Alma Mater Studiorum - Univerità di Bologna

facoltà di scienze matematiche fisiche e naturali
Corso di Laurea in Informatica

Studio e valutazione di sistemi virtuali in JavaScript

Relatore:
Char.mo Prof.
Renzo Davoli

Presentata da:
Marco Melletti

II Sessione
Anno Accademico 2010-2011

Indice

Capitolo 1

Introduzione

Chiunque abbia compilato un form online, navigato pagine web con effetti di trasparenza guidati dal mouse, o semplicemente utilizzato google, ha avuto a che fare con *JavaScript*.

Questo linguaggio è parte integrante dei browser, e viene impiegato dai progettisti di pagine web ormai da quindici anni. Creato per estendere le possibilità grafiche e di controllo di HTML, con l'evolversi dei browser, JavaScript è cresciuto fino a diventare un vero e proprio linguaggio di programmazione, tanto che, recentemente, *Fabrice Bellard*, un noto sviluppatore francese, ha creato una macchina virtuale utilizzando solo HTML e JavaScript.

Jslinux[1] è il nome del software, che si propone di emulare l'hardware di un calcolatore di qualche anno fa, con un processore della famiglia Intell *i386*.

Negli ultimi anni sono comparsi nella rete numerosi emulatori realizzati con gli stessi principi. Per comprendere appieno come sia possibile realizzare un simile programma, si è quindi scelto di studiare, in parallelo, un emulatore di GameBoy, console portatile composta da un hardware decisamente più contenuto. Il software a cui si fa riferimento si chiama *JSGB*[2].

Durante lo studio delle due macchine virtuali si è incappati in lacune di vario tipo, ma principalmente dettate dalla scarsa compatibilità del linguaggio con gli obbiettivi da conseguire. Per due di queste lacune sono state proposte alcune possibili soluzioni.

Lo scopo ultimo di questo studio è stabilire se e come sia possibile realizzare in JavaScript altre macchine virtuali complete, per realizzare il progetto, ideato dal Professor Renzo Davoli, di creare un museo del software online in cui ognuno possa provare a piacimento vecchie macchine, sistemi operativi e programmi.

[1] http://bellard.org/jslinux/
[2] http://codebase.es/jsgb/

Capitolo 2

Stato dell'arte

2.1 JavaScript: un linguaggio in evoluzione

JavaScript nacque nel 1995 come linguaggio di scripting implementato da *Netscape Navigator 2.0*. Le pagine HTML dell'epoca soffrivano di una scarsa dinamicità, erano dei documenti ipertestuali in cui presenziavano soltanto testo ed immagini, l'interattività si esauriva nel caricamento di un'altra pagina, ed il movimento era espresso soltanto dalle gif animate. JavaScript si proponeva di ovviare a queste limitazioni.

Questo linguaggio inizialmente esibiva un limitato raggio di possibilità: veniva utilizzato principalmente per semplici effetti grafici, come le scritte che seguono il puntatore, o per controllare, ancora prima di inviare un form, che i valori immessi fossero conformi a quanto richiesto, oppure ancora per disabilitare il tasto destro del mouse o l'input da tastiera.

Con il passare del tempo e l'evolversi dei web browser, anche JavaScript è cresciuto, aggiungendo metodi per modificare dinamicamente il contenuto di un documento HTML (convenzionalmente denominati metodi *DOM*[1]), per comunicare con il server ed aumentare l'interattività delle pagine web, arrivando all'attuale *AJAX*. Questo termine sta ad indicare una tecnica di sviluppo per realizzare applicazioni web realmente interattive ed asincrone: basato su JavaScript e XML (l'acronimo sta per *Asynchronous JavaScript and XML*) AJAX è implementato in molteplici forme, per citarne una su tutte il *Google Web Toolkit*. Tutte hanno in comune uno scambio di dati asincrono tra client e server, che permette di aggiornare una pagina web modificandone, anche sostanzialmente, i contenuti senza dover esplicitamente ricaricare la suddetta pagina.

Il concetto di asincronia è fondamentale per queste implementazioni, per-

[1] Document Object Model

mette infatti al server di gestire contemporaneamente un numero teoricamente indeterminato di applicazioni che richiedono dati solo quando se ne presenta l'effettiva necessità. Tutto il carico di lavoro estraneo alla trasmissione dei dati è lasciato alla macchina del client.

JavaScript è, ad oggi, alla sua versione 1.5 ed ha superato già da tempo lo status di semplice linguaggio di scripting, guadagnando la definizione di "*linguaggio di scripting orientato agli oggetti*". Ciò significa che si è "elevato" al rango di linguaggi come C++ o Java offrendo possibiltà che vanno ben oltre gli effetti grafici o i controlli sui valori immessi: consente infatti di creare oggetti complessi che contengono dati, metodi o, a loro volta, oggetti. Affiancando questo potente linguaggio ad un suo simile in grado di generare un'interfaccia, anch'esso in costante evoluzione (quale è HTML), è possibile oggi sviluppare applicazioni complesse visualizzabili attraverso il solo browser, uno dei software attualmente di più comune utilizzo. Si superano facilmente, in questo modo, molti dei possibili problemi di portabilità che affliggono i linguaggi "classici", fatta eccezione ovviamente per Java.

I due programmi presi in esame di seguito sono un chiaro esempio di quanto questo linguaggio sia potente e flessibile.

2.2 JSGB & Jslinux

Questi due software sono attualmente fra i migliori esempi di macchine virtuali realizzate in JavaScript. Propongono infatti un ottimo compromesso tra efficienza e funzionalità messe a disposizione, da un lato un emulatore di *GameBoy* completamente funzionante, *JSGB* (di Pedro Ladaria), dall'altro una macchina virtuale in grado di emulare un calcolatore con processore *Intell i386*, che esegue una distribuzione aggiornata di Linux: *Jslinux* (di Fabrice Bellard).

Nessuno dei due programmi, comunque, è completo: l'audio difatti è assente in entrambi, e per quanto riguarda Jslinux, mancano anche l'interfaccia di rete e una scheda grafica, la macchia virtuale permette infatti la visualizzazione di soli caratteri.

2.2.1 Come si presentano

Trattandosi di applicazioni scritte in JavaScript, l'unica maniera per "visualizzarle" è inserirle in una pagina html e utilizzare un browser per aprirla. Ciò permette di sfruttare elementi tipici delle pagine web nell'implementazione dell'interfaccia, come le *dropbox* o le semplici *textbox*, per dialogare con l'utente e selezionare opzioni collaterali all'utilizzo della macchina virtuale.

Si trovano, difatti, affianco ad entrambe gli schermi emulati dai sue sistemi, alcuni oggetti appartenenti all'HTML puro.

Per esempio, in JSGB, la scelta del software da far eseguire all'emulatore, che deve essere effettuata prima dell'avvio dell'emulatore stesso, è realizzata usando il tag html <select> e genera una dropbox che (ri)avvia l'emulazione ogni volta che l'utente sceglie un elemento.

Un esempio di natura differente è quello presente in Jslinux: una textarea è presente di fianco allo "schermo" del terminale e serve come clipboard per gli appunti condivisa. In questo caso la macchina virtuale non ha accesso agli appunti del sistema operativo ospite, per ovviare a questa limitazione è stato dunque resa possibile la copia degli appunti nell'area di testo, che a sua volta li invia a Jslinux.

2.2.2 Il problema dell'audio

L'audio è un tipo di media che si presta poco ad essere generato da un emulatore di questo tipo. È infatti molto semplice riprodurre suoni campionati su un browser, ma far sì che questo ne generi di nuovi secondo particolari specifiche è assai più arduo.

Nel seguito sarà approfondito l'argomento (vedi Cap.3.1).

2.3 Implementazioni a confronto

I due esempi trattati mostrano approcci totalmente differenti nell'implementazione, sia dal punto di vista stilistico, sia per le scelte fatte nella programmazione vera e propria di alcune parti comuni ad entrambi gli emulatori.

Jsgb mostra un approccio più funzionale, meno legato all'hardware originale e un codice più pulito con riferimenti diretti a variabili dichiarate altrove. Ricorda quasi il codice scritto in C.

Jslinux invece non contiene quasi nessuna variabile mirror, fa largo uso di variabili dai nomi poco significativi (il codice in molte parti sembra offuscato) e del comando bind, tipico di JavaScript.

2.3.1 JavaScript `.bind()`

Essendo JavaScript prima di tutto un linguaggio di scripting, quindi completamente interpretato, tende ad avere una gestione dello scope abbastanza lasca. A differenza da C, Java o gli altri maggiori linguaggi di programmazione orientati agli oggetti, in JavaScript lo scope di una funzione, il contesto

cioè in cui viene eseguita, non dipende dal contesto in cui la funzione è definita, ma da quello da cui viene invocata. Questo permette ad una sola funzione di tenere comportamenti differenti in base al contesto, o di avere codominio di dimensione indefinita. Impedisce però di avere veri e propri oggetti che applicano un metodo richiesto da altre entità sfruttando le proprie informazioni interne.

Per offrire controllo sullo scope delle funzioni, JavaScript implementa tre metodi:

`Function.apply(scope, argV)`

invoca la funzione impostando come ambiente di esecuzione il suo primo parametro, e passando alla funzione i parametri contenuti in `argV`, che, se presente, deve essere un vettore;

`Function.call(scope, arg1, arg2, arg3, ...)`

come `apply()` invoca la funzione impostando come environment il primo parametro che riceve, ma gli altri oggetti da passare alla funzione sono i parametri successivi, non un vettore;

`Function.bind(scope, arg1, arg2, arg3, ...)`

questo metodo associa (fino all'invocazione di un altra `bind()`) uno scope alla funzione su cui viene invocato, accetta un numero indefinito di parametri, di cui il primo identifica l'ambiente ed i successivi saranno passati alla funzione come parametri prima di quelli passati dal contesto di esecuzione.

Jslinux fa largo uso del metodo `bind()` per associare gli scope dell'oggetto cpu o dell'mmu od altri alle funzioni che vengono istanziate in tutto il suo codice.

JSGB invece definisce tutte le variabili globali all'inizio dei files che compongono il programma, in modo che siano sempre visibili, e le funzioni le utilizzano direttamente.

2.3.2 Modularità

Al contrario di quanto ci si potrebbe aspettare, Jslinux, che vanta il codice più esteso e complesso, oltre alle pretese più alte, non esibisce una grossa modularizzazione. Infatti le uniche divisioni nette sono tra l'emulatore vero e proprio, il programma che si occupa dell'emulazione di terminale e l'inizializzazione di tutto il sistema. Questo rende difficile l'ampliamento del codice

principale e l'aggiunta di nuovi device. Vero è, che all'interno del codice, non vengono risparmiate le funzioni accessorie che preparano soltanto i parametri. Anche la distinzione tra un device e l'altro è sufficientemente chiara.

Approccio opposto è quello dell'emulatore di GameBoy, strutturato in un insieme di files che descrivono ogni componente dell'hardware emulato, a partire dalla cpu, l'mmu, il controller degli interrupt, lo schermo, addirittura il timer di sistema.

2.4 L'inizializzazione

2.4.1 JSGB

Come risulterà evidente, il GamBoy ha un hardware abbastanza semplice da comprendere e da emulare; ogni componente richiede un'inizializzazione che preparari le strutture interne come registri e memoria, ma nulla di particolarmente complesso.

Caricando la pagina contenente il software di JSGB, lo script si limita a creare delle funzioni associate ai tasti di avvio e arresto dell'emulazione ed un handler per la *dropbox* delle cartucce. Quest'ultimo gestore è quello che da il via alla macchina virtuale, prima avviando l'inizializzazione, poi lanciando l'esecuzione vera e propria.

Ogni volta che viene selezionata una cartuccia dall'elenco, viene invocato il metodo `gb_Insert_Cartridge(fileName, Start)` che prima di tutto ferma l'emulazione se questa è attiva, poi procede a (ri)inizializzare la macchina: vengono riportati a zero i contatori di tempo e frames trascorsi, poi vengono lanciate le funzioni di init dei moduli che compongono l'emulatore, infine viene caricata la cartuccia in memoria ed avviata l'esecuzione.

`jsgb.debugger.js.gb_Init_Debugger()`

Il primo ad essere inizializzato è il modulo di debug, che, se attivo, stampa il contenuto della memoria e dei registri su una table, creata durante la sua inizializzazione. Il metodo non ha altri scopi.

`jsgb.memory.js.gb_Init_Memory()`

Subito dopo tocca alla *MMU*, qui l'inizializzazione si occupa di "vuotare" la mermoria, impostandone tutti i bit a 0, e di scrivere alcuni valori di default nelle posizioni dei registri speciali, principalmente dedicati all'unità di elaborazione grafica, che è la prossima ad essere preparata.

`jsgb.lcd.js.gb_Init_LCD()`

Il blocco di istruzioni più interessante di questo modulo è il primo, quello cioè che recupera informazioni sull'oggetto *canvas*, entità che permette di disegnare a video il contenuto di un *framebuffer*, un contenitore di informazioni su un'immagine, concettualmente molto simile ad una bitmap. La prima cosa da fare è ottenere un collegamento al *canvas*, dopo di ché è importante ottenere il *graphic context* dell'oggetto, che permette di operare realmente con le immagini, di impostarne le dimensioni, gli schemi di colore e di disegnarle sullo schermo. In ultimo, vengono inizializzate le strutture per l'elaborazione delle immagini interne alla *GPU*, quelle accessorie aggiunte nell'emulatore, come il *framebuffer* prima citato, ed alcuni array che specificano quali oggetti grafici debbano essere aggiornati nei vettori specchio della memoria grafica, creati anch'essi nello stesso blocco di istruzioni.

`jsgb.interrupts.js.gb_Init_Interrupts()`

Il metodo associa ad ogni posizione del vettore degli interrupt (che qui è letteralmente implementato attraverso un array) la funzione che risolve la relativa interruzione: al bit 0 dell'interrupt flag impostato, e quindi al valore 1 del flag corrisponde la prima funzione, al bit 1 impostato, valore 2, corrisponde la seconda, al bit 2 impostato, valore 4, corrisponde la terza e così via.

Questa implementazione permette di chiamare direttamente la funzione corretta basandosi sul valore totale del flag degli interrupt pendenti, risparmiando letture in memoria e operazioni che in un hardware emulato possono risultare ridondanti.

`jsgb.cpu.js.gb_Init_CPU()`

Resetta a valori base i registri del processore, il contatore dei cicli di cpu ed i registri mirror che contengono informazioni sul tipo di cartuccia e gestore della memoria utilizzati (il GameBoy ha più metodi di gestione della memoria, poiché questa è molto limitata dalle specifiche hardware.). L'emulazione viene messa in pausa al termine questa sezione, per essere riavviata ad inizializzazione terminata.

`jsgb.input.js.gb_Init_Input()`

Molto semplice è anche questa funzione, che associa ai due hander degli eventi della tastiera `document.onkeydown` e `document.onkeyup` le due funzioni qui

definite `gb_OnKeyDown_Event(e)` e `gb_OnKeyUp_Event(e)`. Questi metodi aggiornano, in base al tasto premuto, i due registri dello stato dell'input.

Ultima fase

Per concludere viene scaricato il codice della cartuccia e quindi caricato nella memoria fisica dalla funzione `gb_ROM_Load(filename)`. Viene infine avviata l'esecuzione.

`XMLHttpRequest()`

Il metodo sopracitato utilizza una richiesta del tipo `XMLHttpRequest()`, la più comunemente usata per effettuare richieste asincrone di dati al server. Il metodo può effettuare richieste *REST* di ogni tipo, in questo caso è sufficiente una *GET* che restituisca in plain text il contenuto di un eseguibile. È il metodo migliore tra quelli attualmente disponibili per caricare blocchi di dati in un'applicazione JavaScript.

2.4.2 Jslinux

Questa macchina virtuale emula il comportamento di un hardware decisamente più complesso di quello precedentemente preso in esame, presenta quindi un'inizializzazione abbastanza estesa, ma non per questo difficile da comprendere. Questa sequenza di istruzioni si occupa di "collegare" la macchina virtuale alla sua interfaccia (il terminale) e di preparare il sistema operativo per essere eseguito.

Al caricamento della pagina viene istanziato un oggetto che conterrà i parametri di inizializzazione dell'emulatore. Vengono quindi inseriti in quest'oggetto il riferimento alla funzione che stampa caratteri sul terminale, la dimensione della memoria fisica dell'emulatore, l'handle per il metodo che restituisce il boot time e le due funzioni per leggere e scrivere sulla clipboard condivisa con il browser.

Jslinux.js:

```
59    /* serial output chars */
60    params.serial_write = term.write.bind(term);
61    /* memory size (in bytes) */
62    params.mem_size = 16 * 1024 * 1024;
63    /* cipboard I/O */
64    params.clipboard_get = clipboard_get;
65    params.clipboard_set = clipboard_set;
66    params.get_boot_time = get_boot_time;
67    pc = new PCEmulator(params);
```

Può quindi partire, con l'ultima istruzione, l'inizializzazione della macchina virtuale vera e propria, divisa in più parti, una per ogni device: prima di tutte è la cpu, che viene preparata istanziando un nuovo oggetto del tipo

`CPU_X86`

Questa sequenza di init si occupa di istanziare le variabili che emulano i registri, le tables ed i buffer della cpu.

Prima i quattro registri fondamentali che risiedono nel vettore `regs[]`:

- `regs[0]` -> *eax* (registro accumulatore),
- `regs[1]` -> *ecx* (registro contatore),
- `regs[2]` -> *edx* (data register),
- `regs[3]` -> *ebx* (base register),

e l'*instruction pointer* (`eip`).

Segue una serie di altri registri della cpu: i primi cinque (`op`, `dst`, `src`, `op2` e `dst2`) vengono utilizzati durante le interpretazioni dei codici istruzione, `op` e `op2` rappresentano il codice principale e quello secondario, se l'istruzione è a 2 bytes, i due `dst` sono i registri di destinazione e `src` quello di origine, quindi gli operandi delle operazioni.

Il *direction flag* (`df`) specifica in che direzione devono essere processate le stringhe (se il flag è impostato le stringe vengono lette da sinistra a destra, nel senso opposte se il suo valore è 0).

Il registro dei *flags* (`eflags`) viene inizializzato a `0x2`, valore che rappresenta il *Parity flag* impostato, indica cioè che l'ultima operazione aritmetica ha dato un risultato con un numero dispari di bit impostati a 1.

Il *contatore dei cicli* della cpu (`cycle_count`), un flag che indica se ci sono o meno interrupt pendenti (`hard_irq`) ed il registro contenente il codice della prossima richiesta di interrupt da risolvere (`hard_intno`).

Il valore del *livello di privilegi* corrente (`cpl`): più basso il valore più ampi i privilegi, se impostato a 0 ci si trova in kernel mode e si ha accesso a tutte le risorse, 2 invece è il livello più comunemente utilizzato per lo user mode.

I *control registers* (`cr0-4`), registri che definiscono lo stato di svariate risorse, di default vengono azzerati e settato il primo bit del registro `cr0`: così facendo si imposta la gestione della memoria in modalità protetta (se il bit viene lasciato a 0 il sistema funziona in real mode).

Vengono infine preparate le tables: l'*interrupt descriptor table* (`idt`) specifica la zona di memoria che contiene il vettore degli interrupt, la *global*

descriptor table (`gdt`) contiene i limiti generali dell'area di memoria segmentabile e la lista dei *segmenti* (`segs`) utilizzati in real mode, che sono gli elementi puntati dalla `gdt`.

Il *task register* (`tr`) punta al (in questo caso contiene il) *task state segment* (`tss`), area di memoria riservata ai descrittori del task corrente: stato del processore, diritti di I/O, stack pointers ed il link al `tss` del task padre.

La *local descriptor table* (`ldt`) contiene le informazioni e i descrittori dei processi utente, ad ogni cambio di esecuzione, cioè quando un processo subentra a quello precedente nell'utilizzo della cpu, questa tabella viene sostituita con quella che descrive il nuovo processo come parte integrante della routine di task switching.

Per finire, lo stato della cpu (`this.halted`), il puntatore alla memoria fisica (`this.phis_mem`), i quattro array del *translation lookaside buffer* (`tlb`), due per la scrittura e due per la lettura in user mode e kernel mode, utilizzati come cache per risolvere gli indirizzi virtuali, il *vettore delle pagine di memoria* (`pages`) e il *page counter* (`pages_count`), che ne indica la dimensione.

L'inizializzazione della macchina virtuale prosegue quindi istanziando la memoria fisica

cpux86-ta.js:

```
7623    cpu.phys_mem_resize(PF.mem_size);
```

CPU_X86.prototype.phys_mem_resize(ea):

```
79      this.mem_size = ea;
80      ea += ((15 + 3) &  3);
81      this.phys_mem = new ArrayBuffer(ea);
82      this.phys_mem8 = new Uint8Array(this.phys_mem, 0, ea);
83      this.phys_mem16 = new Uint16Array(this.phys_mem, 0, ea / 2);
84      this.phys_mem32 = new Int32Array(this.phys_mem, 0, ea / 4);
```

Questa funzione non solo provvede a ridimensionare la memoria, ma la pulisce ogni volta, utilizzando sempre un oggetto nuovo per contenerla. I costruttori `Uint8Array()`, `Uint16Array()` e `Int32Array()` permettono di utilizzare un array esistente come fonte per i dati interpretandolo come se fosse formato rispettivamente da interi a 8, 16 o 32 bit, molto comodo per le differenti rappresentazioni e manipolazioni dei dati in memoria.

Vengono quindi inizializzate le strutture dati contenenti le funzioni per l'I/O. La funzione `init_ioports()` crea sei vettori di 1024 posizioni in cui vengono salvati riferimenti ai metodi per scrivere/leggere bytes, words (2 bytes) o longs (2 words). In ogni posizione di questi vettori vengono salvati i riferimenti a funzioni che restituiscono valori di default indicanti che la porta corrente non è in uso. L'unica porta per cui viene già impostata una funzione differente è la 0x80, per la quale il metodo di scrittura (`ioport80_write()`)

non fa nulla, permettendo così di implementare la pausa dell'I/O (se il processore è troppo veloce rispetto al bus, l'i386 utilizza delle chiamate `OUT B` su questa porta, che è solitamente inutilizzata, mentre aspetta di continuare a mandare dati sul bus).

A questo viene lanciato il blocco di funzioni che inizializzano le "periferiche" il quali il pic (*programmable interrupt controller*), pit (*programmable interval timer*), il *cmos*, la *seriale* (che comunica unicamente col terminale) e la *tastiera*:

Programmable Interrupt Controller (`cpux86-ta.js.qf()`)

Si tratta di un dispositivo che permette di gestire gli inetrrupt con priorità e vettore delle funzioni associate, in questo caso consiste di due controller separati che risiedono su indirizzi differenti del bus (`0x20` e `0xa0`) con *elcr mask* diverse:
`0x20 -> 0 0 0 1 1 1 1 1`
`0xa0 -> 0 1 1 1 1 0 1 1`
queste maschere specificano se ciascuna linea di interrupt debba essere considerata *level triggered* o *edge triggered*, ad ogni bit corrisponde la linea di valore equivalente, da 0 a 7. I due tipi di interrupt si differenziano per il metodo elettronico che viene utilizzato per segnalarli: i level triggered interrupts sono indicati da un particolare stato della linea associata, alto o basso, in base all'implementazione; gli edge triggered sono invece segnalati da una variazione sulla linea, quindi da un impulso che la attraversa.

I due controller separati servono a costruire il modello master-slave dell'intel 8259 (un tipo molto comune di *pic*), la lina 2 di interrupt del controller master (`pics[0]`) fa si che venga risolto un interrupt sull controller slave (`pics[1]`). Nella pratica di in questo caso, la cpu non considera separati i due controller, assume che ce ne sia uno solo con 16 linee. Al momento di risolvere una richiesta è lo stesso controller che si occupa di decidere se l'interrupt da risolvere è sul controller master o sullo slave, ed allo stesso modo imposta il registro del processore `hard_irq` al valore della linea su cui è pendente una richiesta, in modo che la cpu possa reagire nella maniera attesa.

L'handle della funzione `cpu_set_irq()` viene impostata in modo da eseguire un metodo globale (`sf(lf)`) che si limita ad impostare il registro `hard_irq` del processore al valore del suo parametro `lf`. Per finire viene settata la funzione di update dell'*irq* dei due *pic* al metodo generale dell'oggetto `qf`.

cpux86-ta.js.qf(.., sf):
```
7359    this.cpu_set_irq = sf;
```

```
7360      this.pics[0].update_irq = this.update_irq.bind(this);
7361      this.pics[1].update_irq = this.update_irq.bind(this);
```

Ogni *pic* contiene tre registri fondamentali: `irr` (*interrupt request register*) specifica quali interrupt devono ancora essere considerati, `isr` (*in-service register*) specifica quali interrupt sono stati riconosciuti ma aspettano ancora un *End Of Interrupt* (EOI), `imr` (*interrupt mask register*) specifica queli interrupt devono essere ignorati.

Vengono anche istanziati altri registri utilizzati per la risoluzione di interrupt multipli (`priority_add`, `auto_eoi`, `rotate_on_autoeoi`) e per il controllo di flags e maschere.

Programmable Intervall Timer (`cpux86-ta.js.uf()`)

È composto da tre timers al raggiungimento del valore massimo lanciano una richiesta di interrupt. Tipicamente disposti come segue:
0 -> *system timer*
1 -> *refresh dRAM*
2 -> *PC speaker.*

I timers sono inizializzati singolarmente nella funzione `xf(wf)` e hanno tutti la possibilità di leggere il "tempo della cpu", cioè il numero di cicli di processore trascorsi dall'avvio. Si basano su questo valore per la temporizzazione interna.

Vengono riservate alcune posizioni del bus per questo device: quattro a partire dalla `0x40` in scrittura e tre, sempre partendo dallo stesso indirizzo, in lettura, ed una coppia in lettura-scrittura soltanto per gli speaker all"indirizzo `0x61`.

Seriale (`cpux86-ta.js.EF()`)

La periferica seriale (così nominata nel codice) implementa la comunicazione con il terminale, contiene i registri che ne descrivono lo stato e un riferimento alla funzione `term.write(char)` che stampa i caratteri a video. La comunicazione nel senso inverso viene "collegata" durante la fase successiva di setup del terminale.

Caricamento del Sistema Operativo e sequenza di boot

Gli ultimi passi del costruttore `PCEmulator(params)` mettono in comunicazione la clipboard della VM con quella del browser e preparano sei funzioni che leggono/scrivono 8, 16 o 32 bits all'indirizzo del bus specificato.

L'esecuzione torna ora in mano alla chiamata iniziale, che esegue le ultime fasi dell'inizializzazione. Sfruttando le funzioni della macchina virtuale

appena creata, vi carica in memoria il sistema operativo (`vmlinux26.bin`), la directory principale (`root.bin`) ed una sequenza di boot (`lnuxstart.bin`), per ultima la stringa di esecuzione del kernel di Linux. La funzione che effettua queste operazioni è la `CPU_X86.load_binary()`. Come per il caso di *JSGB* per caricare i binari sull'emulatore vengono utilizzate richieste `XMLHttpRequest()`, che si conferma il metodo più funzionale per questo scopo.

Terminale (`term.js`)

Ultima operazione del ciclo di inizializzazione è la creazione ed "apertura" del terminale. Per prima cosa vengono salvate le informazioni utili al terminale, le su dimensioni, una coppia di variabili che indica la posizione attuale del cursore ed una serie di altri valori di riferimento, come colori dei caratteri e dello sfondo, stato del cursore, flags dipendenti dal browser e dal sistema operativo e il collegamento alla funzione che gesirà gli eventi della tastiera. Questa funzione decodifica gli eventi generati dalla tastiera e li invia al processore attraverso un metodo del dispositivo seriale della macchina virtuale (`PCEmulator.serial.send_chars(str)`) come eventi della tastiera compatibili con il processore emulato.

All'apertra del terminale (`term.open()`) viene creato un vettore bidimensionale contenente tutti i caratteri attualmente visibili, che verrà aggiornato ogni volta che la macchina virtuale invoca il metodo `term.write()` e letto ad ogni chiamata della `term.refresh()`. Viene quindi creato l'oggeto visibile, che altro non è che una *table Html* con numero di righe pari alle dimensioni verticali del terminale. Ad ogni invocazione della funzione `term.refresh(ka, la)` le righe comprese tra `ka` e `la` vengono aggiornate con il contenuto del vettore dei caratteri.

La macchina virtuale è ora pronta ed avviata, sta già caricando il Sistema Operativo.

2.5 Il flusso di esecuzione

Le due implementazioni mostrano scelte differenti nella realizzazione del ciclo *fetch-decode-execute*, utilizzando comandi differenti per la temporizzazione e per decodificare gli OPCodes.

2.5.1 L'approccio rigoroso

Jslinux mostra un rigore esemplare nel riprodurre in ogni sua componente il funzionamento della cpu: la sezione di codice che si occupa dell'esecuzione vera e propria ne è la miglior dimostrazione. Alla chiamata della funzione `PCEmulator.start()`, viene preparata la funzione `PCEmulator.timer_func()` per essere lanciata dopo dieci millisecondi, attraverso il metodo di JavaScript `setTimeout(funzione, ritardo)`[2].

La funzione *timer_func()* si occupa di temporizzare le azioni della cpu coerentemente alla velocità di esecuzione del processore emulato. Per prima cosa il metodo aggiorna lo stato degli interrupt, quindi risolve una richiesta se presente ed esegue il successivo OPCode presente in memoria, puntato dal *pc* (`eip`), mediante la chiamata al metodo `CPU_X86.exec(cycle_count)`. Questa routine viene ripetuta finché l'*exec* non restituisce un valore pari a 257 o viene settato a true il flag `PCEmulator.reset_request`, oppure fino a quando non viene esaurito un contatore precedentemente impostato al valore 100000. Nel primo caso, fa partire il timer impostando il ritardo a 10 ms; nel caso in cui la routine termini le sue iterazioni imposta il timeout a 0 ms (quindi avvia direttamente il ciclo successivo). Se invece è stato rishiesto un *reset* oppure la funzione `exec` ha restituito un valore diverso da 256 e 257 l'esecuzione si ferma.

Chi si occupa realmente del ciclo caricamento-decodifica-esecuzione è il metodo `CPU_X86.execInternals()`. La funzione `exec` serve per avviare questo metodo e raccogliere eventuali eccezioni lanciate dallo stesso.

`CPU_X86.execInternals()` si compone di numerosi metodi locali utilizzati per effettuare le operazioni richieste dal programma in esecuzione, uno dei quali, (`hd(Db, b)`), è strutturato alla stessa maniera. Si tratta del metodo che gestisce gli OPCodes di 2 bytes, codici particolari che hanno il primo byte impostato al valore `0xF2`, che indica al processore di utilizzare la seconda tabella di codici istruzione, e il secondo byte (che solitamente rappresenta un operando) indica quale operazione sia da svolgere.

Il codice principale della funzione `CPU_X86.execInternals()`, che si trova dopo tutte le definizioni di metodi locali, sceglie se accedere alla memoria in user o kernel mode in base al registro del livello di privilegi (`CPU_X86.cpl`), legge dalla memoria l'istruzione seguente (*fetch*), entra in un *case switch*

[2] `setTimeout(funzione, ritardo)` e `setIntervall(funzione, intervallo)`,citato successivamente, sono due metodi che il linguaggio mette a disposizione per gestire la temporizzazione degli eventi. Entrambi avviano la funzione specificata una volta passato il tempo specificato. Ciò che differenzia setIntervall da setTimeout è la ripetizione di questo processo fino a che non viene esplicitamente fermato da un'esecuzione del metodo *clearIntervall(intervallo)* sull'oggetto intervall.

che scandisce ogni possibile OPCode (*decode*) ed esegue il metodo relativo (*execute*).

2.5.2 L'approccio pratico

Innanzitutto bisogna notare che differentemente da Jslinux, JSGB ha una scansione temporale basata sui frames visualizzati invece che sui cicli di cpu, questo è dovuto principalmente alla natura dell'hardware che emula: il GameBoy è una console utilizzata per eseguire soprattutto videogames, per cui un requisito fondamentale è la corretta velocità delle azioni, quindi della progressione di frames visualizzati. Questo fattore impone un vincolo importante per la temporizzazione dell'emulazione. Difatti la funzione che scandisce il tempo di azione dell'emulatore è `gb_Frame()`.

Quando il pulsante di avvio viene premuto, viene impostato un timer, attraverso la funzione di Javascript `setIntervall(funzione, intervallo)`, che al termine dell'intervallo specificato lancia la funzione passatagli come parametro, in questo caso proprio la `gb_Frame()`. Questo metodo ripete il ciclo fetch-decode-execute (ed i comandi di aggiornamento dei timer e degli interrupt) fino a che un nuovo frame non viene composto e stampato, oppure l'emulatore messo in pausa. Terminata la composizione di una schermata, la macchina virtuale attende la fine dell'intervallo per iniziare la preparazione di un nuovo frame.

Il ciclo di esecuzione del programma in memoria viene risolto in un'unica istruzione:

jsgb.gameboy.js.gb_Frame():

```
8     if(!gbHalt) OP[MEMR(PC++)](); else gbCPUTicks=4;
```

se l'esecuzione non è stata fermata (`if(!gbHalt)`), viene caricato l'OPCode della prossima istruzione (`MEMR(PC++)`, *fetch*) ed eseguita la funzione che si trova al corrispondente valore della tabella degli OPCodes (`OP[codice]()`, *decode* ed *execute*). Supponendo che il sistema sia fermo (stato *halted*), il contatore dei cicli di cpu deve comunque aumentare per permettere ai timer di avanzare e alla gpu di terminare il frame che sta costruendo.

2.6 L'interfaccia

Come già annunciato, le due macchine virtuali utilizzano approcci decisamente differenti per implementare l'interfaccia grafica; è simile invece il sistema di lettura degli input. Di seguito verranno analizzati i metodi per realizzare lo scambio di informazioni con l'utente.

2.6.1 Dall'utente alla macchina: l'input

L'immissione di informazioni (come caratteri, comandi o sequenze di istruzioni) nelle macchine virtuali può avvenire tramite tutti i device di input fisici collegati alla macchina reale che ospita l'emulatore, ma non solo. Per realizzare il collegamento, ad esempio, tra la tastiera e la macchina virtuale, è necessario scrivere una funzione che rimanga in attesa di informazioni provenienti dalla tastiera. Quando queste arrivano le traduce e le passa alla cpu emulata, come se arrivassero dall'indirizzo della seriale sul bus della macchina virtuale.

È evidente che questa funzione può restare in ascolto sul device fisico tastiera e tradurre le informazione come desidera, interpretando quindi l'input come una tastiera con i tasti disposti secondo lo standard europeo o americano, oppure come un joypad o qualsiasi altro tipo di device; allo stesso modo può tradurre le informazioni provenienti dal mouse e farlo sembrare un joystic analogico alla macchina virtuale. Non è difficile immaginare nemmeno di automatizzare l'immissione da tastiera (o qualsiasi altro dispositivo) senza che questa sia effettivamente presente: è sufficiente scrivere la precedente funzione in modo che in determinati istanti invii alla macchina virtuale i caratteri desiderati.

Le due macchine prese in esame accettano solamente l'input da tastiera, e lo interpretano utilizzando metodi simili, inoltrandolo come una normale tastiera per quanto riguarda Jslinux, traducendolo negli otto tasti del GameBoy in JSGB.

Il punto comune è la "lettura" della tastiera fisica. In entrambi i casi avviene attraverso il metodo più semplice offerto dal linguaggio: i *gestori degli eventi* del documento HTML.

Si tratta di metodi, scritti dal programmatore, che vengono associati agli eventi dell'elemento *document*, il quale rappresenta la pagina HTML corrente. L'atto appena descritto è molto semplice. Per esempio, in JSGB avviene come di seguito:

JSGB

jsgb.input.js.gb_Init_Input():

```
73    document.onkeydown = gb_OnKeyDown_Event;
74    document.onkeyup = gb_OnKeyUp_Event;
```

dove le due funzioni `gb_OnKeyUp_Event(event)` e `gb_OnKeyDown_Event(event)` sono metodi istanziati nello stesso file che accettano un parametro.

Il succitato parametro contiene tutte le informazioni reperibili sull'evento appena accaduto, ed è la risorsa fondamentale per implementare i metodi

di lettura della tastiera. I campi *which* e *keycode* del parametro *event* contengono il codice ascii del carattere o comando immesso; è grazie a questi campi che i gestori delle due macchine virtuali riescono ad interpretare gli eventi della tastiera e ad inoltrarli al processore. (I campi descritti svolgono questa funzione per gli eventi di tipo key[..], di cue esistono in tutto tre tipi: keyup, keydown o keypress; gli eventi HTML comprendono anche molte altre categorie, ad esempio quelli legati al puntatore, o alle richieste HTTP)

Compreso come funzionano i gestori degli eventi, risulta ora relativamente semplice implementare le funzioni che passano le informazioni alla cpu. Per quanto riguarda l'emulatore di GameBoy, basandosi su un hardware molto semplice, il compito è veramente facile: il chip che controlla gli input è composto in tutto da sei linee di controllo, due di interrogazione e quattro di risposta, disposte come in figura. In questo modo un solo registro ad 8 bit è

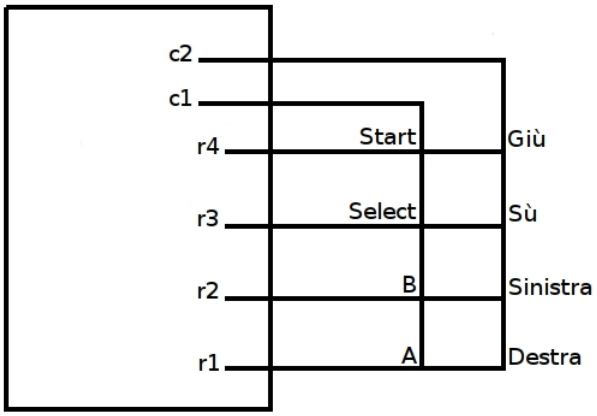

Figura 2.1: GameBoy: chip di controllo della tastiera

sufficiente per rappresentare la tastiera, questo registro è organizzato in modo da avere le linee di risposta (r1-4) che impostano rispettivamente i primi quattro bit, il quinto e sesto bit vengono invece settati dalla cpu, accendendo le linee di controllo (c1-2).

Ad ogni controllo della cpu, cioè ogni volta che imposta ad uno il quinto o il sesto bit del registro, i bit di risposta assumono il valore 0 se il tasto corrispondente è premuto in quel momento, 1 se è rilasciato.

La macchina virtuale tiene in memoria due variabili che rappresentano lo stato di entrambe le linee di controllo. Ad ogni pressione di un tasto sulla tastiera, e per ogni tasto rilasciato, i gestori degli eventi controllano se ne è stato premuto uno di quelli che rappresentano il controller del GameBoy. Se

è questo il caso, la variabile specchio della linea relativa viene aggiornata, e viene impostato l'interrupt della tastiera.

Quando la cpu scrive un valore coerente in questo registro (quindi quando imposta ad 1 i bit 4 o 5), la MMU lancia il metodo di aggiornamento del registro, che consiste nella copia della variabile mirror richiesta nel registro stesso.

Jslinux

Questo caso è di facile implementazione perchè l'hardware da emulare è analogo a quello fisico. Si tratta sempre di una tastiera standard.

Gli *handler* (cioè i gestori) degli eventi della tastiera vengono associati in maniera lievemente differente, utilizzando un metodo apposito dell'oggetto document: `document.addEventListener(event, handler)`

term.js.Term.open():

```
57    document.addEventListenerr(keydown, this.keyDownHandler.bind(this))
58    document.addEventListenerr(keypress, this.keyPressHandler.bind(this))
```

Questi due metodi si occupano di comprendere che tasto viene premuto in base alla configurazione della tastiera del browser ospite, ed inoltrano l'informazione al device *seriale* della macchina virtuale. La funzione `keyDownHandler` agisce quando un tasto viene rilasciato, ed invia all'emulatore i caratteri preceduti dalla stringa di controllo "\x1b" ad indicare che quel carattere non è più premuto. Nel caso in cui sia stata rilasciata una combinazione di tasti che indica un comando preciso, viene inviato un valore specifico per ogni comando.

All'altro capo di questa comunicazione, il device emulato salva il byte ricevuto in un registro interno ed aggiorna gli interrupt tramite la propria funzione apposita. Così, se il byte ricevuto era un carattere o un comando valido, viene lanciata una richiesta di interrupt della seriale e la cpu potrà leggere il byte ricevuto all'indirizzo del bus relativo alla seriale.

2.6.2 Dalla macchina all'utente: l'output

Rimane da analizzare ciò che colpisce l'occhio dell'utente: lo schermo, il terminale, il dispositivo che permette alla macchina virtuale di "inviare" informazioni all'uomo.

Sono di notevole interesse entrambe le realizzazioni, le scelte effettuate sono evidentemente dettate da differenti necessità e risultano in implementazioni molto distanti.

Jslinux

Volendo emulare una macchina datata e soprattutto senza nessun tipo di interfaccia grafica, l'autore del software ha deciso di sviluppare un sistema che non avesse bisogno di una scheda grafica. Ciò che è stato realizzato è un terminale che accetta caratteri come input, senza bisogno di inviare informazioni per renderizzare immagini o schermate. Per costruire un dispositivo di questo tipo sono stati creati tre metodi che costruiscono una *table* HTML ed una matrice di caratteri; impostano il contenuto della matrice con le informazioni provenienti dalla cpu ed aggiornano la table inserendovi il contenuto della matrice dei caratteri.

Il sistema implementato funziona in maniera egregia ed offre il look and feel di un vero terminale, è stato realizzato anche un metodo per gestire una semplice colorazione del testo e dello sfondo: ogni carattere che arriva al terminale definisce nei bit meno significativi il codice del carattere da visualizzare, dal sedicesimo al ventiduesimo due triplette di bit che indicano il colore da utilizzare come sfondo e come primo piano. Le due combinazioni fanno riferimento alle posizioni dei due vettori creati nell'inizializzazione, i quali contengono otto valori esadecimali a sei cifre che indicano proprio un *colore RGB*. In questa maniera il sistema operativo può specificare colori differenti per il testo, ad esempio quando lista i files in una directory (comando `ls`): i file normali in bianco, gli eseguibili in verde e le directory in blu.

JSGB

Il GameBoy ha esigenze differenti da quelle prima descritte, necessita infatti di visualizzare principalmente immagini, ed ha quindi bisogno di un'emulazione di un processore video, per quanto semplice.

JSGB non implementa un vero chip grafico, ma sfruttando alcuni metodi ne emula perfettamente il funzionamento. Le funzioni principali hanno sede nel file jsgb.lcd.js, che si occupa della preparazione del già citato *framebuffer*; nel file jsgb.timers.js è implementata la temporizzazione dell'output video.

La funzione `gb_TIMER_Control()` ha il compito di gestire il timing del sistema, sopstando i timer interni e facendo progredire la *scansione* del frame corrente. Dopo ogni istruzione eseguita dalla cpu e check degli interrupt viene la chiamata a questa funzione. Prima viene aggiornato il timer Divider, poi aggiornato lo stato dello schermo e infine l'Internal Timer. Nella seconda parte viene prima aggiornato il contatore dei cicli di sistema locale all'LCD (come viene indicato nel sorgente). In base a questo valore ci si può trovare in quattro momenti precisi, e quindi differenti situazioni:

- alla fine di un frame, è necessario disegnare a video il contenuto del framebuffer, viene lanciata `gb_Framebuffer_to_LCD()`;
- alla fine di una *scanline*, bisogna quindi attendere il tempo che l'hardware impiega a tornare all'inizio della linea successiva, terminata la corrente, un po come sulle macchine da scrivere bisogna spostare il carrello al termine di una riga prima di iniziare la successiva;
- all'inizio di una nuova linea, viene quindi lanciata la funzione che ne elabora il contenuto: `gb_Draw_Scanline()`
- in un punto interno all'elaborazione di una scanline, resta solo da aspettare.

La funzione che svolge tutto il lavoro è dunque `gb_Draw_Scanline()`. Il suo flusso di esecuzione prima di tutto controlla se nella memoria grafica sono stati modificati i *tiles*, i quadratini 3x3 composti in maniera simile alle bitmap: ogni "punto" del tile è rappresentato da due bit, il cui valore indica il colore di quel punto; viene controllata anche la sezione di memoria in cui sono salvate le *mappe* per lo sfondo: queste rappresentano il modo in cui affiancare i tiles per comporre il background.

Lo sfondo della scanline corrente viene quindi letto dalla memoria e copiato nel vettore che la rappresenta, a questo punto vengono sovrapposti a quanto disegnato gli *sprites*, che sono gli oggetti grafici che possono essere spostati sullo sfondo. Anche gli sprites sono composti da tiles, ma vengono letti direttamente dalla memoria, per cui non c'è bisogno di nessun aggiornamento.

Il disegno degli sprites è più complesso di quello dello sfondo perchè questi possono trovarsi davanti o dietro lo sfondo ed essere specchiati entrambe le direzioni. Altra variabile da considerare è che uno sprite occupa solo una piccola parte dello spazio di visualizzazione, quindi non è triviale conoscere l'esatto gruppo di sprites che si trovano sulla linea correntemente elaborata. JSGB ha risorse sufficienti per permettersi il metodo più semplice per risolvere il problema: controlla ogni sprite in memoria e appena ne trova uno che rientra nella scanline ne aggiunge il contenuto al vettore d'uscita.

Rimane soltanto la funzione che effettivamente disegna l'immagine sul browser: `gb_Framebuffer_to_LCD()`. L'elemento utilizzato da questo software per "stampare a video" le immagini è il tag HTML `<canvas>`. Questo tag riserva uno spazio in cui è possibile disegnare forme e scritte colorate oppure di accedere direttamente ai dati dell'immagine visualizzata e modificarli, in modo da poter impostare ogni pixel al colore RGBA desiderato.

Quest'ultimo è proprio il metodo utilizzato dalla macchina virtuale: all'invocazione della funzione appena citata viene scorso l'intero framebuffer, che contiene soltanto il valore a due bit del colore di ogni pixel che compone lo schermo, e questo colore viene tradotto in RGB attraverso un semplice array che specifica i valori esadecimali dei colori possibili. I valori esadecimali vengono in ultimo copiati nei dati del canvas e l'immagine così ottenuta è visualizzata grazie alla chiamata

jsgb.lcd.js.gb_Framebuffer_to_LCD():

```
179     gbLCDCtx.putImageData(gbLCDImage, 0, 0);
```

Capitolo 3

Idee per possibili evoluzioni

3.1 Generare suoni

HTML5 mette a disposizione un nuovo tag1 <*audio*>: lo scopo principale è quello di offrire un elemento in grado di gestire in maniera semplice e automatizzata la riproduzione di stream audio. L'uso più generico che si fa di questo tag è quello di creare lettori audio interattivi. I browser attuali supportano tipi diversi di file audio caricati in questa maniera, nello specifico i file Wav sono supportati da Firefox, Opera, Chrome e Safari, ma non da Internet Explorer, che invece accetta gli MP3, che non sono utilizzabili in Firefox e Opera, e ancora gli Ogg Vorbis non sono supportati da Safari ed Internet Explorer. Il costrutto HTML permette di inserire diverse fonti, ciascuna in uno dei formati supportati, ma lascia aperta anche una possibilità molto più ricca ed interessante: scrivere i propri samples di suoni che verranno eseguiti da un sintetizzatore interno al browser.

Esistono due maniere per generare suoni con il browser: la prima consiste nella creazione di un header WAVE contenete la rappresentazione in samples del suono e poi riprodurlo, la seconda sfrutta metodi specifici implementati da Firefox e permette di lasciare un player "in esecuzione" ed inviargli suoni bufferati man mano.

3.1.1 Maggiore compatibilità: l'header WAVE

Esistono, in rete, alcuni esempi di questo approccio (Cf. [6]). Il vantaggio principale è la compatibilità con la maggior parte dei browser, difatti tutti i navigatori che permettono di riprodurre file Wave possono generare suoni in questa maniera.

Nello specifico si tratta di scrivere una funzione che codifichi i dati di un

suono nel formato *LPCM*[1], un metodo per creare i campioni di suono, uno che crei l'header dei vari suoni ed infine una funzione che li riproduca quando necessario.

Le ultime due funzioni sono triviali, le prime due richiedono approfondimento. Prima di tutto, il formato a cui si deve arrivare, LPCM, è una *sequenza di rappresentazioni lineari dell'ampiezza di una forma d'onda in una scansione temporale*. Le informazioni necessarie per concretizzare quest'onda devono essere specificate nel file e comprendono:

- la risoluzione del file, ad esempio 8 o 16 bit;
- la frequenza di campionamento: 16.000, 24.000, 44.100, 48.000, ... scansioni per secondo;
- il segno dei dati, possono essere ampiezze espresse con o senza segno;
- il numero di canali;
- l'interleaving dei canali, cioè la possibile disposizione di più canali in maniera non contigua;
- la disposizione dei bytes, little o big endian;
- i dati veri e propri, cioè le ampiezze di ogni scansione in successione.

Le già citate ampiezze d'onda hanno limiti fissi in base alla risoluzione del file e al parametro del segno: un sample a 16 bit per esempio può assumere valori che vanno da -32768 a 32767 se sono valori con segno, altrimenti i limiti saranno 0 e 65535.

Bisogna dunque creare dei sample che siano facilmente convertibili in questo formato. Un esempio abbastanza adatto è quello offerto da *Supercollider*[2], un linguaggio per la gestione di oscillatori e sintetizzatori, principalmente utilizzato per il *live coding* musicale[3].

In Supercollider gli oscillatori, che rappresentano le onde, assumono in ogni istante il valore del seno dell'onda rappresentata. Ciò significa che l'ampiezza di un onda in ogni momento è compresa tra -1 e 1, valori decisamente adatti ad essere convertiti in un intervallo arbitrario.

Questo linguaggio offre inoltre ottimi esempi di effetti e modifiche controllate delle forme d'onda realizzati tramite semplici funzioni matematiche applicate ai valori degli oscillatori.

[1] Linear Pulse-Code Modulation

[2] http://www.audiosynth.com

[3] Una pratica che consiste nella scrittura di software in tempo reale che generi musica.

Traendo ispirazione da quanto considerato, è quindi possibile realizzare una serie di funzioni per creare e modificare onde sonore, convertirle in LPCM e riprodurle in un browser quando richiesto.

Purtroppo il limite di questo sistema è la necessità di salvare un sample audio prima di riprodurlo, il che impedisce di realizzare un convertitore come i DAC[4] presenti nelle schede audio, che convertono, sintetizzano e riproducono "on the fly" l'audio digitale.

È comunque possibile implementare un rudimentale chip audio monofonico ad 8 bit. A questo scopo sarà necessario generare in fase di inizializzazione una serie di "onde quadre", come quelle dei vecchi circuiti succitati, che differiscano solo nella frequenza, in modo da poterle riprodurre quando arriva la relativa richiesta al dispositivo.

Per ottimizzare le dimensioni dell'applicazione, invece di generare molti campione, se ne può utilizzare uno solo variando il sample-rate così da ottenere frequenze differenti.

Rimane un ultima questione: la temporizzazione. Ammettendo di aver generato durante l'inizializzazione una serie di samples che copre tutta la gamma dei 256 suoni riproducibili da un chip 8-bit, bisogna riprodurli solo per il tempo specificato dall'applicazione. Il tag <audio> permette di riprodurre un suono a ciclo continuo grazie al parametro *loop*, fino a che non viene inviato il comando per fermare l'esecuzione.

Sfortunatamente questa funzionalità non è ancora del tutto perfezionata, quindi è possibile avere dei ritardi fra un ciclo e l'altro o non avere proprio un loop. La soluzione che si adatta meglio alla maggior parte dei browser al momento è quella di utilizzare due audio player che si avviano a vicenda quando sono in procinto di terminare l'esecuzione. In questo modo sarà necessaria una funzione, avviata da un timer poco prima che termini la riproduzione del player A, che avvii il B, e riavvolga quindi la riproduzione del primo. Allo stesso modo un secondo timer attenderà il tempo minimo di riproduzione su B per avviare A e riavvolgere il sercondo.

Sarà quindi sufficiente riprodurre ogni suono a ciclo utilizzando il metodo appena descritto e fermarlo quando è terminato il suo tempo di riproduzione.

3.1.2 Sintesi reale: le funzioni *moz*

Mozilla Firefox è l'unico, tra i maggiori browser odierni, a fornire un set di funzioni di "basso livello" per riprodurre direttamente dati audio non codificati generati da uno script. Questa serie di metodi ha in comune la

[4] Digital-to-Analog Converter

prima sillaba degli identificatori, moz: `mozSetup(channels, sampleRate)`, `mozWriteAudio(buffer)`, `mozCurrentSampleOffset()`.

Il Wiki di Mozilla (Cf. [7]), nella documentazione riguardante il tag <audio>, offre un esempio interessante, un generatore di toni[5]. Questo script crea un riproduttore che continua a processare audio non codificato, ed una funzione che ad una cadenza regolare popola il buffer del player. Quando viene premuto il pulsante di pausa, in realtà il riproduttore continua a processare onde di 0 Hz, e quindi il risultato è l'assenza di suono.

Bastano poche modifiche a questo esempio per ottenere il codice adatto a generare in maniera semplice i suoni prodotti da un chip a 8 bit:

- bisogna minimizzare il ritardo (*lag*) tra il cambiamento di frequenza e l'effettiva variazione sonora;
- è necessario interfacciare il player al chip della macchina virtuale;
- sarebbe opportuno generare onde quadre al posto delle sinusoidi che genera lo script.

Minimizzare il lag

Il codice considerato utilizza un buffer di 500 millisecondi che viene aggiornato ogni 100 millisecondi, questo genera un ritardo udibile tra la la variazione della frequenza sintetizzata e l'effettivo cambiamento del suono. È sufficiente provare il codice per accorgersene: il suono non parte all'immediata pressione del tasto play, ma con un leggero ritardo, e con lo stesso ritardo cambia, o si ferma, quando viene selezionata una frequenza differente oppure premuto il tasto stop.

Per abbattere questo ritardo bisogna diminuire la dimensione del buffer, e conseguentemente la frequenza di aggiornamento. Portando questi due valori rispettivamente a 120 ms e 10 ms, si ottiene un buon compromesso, difatti il sistema non viene sovraccaricato dalle richieste troppo frequenti e il lag (ancora minimamente udibile) è ridotto ad un minimo accettabile.

Generare onde quadre

Come già accennato, una caratteristica dei vecchi chip audio monofonici, era il tipo di onde (e quindi di suoni) che generavano. Trattandosi di circuiti abbastanza semplici, non contenevano un vero e proprio oscillatore in grado di generare un range di valori abbastanza ampio da produrre onde sinusoidali,

[5] http://wiki.mozilla.org/Audio_Data_API#Complete_Example:_Creating_a_Web_Based_Tone_Generator

erano molto più simili al clock della cpu: generando variazioni di tensione tra due livelli, uno alto e uno basso, il suono riproducibile è soltanto quello descritto da un'onda quadra.

Queste onde sono così definite a causa della loro rappresentazione grafica: una serie di segmenti rettilinei paralleli all'asse x con ordinata ai valori massimo e minimo assunti dal grafico, e segmenti verticali che rappresentano le variazioni di tensione, i passaggi cioè dal valore massimo a quello minimo, formando un motivo ondulatorio squadrato.

La connotazione sonora caratteristica di queste onde è il ronzio, tipico delle sonorità *chiptune*[6].

Per generare un onda di questo tipo sono possibili svariati metodi, di seguito ne verranno mostrati due.

Considerando lo sviluppo di questa forma d'onda, la maniera più lineare per descriverla in ogni frame, conoscendone frequenza e sampleRate, è quella di ottenere la dimensione in sample di un ciclo intero d'oscillazione (dividendo il sampleRate per la frequenza) e successivamente, con due controlli annidati, stabilire se nel frame corrente l'onda ha un massimo, un minimo o una transizione:

beeper.html, requestSoundData():

```
72    var k = sampleRate / frequency;
73    var m = k / 2;
74
75    for(var i=0, size=soundData.length; i<size; i++) {
76       var pos = currentSoundSample % k;
77       //se ci troviamo in un punto di massimo o minimo
78       if(pos && (pos != m))
79       {
80          //se è minore del punto medio assume valore massimo
81          if(pos < m)
82             soundData[i] = 0.5;
83          //altrimenti assume valore minimo
84          else
85             soundData[i] = -0.5;
86       }
87       //altrimenti è un valore di transizione e assume 0
88       else
89          soundData[i] = 0;
90       currentSoundSample++;
91    }
```

[6] Chiptune indica un genere di musica elettronica prodotta spesso utilizzando i chip di vecchi computer e console per videogames, conosciuto anche con il nome di musica ad 8 bit

Il secondo metodo proposto sfrutta la sinusoide generata dall'esempio di Mozilla: prima di restituire il valore calcolato applicando la funzione seno al valore dell'onda, tale valore viene arrotondato ad un intero, quindi ad un valore tra -1, 0 e 1:

beeper.html, requestSoundData():

```
72    var k = 2* Math.PI * frequency / sampleRate;
73
74    for (var i=0, size=soundData.length; i<size; i++) {
75       soundData[i] = Math.round(Math.sin(k * currentSoundSample++));
76    }
```

Il secondo sistema risulta essere quello più efficace del precedente, poiché quest'ultimo alle volte genera dei click audio in maniera casuale.

Utilizzando una funzione apposita non è difficile far riprodurre delle vere e prorpie note a questo sistema, espresse come un offset di semitoni: la funzione matematica $f = 440 * 2^{n/12} Hz$ descrive l'andamento delle frequenze in base alle note. Nella formula, f rappresenta la frequenza della nota, e n il numero di semitoni di distanza dal *la centrale*, che ha frequenza 440 Hz.

Considerando le note come una successione di semitoni, e riservando un valore per l'assenza di musica, si costruisce un rudimentale sintetizzatore simile al midi, per come vengono espresse le note. In questo schema si considera la nota 0 come una pausa, dalla 1 alla 72 le note dal Do più basso a quello più alto sei ottave sopra. I valori oltre questa soglia possono essere riservati per sintetizzatori che generino onde differenti, ad esempio altre sei ottave per onde sinusoidali, un valore per ottenere rumore bianco e così via.

Emulare un chip audio

Partendo dall'esempio sopra fornito, il lavoro di "costruzione" di un emulatore di chip audio come il *MOS SID 6581* del Commodore 64 o dello *Yamaha YM3812* delle vecchie schede audio AdLib e SoundBlaster si riduce alla codifica degli oscillatori e dei pochi effetti che vi possono applicare.

Sarà poi sufficiente sostituire il codice che genera l'output degli oscillatori a quello della funzione `requestSoundData(soundData)`; oppure, per mantenere un approccio modulare, far sì che gli oscillatori scrivano su di un buffer, che verrà letto e mandato in riproduzione dalla funzione succitata.

3.2 Scheda di rete per Jslinux

Una delle mancanze di questa macchina virtuale è la possibilità di entrare in contatto con altre macchine, virtuali e non. Per giungere a questo scopo è necessario prima di tutto equipaggiare Jslinux con una scheda di rete emulata.

Come suggerito dall'autore stesso del software, l'hardware che più si adatta a questo proposito, per efficacia e semplicità di implementazione, è la scheda Ethernet *NE2000*. Si tratta di una delle prime schede di rete a basso costo, supportata da una vasta gamma di sistemi operativi.

Altro grosso vantaggio della NE2000 è il bus per cui è realizzata: l'*ISA*[7], un bus parallelo, per PC IBM compatibili, dalla struttura sufficientemente semplice, e in linea con il resto dell'hardware emulato.

I primi due elementi da implementare sono quindi un bus ISA e una scheda di rete NE2000. Esistono svariate macchine virtuali Open Source in cui sono realizzati questi due hardware, non sarà quindi un compito troppo pesante quello di "portare" in JavaScript quelle implementazioni.

Ciò che è realmente complesso è il trasferimento di dati dalla porta Ethernet virtuale della scheda NE2000 alla rete e viceversa.

3.2.1 Livelli di astrazione

L'ambiente in cui funziona JavaScript, come già detto, è il browser, un'applicazione che non ha diretto accesso alla scheda di rete della macchina ospite, e il linguaggio non mette a disposizione metodi per superare questo limite. Non è quindi direttamente possibile instradare all'hardware le informazioni che arrivano dalla scheda di rete emulata.

Il limite di Javascript è proprio il livello di astrazione a cui lavora: l'infrastruttura *TCP/IP* su cui si basa Internet rappresenta il modus operandi dei protocolli di rete utilizzati per la trasmissione di informazioni.

Si può immaginare l'insieme dei protocolli di internet diviso in quattro super-gruppi:

- livello applicazioni (application layer);
- livello di trasporto (transport layer);
- livello di rete (internet layer);
- livello di collegamento (link layer).

Ognuno di questi gruppi svolge compiti differenti e non è a conoscenza di quale protocollo lavori al livello inferiore, né se sia presente un livello superiore. I gruppi sono ordinati per ordine decrescente di livello, più si scende, più ci si avvicina alla comunicazione vera e propria dei dati.

Ethernet è una tecnologia del livello di collegamento, si occupa di inviare o ricevere pacchetti a/da indirizzi appartenenti alla rete locale (LAN) a cui

[7] Industry Standard Architecture

appartiene, creando una connessione diretta tra le macchine (*host-to-host*), che viene mantenuta aperta finché il cavo di rete è collegato alla scheda di rete.

Il browser, invece, per comunicare utilizza protocolli appartenenti al livello applicazione, come *HTTP* o *FTP*, che sono pensati per connessioni brevi tra due interlocutori, senza vincoli di indirizzo (possono quindi appartenere a due sottoreti qualsiasi, non necessariamente alla stessa LAN); connessioni che rimangono attive per il solo tempo necessario a trasferire i dati richiesti dal client al server.

3.2.2 Una possibile soluzione

JavaScript, tuttavia, mette a disposizione un tipo di protocollo che si avvicina a quelli di basso livello: il *Websocket*. È possibile, utilizzando questo metodo, collegare due applicazioni remote. Si tratta infatti di una connessione simile agli *InternetSocket*[8] veri e propri, in grado di mettere in comunicazione applicazioni remote.

Esiste inoltre un'applicazione, *Websockify*, che può essere interposta, come un *proxy*[9], tra l'applicazione che utilizza i Websocket e il suo interlocutore, ed agisce "convertendo" i Websocket in veri InternetSocket.

Ancora non è possibile, con le sole tecnologie appena descritte, realizzare una comunicazione Ethernet in JavaScript. L'elemento mancante è qualche tipo di software che incanale le informazioni ricevute in formato rawTCP dal socket in una connessione Ethernet.

Il programma che realizza questo passaggio è *VDE*[10]. Questo software permette di creare reti locali virtuali, composte da versioni virtuali di tutti i componenti di una rete locale: cavi, host, switch, e slirp[11].

Per risolvere quindi il problema succitato di connettere la scheda di rete emulata ad una LAN, sarà necessario utilizzare le tecnologie esposte: prima di tutto i dati uscenti dalla scheda di rete devono essere incanalati in un Webscoket; questo verrà "tradotto" da Websockify in un InternetSocket

[8] L'InternetSocket, o semplicemente socket, indica un capo di una comunicazione bidirezionale tra processi che passa tramite una rete informatica. È un metodo di comunicazione che trasmette dati "raw TCP", cioè informazioni incapsulate direttamente in pacchetti TCP, senza utilizzare protocolli di livello maggiore, tra applicazioni funzionanti su macchine connesse tramite una rete.

[9] Un proxy è un'applicazione, o una macchina, che agisce come intermediario per le richieste di un client, interrogando altri server e restituendo la risposta ottenuta.

[10] Virtual Distributed Ethernet

[11] Uno slirp è un programma che emula una connessione ad Internet tramite terminale. Nella suite VDE rappresenta il punto della LAN che comunica con Internet.

collegato ad un host di VDE; l'host sarà collegato mediante un cavo virtuale allo slirp, e quindi la comunicazione giungerà ad Internet.

3.2.3 Tradeoff

Rimane ora da valutare quale sia la macchina che deve eseguire questa serie di connessioni: un server dedicato o il client su cui gira l'emulatore?

Sono due i fattori che influenzano questa. Nel caso in cui si decida di lasciare la gestione della connessione al client, sarà necessario fornire un pacchetto che contenga Websockfy e VDE, possibilmente già configurati, che l'utente dovtà eseguire sulla sua macchina per permettere all'emulatore di connettersi, viene quindi a perdersi il vantaggio di dover utilizzare esclusivamente il browser. D'altro canto se si decide di dedicare un server per questo servizio, il numero di utenti che potranno connettersi alla rete tramite la macchina virtuale sarà limitato al numero di connessioni simultanee che il server sopporta.

Questi elementi influenzano, comunque, soltanto la connessione di rete, difatti qualsiasi caso sarà sempre possibile usufruire della macchina virtuale in modalità offline.

3.3 Macchine virtuali in JavaScript

L'analisi appena terminata permette di comprendere quali siano alcuni tra i più efficaci approcci utili a creare una macchina virtuale scritta in questo linguaggio.

Sono state considerate per esteso due tra le maggiori lacune che, ad oggi, affliggono questi software.

Si propone quindi, come obbiettivo futuro, quello di implementare una macchina virtuale, completa in tutte le sue parti, in grado di permettere a ciascun utente di utilizzare hardware e software ormai facenti parte della "storia" dell'informatica.

Seguendo l'idea del Professor Renzo Davoli di creare un museo del software, ed affiancandosi al suo già attivo zoo di sistemi operativi open source[12], si vuole quindi realizzare un'interfaccia, che non abbia limitazioni in termini di utenti simultaneamente connessi, da sostituire al Free Live OS Zoo display[13], e creare un vero e proprio museo online in cui situare emulatori di PC/IBM, Apple II, Commodore 64, Amiga e simili.

[12] http://www.oszoo.org
[13] http://floz.v2.cs.unibo.it:8880/

Bibliografia

[1] Alberto Bottarini. *Guida Javascript: tecniche avanzate.* http://javascript.html.it/, 2009.

[2] Abraham Silberschatz, Peter B. Galvin, Greg Gagne. *Sistemi operativi. Concetti ed esempi.* Pearson, 2006.

[3] MazeGen. *X86 Opcode and Instruction Reference.* http://ref.x86asm.net/, 2009.

[4] Leoš Literak. *80x86 instruction set.* http://www.penguin.cz/ literakl/intel/, 1999.

[5] Imran Nazar. *GameBoy Emulation in JavaScript.* http://imrannazar.com/, 2010-2011.

[6] Steven Wittens. *JavaScript audio synthesis with HTML 5.* http://acko.net/, 2009.

[7] David Humphrey, Corban Brook, Al MacDonald, Yury Delendik, Ricard Marxer, Charles Cliffe. *Defining an Enhanced API for Audio (Draft Recommendation).* https://wiki.mozilla.org/, 2011.

[8] L. Peterson, B. Davie. *Computer Networks: A Systems Approach.* Morgan Kaufmann, Fourth Edition.