioni a seconda della quantità di livelli, dovute alla potenza di calcolo della macchina: a maggior numeri di livelli equivale più tempo di attesa nell'estrazione dei dati vettoriali; non sarebbe errato

supporre quindi che ad una maggiore potenza di calcolo equivarrebbero degli export più veloci.

Si prenda in esame infine il sesto test, in cui è stato possibile confrontare anche la soluzione proposta da Adobe: il tempo di esportazione è di poco inferiore rispetto l'utilizzo dell'estensione (4 secondi); a questo punto però è stato preso in considerazione anche tutto ciò che comporta la soluzione Adobe, ovvero il fatto di generare dei file *statici*. È stato quindi chiesto di *modificare* il progetto, nello specifico aggiungendo un solo fotogramma alla fine di un livello ed è qui che la soluzione Adobe ha mostrato delle lacune: per aggiornare il progetto su After Effects non c'è stato altro modo se non reimportare l'intero file FLA, generando nuovi file e perdendo tempo nella riorganizzazione del progetto per la cancellazione dei file superflui (nonché per eliminarli dalla macchina); con l'estensione invece questa procedura è risultata molto più *fluida*, potendo scegliere quale livello esportare (cosa non concessa dalla soluzione Adobe) e aggiornare con molta più facilità il progetto After Effects.

È pur vero che rimane una questione di punti di vista, in quanto a seconda della situazione un professionista potrebbe preferire l'una o l'altra soluzione. I candidati che infatti non hanno esplicitamente espresso di preferire l'utilizzo dell'estensione, hanno comunque confermato che a seconda dei casi potrebbero prenderne in considerazione l'impiego nonostante la soluzione Adobe.

Conclusioni

Volendo riassumere quanto affrontato in questo lavoro di tesi, questa si configurerebbe come un *percorso* atto innanzitutto a fornire una visione sulle possibilità applicative delle competenze di un ingegnere del Cinema: è mia convinzione infatti che l'analisi della figura professionale rappresentata dal *Technical Director*, con le sue affinità con il profilo dell'ingegnere del Cinema, seppur non necessaria ai fini del progetto di tesi, ne rappresenti un valore aggiunto.

Si ritiene inoltre che la cura dedicata al processo di documentazione di tutte le fasi di ricerca e sviluppo del progetto abbia anch'essa particolare valore, avendo avuto modo di verificare delle relative mancanze nell'ecosistema Adobe di fornire una *guida* univocamente riconosciuta nell'ambito dello sviluppo per i propri software: questo è un campo relativamente nuovo e inesplorato ai più ed è stato quindi obiettivo di questa tesi fornire delle conoscenze *manualistiche* per rendere il più accessibile possibile le informazioni espresse.

La fine di questo percorso ha portato alla realizzazione della prima versione di un'estensione CEP che si prefigge come obiettivo quello della risoluzione dei problemi d'interazione tra i programmi Animate e After Effects. A seguito dei test riportati si può dire senza ombra di dubbio di aver apportato un effettivo miglioramento nel panorama di lavoro con questi due software. Quindi, comprovata l'efficacia dell'estensione, viene da interrogarsi sul futuro che questa possa avere.

Innanzitutto, un futuro nel proprio miglioramento: non si concluderà infatti qui lo sviluppo dell'estensione, la quale ha appena scalfito la superficie delle proprie potenzialità; i primi miglioramenti prospettati riguardano innanzitutto l'inclusione dei

simboli e dei *testi* del DOM di Animate tra i tipi di oggetti esportabili dall'estensione; successivamente verrà anche aperta la via di comunicazione inversa a quella attuale, permettendo il trasferimento dei dati da After Effects a Animate, e aumentando così ulteriormente la *fluidità* e *dinamicità* del flusso di lavoro tra i due software. E molto altro è in mente per l'evoluzione del progetto.

Ma questa evoluzione probabilmente non verrà affrontata da solo: durante tutto lo svolgimento del progetto ci si è infatti interrogati più volte, oltre che sul futuro *operativo* dell'estensione, quale sarebbe stato il suo futuro *distributivo*. Il panorama delle estensioni CEP offre infatti dei risvolti economici non indifferenti: il mercato esiste ed è concreto, ed uno sviluppatore affermato potrebbe tranquillamente dedicarcisi tutta la vita. Basti pensare alla già citata piattaforma *aescripts+aeplugins*, l'e-commerce più vasto e sempre in attività di compravendita di estensioni, scripts e plug-ins Adobe. Convinto della validità del progetto, ci si poteva quindi prospettare un rientro economico dell'estensione.

Ma così facendo, non si invaliderebbe inevitabilmente quanto approfondito sull'*open source*? Dopotutto, qual è stato il fine dello sviluppo di questo progetto: il profitto o l'apporto d'innovazione? La vendita dell'estensione rappresenterebbe inevitabilmente un muro nella sua distribuzione. Ed è volontà primaria del progetto quella di *affermarsi* nella *community*. Non è forse compito di ogni sviluppatore supportare la crescita e l'apertura della *community* di cui fa parte?

La ricerca sull'*open source* ha infatti radicalmente cambiato il modo di intendere i software con cui lavoro; dopo una tale ricerca, non si può fare altro che abbracciare la filosofia *open source*: i software, gli strumenti, non devono rappresentare il fine ultimo della ricerca, ma devono essere messi a disposizione di tutti affinché siano latori di nuova creatività.

E tornando all'evoluzione del progetto, l'apertura gioverebbe anche al progetto stesso, in quanto riceverebbe un aiuto e un sostegno che potrebbero anche portare a svolte inaspettate in futuro.

È volontà ultima di questo progetto di tesi quindi, una volta preparata una prima versione stabile dell'estensione, la sua pubblicazione in *open source*. E questa è, a conti fatti, l'unica degna conclusione della ricerca.

Bibliografia

[1] L. Manovich, «The New Language of Cinema,» in *The Language of New Media*, 2001.

[2] Blackmagic Design, «DaVinci Resolve 16 | Blackmagic Design,» Blackmagic Design, 2019. [Online]. Available: https://www.blackmagicdesign.com/products/davinciresolve/. [Consultato il giorno 7 Ottobre 2019].

[3] Blender Foundation, «Home of the Blender project - Free and Open 3D Creation Software,» Blender Foundation, 2019. [Online]. Available: https://www.blender.org/. [Consultato il giorno 7 Ottobre 2019].

[4] Imageworks, «Sony Pictures Imageworks - Open Source,» Sony Pictures Imageworks, 2019. [Online]. Available: http://opensource.imageworks.com/. [Consultato il giorno 7 Ottobre 2019].

[5] A. S. Foundation, «About - ASWF,» ASWF, 2018. [Online]. Available: https://www.aswf.io/about/. [Consultato il giorno 7 Ottobre 2019].

[6] M. Asay, «How and Why Adobe is Making Open Source A Strategic Priority,» 15 Maggio 2019. [Online]. Available: https://theblog.adobe.com/how-and-why-adobe-is-making-open-source-a-strategic-priority/. [Consultato il giorno 7 Ottobre 2019].

[7] A. Mentor, «What does a Technical Director in Animation do? Blue Sky's Prashanth Pandurangaiah tells us!,» Animation Mentor, 10 Maggio 2017. [Online]. Available: https://blog.animationmentor.com/what-does-a-technical-director-in-animation-do-blue-skys-prashanth-pandurangaiah-tells-us/. [Consultato il giorno 7 Ottobre 2019].

[8] J. Wolfe, «'Piper' and the Development of Pixar's Presto Sculpting Brush,» 5 Febbraio 2017. [Online]. Available: https://www.awn.com/animationworld/piper-and-development-pixar-s-presto-sculpting-brush. [Consultato il giorno 7 Ottobre 2019].

[9] D. Coders, «LOVE MOVIES? LEARN TO CODE PYTHON AND YOU MIGHT WORK FOR ILM,» Davinci Coders, 10 Febbraio 2017. [Online]. Available: https://www.davincicoders.com/codingblog/2017/2/10/love-movies-learn-to-code-python-and-you-might-work-for-ilm. [Consultato il giorno 7 Ottobre 2019].

[10] B. Pohl, «Virtual Production with Unreal Engine: A New Era of Filmmaking,» Epic Games, 25 Ottobre 2018. [Online]. Available: https://www.unrealengine.com/en-US/blog/virtual-production-with-unreal-engine-a-new-era-of-filmmaking. [Consultato il giorno 7 Ottobre 2019].

[11] Adobe Inc., «Adobe Support Community,» Adobe Inc., 2019. [Online]. Available: https://community.adobe.com/. [Consultato il giorno 7 Ottobre 2019].

[12] Adobe Inc., «Adobe Help Center,» Adobe Inc., 2019. [Online]. Available: https://helpx.adobe.com/support.html. [Consultato il giorno 7 Ottobre 2019].

[13] Adobe Inc., «Adobe I/O,» Adobe Inc., 2019. [Online]. Available: https://www.adobe.io/. [Consultato il giorno 7 Ottobre 2019].

[14] Adobe Inc., «After Effects Scripting Guide,» Adobe Inc., 2019. [Online]. Available: https://buildmedia.readthedocs.org/media/pdf/after-effects-scripting-guide/latest/after-effects-scripting-guide.pdf#page=261. [Consultato il giorno 7 Ottobre 2019].

[15] Adobe Inc., «Shape Layer Match Names — After Effects Scripting Guide 0.0.1 documentation,» Adobe Inc., 2019. [Online]. Available: http://docs.aenhancers.com/matchnames/layer/shapelayer/. [Consultato il giorno 7 Ottobre 2019].

[16] Adobe Inc., «GitHub - Adobe-CEP/Samples: Code samples for CEP extensions,» Adobe Inc., 2019. [Online]. Available: https://github.com/Adobe-CEP/Samples. [Consultato il giorno 7 Ottobre 2019].

[17] A. Hall, «cep-sample-extension/testExtension at master · andyhall/cep-sample-extension · GitHub,» 2019. [Online]. Available: https://github.com/andyhall/cep-sample-extension/tree/master/testExtension. [Consultato il giorno 7 Ottobre 2019].

[18] A. Hall, «//andy hall | chatter from an indie dev in Tokyo,» Andy Hall, 2019. [Online]. Available: http://aphall.com/. [Consultato il giorno 7 Ottobre 2019].

[19] Adobe Inc., «Packaging and Signing Adobe Extensions - How signing works,» Adobe Inc., 2019. [Online]. Available: http://wwwimages.adobe.com/www.adobe.com/content/dam/acom/en/devnet/creativesuite/pdfs/SigningTechNote_CC.pdf#page=5. [Consultato il giorno 7 Ottobre 2019].

[20] Adobe Inc., «Adobe - Exchange : Download the Adobe Extension Manager,» Adobe Inc., 2019. [Online]. Available: https://www.adobe.com/exchange/em_download/. [Consultato il giorno 7 Ottobre 2019].

[21] aescripts + aeplugins, «ZXP Installer - aescripts + aeplugins - aescripts.com,» aescripts + aeplugins, 26 Aprile 2016. [Online]. Available: https://aescripts.com/learn/zxp-installer/. [Consultato il giorno 7 Ottobre 2019].